Week 6&7: MDPS_POMDP

Resources:
1. MDPS
    1.1. Introduction
    1.2. Assumption of MDPS
    1.3. Domain Models
        1.3.1. Transition Models :
        1.3.2. Reward:
        1.3.3. policy:
        1.3.4. Utility:
    1.4. Definition of MDPs:
2. Solution of MDPS
    2.1. Optimal Policy
    2.2. Value Iteration
        2.2.1. Bellman Equation
        2.2.2. The value iteration algorithm
        2.2.3. Convergence of value iteration
    2.3. Dynamic Programming: Policy Iteration
    2.4. Dynamic Programming
3. Online Planning
    3.1. BackGround
    3.2. Basic Ideas
    3.3. Two basic ways
        Used model
        Simulation-based planning
    3.4. Trajectory-Tree-Based Dynamic Programming
    3.5. Monte Carlo Tree Search (MCTS)
        Basic Ideas
        Process
        Example
        Convergence
        Trade-off
        Another Perspective
4. Partially Observable MDPS
    4.1. Definition
    4.2. Belief State
    4.3. Actions
    4.4. Decision Cycles
    4.5. Value Iteration for POMDPs
        4.5.1. Reward function for belief state
        4.5.2. Terminology and Assumption
        4.5.3. POMDPs value iteration
        4.5.4. Point-based backup
    4.6. POMDPS as continuous-state MDPs
        4.6.1. Model
        4.6.2. Solving POMDPS from continuous-state MDPs perspective

4.6.3. Example

Resources:

- Mainly from Courses IN4010-12 Artificial Intelligence Course at TU Delft
- For "Solutions From MDP" part, revised based on the 4th Chapter of Book *Reinforcement Learning: An Introduction (2rd Edition)*

1. MDPS

1.1. Introduction

In this week, we focus on **Sequential Decision Problems**: in which the agent’s utility depends on a sequence of decisions

1.2. Assumption of MDPS

1. the environment is fully observable
2. actions are unreliable, and transitions are Markovian
3. simply stipulate that in each state s, the agent receives a reward

1.3. Domain Models

1.3.1. Transition Models :

- Markovian Property

1.3.2. Reward:

- The balance of risk and reward changes depending on the value of R(s) for the nonterminal states.

1.3.3. policy:

what the agent should do for any state that the agent might reach $\pi(s)$

- we use the expected utility to measure a policy
- optimal policy: with highest utility

1.3.4. Utility:

the sum of the rewards received

Finite Horizon : Does not mean all state sequences are infinite; it just means that there is no fixed deadline

- always has a stationary policy

infinite horizon

- always do not has a stationary policy

The way to calculate the utility of state of sequences:

Additive Rewards

$$U_h\left([s_0, s_1, s_2, \ldots]\right) = R\left(s_0\right) + R\left(s_1\right) + R\left(s_2\right) + \cdots$$

Discounted Rewards

$$U_h([s_0, s_1, s_2, \cdots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$$

For Infinite Sequences

We can calculate the average reward

1.4. Definition of MDPs:

MDPs: a **sequential decision problem** for a **fully observable, stochastic environment** with a **Markovian transition model and additive rewards**

2. Solution of MDPS

2.1. Optimal Policy

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

- When using the discounted utilities with infinite horizons: the optimal policy is independent of the starting state
- R(s) is the “short term” reward for being in s, whereas U(s) is the “long term” total reward from s onward

2.2. Value Iteration

The basic idea is to calculate the utility of each state and then use the state utilities to select an optimal action in each state.

2.2.1. Bellman Equation

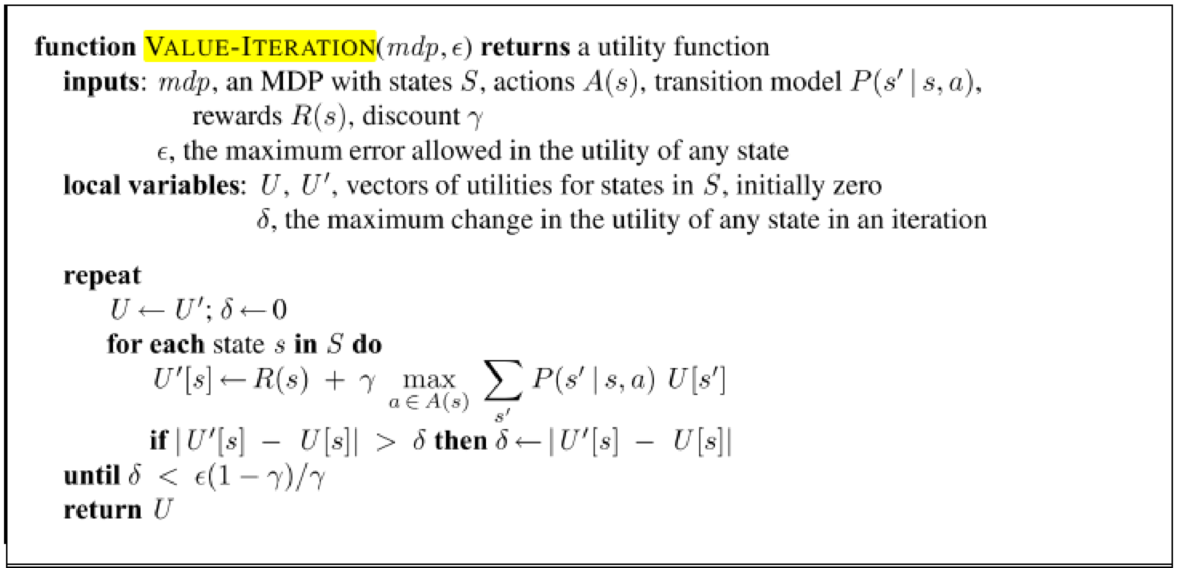
$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P\left(s' \mid s, a\right) U\left(s'\right)$$

- the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action

2.2.2. The value iteration algorithm

- If there are n possible states, then there are n Bellman equations, one for each state. The n equations contain n unknowns. → can be solved
- But max function is nonlinear

Bellman Update



```
function VALUE-ITERATION($mdp, \epsilon$) returns a utility function
  inputs: $mdp$, an MDP with states $S$, actions $A(s)$, transition model $P(s' \mid s, a)$,
              rewards $R(s)$, discount $\gamma$
          $\epsilon$, the maximum error allowed in the utility of any state
  local variables: $U$, $U'$, vectors of utilities for states in $S$, initially zero
                   $\delta$, the maximum change in the utility of any state in an iteration

  repeat
     $U \leftarrow U'$; $\delta \leftarrow 0$
     for each state $s$ in $S$ do
        $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$
        if $|U'[s] - U[s]| > \delta$ then $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return $U$
```

Figure 17.4 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (17.8).

2.2.3. Convergence of value iteration

Contraction

a contraction is a function of one argument that, when applied to
two different inputs in turn, produces two output values that are "closer together," by at least some constant factor, than the original inputs

Theorem 1:

the Bellman update is a contraction by a factor of γ on the space ofutility vectors.

$$\|BU_i - BU_i'\| \leq \gamma \|U_i - U_i'\|$$

Where

$$U_{i+1} \leftarrow BU_i$$
$$\|U\| = \max_s |U(s)|$$

- value iteration always **converges** to a unique solution of the Bellman equations whenever **γ < 1**

Theorem 2:

Theorem 2 is about how well it will do ifit makes its decisions on the basis ofthis utility function

policy loss:

$$\text{if } \|U_i - U\| < \epsilon \text{ then } \|U^{\pi_i} - U\| < 2\epsilon\gamma/(1-\gamma)$$

Where

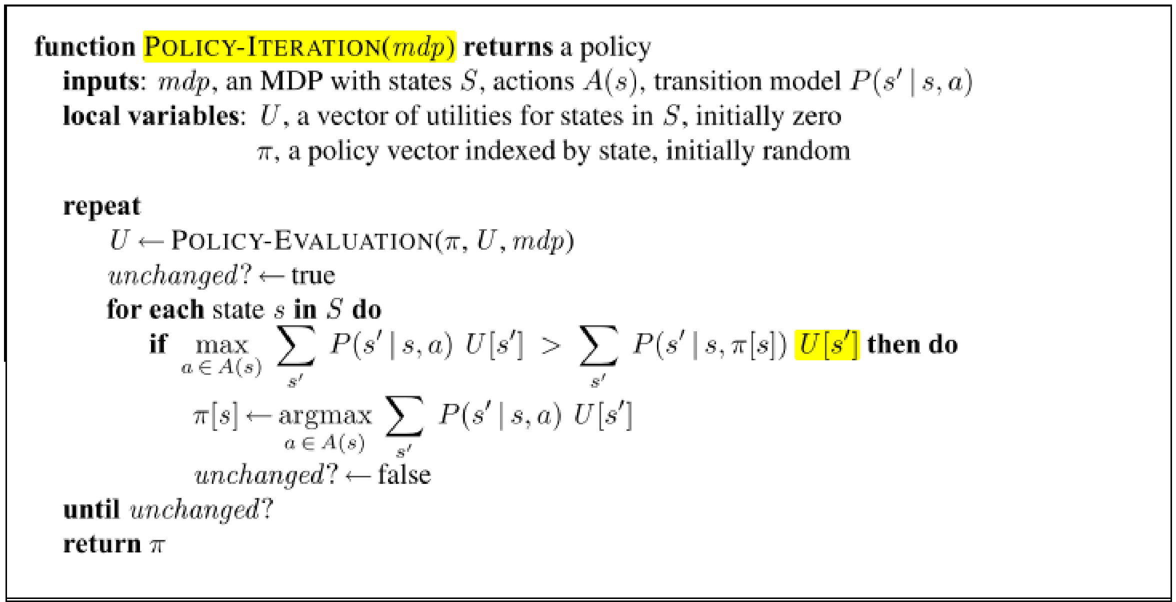
$$\|U^{\pi_i} - U\| \text{ is the most the agent can lose by executing } \pi_i \text{ instead of the optimal policy } \pi^*$$

2.3. Dynamic Programming: Policy Iteration

Two phases:

1. Policy Evaluation: evaluate the utility of each state of a given policy
2. Policy Improvement: Calculate a new MEU policy, using one-step look-ahead based on U_i

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P\left(s' \mid s, \pi_i(s)\right) U_i\left(s'\right)$$



```
function POLICY-ITERATION($mdp$) returns a policy
  inputs: $mdp$, an MDP with states $S$, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: $U$, a vector of utilities for states in $S$, initially zero
                   $\pi$, a policy vector indexed by state, initially random

  repeat
      $U \leftarrow$ POLICY-EVALUATION($\pi, U, mdp$)
      $unchanged? \leftarrow$ true
      for each state $s$ in $S$ do
          if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s'] > \sum_{s'} P(s' \mid s, \pi[s])\ U[s']$ then do
              $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
              $unchanged? \leftarrow$ false
  until $unchanged?$
  return $\pi$
```

Figure 17.7 The policy iteration algorithm for calculating an optimal policy.

- Because there are only finitely many policies for a finite state space, and each iteration can be shown to yield a better policy, policy iteration must terminate

Intuition

Policy Iteration is like the *perpendicular search* in the Optimization Algorithms. The only difference is in perpendicular search we update subject function after each change, but in Policy Iteration, it updates after a iteration.

Compare

1. Value Iteration:

Based on last iteration, we update utility of a state without find an action.

1. Policy Iteration

Based on last iteration (a simplier value iteration process), we calculate utility and then change polilcy

2.4. Dynamic Programming

Use iteration update a table U(state, action)

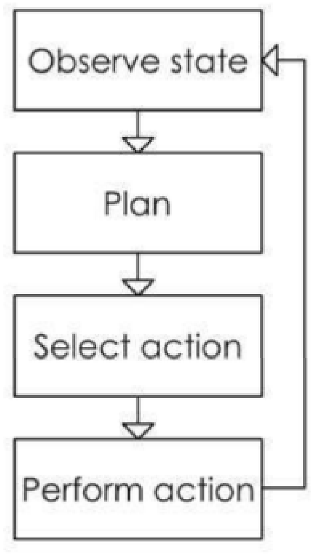
3. Online Planning

3.1. BackGround

For complex problesm, representing a policy as a table may not feasible, so online planning can help

3.2. Basic Ideas

Online planning interleave planning and exetuation



3.3. Two basic ways

Used model

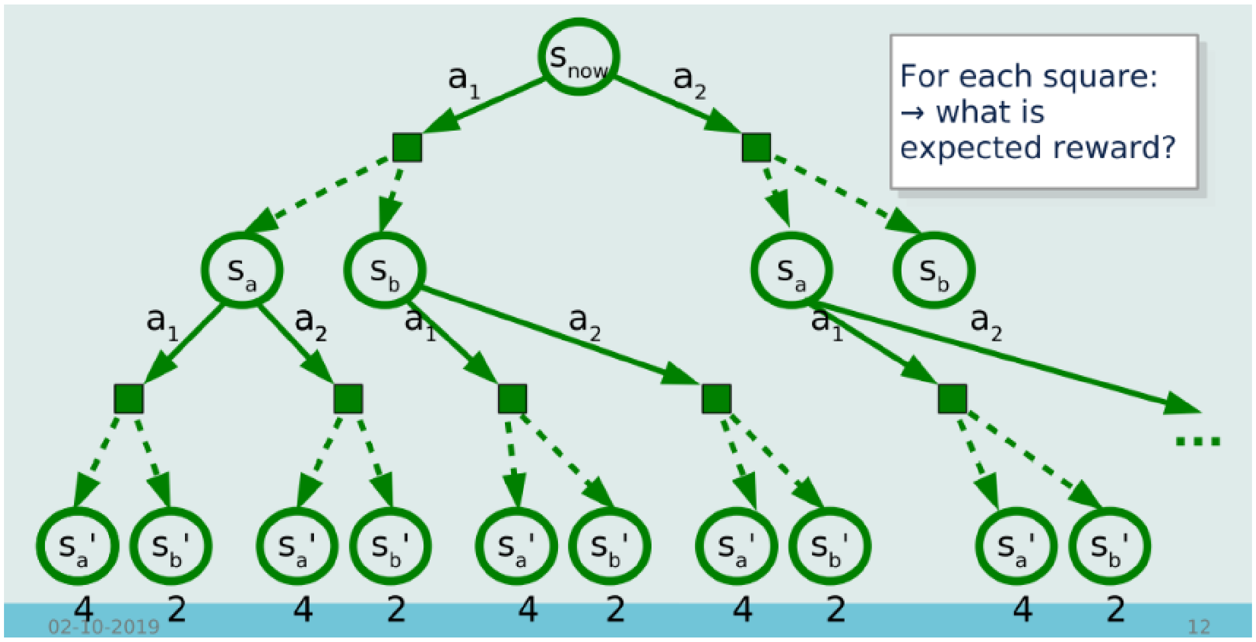
for example dynamic programming

Simulation-based planning

for example Monte Carlo Tree Search(MCTS)

3.4. Trajectory-Tree-Based Dynamic Programming

Different from previous dynamic programming, in online programming we use trajectory trees to do dynamic programming



3.5. Monte Carlo Tree Search (MCTS)

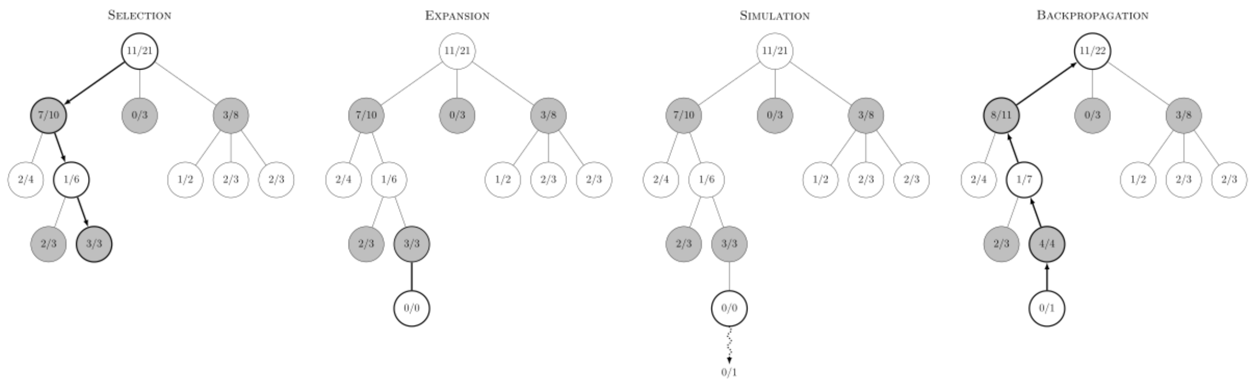
The trajectory tree can still be two large

Basic Ideas

- incrementally constructing a sampled version of the tree
- focusing on promising regions
- Sampling result itself contains the possibility information

Process

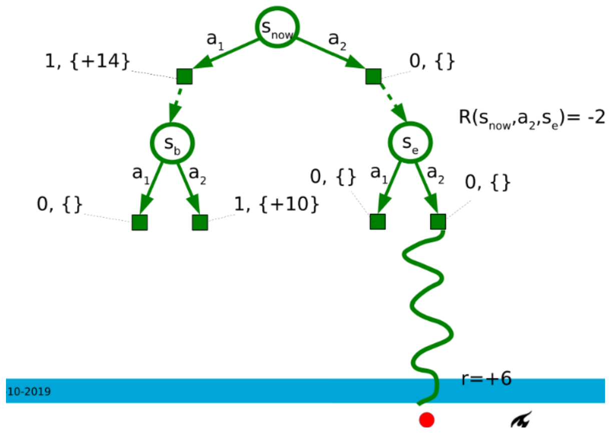
- **Selection**: Start from **root** R and select successive **child nodes** until a **leaf** node L is reached (Sometimes the choose is choose the highest utility way and need trade-off). The root is the current game state and a **leaf** is any node that has a potential child from which no simulation (playout) has yet been initiated. The section below says more about a way of biasing choice of child nodes that lets the game tree expand towards the most promising moves, which is the essence of Monte Carlo tree search.
- **Expansion**: Unless L ends the game decisively (e.g. win/loss/draw) for either player, **create one (or more) child nodes and choose node C from one of them**. Child nodes are any valid moves from the game position defined by .
- **Simulation**: Complete one random playout from node C. This step is sometimes also called **playout or rollout**. A playout may be as simple as choosing uniform random moves until the game is decided (for example in chess, the game is won, lost, or drawn).
- **Backpropagation**: Use the result of the playout to update information in the nodes on the path from C to R .

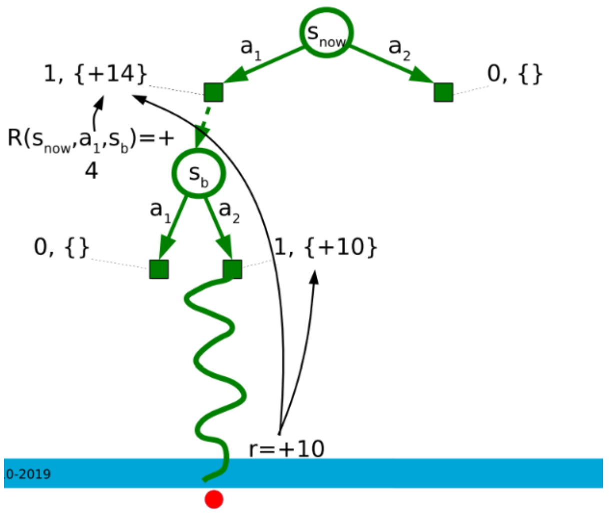


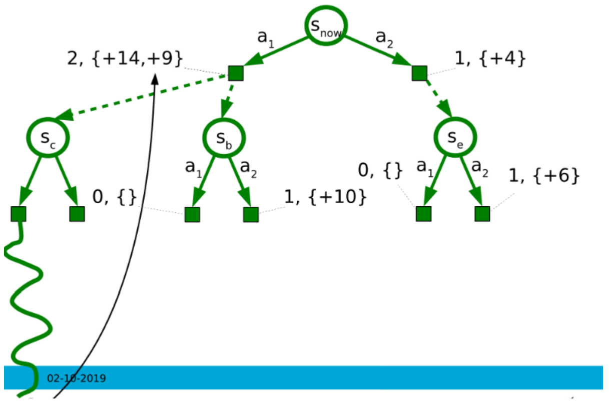
More about Selection Steps:

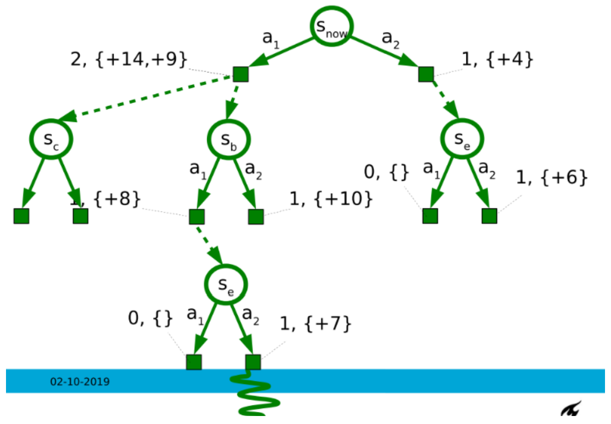
Selection chooses a child to be searched based on previous information. It **controls the balance between exploitation and exploration**. On the one hand, the task consists of selecting the move that leads to the best results so far (exploitation). On the other hand, the less promising moves still have to be tried, due to the uncertainty of the simulations (exploration)

Example









Convergence

MCTS does converge, but requires good strategy

Trade-off

MCTS need to balance between:

Exploitation: focus on good branches

Exploration: see if there could be better branches

<u>**Possible Approach: the UCT algorithm**</u>

$$\underbrace{U(h,a)}_{\text{upper confidence bound of node } h} = \underbrace{Q(h,a)}_{\text{mean return}} + \underbrace{c\sqrt{\log(N_h+1)/N_{h,a}}}_{\text{exploration bonus}}$$

As time goes, at **nodes h**, if we take almost all **a** actions rather than **b** actions, the Utility of doing **b** actions at nodes **h** will grow very big

Another Perspective

Regard as a **policy improvement operator:** you give it a policy, and MCTS makes it better by applying additional search

4. Partially Observable MDPS

4.1. Definition

A POMDP has the same elements as an MDP:

- transition model $P(s\prime|s,\,a)$
- action $A(s)$
- reward function $R(s)$

But different with MDP, it **cannot exactly know which state it is in**, so it also has:

- **sensor model** $P(e|s)$ **,** the sensor model specifies the probability of perceiving evidence e in state s

4.2. Belief State

In order to handle the problem that the agent cannot exactly know which state it is in, it should maintain a **belief state**:

- the set of actual states the agent might be in
- becomes a **probability distribution** over all possible states

Update:

- if we know the current belief and subsequent precept

$$b'\left(s'\right)=\alpha P\left(e \mid s'\right) \sum_{s} P\left(s' \mid s, a\right) b(s)$$
$$b' = FORWARD(b,a,e)$$

- However, **when deciding actions (calculate reward function), we do not know the sequent percept yet**, so we should find another way to update belief, the subsequent percept is not yet known, so the agent might arrive in one of several possible belief states b′,

$$\begin{aligned}P(b' \mid b,a) &= P(b' \mid a,b) = \sum_e P(b' \mid e,a,b)\, P(e \mid a,b) \\ &= \sum P(b' \mid e,a,b) \sum_{e'} P(e \mid s') \sum_s P(s' \mid s,a)\, b(s)\end{aligned}$$

$$\begin{aligned}P(e \mid a,b) &= \sum_{s'} P(e \mid a,s',b)\, P(s' \mid a,b) \\ &= \sum_{s'} P(e \mid s')\, P(s' \mid a,b) \\ &= \sum_{s'} P(e \mid s') \sum_s P(s' \mid s,a)\, b(s).\end{aligned}$$

4.3. Actions

- the optimal action depends only on the agent's current belief state

$$b \;\rightarrow\; \pi$$

- Different from pervious belief update, when deciding actions, we do not know the sequent percept yet, so we should find another way to update belief

4.4. Decision Cycles

1. Given the current belief state b, execute the **action** $a = \pi(b)$.
2. Receive percept e
3. Set the current belief state to **FORWARD(b, a, e)** and repeat

4.5. Value Iteration for POMDPs

4.5.1. Reward function for belief state

$$\rho(b) = \sum_s b(s) R(s)$$

with that, **solving a POMDP on a physical state space** can be reduced to **solving an MDP on the corresponding belief-state space**

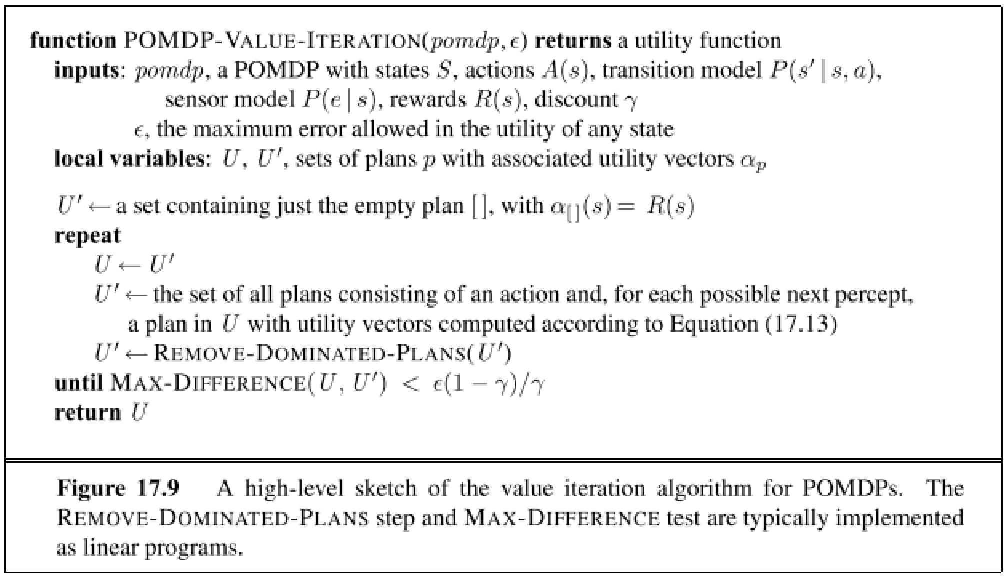
4.5.2. Terminology and Assumption

- Let the **utility** of executing a fixed conditional plan p starting in physical state s be $\alpha_p(s)$. Then the expected utility of executing p in belief state b is just $\sum_s b(s)\alpha_p(s)$, or $b \cdot \alpha_p$, and it is **piecewise linear and convex**
- the expected utility of b under the optimal policy is just the utility of that conditional plan:

$$U(b) = U^{\pi^*}(b) = \max_p b \cdot \alpha_p$$

- we call plan that is suboptimal in entire space are **dominated** plan

4.5.3. POMDPs value iteration



```
function POMDP-VALUE-ITERATION($pomdp, \epsilon$) returns a utility function
  inputs: $pomdp$, a POMDP with states $S$, actions $A(s)$, transition model $P(s' \mid s, a)$,
              sensor model $P(e \mid s)$, rewards $R(s)$, discount $\gamma$
          $\epsilon$, the maximum error allowed in the utility of any state
  local variables: $U$, $U'$, sets of plans $p$ with associated utility vectors $\alpha_p$

  $U' \leftarrow$ a set containing just the empty plan [ ], with $\alpha_{[\,]}(s) = R(s)$
  repeat
      $U \leftarrow U'$
      $U' \leftarrow$ the set of all plans consisting of an action and, for each possible next percept,
          a plan in $U$ with utility vectors computed according to Equation (17.13)
      $U' \leftarrow$ REMOVE-DOMINATED-PLANS($U'$)
  until MAX-DIFFERENCE($U$, $U'$) $< \epsilon(1-\gamma)/\gamma$
  return $U$
```

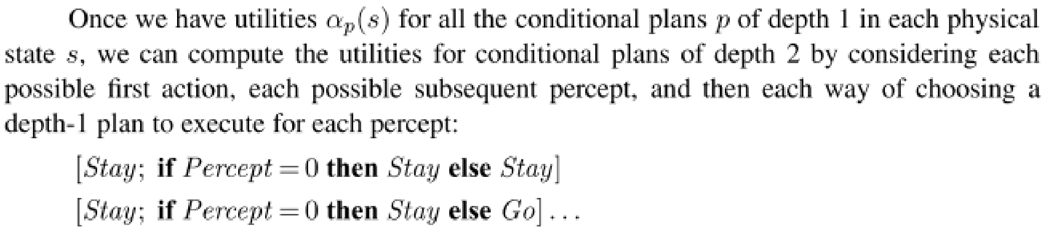
Figure 17.9 A high-level sketch of the value iteration algorithm for POMDPs. The REMOVE-DOMINATED-PLANS step and MAX-DIFFERENCE test are typically implemented as linear programs.

$$\alpha_p(s) = R(s) + \gamma \left(\sum_{s'} P(s' \mid s, a) \sum_{e} P(e \mid s')\, \alpha_{p.e}(s') \right)$$

From the first point in the assumption and terminology, for example, a two state space, the utility of a plan is a linear line in the plane, $V_P(b) = b \cdot \alpha_p$

Another important part is **how α is updated**?

it expanded when we consider **deepr conditional plan**



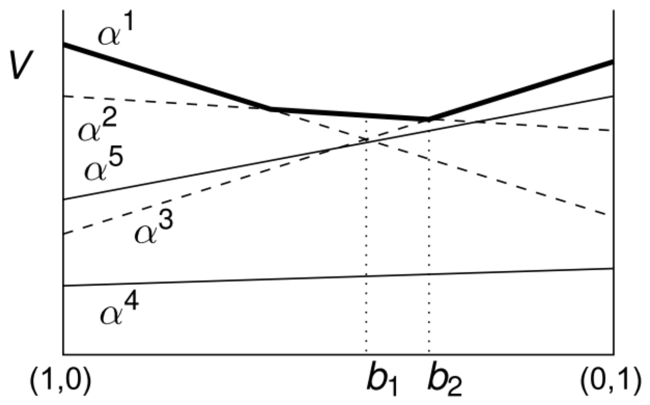
Once we have utilities $\alpha_p(s)$ for all the conditional plans p of depth 1 in each physical state s, we can compute the utilities for conditional plans of depth 2 by considering each possible first action, each possible subsequent percept, and then each way of choosing a depth-1 plan to execute for each percept:

$[Stay;$ **if** $Percept = 0$ **then** $Stay$ **else** $Stay]$
$[Stay;$ **if** $Percept = 0$ **then** $Stay$ **else** $Go] \ldots$

<u>**Drawbacks and Improvements**</u>

- the n**umber of vetors generated** for t+1 steps to go using the exhaustive POMDP backup operator is exponential in the number of observations
- so after each step, we can do some **vetor pruning** to shrink the space

Vector pruning



Linear program for pruning:

variables: $\forall s \in S, b(s); x$
maximize: x
subject to:

$$b \cdot (\alpha - \alpha') \geq x, \forall \alpha' \in V, \alpha' \neq \alpha$$
$$b \in \Delta(S)$$

4.5.4. Point-based backup

Background

Although with vetor pruning, the space may still be very large

Idea

- We can only focus on **important belief points,** it kind like online RL, explore, generate specified beliefs, improve these beliefs
- Given value function V_n and a particular belief point b we can easily compute the vector α^k_{n+1} of HV_n such that

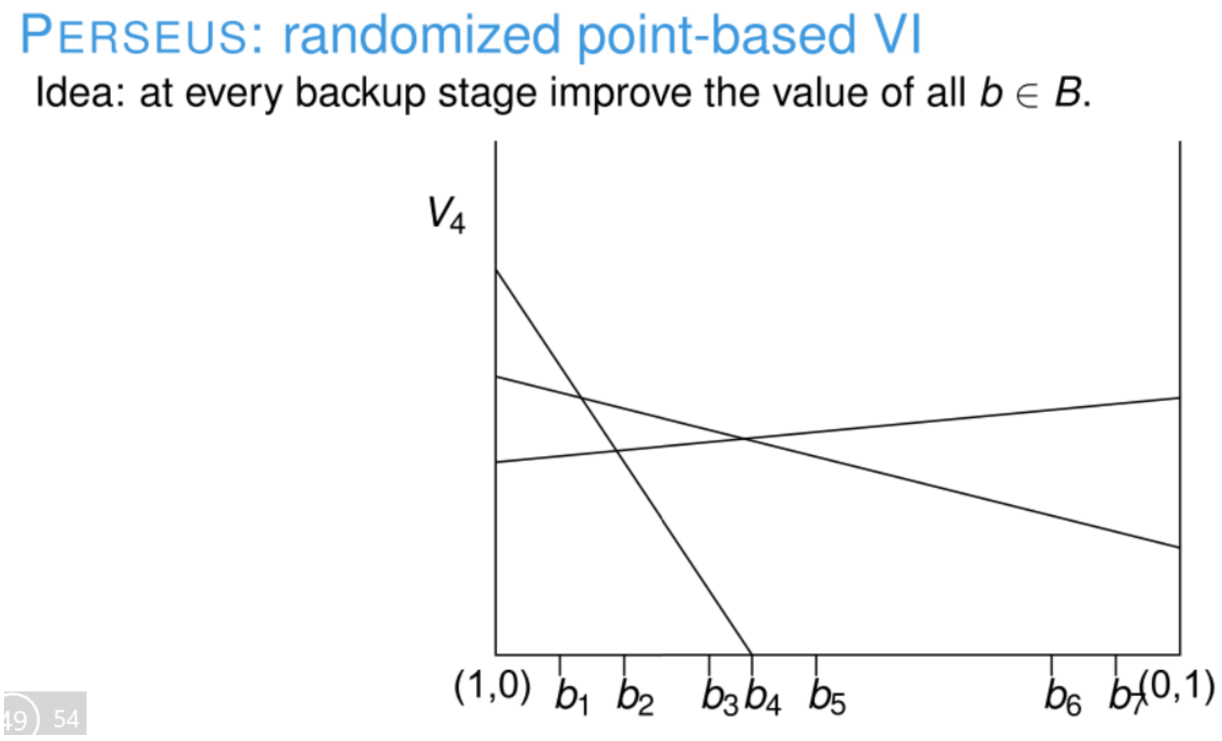
$$\alpha^b_{n+1} = \underset{\{\alpha^k_{n+1}\}_k}{\arg\max} b \cdot \alpha^k_{n+1}$$

where $\{\alpha^k_{n+1}\} k = 1^{|HVn|}$ is the (unknown) set of vectors for HV_n, we will denote this operation $\alpha^b_{n+1} = \text{backup}(b)$

Process

Point-based (approximate) value iteration plans only on **a limited set of reachable belief points:**

- Let the robot **explore** the environment.
- Collect **a set B of belief points**.
- Run **approximate value iteration** on B.



PERSEUS: randomized point-based VI

Idea: at every backup stage improve the value of all $b \in B$.

4.6. POMDPS as continuous-state MDPs

4.6.1. Model

A POMDP can be treated as a **continuous-state (belief-state) MDP**:

- **Continuous state space** Δ: a simplex in $[0,1]^{|S|-1}$.
- Stochastic Markovian transition model $p(b_a^o|b,a) = p(o|b,a)$. This is normalizer of Bayes' rule.
- Reward function $R(b,a) = \sum_s R(s,a)b(s)$. This is the average reward with respsect to $b(s)$
- The agent fully 'observes' the new belief-state the new belief-state b_a^o after executing a and observing o

4.6.2. Solving POMDPS from continuous-state MDPs perspective

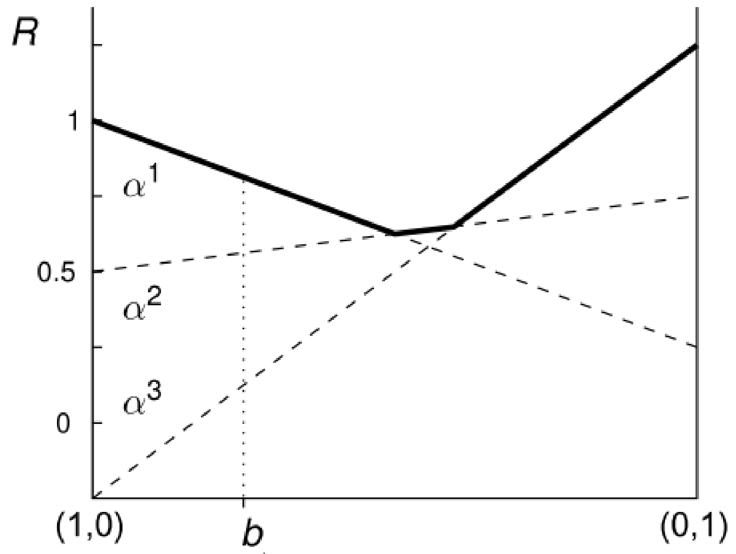
- A solution to a POMDP is a **policy**, i.e., a mapping $\pi : \Delta \rightarrow A$ from **beliefs to actions.**
- The optimal value V^* of a POMDP satisfies the Bellman optimality equation $V^* = HV^*$

$$V^*(b) = \max_a \left[R(b,a) + \gamma \sum_o p(o \mid b,a) V^*\left(b_a^o\right) \right]$$
$$R(s,a) = \sum_s b(s) R(s,a)$$

- Value iteration repeatedly applies $V_{n+1} = HV_n$ starting from an initial V_0
- Computing the optimal value function is **a hard problem**

4.6.3. Example

Example V_0



$R(s,a)$	a_1	a_2	a_3
s_1	1.00	0.50	−0.25
s_2	0.25	0.75	1.25

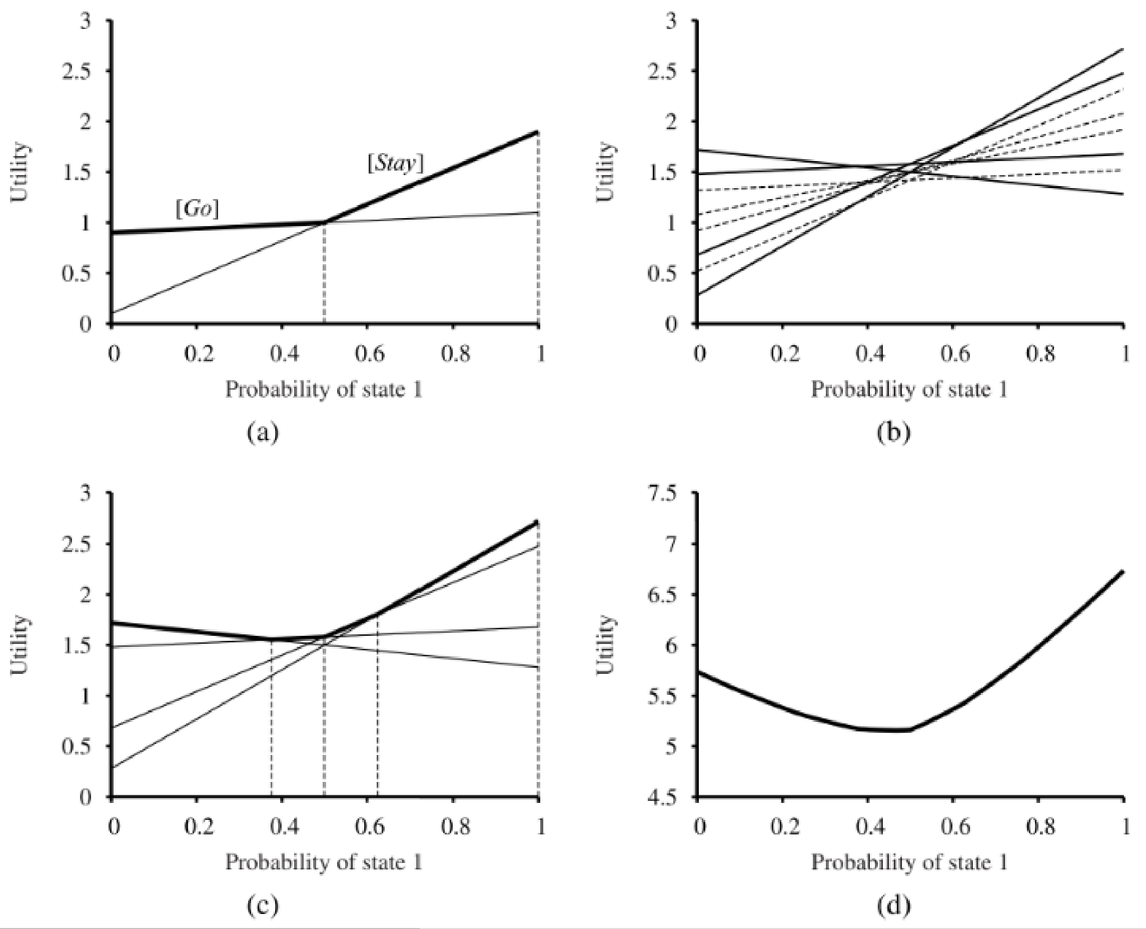


Figure 17.8 (a) Utility of two one-step plans as a function of the initial belief state $b(1)$ for the two-state world, with the corresponding utility function shown in bold. (b) Utilities for 8 distinct two-step plans. (c) Utilities for four undominated two-step plans. (d) Utility function for optimal eight-step plans.