Probabilistic Planning

Blai Bonet

Universidad Simón Bolívar

5th Int. Seminar on New Issues in AI

Madrid, Spain 2012

(. . . references at the end . . .)

General Ideas

‘Planning’ is the **model-based approach** for autonomous behaviour

Focus on most common **planning models** and **algorithms** for:

- **non-deterministic** (probabilistic) actuators (actions)

Ultimate goal is to build planners that solve a class of models

(Intro based on IJCAI’11 tutorial by H. Geffner)

Models, Languages, and Solvers

A planner is a solver over a class of models (problems)

$$Model \quad \Longrightarrow \quad \boxed{\; Planner \;} \quad \Longrightarrow \quad Controller$$

Models, Languages, and Solvers

A planner is a solver over a class of models (problems)

$$Model \quad \Longrightarrow \quad \boxed{\;Planner\;} \quad \Longrightarrow \quad Controller$$

- What is the model? How is the model specified?
- What is a controller? How is the controller specified?

Models

Broad classes given by problem features:

- **actions:** deterministic, non-deterministic, probabilistic
- **agent's information:** complete, partial, none
- **goals:** reachability, maintainability, fairness, LTL, . . .
- **costs:** non-uniform, rewards, non-Markovian, . . .
- **horizon:** finite or infinite
- **time:** discrete or continuous
- . . .

Models

Broad classes given by problem features:

- **actions:** deterministic, non-deterministic, probabilistic
- **agent's information:** complete, partial, none
- **goals:** reachability, maintainability, fairness, LTL, . . .
- **costs:** non-uniform, rewards, non-Markovian, . . .
- **horizon:** finite or infinite
- **time:** discrete or continuous
- . . .

. . . and **combinations** and **restrictions** that define interesting subclasses

Models: Controllers

Solution for a problem is a **controller** that tells the agent what to do at each time point

Form of the controller **depends** on the problem class

E.g., controllers for a deterministic problem with full information **aren't** *of the same form as controllers for a probabilistic problem with incomplete information*

Models: Controllers

Solution for a problem is a **controller** that tells the agent what to do at each time point

Form of the controller **depends** on the problem class

> *E.g., controllers for a deterministic problem with full information* **aren't** *of the same form as controllers for a probabilistic problem with incomplete information*

Characteristics of controllers:

- **consistency:** is the action selected an **executable** action?
- **validity:** does the selected action sequence **achieve** the goal?
- **completeness:** is there a controller that **solves** the problem?

Languages

Models and controllers specified with **representation languages**

Expressivity and **succinctness** have impact on the **complexity** for computing a solution

Languages

Models and controllers specified with **representation languages**

Expressivity and **succinctness** have impact on the **complexity** for computing a solution

Different types of languages:

- **flat languages:** states and actions have no (internal) structure (*good for understanding the model, solutions and algorithms*)

Languages

Models and controllers specified with **representation languages**

Expressivity and **succinctness** have impact on the **complexity** for computing a solution

Different types of languages:

- **flat languages:** states and actions have no (internal) structure
  (*good for understanding the model, solutions and algorithms*)
- **factored languages:** states and actions are specified with variables
  (*good for describing complex problem with few bits*)

Languages

Models and controllers specified with **representation languages**

Expressivity and **succinctness** have impact on the **complexity** for computing a solution

Different types of languages:

- **flat languages:** states and actions have no (internal) structure
  (*good for understanding the model, solutions and algorithms*)
- **factored languages:** states and actions are specified with variables
  (*good for describing complex problem with few bits*)
- **implicit, thru functions:** states and actions **directly** coded
  (*good for efficiency, used to deploy*)

Solvers

Algorithms whose input is a model and output is a controller

Characteristics of solvers:

- **soundness:** the output controller is a valid controller
- **completeness:** if there is a controller that solves problem, the solver outputs one such controller; else, it reports unsolvability
- **optimality:** the output controller is best (under certain criteria)

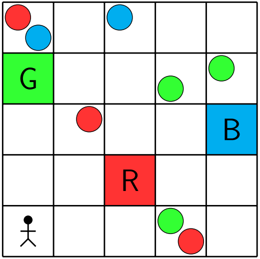
Three Levels

- **Mathematical models** for crisp formulation of classes and solutions
- **Algorithms** that solve these models, which are specified with . . .
- **Languages** that describe the inputs and outputs

Outline

- Introduction (almost done!)
- Part I: Markov Decision Processes (MDPs)
- Part II: Algorithms
- Part III: Heuristics
- Part IV: Monte-Carlo Planning

Example: Collecting Colored Balls



Task: agent picks and delivers balls

Goal: all balls delivered in correct places

Actions: Move, Pick, Drop

Costs: 1 for each action, -100 for 'good' drop

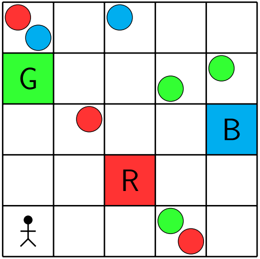
Example: Collecting Colored Balls

Task: agent picks and delivers balls

Goal: all balls delivered in correct places

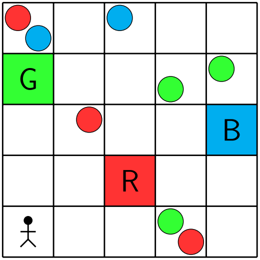
Actions: Move, Pick, Drop

Costs: 1 for each action, -100 for ‘good’ drop



- if deterministic actions and initial state known, problem is **classical**
- if stochastic actions and state is observable, problem is **MDP**
- if stochastic actions and partial information, problem is **POMDP**

Example: Collecting Colored Balls



Task: agent picks and delivers balls

Goal: all balls delivered in correct places

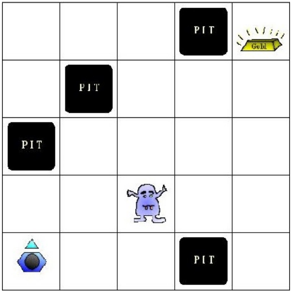
Actions: Move, Pick, Drop

Costs: 1 for each action, -100 for ‘good’ drop

- if deterministic actions and initial state known, problem is **classical**
- if stochastic actions and state is observable, problem is **MDP**
- if stochastic actions and partial information, problem is **POMDP**

Different combinations of uncertainty and feedback:
three problems, three models

Another Example: Wumpus World



Performance measure:

- Gold (reward 1000), death (cost 1000)
- 1 unit cost per movement, 10 for throwing arrow

Environment:

- Cells adjacent to Wumpus smell
- Cells adjacent to Pit are breezy
- Glitter if in same cell as gold
- Shooting kill Wumpus if facing it
- Only one arrow available for shooting
- Grabbing gold picks it if in same cell

Actuators: TurnLeft, TurnRight, MoveForward, Grab, Shoot

Sensors: Smell, Breeze, Glitter

Part I

Markov Decision Processes (MDPs)

Goals of Part I: MDPs

- Models for probabilistic planning
    - Understand the underlying model
    - Understand the solutions for these models
    - Familiarity with notation and formal methods

Classical Planning: Simplest Model

Planning with **deterministic** actions under **complete knowledge**

Characterized by:

- a finite **state space** S
- a finite set of **actions** A; $A(s)$ are actions **executable** at s
- **deterministic** transition function $f : S \times A \to S$ such that $f(s, a)$ is state after applying action $a \in A(s)$ in state s
- **known** initial state s_{init}
- subset $G \subseteq S$ of **goal states**
- **positive costs** $c(s, a)$ of applying action a in state s

(often, $c(s, a)$ only depends on a)

Classical Planning: Simplest Model

Planning with **deterministic** actions under **complete knowledge**

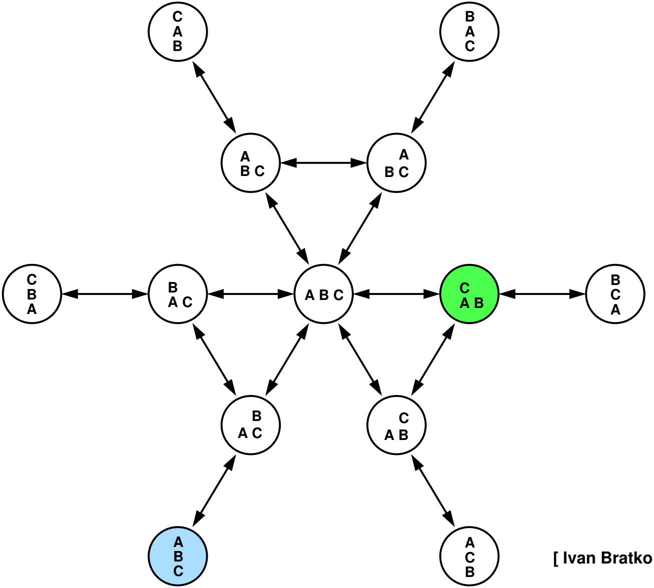
Characterized by:

- a finite **state space** S
- a finite set of **actions** A; $A(s)$ are actions **executable** at s
- **deterministic** transition function $f : S \times A \rightarrow S$ such that $f(s, a)$ is state after applying action $a \in A(s)$ in state s
- **known** initial state s_{init}
- subset $G \subseteq S$ of **goal states**
- **positive costs** $c(s, a)$ of applying action a in state s

(often, $c(s, a)$ only depends on a)

Abstract model that works at 'flat' representation of problem

Classical Planning: Blocksworld



[Ivan Bratko]

Classical Planning: Solutions

Since the initial state is **known** and the effects of the actions can be **predicted**, a controller is a **fixed** action sequence $\pi = \langle a_0, a_1, \ldots, a_n \rangle$

The sequence defines a **state trajectory** $\langle s_0, s_1, \ldots, s_{n+1} \rangle$ where:

- $s_0 = s_{init}$ is the initial state
- $a_i \in A(s_i)$ is an applicable action at state s_i, $i = 0, \ldots, n$
- $s_{i+1} = f(s_i, a_i)$ is the result of applying action a_i at state s_i

The controller is **valid** (i.e., solution) iff s_{n+1} is a goal state

Its **cost** is $c(\pi) = c(s_0, a_0) + c(s_1, a_1) + \cdots + c(s_n, a_n)$

It is **optimal** if its cost is minimum among all solutions

Actions with Uncertain Effects

- Certain problems have actions whose behaviour is **non-deterministic**

  E.g., tossing a coin or rolling a dice are actions whose outcomes cannot be predicted with certainty

- In other cases, uncertainty is the result of a **coarse model** that doesn't include all the information required to predict the outcomes of actions

In both cases, it is necessary to consider problems with non-deterministic actions

Extending the Classical Model with Non-Det Actions but Complete Information

- A finite state space S
- a finite set of actions A; $A(s)$ are actions executable at sS
- **non-deterministic** transition function $F : S \times A \rightarrow 2^S$ such that $F(s,a)$ is set of states that **may** result after executing a at s
- initial state s_{init}
- subset $G \subseteq S$ of goal states
- positive costs $c(s,a)$ of applying action a in state s

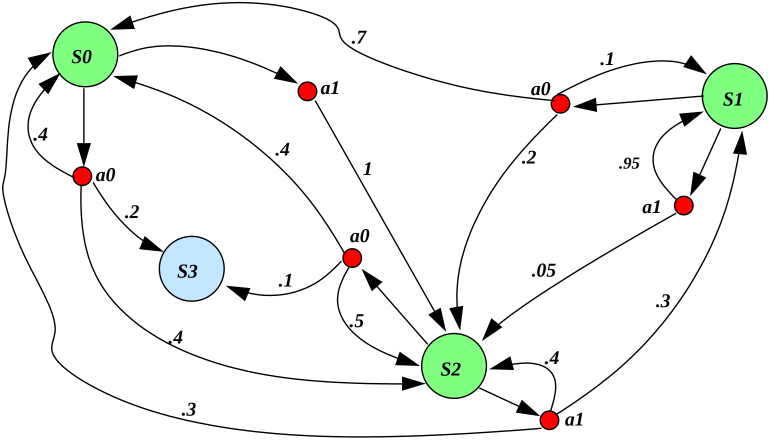
States are assumed to be fully observable

Mathematical Model for Probabilistic Planning

- A finite state space S
- a finite set of actions A; $A(s)$ are actions executable at sS
- **stochastic** transitions given by **distributions** $p(\cdot|s,a)$ where $p(s'|s,a)$ is the probability of reaching s' when a is executed at s
- initial state s_{init}
- subset $G \subseteq S$ of goal states
- positive costs $c(s,a)$ of applying action a in state s

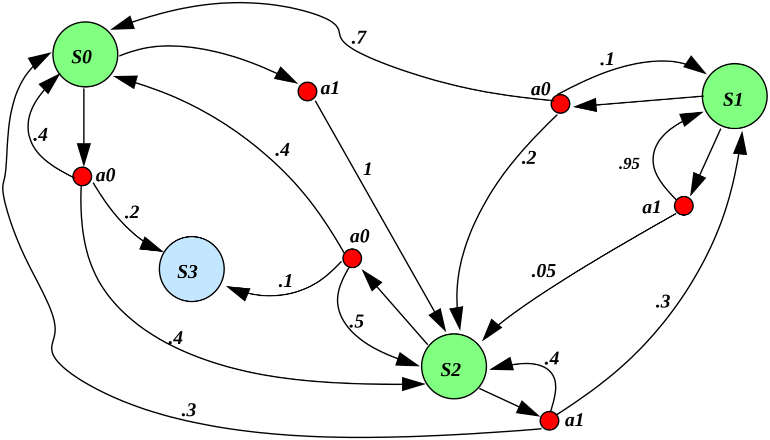
States are assumed to be fully observable

Example: Simple Problem



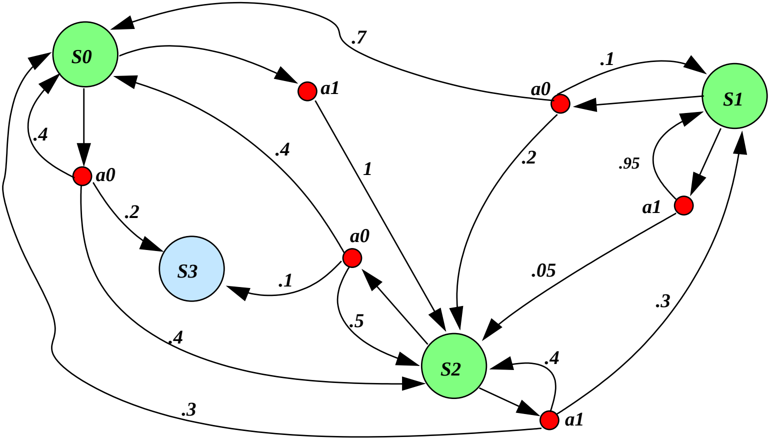
- 4 states; $S = \{s_0, \dots, s_3\}$
- 2 actions; $A = \{a_0, a_1\}$
- 1 goal; $G = \{s_3\}$

Example: Simple Problem



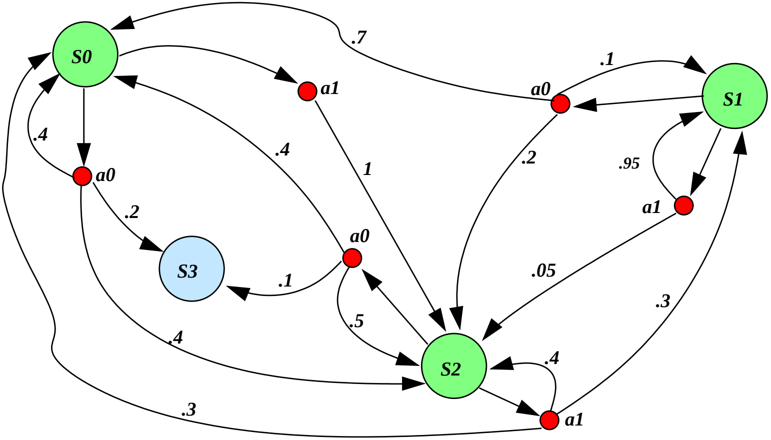
- 4 states; $S = \{s_0, \ldots, s_3\}$
- 2 actions; $A = \{a_0, a_1\}$
- 1 goal; $G = \{s_3\}$
- $p(s_2|s_0, a_1) = 1.0$

Example: Simple Problem



- 4 states; $S = \{s_0, \ldots, s_3\}$
- 2 actions; $A = \{a_0, a_1\}$
- 1 goal; $G = \{s_3\}$
- $p(s_2|s_0, a_1) = 1.0$
- $p(s_0|s_1, a_0) = 0.7$

Example: Simple Problem



- 4 states; $S = \{s_0, \ldots, s_3\}$
- 2 actions; $A = \{a_0, a_1\}$
- 1 goal; $G = \{s_3\}$

- $p(s_2|s_0, a_1) = 1.0$
- $p(s_0|s_1, a_0) = 0.7$
- $p(s_2|s_2, a_1) = 0.4$

Controllers

A controller **cannot** be a sequence of actions because the agent **cannot predict with certainty** what would be the future state

However, since states are fully observable, the agent can be **prepared** for any **possible future state**

Such controller is called **contingent** with full observability

Contingent Plans

Many ways to represent contingent plans. Most general correspond to **sequence of functions** that map states into actions

Contingent Plans

Many ways to represent contingent plans. Most general correspond to **sequence of functions** that map states into actions

Definition

A **contingent plan** *is a sequence* $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ *of* **decision functions** $\mu_i : S \to A$ *such that the agent executes action* $\mu_i(s)$ *when the state at time* i *is* s

Contingent Plans

Many ways to represent contingent plans. Most general correspond to **sequence of functions** that map states into actions

Definition

A **contingent plan** *is a sequence* $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ *of* **decision functions** $\mu_i : S \to A$ *such that the agent executes action* $\mu_i(s)$ *when the state at time* i *is* s

The plan is **consistent** if for every s and i, $\mu_i(s) \in A(s)$

Contingent Plans

Many ways to represent contingent plans. Most general correspond to **sequence of functions** that map states into actions

Definition

A **contingent plan** *is a sequence* $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ *of* **decision functions** $\mu_i : S \to A$ *such that the agent executes action* $\mu_i(s)$ *when the state at time* i *is* s

The plan is **consistent** if for every s and i, $\mu_i(s) \in A(s)$

Because of non-determinism, a **fixed** plan π executed at **fixed** initial state s may generate **more than one** state trajectory

Example: Solution

$$\mu_0 = (a_0, a_0, a_0)$$

$$\mu_1 = (a_0, a_0, a_1)$$

$$\mu_2 = (a_0, a_1, a_0)$$

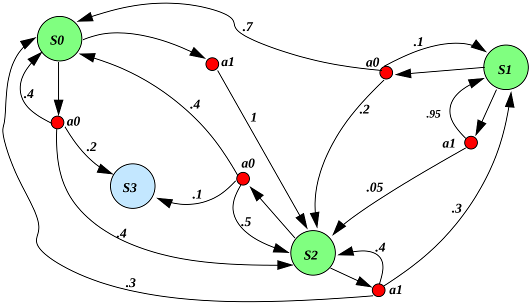
$$\mu_3 = (a_0, a_1, a_1)$$

$$\mu_4 = (a_1, a_0, a_0)$$

$$\mu_5 = (a_1, a_0, a_1)$$

$$\mu_6 = (a_1, a_1, a_0)$$

$$\mu_7 = (a_1, a_1, a_1)$$



$$\pi_0 = \langle \mu_0, \mu_1, \mu_0, \mu_1, \mu_0, \mu_1, \mu_0, \mu_1, \mu_0, \ldots \rangle$$

$$\pi_1 = \langle \mu_5, \mu_5, \mu_5, \mu_5, \mu_5, \ldots \rangle$$

$$\pi_2 = \langle \mu_0, \mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6, \mu_7, \mu_0, \ldots \rangle$$

$$\pi_3 = \langle \mu_2, \mu_3, \mu_5, \mu_7, \mu_2, \ldots \rangle$$

Contingent Plans

For plan $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ and initial state s, the possible trajectories are the sequences $\langle s_0, s_1, \ldots \rangle$ such that

- $s_0 = s$
- $s_{i+1} \in F(s_i, \mu_i(s_i))$
- if $s_i \in G$, then $s_{i+1} = s_i$ ⟵ (mathematically convenient)

Contingent Plans

For plan $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ and initial state s, the possible trajectories are the sequences $\langle s_0, s_1, \ldots \rangle$ such that

- $s_0 = s$
- $s_{i+1} \in F(s_i, \mu_i(s_i))$
- if $s_i \in G$, then $s_{i+1} = s_i$ $\quad \longleftarrow \quad$ (mathematically convenient)

How do we define the cost of a controller?

Contingent Plans

For plan $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ and initial state s, the possible trajectories are the sequences $\langle s_0, s_1, \ldots \rangle$ such that

- $s_0 = s$
- $s_{i+1} \in F(s_i, \mu_i(s_i))$
- if $s_i \in G$, then $s_{i+1} = s_i$ $\longleftarrow$ (mathematically convenient)

How do we define the cost of a controller?

What is a valid controller (solution)?

Contingent Plans

For plan $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ and initial state s, the possible trajectories are the sequences $\langle s_0, s_1, \ldots \rangle$ such that

- $s_0 = s$
- $s_{i+1} \in F(s_i, \mu_i(s_i))$
- if $s_i \in G$, then $s_{i+1} = s_i$ $\longleftarrow$ (mathematically convenient)

How do we define the cost of a controller?

What is a valid controller (solution)?

How do we compare two controllers?

Example: Trajectories

$\mu_0 = (a_0, a_0, a_0)$

$\mu_1 = (a_0, a_0, a_1)$

$\mu_2 = (a_0, a_1, a_0)$

$\mu_3 = (a_0, a_1, a_1)$

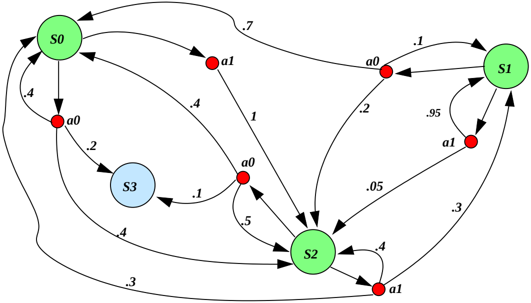
$\mu_4 = (a_1, a_0, a_0)$

$\mu_5 = (a_1, a_0, a_1)$

$\mu_6 = (a_1, a_1, a_0)$

$\mu_7 = (a_1, a_1, a_1)$

$\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$



Trajectories starting at s_0:

$\langle s_0, s_2, s_3, s_3, \ldots \rangle$

$\langle s_0, s_2, s_0, s_2, s_3, \ldots \rangle$

$\langle s_0, s_2, s_2, s_2, s_2, s_2, s_3, \ldots \rangle$

Cost of Plans (Intuition)

Each trajectory $\tau = \langle s_0, s_1, \ldots \rangle$ has **probability**

$$P(\tau) = p(s_1|s_0, \mu_0(s_0)) \cdot p(s_2|s_1, \mu_1(s_1)) \cdots$$

where $p(s|s,a) = 1$ for all $a \in A$ when $s \in G$ (convenience)

Cost of Plans (Intuition)

Each trajectory $\tau = \langle s_0, s_1, \ldots \rangle$ has **probability**

$$P(\tau) = p(s_1|s_0, \mu_0(s_0)) \cdot p(s_2|s_1, \mu_1(s_1)) \cdots$$

where $p(s|s, a) = 1$ for all $a \in A$ when $s \in G$ (convenience)

Each trajectory has **cost**

$$c(\tau) = c(s_0, \mu_0(s_0)) + c(s_1, \mu_1(s_1)) + \cdots$$

where $c(s, a) = 0$ for all $a \in A$ and $s \in G$ (convenience)

Cost of Plans (Intuition)

Each trajectory $\tau = \langle s_0, s_1, \ldots \rangle$ has **probability**

$$P(\tau) = p(s_1|s_0, \mu_0(s_0)) \cdot p(s_2|s_1, \mu_1(s_1)) \cdots$$

where $p(s|s,a) = 1$ for all $a \in A$ when $s \in G$ (convenience)

Each trajectory has **cost**

$$c(\tau) = c(s_0, \mu_0(s_0)) + c(s_1, \mu_1(s_1)) + \cdots$$

where $c(s,a) = 0$ for all $a \in A$ and $s \in G$ (convenience)

Therefore, the **cost of policy** π at state s is

$$J_\pi(s) = \textstyle\sum_\tau c(\tau) \cdot P(\tau) \qquad \textbf{(expected cost)}$$

Example: Cost of Plan

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$

Trajectories can be reduced to (using $p = \frac{2}{10}$ and $q = \frac{8}{10}$):

$\tau = \langle s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = p$ and $c(\tau) = 1 + 2$

Example: Cost of Plan

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$

Trajectories can be reduced to (using $p = \frac{2}{10}$ and $q = \frac{8}{10}$):

$\tau = \langle s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = p$ and $c(\tau) = 1 + 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq$ and $c(\tau) = 2 + 2 \cdot 2$

Example: Cost of Plan

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$

Trajectories can be reduced to (using $p = \frac{2}{10}$ and $q = \frac{8}{10}$):

$\tau = \langle s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = p$ and $c(\tau) = 1 + 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq$ and $c(\tau) = 2 + 2 \cdot 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_0, s_2, s_3, \ldots \rangle$ with $P(\tau) = pq^2$ and $c(\tau) = 3 + 3 \cdot 2$

Example: Cost of Plan

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$

Trajectories can be reduced to (using $p = \frac{2}{10}$ and $q = \frac{8}{10}$):

$\tau = \langle s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = p$ and $c(\tau) = 1 + 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq$ and $c(\tau) = 2 + 2 \cdot 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_0, s_2, s_3, \ldots \rangle$ with $P(\tau) = pq^2$ and $c(\tau) = 3 + 3 \cdot 2$

$\tau = \langle \underbrace{s_0, s_2}_{k+1 \text{ times}}, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq^k$ and $c(\tau) = 3(k + 1)$

Example: Cost of Plan

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$

Trajectories can be reduced to (using $p = \frac{2}{10}$ and $q = \frac{8}{10}$):

$\tau = \langle s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = p$ and $c(\tau) = 1 + 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq$ and $c(\tau) = 2 + 2 \cdot 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_0, s_2, s_3, \ldots \rangle$ with $P(\tau) = pq^2$ and $c(\tau) = 3 + 3 \cdot 2$

$\tau = \langle \underbrace{s_0, s_2}_{k+1 \text{ times}}, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq^k$ and $c(\tau) = 3(k+1)$

Cost of policy from s_0:

$$J_\pi(s_0) = \sum_{k \geq 0} 3(k+1)pq^k$$

Example: Cost of Plan

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$

Trajectories can be reduced to (using $p = \frac{2}{10}$ and $q = \frac{8}{10}$):

$\tau = \langle s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = p$ and $c(\tau) = 1 + 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq$ and $c(\tau) = 2 + 2 \cdot 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_0, s_2, s_3, \ldots \rangle$ with $P(\tau) = pq^2$ and $c(\tau) = 3 + 3 \cdot 2$

$\tau = \langle \underbrace{s_0, s_2}_{k+1 \text{ times}}, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq^k$ and $c(\tau) = 3(k+1)$

Cost of policy from s_0:

$$J_\pi(s_0) = \sum_{k \geq 0} 3(k+1)pq^k = 3p \sum_{k \geq 0} (k+1)q^k$$

Example: Cost of Plan

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$

Trajectories can be reduced to (using $p = \frac{2}{10}$ and $q = \frac{8}{10}$):

$\tau = \langle s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = p$ and $c(\tau) = 1 + 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq$ and $c(\tau) = 2 + 2 \cdot 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_0, s_2, s_3, \ldots \rangle$ with $P(\tau) = pq^2$ and $c(\tau) = 3 + 3 \cdot 2$

$\tau = \langle \underbrace{s_0, s_2}_{k+1 \text{ times}}, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq^k$ and $c(\tau) = 3(k+1)$

Cost of policy from s_0:

$$J_\pi(s_0) = \sum_{k \geq 0} 3(k+1)pq^k = 3p \sum_{k \geq 0} (k+1)q^k = 3p \sum_{k \geq 0} [kq^k + q^k]$$

Example: Cost of Plan

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$

Trajectories can be reduced to (using $p = \frac{2}{10}$ and $q = \frac{8}{10}$):

$\tau = \langle s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = p$ and $c(\tau) = 1 + 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq$ and $c(\tau) = 2 + 2 \cdot 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_0, s_2, s_3, \ldots \rangle$ with $P(\tau) = pq^2$ and $c(\tau) = 3 + 3 \cdot 2$

$\tau = \langle \underbrace{s_0, s_2}_{k+1 \text{ times}}, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq^k$ and $c(\tau) = 3(k+1)$

Cost of policy from s_0:

$$
\begin{aligned}
J_\pi(s_0) &= \sum_{k \geq 0} 3(k+1)pq^k = 3p \sum_{k \geq 0} (k+1)q^k = 3p \sum_{k \geq 0} [kq^k + q^k] \\
&= 3p \left[\frac{q}{(1-q)^2} + \frac{1}{1-q} \right]
\end{aligned}
$$

Example: Cost of Plan

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$

Trajectories can be reduced to (using $p = \frac{2}{10}$ and $q = \frac{8}{10}$):

$\tau = \langle s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = p$ and $c(\tau) = 1 + 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq$ and $c(\tau) = 2 + 2 \cdot 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_0, s_2, s_3, \ldots \rangle$ with $P(\tau) = pq^2$ and $c(\tau) = 3 + 3 \cdot 2$

$\tau = \langle \underbrace{s_0, s_2}_{k+1 \text{ times}}, s_3, s_3, \ldots \rangle$ with $P(\tau) = pq^k$ and $c(\tau) = 3(k+1)$

Cost of policy from s_0:

$$\begin{aligned} J_\pi(s_0) &= \sum_{k \geq 0} 3(k+1)pq^k = 3p \sum_{k \geq 0} (k+1)q^k = 3p \sum_{k \geq 0} [kq^k + q^k] \\ &= 3p \left[\frac{q}{(1-q)^2} + \frac{1}{1-q} \right] = \frac{3p}{(1-q)^2} \end{aligned}$$

Example: Cost of Plan

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \dots \rangle$

Trajectories can be reduced to (using $p = \frac{2}{10}$ and $q = \frac{8}{10}$):

$\tau = \langle s_0, s_2, s_3, s_3, \dots \rangle$ with $P(\tau) = p$ and $c(\tau) = 1 + 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_3, s_3, \dots \rangle$ with $P(\tau) = pq$ and $c(\tau) = 2 + 2 \cdot 2$

$\tau = \langle s_0, s_2, s_0, s_2, s_0, s_2, s_3, \dots \rangle$ with $P(\tau) = pq^2$ and $c(\tau) = 3 + 3 \cdot 2$

$\tau = \langle \underbrace{s_0, s_2}_{k+1 \text{ times}}, s_3, s_3, \dots \rangle$ with $P(\tau) = pq^k$ and $c(\tau) = 3(k+1)$

Cost of policy from s_0:

$$\begin{aligned} J_\pi(s_0) &= \sum_{k \geq 0} 3(k+1)pq^k = 3p \sum_{k \geq 0} (k+1)q^k = 3p \sum_{k \geq 0} [kq^k + q^k] \\ &= 3p \left[\frac{q}{(1-q)^2} + \frac{1}{1-q} \right] = \frac{3p}{(1-q)^2} = 15 \end{aligned}$$

Cost of Plans (Formal)

Under **fixed** controller $\pi = \langle \mu_0, \mu_1, \ldots \rangle$, the system becomes a **Markov chain** with transition probabilities $p_i(s'|s) = p(s'|s, \mu_i(s))$

Cost of Plans (Formal)

Under **fixed** controller $\pi = \langle \mu_0, \mu_1, \ldots \rangle$, the system becomes a **Markov chain** with transition probabilities $p_i(s'|s) = p(s'|s, \mu_i(s))$

These transitions define **probabilities** P^π_s and **expectations** E^π_s over the trajectories generated by π starting at s

Cost of Plans (Formal)

Under **fixed** controller $\pi = \langle \mu_0, \mu_1, \ldots \rangle$, the system becomes a **Markov chain** with transition probabilities $p_i(s'|s) = p(s'|s, \mu_i(s))$

These transitions define **probabilities** P_s^π and **expectations** E_s^π over the trajectories generated by π starting at s

Let X_i be the r.v. that is the state of the chain at time i; e.g.,

Cost of Plans (Formal)

Under **fixed** controller $\pi = \langle \mu_0, \mu_1, \ldots \rangle$, the system becomes a **Markov chain** with transition probabilities $p_i(s'|s) = p(s'|s, \mu_i(s))$

These transitions define **probabilities** P_s^π and **expectations** E_s^π over the trajectories generated by π starting at s

Let X_i be the r.v. that is the state of the chain at time i; e.g.,

- $P_s^\pi(X_{10} = s')$ is the probability that the state at time 10 will be s' given that we execute π starting from s

Cost of Plans (Formal)

Under **fixed** controller $\pi = \langle \mu_0, \mu_1, \ldots \rangle$, the system becomes a **Markov chain** with transition probabilities $p_i(s'|s) = p(s'|s, \mu_i(s))$

These transitions define **probabilities** P_s^π and **expectations** E_s^π over the trajectories generated by π starting at s

Let X_i be the r.v. that is the state of the chain at time i; e.g.,

- $P_s^\pi(X_{10} = s')$ is the probability that the state at time 10 will be s' given that we execute π starting from s
- $E_s^\pi[c(X_{10}, \mu_{10}(X_{10}))]$ is the expected cost incurred by the agent at time 10 given that we execute π starting from s

Cost of Controllers (Formal)

Definition

The **cost of policy** π *at state* s *is defined as*

$$J_\pi(s) = E_s^\pi\left[\sum_{i=0}^{\infty} c(X_i, \mu_i(X_i))\right]$$

- J_π is a **vector of costs** $J_\pi(s)$ for each state s
- J_π is called the **value function** for π
- Policy π is better than π' **at state** s iff $J_\pi(s) < J_{\pi'}(s)$

Solutions (Valid Controllers)

Definition

Policy π is **valid for state** s *if π reaches a goal* **with probability 1** *from state s*

Definition

A policy π is **valid** *if it is valid for each state s*

Solutions (Valid Controllers)

Definition

Policy π is **valid for state** s *if π reaches a goal* **with probability 1** *from state s*

Definition

A policy π is **valid** *if it is valid for each state s*

In probabilistic planning, we are interested in solutions valid for the initial state

Time to Arrive to the Goal

We want to calculate the **“time to arrive to the goal”**,
for fixed policy π and initial state s

This time is a r.v. because there are many possible trajectories,
each with different probability

Time to Arrive to the Goal

We want to calculate the **“time to arrive to the goal”**,
for fixed policy π and initial state s

This time is a r.v. because there are many possible trajectories, each with different probability

For trajectory $\tau = \langle X_0, X_1, \ldots \rangle$, let $T(\tau) = \min\{i : X_i \in G\}$
(i.e. **the time of arrival to the goal**)

If τ doesn't contain a goal state, $T(\tau) = \infty$

Time to Arrive to the Goal

We want to calculate the **"time to arrive to the goal"**, for fixed policy π and initial state s

This time is a r.v. because there are many possible trajectories, each with different probability

For trajectory $\tau = \langle X_0, X_1, \ldots \rangle$, let $T(\tau) = \min\{i : X_i \in G\}$ (i.e. **the time of arrival to the goal**)

If τ doesn't contain a goal state, $T(\tau) = \infty$

The validity of π is **expressed in symbols** as:

- π is valid for s iff $P_s^\pi(T = \infty) = 0$
- π is valid iff it is valid for all states

Optimal Solutions

Definition

Policy π *is* **optimal for** s *if* $J_\pi(s) \leq J_{\pi'}(s)$ *for all policies* π'

Definition

Policy π *is (globally)* **optimal** *if it is optimal for all states*

Optimal Solutions

Definition

Policy π *is* **optimal for** s *if* $J_\pi(s) \leq J_{\pi'}(s)$ *for all policies* π'

Definition

Policy π *is (globally)* **optimal** *if it is optimal for all states*

In probabilistic planning, we are interested in:

- **Solutions for the initial state**
- **Optimal solutions for the initial state**

Computability Issues

The **size** of a controller $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ is in principle **infinite** because the decision functions may vary with time

Computability Issues

The **size** of a controller $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ is in principle **infinite** because the decision functions may vary with time

How do we store a controller?

How do we compute a controller?

Computability Issues

The **size** of a controller $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ is in principle **infinite** because the decision functions may vary with time

How do we store a controller?

How do we compute a controller?

A policy $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ is **stationary** if $\mu = \mu_i$ for all $i \geq 0$; i.e. decision function doesn't depend on time

- Such a policy is simply denoted by μ
- The **size** of μ is just $|S| \log |A|$!!!

Computability Issues

The **size** of a controller $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ is in principle **infinite** because the decision functions may vary with time

How do we store a controller?

How do we compute a controller?

A policy $\pi = \langle \mu_0, \mu_1, \ldots \rangle$ is **stationary** if $\mu = \mu_i$ for all $i \geq 0$; i.e. decision function doesn't depend on time

- Such a policy is simply denoted by μ
- The **size** of μ is just $|S| \log |A|$!!!

What can be captured by stationary policies?

Recursion I: Cost of Stationary Policies

Under stationary μ, the chain is **homogenuous in time** and satisfies the **Markov property**

Recursion I: Cost of Stationary Policies

Under stationary μ, the chain is **homogenuous in time** and satisfies the **Markov property**

Moreover, it is easy to show that J_μ satisfies the recursion:

$$J_\mu(s) = c(s, \mu(s)) + \sum_{s'} p(s'|s, \mu(s)) J_\mu(s')$$

Example: Stationary Policy

Policy: $\pi = \langle \mu_6, \mu_6, \mu_6, \ldots \rangle$

Equations:

$$\begin{aligned}
J_{\mu_6}(s_0) &= 1 + J_{\mu_6}(s_2) \\
J_{\mu_6}(s_1) &= 1 + \tfrac{19}{20} J_{\mu_6}(s_1) + \tfrac{1}{20} J_{\mu_6}(s_2) \\
J_{\mu_6}(s_2) &= 1 + \tfrac{2}{5} J_{\mu_6}(s_0) + \tfrac{1}{2} J_{\mu_6}(s_2)
\end{aligned}$$

Solution:

$$\begin{aligned}
J_{\mu_6}(s_0) &= 15 \\
J_{\mu_6}(s_1) &= 34 \\
J_{\mu_6}(s_2) &= 14
\end{aligned}$$

Proper Policies

Important property of stationary policies (widely used in OR)

Definition

A stationary policy μ is **proper** *if*

$$\rho_\mu = \max_{s \in S} \; P_s^\mu(X_N \notin G) < 1$$

where $N = |S|$ is the number of states

Properness is a **global property** because it depends on all the states

Basic Properties of Stationary Policies

Theorem

μ is valid for s iff $E_s^\mu T < \infty$

Theorem

μ is valid for s iff $J_\mu(s) < \infty$

Theorem

μ is valid iff μ is proper

Fundamental Operators

For stationary policy μ, define the **operator** T_μ, that maps vectors into vectors, as

$$(T_\mu J)(s) = c(s, \mu(s)) + \sum_{s'} p(s'|s, \mu(s)) J(s')$$

I.e., if J is a vector, then TJ is a vector

Fundamental Operators

For stationary policy μ, define the **operator** T_μ, that maps vectors into vectors, as

$$(T_\mu J)(s) = c(s,\mu(s)) + \sum_{s'} p(s'|s,\mu(s))J(s')$$

I.e., if J is a vector, then TJ is a vector

Likewise, define the **operator** T as

$$(TJ)(s) = \min_{a\in A(s)} c(s,a) + \sum_{s'} p(s'|s,a)J(s')$$

Assume all functions (vectors) satisfy $J(s) = 0$ for goals s

Fixed Points

Operators T_μ and T are **monotone** and **continuous**

Therefore, both have a **unique least fixed points** (LFP)

Theorem

The LFP of T_μ *is* J_μ*; i.e.,* $J_\mu = T_\mu J_\mu$

Recursion II: Bellman Equation

Let J^* be the LFP of T; i.e., $J^* = TJ^*$

Bellman Equation

$$J^*(s) = \min_{a \in A(s)} c(s,a) + \sum_{s'} p(s'|s,a) J^*(s')$$

Theorem

$J^* \leq J_\pi$ for all π (stationary or not)

Greedy Policies

The **greedy** (stationary) policy μ for value function J is

$$\mu(s) = \operatorname*{argmin}_{a \in A(s)} c(s,a) + \textstyle\sum_{s'} p(s'|s,a) J(s')$$

Greedy Policies

The **greedy** (stationary) policy μ for value function J is

$$\mu(s) = \operatorname*{argmin}_{a \in A(s)} c(s,a) + \textstyle\sum_{s'} p(s'|s,a) J(s')$$

Observe

$$\begin{aligned}
(T_\mu J)(s) &= c(s,\mu(s)) + \textstyle\sum_{s'} p(s'|s,\mu(s)) J(s') \\
&= \min_a c(s,a) + \textstyle\sum_{s'} p(s'|s,a) J(s) \\
&= (TJ)(s)
\end{aligned}$$

Thus, μ is greedy for J iff $T_\mu J = TJ$

Optimal Greedy Policies

Let μ^* be the greedy policy for J^* ; i.e.,

$$\mu^*(s) = \min_{a \in A(s)} c(s,a) + \textstyle\sum_{s'} p(s'|s,a) J^*(s')$$

Optimal Greedy Policies

Let μ^* be the greedy policy for J^* ; i.e.,

$$\mu^*(s) = \min_{a \in A(s)} c(s,a) + \textstyle\sum_{s'} p(s'|s,a) J^*(s')$$

Theorem (Main)

$J^* = J_{\mu^*}$ *and thus* μ^* *is an* **optimal solution**

Optimal Greedy Policies

Let μ^* be the greedy policy for J^* ; i.e.,

$$\mu^*(s) = \min_{a \in A(s)} c(s,a) + \textstyle\sum_{s'} p(s'|s,a) J^*(s')$$

Theorem (Main)

$J^* = J_{\mu^*}$ and thus μ^* is an **optimal solution**

Most important implications:

- We can **focus** only on stationary policies without compromising optimality
- We can **focus** on computing J^* (the solution of Bellman Equation) because the greedy policy wrt it is optimal

Convergence (Bases for Algorithms)

Theorem

If μ *is a valid policy, then* $T_\mu^k J \to J_\mu$ *for all vectors* J *with* $\|J\| < \infty$

Convergence (Bases for Algorithms)

Theorem

If μ *is a valid policy, then* $T_\mu^k J \to J_\mu$ *for all vectors* J *with* $\|J\| < \infty$

Theorem

If $T_\mu J \leq J$ *for some* J *such that* $\|J\| < \infty$*, then* μ *is a valid policy*

Convergence (Bases for Algorithms)

Theorem

If μ is a valid policy, then $T_\mu^k J \to J_\mu$ for all vectors J with $\|J\| < \infty$

Theorem

If $T_\mu J \leq J$ for some J such that $\|J\| < \infty$, then μ is a valid policy

Theorem (Basis for Value Iteration)

If there is a valid solution, then $T^k J \to J^*$ for all J with $\|J\| < \infty$

Convergence (Bases for Algorithms)

Let μ_0 be a **proper** policy

Define the following stationary policies:

- μ_1 greedy for J_{μ_0}

Convergence (Bases for Algorithms)

Let μ_0 be a **proper** policy

Define the following stationary policies:

- μ_1 greedy for J_{μ_0}
- μ_2 greedy for J_{μ_1}

Convergence (Bases for Algorithms)

Let μ_0 be a **proper** policy

Define the following stationary policies:

- μ_1 greedy for J_{μ_0}
- μ_2 greedy for J_{μ_1}
- $\ldots$
- μ_{k+1} greedy for J_{μ_k}

Convergence (Bases for Algorithms)

Let μ_0 be a **proper** policy

Define the following stationary policies:

- μ_1 greedy for J_{μ_0}
- μ_2 greedy for J_{μ_1}
- $\ldots$
- μ_{k+1} greedy for J_{μ_k}

Theorem (Basis for Policy Iteration)

μ_k *converges to an optimal policy in a finite number of iterates*

Convergence (Bases for Algorithms)

Let μ_0 be a **proper** policy

Define the following stationary policies:

- μ_1 greedy for J_{μ_0}
- μ_2 greedy for J_{μ_1}
- $\ldots$
- μ_{k+1} greedy for J_{μ_k}

Theorem (Basis for Policy Iteration)

μ_k converges to an optimal policy in a finite number of iterates

Theorem

If there is a solution, the fully random policy is proper

Suboptimality of Policies

The **suboptimality** of policy π at **state** s is $|J_\pi(s) - J^*(s)|$

The **suboptimality** of policy π is $\|J_\pi - J^*\| = \max_s |J_\pi(s) - J^*(s)|$

Summary

- Solutions are functions that map states into actions
- Cost of solutions is expected cost over trajectories
- There is a stationary policy μ^* that is optimal
- Global solutions vs. solutions for s_{init}
- Cost function J_μ is LFP of operator T_μ
- J_{μ^*} satisfies the Bellman equation and is LFP of Bellman operator

Part II

Algorithms

Goals

- Basic Algorithms
  - Value Iteration and Asynchronous Value Iteration
  - Policy Iteration
  - Linear Programming

- Heuristic Search Algorithms
  - Real-Time Dynamic Programming
  - LAO*
  - Labeled Real-Time Dynamic Programming
  - Others

Value Iteration (VI)

Computes a sequence of iterates J_k using the Bellman Equation as assignment:

$$J_{k+1}(s) = \min_{a \in A(s)} c(s,a) + \sum_{s'} p(s'|s,a) J_k(s')$$

I.e., $J_{k+1} = TJ_k$. The initial iterate is J_0

The iteration stops when the **residual** $\|J_{k+1} - J_k\| < \epsilon$

- Enough to store two vectors: J_k (current) and J_{k+1} (new)
- **Gauss-Seidel**: store one vector (performs updates **in place**)

Value Iteration (VI)

Theorem

If there is a solution, $\|J_{k+1} - J_k\| \to 0$ *from every initial* J_0 *with* $\|J_0\| < \infty$

Corollary

If there is solution, VI terminates in finite time

Value Iteration (VI)

Theorem

If there is a solution, $\|J_{k+1} - J_k\| \to 0$ *from every initial* J_0 *with* $\|J_0\| < \infty$

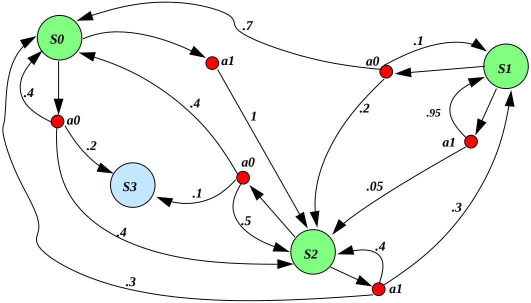
Corollary

If there is solution, VI terminates in finite time

Open Question

Upon termination at iterate $k+1$ *with residual* $< \epsilon$*,*
what is the suboptimality of the greedy policy μ_k *for* J_k*?*

Example: Value Iteration



Example: Value Iteration

$$J_0 = (0.00, 0.00, 0.00)$$

$$J_1 = (1.00, 1.00, 1.00)$$

$$J_2 = (1.80, 2.00, 1.90)$$

$$J_3 = (2.48, 2.84, 2.67)$$

$$\dots$$

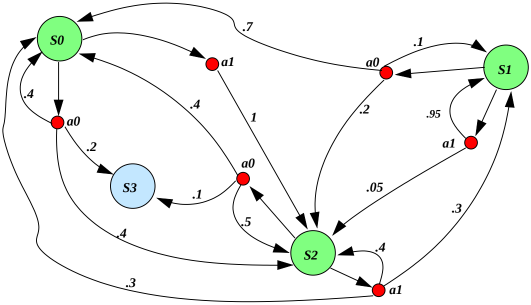
$$J_{10} = (5.12, 6.10, 5.67)$$

$$\dots$$

$$J_{100} = (6.42, 7.69, 7.14)$$

$$\dots$$

$$J^* = (6.42, 7.69, 7.14)$$



Example: Value Iteration

$$J_0 = (0.00, 0.00, 0.00)$$

$$J_1 = (1.00, 1.00, 1.00)$$

$$J_2 = (1.80, 2.00, 1.90)$$

$$J_3 = (2.48, 2.84, 2.67)$$

$$\dots$$

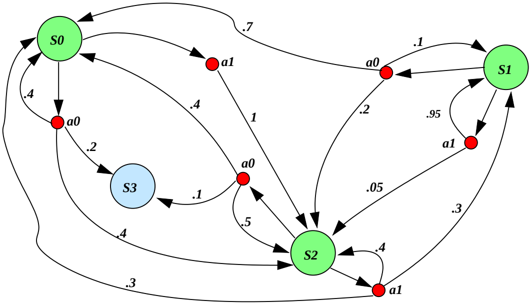
$$J_{10} = (5.12, 6.10, 5.67)$$

$$\dots$$

$$J_{100} = (6.42, 7.69, 7.14)$$

$$\dots$$

$$J^* = (6.42, 7.69, 7.14)$$



$$\mu^*(s_0) = \operatorname{argmin}\{1 + \tfrac{2}{5}J^*(s_0) + \tfrac{2}{5}J^*(s_2), 1 + J^*(s_2)\} = a_0$$

$$\mu^*(s_1) = \operatorname{argmin}\{1 + \tfrac{7}{10}J^*(s_0) + \tfrac{1}{10}J^*(s_1) + \tfrac{1}{5}J^*(s_2), 1 + \tfrac{19}{20}J^*(s_1) + \tfrac{1}{20}J^*(s_2)\} = a_0$$

$$\mu^*(s_2) = \operatorname{argmin}\{1 + \tfrac{2}{5}J^*(s_1) + \tfrac{1}{2}J^*(s_2), 1 + \tfrac{3}{10}J^*(s_0) + \tfrac{3}{10}J^*(s_1) + \tfrac{2}{5}J^*(s_2)\} = a_0$$

Asynchronous Value Iteration

VI is sometimes called **Parallel** VI because it updates all states at each iteration

However, this is not needed!

Asynchronous Value Iteration

VI is sometimes called **Parallel** VI because it updates all states at each iteration

However, this is not needed!

Let S_k be the **set of states updated at iteration** k; i.e.,

$$J_{k+1}(s) = \begin{cases} (TJ_k)(s) & \text{if } s \in S_k \\ J_k(s) & \text{otherwise} \end{cases}$$

Asynchronous Value Iteration

VI is sometimes called **Parallel** VI because it updates all states at each iteration

However, this is not needed!

Let S_k be the **set of states updated at iteration** k; i.e.,

$$J_{k+1}(s) = \begin{cases} (TJ_k)(s) & \text{if } s \in S_k \\ J_k(s) & \text{otherwise} \end{cases}$$

Theorem

If there is solution and every state is updated **infinitely often**, *then* $J_k \to J^*$ *as* $k \to \infty$

Policy Iteration (PI)

Computes a sequence of policies starting from a **proper** policy μ_0:

- μ_1 is greedy for J_{μ_0}
- μ_2 is greedy for J_{μ_1}
- μ_{k+1} is greedy for J_{μ_k}
- **Stop** when $J_{\mu_{k+1}} = J_{\mu_k}$ (or $\mu_{k+1} = \mu_k$)

Policy Iteration (PI)

Computes a sequence of policies starting from a **proper** policy μ_0:

- μ_1 is greedy for J_{μ_0}
- μ_2 is greedy for J_{μ_1}
- μ_{k+1} is greedy for J_{μ_k}
- **Stop** when $J_{\mu_{k+1}} = J_{\mu_k}$ (or $\mu_{k+1} = \mu_k$)

Given vector J_{μ_k}, μ_{k+1} is calculated with equation

$$\mu_{k+1}(s) = \mathrm{argmin}_{a \in A(s)}\, c(s,a) + \textstyle\sum_{s'} p(s'|s,a) J_{\mu_k}(s')$$

Policy Iteration (PI)

Computes a sequence of policies starting from a **proper** policy μ_0:

- μ_1 is greedy for J_{μ_0}
- μ_2 is greedy for J_{μ_1}
- μ_{k+1} is greedy for J_{μ_k}
- **Stop** when $J_{\mu_{k+1}} = J_{\mu_k}$ (or $\mu_{k+1} = \mu_k$)

Given vector J_{μ_k}, μ_{k+1} is calculated with equation

$$\mu_{k+1}(s) = \operatorname{argmin}_{a \in A(s)} c(s,a) + \textstyle\sum_{s'} p(s'|s,a) J_{\mu_k}(s')$$

Given (stationary and proper) policy μ, J_μ is the solution of the **linear system** of equations (one equation per state) given by

$$J(s) = c(s,\mu(s)) + \textstyle\sum_{s'} p(s'|s,\mu(s)) J(s') \qquad s \in S$$

To solve it, one can invert a matrix or use other numerical methods

Policy Iteration (PI)

If μ_0 isn't proper, J_{μ_0} is unbounded for at least one state:

- policy evaluation is not well-defined
- PI may **loop forever**

If μ_0 is proper, then **all policies** μ_k **are proper**

Policy Iteration (PI)

If μ_0 isn't proper, J_{μ_0} is unbounded for at least one state:

- policy evaluation is not well-defined
- PI may **loop forever**

If μ_0 is proper, then **all policies** μ_k **are proper**

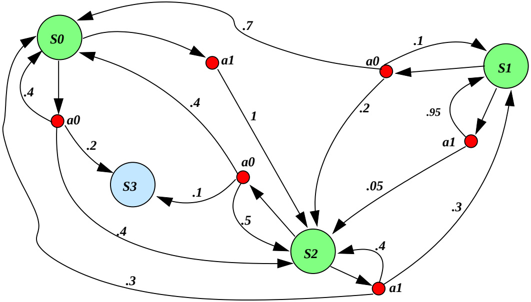
Theorem

Given an initial proper policy, PI terminates in finite time with an optimal policy

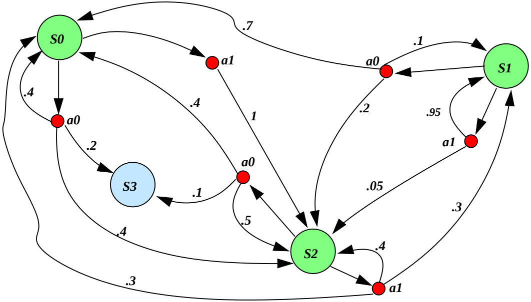
Theorem

Given an initial proper policy, the number of iterations of PI is bounded by the number of stationary policies which is $|A|^{|S|}$

Example: Policy Iteration

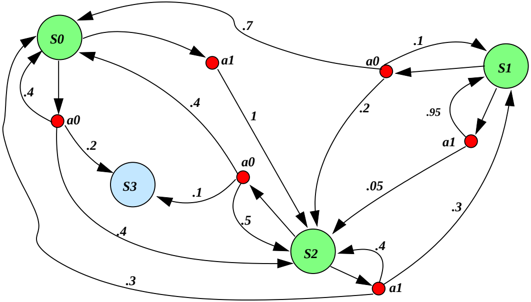


Example: Policy Iteration



$$\mu_0 = (a_1, a_1, a_0)$$
$$J_{\mu_0} = (15.00, 34.00, 14.00)$$
$$\mu_1 = (a_0, a_0, a_0)$$
$$J_{\mu_1} = (6.42, 7.69, 7.14)$$ **(optimal)**

Example: Policy Iteration



$$\mu_0 = (a_1, a_1, a_0)$$

$$J_{\mu_0} = (15.00, 34.00, 14.00)$$

$$\mu_1 = (a_0, a_0, a_0)$$

$J_{\mu_1} = (6.42, 7.69, 7.14)$ **(optimal)**

If $\mu_0 = (a_1, a_1, a_1)$, the policy is **improper** and PI **loops forever!**

Modified Policy Iteration (MPI)

The computation of J_{μ_k} (**policy evaluation**) is the most time-consuming step in PI

Modified Policy Iteration differs from PI in two aspects:

1) Policy evaluation is done **iteratively** by computing a sequence $J_{\mu_k}^0, J_{\mu_k}^1, J_{\mu_k}^2, \ldots$ of value function with

$$J_{\mu_k}^0 = 0$$
$$J_{\mu_k}^{m+1} = T_{\mu_k} J_{\mu_k}^m$$

This is the **inner loop**, stopped when $\|J_{\mu_k}^{m+1} - J_{\mu_k}^m\| < \delta$

Modified Policy Iteration (MPI)

2) The **outer loop**, that computes the policies $\mu_0, \mu_1, \mu_2, \ldots$, is stopped when $\|J_{\mu_{k+1}}^{m_{k+1}} - J_{\mu_k}^{m_k}\| < \epsilon$

That is, MPI performs **approximated** policy evaluation and **limited** policy improvement

For problems with discount (not covered in these lectures), there are suboptimality guarantees as function of ϵ and δ

Linear Programming (LP)

The optimal value function J^* can be computed as the solution of a **linear program** with non-negative variables, one variable x_s per state s, and $|S| \times |A|$ constraints

Linear Program

Maximize $\sum_s x_s$

Subject to

$$c(s,a) + \sum_s p(s'|s,a) x_{s'} \geq x_s \qquad s \in S,\ a \in A(s)$$

$$x_s \geq 0 \qquad s \in S$$

Linear Programming (LP)

Theorem

If there is solution, the LP has bounded solution $\{x_s\}_{s \in S}$ *and* $J^*(s) = x_s$ *for all* $s \in S$

Linear Programming (LP)

Theorem

If there is solution, the LP has bounded solution $\{x_s\}_{s \in S}$ *and* $J^*(s) = x_s$ *for all* $s \in S$

In practice, VI is faster than PI, MPI and LP

Discussion

Complete methods, as the above, compute **entire** solutions (policies) that work for all states

In probabilistic planning, we are only interested in solutions for the initial state

Worse, the problem may have a solution for s_{init} and not have entire solution (e.g., when there are **avoidable dead-end** states). In such cases, the previous methods do not work

Search-based methods are designed to compute **partial solutions** that work for the initial state

Partial Solutions

A partial (stationary) policy is a **partial function** $\mu : S \rightarrow A$

Executing μ from state s, generates trajectories $\tau = \langle s_0, s_1, \ldots \rangle$, but now μ must be defined on all s_i. If not, the trajectory gets 'truncated' at the first state at which μ is undefined

The states **reachable** by μ from s is the set $R_\mu(s)$ of states appearing in the trajectories of μ from s

Partial Solutions

A partial (stationary) policy is a **partial function** $\mu : S \to A$

Executing μ from state s, generates trajectories $\tau = \langle s_0, s_1, \ldots \rangle$, but now μ must be defined on all s_i. If not, the trajectory gets 'truncated' at the first state at which μ is undefined

The states **reachable** by μ from s is the set $R_\mu(s)$ of states appearing in the trajectories of μ from s

We say that:

- μ is **closed on state** s ff μ is defined on all states in $R_\mu(s)$
- μ is **closed** if it is closed on every state on which it is defined

Partial Solutions

A partial (stationary) policy is a **partial function** $\mu : S \rightarrow A$

Executing μ from state s, generates trajectories $\tau = \langle s_0, s_1, \ldots \rangle$, but now μ must be defined on all s_i. If not, the trajectory gets 'truncated' at the first state at which μ is undefined

The states **reachable** by μ from s is the set $R_\mu(s)$ of states appearing in the trajectories of μ from s

We say that:

- μ is **closed on state** s ff μ is defined on all states in $R_\mu(s)$
- μ is **closed** if it is closed on every state on which it is defined

The next algorithms compute partial policies closed on the initial state

Goals

- Basic Algorithms
  - Value Iteration and Asynchronous Value Iteration
  - Policy Iteration
  - Linear Programming

- Heuristic Search Algorithms
  - Real-Time Dynamic Programming
  - LAO*
  - Labeled Real-Time Dynamic Programming
  - Others

Classical Planning: Algorithms

Classical planning is a **path-finding problem** over a huge graph

Many algorithms available, among others:

- Blind search: DFS, BFS, DFID, ...
- Heuristic search: A*, IDA*, WA*, ...
- Greedy: greedy best-first search, Enforced HC, local search, ...
- On-line search: LRTA* and variants

Classical Planning: Best-First Search (DD and RO)

```
$open := \emptyset$          [priority queue w/ nodes $\langle s, g, h\rangle$ ordered by $g + h$]
$closed := \emptyset$        [collection of closed nodes]
$\textsc{push}(\langle s_{init}, 0, h(s_{init})\rangle, open)$
while $open \neq \emptyset$ do
    $\langle s, g, h\rangle := \textsc{pop}(open)$
    if $s \notin closed$ or $g < dist[s]$ then
        $closed := closed \cup \{s\}$
        $dist[s] := g$
        if $s$ is goal then return $(s, g)$
        foreach $a \in A(s)$ do
            $s' := f(s, a)$
            if $h(s') < \infty$ then
                $\textsc{push}(\langle s', d + cost(s, a), h(s')\rangle, open)$
```

(From lectures of B. Nebel, R. Mattmüller and T. Keller)

Classical Planning: Learning Real-Time A* (LRTA*)

```
Let $H$ be empty hash table with entries $H(s)$ initialized to $h(s)$ as needed
repeat
    Set $s := s_{init}$
    while $s$ isn't goal do
        foreach action $a \in A(s)$ do
            Let $s' := f(s, a)$
            Set $Q(s, a) := c(s, a) + H(s')$
        Select best action $\mathbf{a} := \mathrm{argmin}_{a \in A(s)} Q(s, a)$
        Update value $H(s) := Q(s, \mathbf{a})$
        Set $s := f(s, \mathbf{a})$
    end while
until some termination condition
```

Learning Real-Time A* (LRTA*)

- **On-line algorithm** that interleaves planning/execution
- Performs multiple **trials**
- Best action chosen greedily by **one-step lookahead** using values stored in hash table
- Can't get trapped into loops because values are **continuously updated**
- Converges to optimal path under certain conditions
- Uses heuristic function h, the **better** the heuristic the **faster** the convergence
- Can be converted into **offline** algorithm

Real-Time Dynamic Programming (RTDP)

```
Let $H$ be empty hash table with entries $H(s)$ initialized to $h(s)$ as needed
repeat
    Set $s := s_{init}$
    while $s$ isn't goal do
        foreach action $a \in A(s)$ do
            Set $Q(s,a) := c(s,a) + \sum_{s' \in S} p(s'|s,a)H(s')$
        Select best action $\mathbf{a} := \operatorname{argmin}_{a \in A(s)} Q(s,a)$
        Update value $H(s) := Q(s,\mathbf{a})$
        Sample next state $s'$ with probability $p(s'|s,\mathbf{a})$ and set $s := s'$
    end while
until some termination condition
```

Real-Time Dynamic Programming (RTDP)

- **On-line algorithm** that interleaves planning/execution
- Performs multiple **trials**
- Best action chosen greedily by **one-step lookahead** using value function stored in hash table
- Can't get trapped into loops because values are **continuously updated**
- Converges to optimal policy under certain conditions
- Uses heuristic function h, the **better** the heuristic the **faster** the convergence
- Can be converted into **offline** algorithm
- Generalizes Learning Real-Time A*

Properties of Heuristics

Heuristic $h : S \to \mathbb{R}^+$ is **admissible** if $h \leq J^*$

Heuristic $h : S \to \mathbb{R}^+$ is **consistent** if $h \leq Th$

Properties of Heuristics

Heuristic $h : S \to \mathbb{R}^+$ is **admissible** if $h \leq J^*$

Heuristic $h : S \to \mathbb{R}^+$ is **consistent** if $h \leq Th$

Lemma

If h is consistent, h is admissible

Lemma

Let h be consistent (resp. admissible) and $h' = h$ except at s' where

$$h'(s') = (Th)(s')$$

Then, h' is consistent (resp. admissible)

The constant-zero heuristic is **admissible and consistent**

Convergence of RTDP

Theorem

If there is a solution for the reachable states from s_{init}*, then RTDP converges to a (partial) value function.*

The (partial) policy greedy with respect to this value function is a **valid** *solution for the* **initial state***.*

Convergence of RTDP

Theorem

If there is a solution for the reachable states from s_{init}, then RTDP converges to a (partial) value function.

The (partial) policy greedy with respect to this value function is a **valid** *solution for the* **initial state***.*

Theorem

If, in addition, the heuristic is **admissible***, then RTDP converges to a value function whose value on the relevant states coincides with J^*.*

Hence, the partial policy greedy with respect to this value function is an **optimal** *solution for the* **initial state***.*

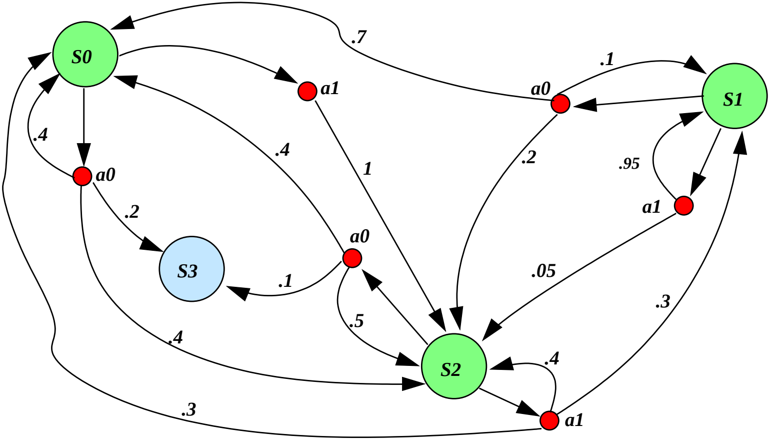
AND/OR Graphs

An AND/OR graph is a **rooted digraph** made of AND nodes and OR nodes:

- an OR node models the **choice** of an action at the state represented by the node
- an AND node models the (multiple) **outcomes** of the action represented by the node

If deterministic actions, the AND/OR graph is a digraph

Example: AND/OR Graph



Solutions for AND/OR Graphs

A solution for an AND/OR graph is a **subgraph** that satisfies:

- the root node, that represents the initial state, belongs to the solution
- for every internal OR node in the solution, exactly one of its children belongs to the solution
- for every AND node in the solution, all of its children belong to the solution

Solutions for AND/OR Graphs

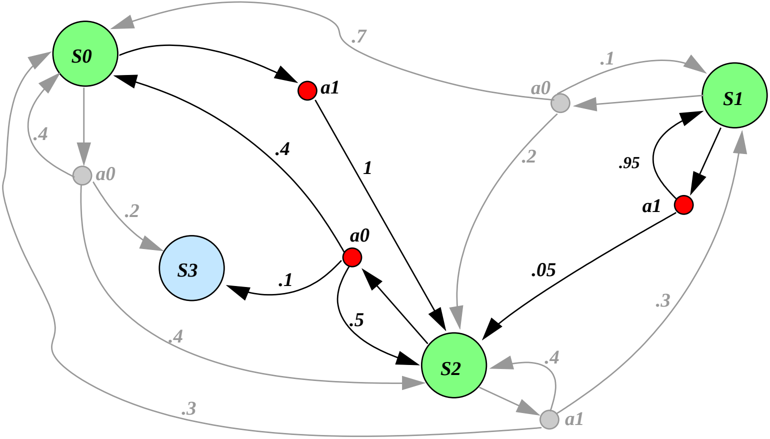
A solution for an AND/OR graph is a **subgraph** that satisfies:

- the root node, that represents the initial state, belongs to the solution
- for every internal OR node in the solution, exactly one of its children belongs to the solution
- for every AND node in the solution, all of its children belong to the solution

The solution is **complete** if every maximal directed path ends in a terminal (goal) node

Otherwise, the solution is **partial**

Example: Solution for AND/OR Graph



Best-First Search for AND/OR Graphs (AO*)

Best First: iteratively, expand nodes on the fringe of best **partial solution** until it becomes complete

Optimal because cost of best partial solution is lower bound of any complete solution (if heuristic is admissible)

Best partial solution determined **greedily** by choosing, for each OR node, the action with **best (expected) value**

Best-First Search for AND/OR Graphs (AO*)

Best First: iteratively, expand nodes on the fringe of best **partial solution** until it becomes complete

Optimal because cost of best partial solution is lower bound of any complete solution (if heuristic is admissible)

Best partial solution determined **greedily** by choosing, for each OR node, the action with **best (expected) value**

AO* solves the DP recursion in **acyclic spaces** by:

- **Expansion:** expands one or more nodes on the fringe of best partial solution
- **Cost Revision:** propagates the new values on the fringe upwards to the root using **backward induction**

LAO*

LAO* generalizes AO* for **AND/OR graphs with cycles**

LAO*

LAO* generalizes AO* for **AND/OR graphs with cycles**

Maintains the expansion step of AO* but changes the cost-revision step from backward induction to **policy evaluation** of the partial solution

LAO*

LAO* generalizes AO* for **AND/OR graphs with cycles**

Maintains the expansion step of AO* but changes the cost-revision step from backward induction to **policy evaluation** of the partial solution

Improved LAO* (ILAO*):

- expands all open nodes on the fringe of current solution
- performs just **one backup** for each node in current solution

LAO*

LAO* generalizes AO* for **AND/OR graphs with cycles**

Maintains the expansion step of AO* but changes the cost-revision step from backward induction to **policy evaluation** of the partial solution

Improved LAO* (ILAO*):

- expands all open nodes on the fringe of current solution
- performs just **one backup** for each node in current solution

As a result, current partial solution is not **guaranteed** to be a best partial solution

Hence, stopping criteria is **strengthened to ensure optimality**

Improved LAO*

```
Explicit graph initially consists of the start state $s_{init}$
repeat
    Depth-first traversal of states in current best (partial) solution graph
    foreach visited state $s$ in postorder traversal do
        if state $s$ isn't expanded then
            Expand $s$ by generating each successor $s'$ and initializing $H(s')$ to $h(s')$
        end if
        Set $H(s) := \min_{a \in A(s)} c(s,a) + \sum_{s' \in S} p(s'|s,a) H(s')$ and mark best action
    end foreach
until best solution graph has no unexpanded tips and residual $< \epsilon$
```

Improved LAO*

The expansion and cost-revision steps of ILAO* performed in the **same depth-first traversal** of the partial solution graph

Stopping criteria extended with a **test on residual**

Improved LAO*

The expansion and cost-revision steps of ILAO* performed in the **same depth-first traversal** of the partial solution graph

Stopping criteria extended with a **test on residual**

Theorem

If there is solution for s_{init} *and* h *is consistent, LAO* and ILAO* terminate with solution for* s_{init} *and residual* $< \epsilon$

Faster Convergence

ILAO* converges much faster than RTDP because

- performs **systematic exploration** of the state space rather than **stochastic exploration**
- has an **explicit convergence test**

Both ideas can be incorporated into RTDP

Labeling States

RTDP keeps visiting reachable states even when the value function has **converged** over them (aka solved states)

Updates on solved states are **wasteful** because the value function doesn't change

Hence, it makes sense to **detect solved states** and not perform updates on them

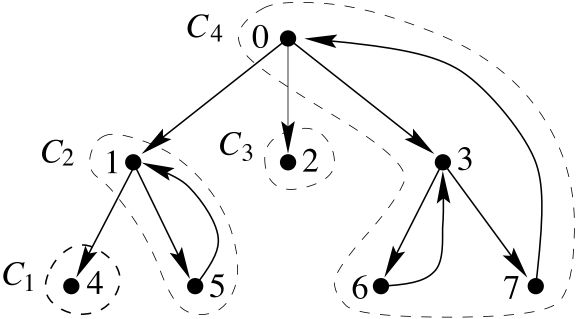
Solved States

A state s is **solved** for J when s and all states reachable from s using the greedy policy for J have residual $< \epsilon$

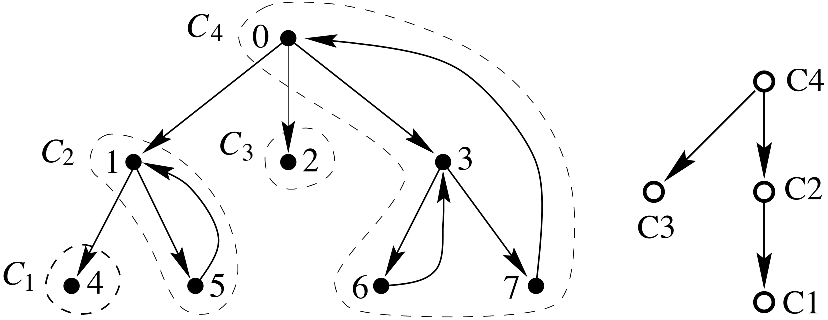
If the **solution graph contains cycles**, labeling states as 'solved' **cannot** be done by backward induction

However, the solution graph can be decomposed into strongly-connected components (SCCs) that make up an **acyclic graph** that can be labeled

Example: Strongly-Connected Components (SCCs)



Example: Strongly-Connected Components (SCCs)



Detecting Solved States

A depth-first traversal from s that chooses actions greedily with respect to J can be used to test if s is solved:

- **backtrack** at solved states returning **true**
- **backtrack** at states with residual $\geq \epsilon$ returning **false**

Detecting Solved States

A depth-first traversal from s that chooses actions greedily with respect to J can be used to test if s is solved:

- **backtrack** at solved states returning **true**
- **backtrack** at states with residual $\geq \epsilon$ returning **false**

If updates are performed at states with residual $\geq \epsilon$ and their ancestors, the traversal either

- **detects** a solved state, or
- performs at least **one update** that changes the value of some state in more than ϵ

Detecting Solved States

A depth-first traversal from s that chooses actions greedily with respect to J can be used to test if s is solved:

- **backtrack** at solved states returning **true**
- **backtrack** at states with residual $\geq \epsilon$ returning **false**

If updates are performed at states with residual $\geq \epsilon$ and their ancestors, the traversal either

- **detects** a solved state, or
- performs at least **one update** that changes the value of some state in more than ϵ

This algorithm is called CheckSolved

CheckSolved

```
Let $rv := true$; $open := \emptyset$; $closed := \emptyset$
if not labeled $s$ then $\textsc{push}(s, open)$

while $open \neq \emptyset$ do
    $s := \textsc{pop}(open)$; $\textsc{push}(s, closed)$
    if $\textsc{residual}(s) > \epsilon$ then $rv := false$; continue

    $a := \textsc{best-action}(s)$
    foreach $s'$ with $P(s'|s, a) > 0$ do
        if not labeled $s'$ and $s' \notin open \cup closed$ then
            $\textsc{push}(s, open)$
endwhile

if $rv = true$ then
    foreach $s' \in closed$ do label $s$
else
    while $closed \neq \emptyset$ do
        $s := \textsc{pop}(closed)$
        $\textsc{dp-update}(s)$
return $rv$
```

Labeled RTDP (LRTDP)

RTDP in which the goal states are initially marked as solved and the trials are modified to:

- **terminate** at solved states rather than goal states
- at termination, call **CheckSolved** on all states in the trial (in reverse order) until it returns **false**
- **terminate trials** when the initial state is labeled as solved

Labeled RTDP (LRTDP)

LRTDP achieves the following:

- **crisp** termination condition
- final function has residual $< \epsilon$ on states reachable from s_{init}
- doesn't perform updates over converged states
- the search is still stochastic but it is **“more systematic”**

Labeled RTDP (LRTDP)

LRTDP achieves the following:

- **crisp** termination condition
- final function has residual $< \epsilon$ on states reachable from s_{init}
- doesn't perform updates over converged states
- the search is still stochastic but it is **"more systematic"**

Theorem

If there is solution for all reachable states from s_{init}, and h is **consistent**, *LRTDP terminates with an optimal solution for s_{init} in a number of trials bounded by $\epsilon^{-1} \sum_s J^*(s) - h(s)$*

Using Non-Admissible Heuristics

LAO* and LRTDP can be used with **non-admissible** heuristics, yet one looses the guarantees on optimality

Theorem

If there is a solution for s_{ini} and h is non-admissible, then LAO (and improved LAO*) terminates with a solution for the initial state*

Theorem

If there is a solution for the reachable states from s_{init} and h is non-admissible, then RTDP terminates with a solution for the initial state

Heuristic Dynamic Programming (HDP)

Tarjan's algorithm for computing SCCs is a depth-first traversal that computes the SCCs and their **acyclic structure**

It can be modified to:

- **backtrack** on solved states
- **expand** (and update the value) of **non-goal tip nodes**
- **update** the value of states with residual $\geq \epsilon$
- **update** the value of **ancestors** of updated nodes
- when detecting an SCC of nodes with residual $< \epsilon$, **label all nodes** in the SCC as **solved**

(Modified) Tarjan's algorithm can be used to find optimal solutions:

```
while $s_{init}$ isn't solved do TarjanSCC($s_{init}$)
```

General Template: Find-and-Revise

```
Start with a consistent function $J := h$

repeat
    Find a state $s$ in the greedy graph for $J$ with $\textsc{Residual}(s) > \epsilon$
    Revise $J$ at $s$
until no such state $s$ is found

return $J$
```

General Template: Find-and-Revise

```
Start with a consistent function $J := h$

repeat
    Find a state $s$ in the greedy graph for $J$ with $\textsc{Residual}(s) > \epsilon$
    Revise $J$ at $s$
until no such state $s$ is found

return $J$
```

- J remains **consistent** (lower bound) after revisions (updates)
- number of iterations until convergence bounded as in RTDP; i.e., by $\epsilon^{-1} \sum_s J^*(s) - h(s)$

Other Algorithms

Bounds: admissible heuristics are LBs. With UBs, one can:

- use difference of bounds to bound suboptimality
- use difference of bounds to focus the search

Algorithms that use both bounds are BRTDP, FRTDP, . . .

AND/OR Graphs: used to model a variety of problems.
LDFS is a unified algorithm for AND/OR graphs that is based of depth-first search and DP updates

Symbolic Search: many variants of above algorithms as well as others that implement search in symbolic representations and **factored MDPs**

Summary

- Explicit algorithms such as VI and PI work well for small problems
- Explicit algorithms compute (entire) solutions
- LAO* and LRTDP compute solutions for the initial state:
  - if heuristic is admissible, both compute optimal solutions
  - if heuristic is non-admissible, both compute solutions
  - number of updates depends on quality of heuristic
- There are other search algorithms

Part III

Heuristics (few thoughts)

Recap: Properties of Heuristics

Heuristic $h : S \to \mathbb{R}^+$ is **admissible** if $h \leq J^*$

Heuristic $h : S \to \mathbb{R}^+$ is **consistent** if $h \leq Th$

Lemma

If h is consistent, h is admissible

Search-based algorithms compute:

- Optimal solution for initial state if heuristic is admissible
- Solution for initial state for any heuristic

How to Obtain Admissible Heuristics?

Relax problem → **Solve optimally** → **Admissible heuristic**

How to Obtain Admissible Heuristics?

Relax problem → **Solve optimally** → **Admissible heuristic**

How to relax?

- Remove non-determinism
- State abstraction (?)

How to Obtain Admissible Heuristics?

Relax problem → **Solve optimally** → **Admissible heuristic**

How to relax?

- Remove non-determinism
- State abstraction (?)

How to solve relaxation?

- Use available solver
- Use search with admissible heuristic
- Substitute with admissible heuristic for relaxation

Determinization: Min-Min Heuristic

Determinization **obtained** by transforming Bellman equation

$$J^*(s) = \min_{a \in A(s)} c(s,a) + \sum_{s' \in s} p(s'|s,a) J^*(s')$$

into

$$J^*_{min}(s) = \min_{a \in A(s)} c(s,a) + \min\{J^*_{min}(s') : p(s'|s,a) > 0\}$$

Determinization: Min-Min Heuristic

Determinization **obtained** by transforming Bellman equation

$$J^*(s) = \min_{a \in A(s)} c(s,a) + \sum_{s' \in s} p(s'|s,a) J^*(s')$$

into

$$J^*_{min}(s) = \min_{a \in A(s)} c(s,a) + \min\{J^*_{min}(s') : p(s'|s,a) > 0\}$$

Obs: This is Bellman equation for **deterministic** problem

Theorem

$J^*_{min}(s)$ is consistent and thus $J^*_{min}(s) \leq J^*(s)$

Determinization: Min-Min Heuristic

Determinization **obtained** by transforming Bellman equation

$$J^*(s) = \min_{a \in A(s)} c(s,a) + \sum_{s' \in s} p(s'|s,a) J^*(s')$$

into

$$J^*_{min}(s) = \min_{a \in A(s)} c(s,a) + \min\{J^*_{min}(s') : p(s'|s,a) > 0\}$$

Obs: This is Bellman equation for **deterministic** problem

Theorem

$J^*_{min}(s)$ *is consistent and thus* $J^*_{min}(s) \leq J^*(s)$

Solve with search algorithm, or use **admissible** estimate for J^*_{min}

Abstractions

Abstraction of problem P with space S is problem P' with space S' together with **abstraction function** $\alpha : S \to S'$

Interested in “small” abstractions; i.e., $|S'| \ll |S|$

Abstractions

Abstraction of problem P with space S is problem P' with space S' together with **abstraction function** $\alpha : S \to S'$

Interested in “small” abstractions; i.e., $|S'| \ll |S|$

Abstraction is **admissible** if $J^*_{P'}(\alpha(s)) \leq J^*_P(s)$

Abstraction is **bounded** if $J^*_{P'}(\alpha(s)) = \infty \implies J^*_P(s) = \infty$

Abstractions

Abstraction of problem P with space S is problem P' with space S' together with **abstraction function** $\alpha : S \to S'$

Interested in "small" abstractions; i.e., $|S'| \ll |S|$

Abstraction is **admissible** if $J^*_{P'}(\alpha(s)) \leq J^*_P(s)$

Abstraction is **bounded** if $J^*_{P'}(\alpha(s)) = \infty \implies J^*_P(s) = \infty$

how to compute admissible abstractions?

how to compute bounded abstractions?

Summary

- Not much known about heuristics for probabilistic planning
- There are (search) algorithms but cannot be **exploited**
- Heuristics to be **effective** must be computed at representation level, like done in classical planning
- Heuristics for classical planning can be **lifted** for probabilistic planning through **determinization**

Summary

- Not much known about heuristics for probabilistic planning
- There are (search) algorithms but cannot be **exploited**
- Heuristics to be **effective** must be computed at representation level, like done in classical planning
- Heuristics for classical planning can be **lifted** for probabilistic planning through **determinization**

Lots of things to be done about heuristics!

Part IV

Monte-Carlo Planning

Goals

- Monte-Carlo Planning
- Uniform Monte-Carlo
- Adaptive Monte-Carlo

(based on ICAPS'10 tutorial on Monte-Carlo Planning by A. Fern)

Motivation

- Often, not interested in computing an **explicit** policy; it is enough to have a method for **action selection**
- May have no good heuristic to **prune** irrelevant parts of the space
- State space can be prohibitively large, even store a policy or value function over the **relevant states**
- May have no **explicit model**, but just **simulator**
- May have (somewhat) good **base policy** for the problem instead of a heuristic

Anyone of these may render complete algorithms useless!

Simulators and Action Selection Mechanisms

Definition (Simulator)

A simulator is a computer program that given a state and action, generates a successor state and reward **distributed** *according to the problem dynamics and rewards (known or unknown)*

Definition (Action Selection Mechanism)

An action-selection mechanism is a computer program that given a state, returns an action that is applicable at the state; i.e., it is a **policy** *represented* **implicitly**

Monte-Carlo Planning

Given state and **time window** for making a decision, **interact** with a simulator (for given time) and then choose an action

Monte-Carlo planning is often described in problems with **rewards** instead of **costs**; both views are valid and **interchangeable**

Monte-Carlo planning is described in problems with **discount**, but it is also used in **undiscounted** problems

Single-State Monte-Carlo Planning

Problem:

- single state s and k actions $a_1, \ldots, a_k$
- rewards $r(s, a_i) \in [0, 1]$ are **unknown** and **stochastic**
- simulator **samples rewards** according to their hidden distributions

Objective:

- **maximize profit** in a given time window
- must **explore** and **exploit**!

Single-State Monte-Carlo Planning

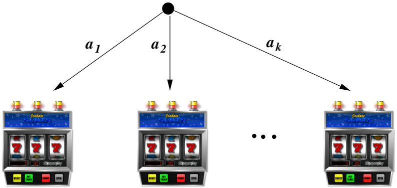
Problem:

- single state s and k actions $a_1, \ldots, a_k$
- rewards $r(s, a_i) \in [0, 1]$ are **unknown** and **stochastic**
- simulator **samples rewards** according to their hidden distributions

Objective:

- **maximize profit** in a given time window
- must **explore** and **exploit**!

This problem is called the **Multi-Armed Bandit Problem** (MABP)



Uniform Bandit Algorithm

- Pull arms **uniformly** (each, the same number w of times)
- Then, for each bandit i, get sampled rewards $\hat{r}_{i1}, \hat{r}_{i2}, \ldots, \hat{r}_{iw}$
- Select arm a_i with best **average reward** $\frac{1}{w}\sum_{j=1}^{w} \hat{r}_{ij}$

Uniform Bandit Algorithm

- Pull arms **uniformly** (each, the same number w of times)
- Then, for each bandit i, get sampled rewards $\hat{r}_{i1}, \hat{r}_{i2}, \ldots, \hat{r}_{iw}$
- Select arm a_i with best **average reward** $\frac{1}{w}\sum_{j=1}^{w} \hat{r}_{ij}$

Theorem (PAC Result)

If $w \geq \left(\frac{R_{\max}}{\epsilon}\right)^2 \ln \frac{k}{\delta}$ *for all arms simultaneously, then*

$$\left| E[R(s, a_i)] - \frac{1}{w}\sum_{j=1}^{w} \hat{r}_{ij} \right| \leq \epsilon$$

with probability at least $1 - \delta$

- ϵ-accuracy with probability at least $1 - \delta$
- # calls to simulator $= O(\frac{k}{\epsilon^2} \ln \frac{k}{\delta})$

Finite-Horizon MDPs

The process goes for h stages (decisions) **only**

The value functions are $J_\mu(s,i)$ for policy μ and $J^*(s,i)$ for optimal value function, $0 \le i \le h$:

$$J_\mu(s,0) = 0 \quad \text{(process is terminated)}$$

$$J_\mu(s,i) = r(s,\mu(s,i)) + \sum_{s'} p(s'|s,\mu(s)) J_\mu(s',i-1)$$

Finite-Horizon MDPs

The process goes for h stages (decisions) **only**

The value functions are $J_\mu(s,i)$ for policy μ and $J^*(s,i)$ for optimal value function, $0 \leq i \leq h$:

$$J_\mu(s,0) = 0 \quad \text{(process is terminated)}$$

$$J_\mu(s,i) = r(s,\mu(s,i)) + \sum_{s'} p(s'|s,\mu(s)) J_\mu(s',i-1)$$

$$J^*(s,0) = 0 \quad \text{(process is terminated)}$$

$$J^*(s,i) = \max_{a \in A(s)} r(s,a) + \sum_{s'} p(s'|s,a) J^*(s',i-1)$$

Finite-Horizon MDPs

The process goes for h stages (decisions) **only**

The value functions are $J_\mu(s,i)$ for policy μ and $J^*(s,i)$ for optimal value function, $0 \leq i \leq h$:

$$J_\mu(s,0) = 0 \quad \text{(process is terminated)}$$

$$J_\mu(s,i) = r(s,\mu(s,i)) + \sum_{s'} p(s'|s,\mu(s)) J_\mu(s',i-1)$$

$$J^*(s,0) = 0 \quad \text{(process is terminated)}$$

$$J^*(s,i) = \max_{a \in A(s)} r(s,a) + \sum_{s'} p(s'|s,a) J^*(s',i-1)$$

Greedy policy μ for vector J, $1 \leq i \leq h$:

$$\mu(s,i) = \operatorname*{argmax}_{a \in A(s)} r(s,a) + \sum_{s'} p(s'|s,a) J(s',i-1)$$

Estimating Quality of Base Policies by Sampling

For (implicit) base policy μ, we can estimate its quality by **sampling**

Estimating Quality of Base Policies by Sampling

For (implicit) base policy μ, we can estimate its quality by **sampling**

A **simulated rollout** of μ starting at s is obtained by:

```
let $j = 0$ and $s_0 = s$
while $j < h$ do
    select action $a_j$ at $s_j$ using $\mu$; i.e., $a_j = \mu(s_j, h - j)$
    use simulator to sample reward $\hat{r}_j$ and state $s'$
    set $s_{j+1} := s'$ and increase $j$
end while
```

Estimating Quality of Base Policies by Sampling

For (implicit) base policy μ, we can estimate its quality by **sampling**

A **simulated rollout** of μ starting at s is obtained by:

```
let $j = 0$ and $s_0 = s$
while $j < h$ do
    select action $a_j$ at $s_j$ using $\mu$; i.e., $a_j = \mu(s_j, h - j)$
    use simulator to sample reward $\hat{r}_j$ and state $s'$
    set $s_{j+1} := s'$ and increase $j$
end while
```

- $J_\mu(s, h)$ can be **estimated** as $\sum_{j=0}^{h-1} \hat{r}_j$

Estimating Quality of Base Policies by Sampling

For (implicit) base policy μ, we can estimate its quality by **sampling**

A **simulated rollout** of μ starting at s is obtained by:

```
let $j = 0$ and $s_0 = s$
while $j < h$ do
    select action $a_j$ at $s_j$ using $\mu$; i.e., $a_j = \mu(s_j, h - j)$
    use simulator to sample reward $\hat{r}_j$ and state $s'$
    set $s_{j+1} := s'$ and increase $j$
end while
```

- $J_\mu(s, h)$ can be **estimated** as $\sum_{j=0}^{h-1} \hat{r}_j$
- Can repeat w times to get **better estimate:** $\frac{1}{w} \sum_{i=1}^{w} \sum_{j=0}^{h-1} \hat{r}_{ij}$

Estimating Quality of Base Policies by Sampling

For (implicit) base policy μ, we can estimate its quality by **sampling**

A **simulated rollout** of μ starting at s is obtained by:

```
let $j = 0$ and $s_0 = s$
while $j < h$ do
    select action $a_j$ at $s_j$ using $\mu$; i.e., $a_j = \mu(s_j, h - j)$
    use simulator to sample reward $\hat{r}_j$ and state $s'$
    set $s_{j+1} := s'$ and increase $j$
end while
```

- $J_\mu(s,h)$ can be **estimated** as $\sum_{j=0}^{h-1} \hat{r}_j$
- Can repeat w times to get **better estimate:** $\frac{1}{w} \sum_{i=1}^{w} \sum_{j=0}^{h-1} \hat{r}_{ij}$
- Accuracy bounds (PAC) can be obtained as function of $\epsilon, \delta, |A|, w$

Action Selection as a Multi-Armed Bandit Problem

The problem of selecting **best action** at state s and then **following** base policy μ for h steps (in general MDPs) is similar to MABP:

- each action leads to a state from which the policy μ is executed

Action Selection as a Multi-Armed Bandit Problem

The problem of selecting **best action** at state s and then **following** base policy μ for h steps (in general MDPs) is similar to MABP:

- each action leads to a state from which the policy μ is executed
- the expected reward of taking action a at state s is

$$Q_\mu(s, a, h) = r(s, a) + \textstyle\sum_{s'} p(s'|s, a) J_\mu(s', h-1)$$

Action Selection as a Multi-Armed Bandit Problem

The problem of selecting **best action** at state s and then **following** base policy μ for h steps (in general MDPs) is similar to MABP:

- each action leads to a state from which the policy μ is executed
- the expected reward of taking action a at state s is

$$Q_\mu(s,a,h) = r(s,a) + \textstyle\sum_{s'} p(s'|s,a) J_\mu(s', h-1)$$

- it can be **estimated** with function $SimQ(s,a,\mu,h)$

```
$SimQ(s,a,\mu,h)$
    sample $(\hat{r}, s')$ that result of executing $a$ at $s$
    set $\hat{q} := \hat{r}$
    for $i = 1$ to $h-1$ do
        sample $(\hat{r}, s'')$ that result of executing $\mu(s', h-i)$ at $s'$
        set $\hat{q} := \hat{q} + r$ and $s' := s''$
    end for
    return $\hat{q}$
```

Action Selection as a Multi-Armed Bandit Problem

For state s, base policy μ, and depth h, do:

- run $SimQ(s, a, \mu, h)$ w times to get estimations $\hat{q}_{a1}, \ldots, \hat{q}_{aw}$
- estimate Q_μ-value for action a as $\hat{Q}_\mu(s, a, h) = \frac{1}{w} \sum_{i=1}^{w} \hat{q}_{ai}$
- select action a that maximizes $\hat{Q}_\mu(s, a, h)$

Action Selection as a Multi-Armed Bandit Problem

For state s, base policy μ, and depth h, do:

- run $SimQ(s,a,\mu,h)$ w times to get estimations $\hat{q}_{a1},\ldots,\hat{q}_{aw}$
- estimate Q_μ-value for action a as $\hat{Q}_\mu(s,a,h) = \frac{1}{w}\sum_{i=1}^{w}\hat{q}_{ai}$
- select action a that maximizes $\hat{Q}_\mu(s,a,h)$

This is the **Policy Rollout** algorithm applied to base policy μ

calls to simulator per decision $= |A|wh$

Multi-Stage Rollouts

The Policy Rollout of μ is a policy; let's refer to it by $Rollout_\mu$

Multi-Stage Rollouts

The Policy Rollout of μ is a policy; let's refer to it by $Rollout_\mu$

We can apply Policy Rollout to base policy $Rollout_\mu$

The result is a policy called **2-Stage Rollout** of μ; $Rollout_\mu^2$

Multi-Stage Rollouts

The Policy Rollout of μ is a policy; let's refer to it by $Rollout_{\mu}$

We can apply Policy Rollout to base policy $Rollout_{\mu}$

The result is a policy called **2-Stage Rollout** of μ; $Rollout_{\mu}^{2}$

In general, we can apply Policy Rollout to base policy $Rollout_{\mu}^{k-1}$ to obtain the k-**Stage Rollout** of μ; $Rollout_{\mu}^{k}$

Multi-Stage Rollouts

The Policy Rollout of μ is a policy; let's refer to it by $Rollout_{\mu}$

We can apply Policy Rollout to base policy $Rollout_{\mu}$

The result is a policy called **2-Stage Rollout** of μ; $Rollout_{\mu}^{2}$

In general, we can apply Policy Rollout to base policy $Rollout_{\mu}^{k-1}$ to obtain the k-**Stage Rollout** of μ; $Rollout_{\mu}^{k}$

None of these policies consume space, but the time to compute them is exponential in k:

- $Rollout_{\mu}$ requires $|A|wh$ simulator calls
- $Rollout_{\mu}^{2}$ requires $(|A|wh)^{2}$ simulator calls
- $Rollout_{\mu}^{k}$ requires $(|A|wh)^{k}$ simulator calls

Rollouts and Policy Iteration

As the horizon is finite, Policy Iteration **always** converges

For base policy μ, PI computes sequence $\langle \mu_0 = \mu, \mu_1, \ldots \rangle$ of policies

Rollouts and Policy Iteration

As the horizon is finite, Policy Iteration **always** converges

For base policy μ, PI computes sequence $\langle \mu_0 = \mu, \mu_1, \ldots \rangle$ of policies

For large w, $\hat{Q}_\mu(s,a,h) \simeq r(s,a) + \sum_{s'} p(s'|s,a) J_\mu(s', h-1)$

Rollouts and Policy Iteration

As the horizon is finite, Policy Iteration **always** converges

For base policy μ, PI computes sequence $\langle \mu_0 = \mu, \mu_1, \ldots \rangle$ of policies

For large w, $\hat{Q}_\mu(s,a,h) \simeq r(s,a) + \sum_{s'} p(s'|s,a) J_\mu(s', h-1)$

- $Rollout_\mu = \mu_1$ (1st iterate of PI) for sufficiently large w

Rollouts and Policy Iteration

As the horizon is finite, Policy Iteration **always** converges

For base policy μ, PI computes sequence $\langle \mu_0 = \mu, \mu_1, \ldots \rangle$ of policies

For large w, $\hat{Q}_\mu(s,a,h) \simeq r(s,a) + \sum_{s'} p(s'|s,a) J_\mu(s', h-1)$

- $Rollout_\mu = \mu_1$ (1st iterate of PI) for sufficiently large w
- $Rollout^2_\mu = \mu_2$ (2nd iterate of PI) for sufficiently large w

Rollouts and Policy Iteration

As the horizon is finite, Policy Iteration **always** converges

For base policy μ, PI computes sequence $\langle \mu_0 = \mu, \mu_1, \ldots \rangle$ of policies

For large w, $\hat{Q}_\mu(s,a,h) \simeq r(s,a) + \sum_{s'} p(s'|s,a) J_\mu(s', h-1)$

- $Rollout_\mu = \mu_1$ (1st iterate of PI) for sufficiently large w
- $Rollout_\mu^2 = \mu_2$ (2nd iterate of PI) for sufficiently large w
- $Rollout_\mu^k = \mu_k$; i.e., **multi-stage rollout implements PI!**

Rollouts and Policy Iteration

As the horizon is finite, Policy Iteration **always** converges

For base policy μ, PI computes sequence $\langle \mu_0 = \mu, \mu_1, \ldots \rangle$ of policies

For large w, $\hat{Q}_\mu(s,a,h) \simeq r(s,a) + \sum_{s'} p(s'|s,a) J_\mu(s', h-1)$

- $Rollout_\mu = \mu_1$ (1st iterate of PI) for sufficiently large w
- $Rollout_\mu^2 = \mu_2$ (2nd iterate of PI) for sufficiently large w
- $Rollout_\mu^k = \mu_k$; i.e., **multi-stage rollout implements PI!**

Theorem

For sufficiently large w *and* k, $Rollout_\mu^k$ *is optimal*

Recursive Sampling (aka Sparse Sampling)

With sampling, we can estimate $J_\mu(s,h)$ for base policy μ

Can we use sampling to estimate $J^*(s,h)$ directly?

Recursive Sampling (aka Sparse Sampling)

With sampling, we can estimate $J_\mu(s,h)$ for base policy μ

Can we use sampling to estimate $J^*(s,h)$ directly?

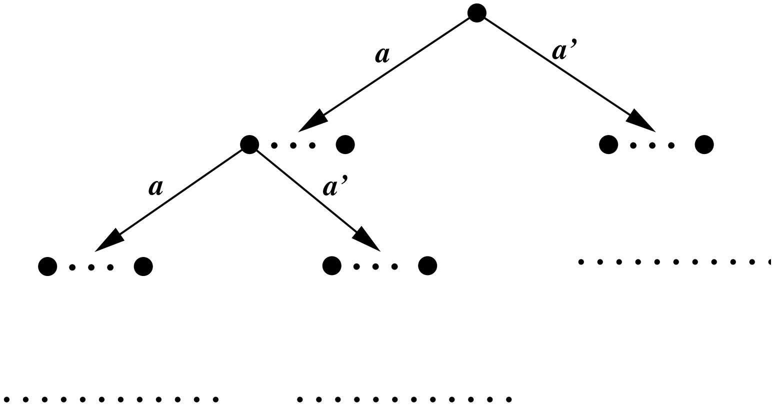
Idea: use **recursion** based on Bellman Equation

$$Q^*(s,a,0) = 0$$

$$Q^*(s,a,h) = r(s,a) + \sum_{s'} p(s'|s,a) J^*(s,h-1)$$

$$J^*(s,h) = \max_{a \in A(s)} Q^*(s,a,h)$$

Recursive Sampling



Recursive Sampling: Pseudocode

```
$SimQ^*(s, a, h, w)$
    set $\hat{q} := 0$
    for $i = 1$ to $w$ do
        sample $(\hat{r}, s')$ that result of executing $a$ at $s$
        set $best := -\infty$
        foreach $a' \in A(s')$ do
            set $new := SimQ^*(s', a', h-1, w)$
            set $best := \max\{best, new\}$
        end foreach
        set $\hat{q} := \hat{q} + \hat{r} + best$
    end for
    return $\frac{\hat{q}}{w}$
```

Recursive Sampling: Properties

- For large w, $SimQ^*(s,a,h,w) \simeq Q^*(s,a,h)$
- Hence, for large w, can be used to choose **optimal actions**
- Estimation doesn't depend on **number of states!!**
- There are bounds on accurracy but for **impractical** values for w
- The actions (space) is sampled **uniformly**; i.e., doesn't **bias exploration** towards most promising areas of the space

This algorithm is called Sparse Sampling

Adaptive Sampling

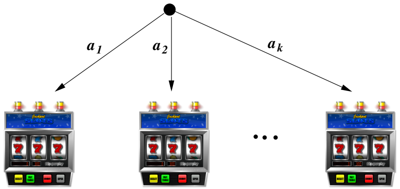
Recursive Sampling is uniform but it should be **adaptive** focusing the effort in most promising parts of the space

An adaptive algorithm **balances** exploration in terms of the sampled rewards. There are **competing needs:**

- actions w/ higher sampled reward should be preferred (exploitation)
- actions that had been explored less should be preferred (exploration)

Important theoretical results for Multi-Armed Bandit Problem

Adaptive Sampling for Multi-Armed Bandits (UCB)



Keep track of number $n(i)$ of times arm i had been 'pulled' and the average sampled reward $\hat{r}_i$ for arm i:

The **UCB rule** says:

$$\text{Pull arm that maximizes } \hat{r}_i + \sqrt{\frac{2\ln n}{n(i)}}$$

where n is the total number of pulls

Upper Confidence Bound (UCB)

Pull arm that maximizes $\underbrace{\hat{r}_i}_{\text{avg. samp. reward}} + \underbrace{\sqrt{2 \ln n / n(i)}}_{\text{exploration bonus}}$

- At the beginning, the exploration bonus ‘dominates’ and arms are pulled, gathering information about them
- The accuracy of the estimate $\hat{r}_i$ increases as the number of pulls to arm i increases
- As the number of pulls increase, the exporation bonus decreases and the ‘quality’ term dominates

Upper Confidence Bound (UCB)

Pull arm that maximizes $\underbrace{\hat{r}_i}_{\text{avg. samp. reward}} + \underbrace{\sqrt{2\ln n/n(i)}}_{\text{exploration bonus}}$

- At the beginning, the exploration bonus ‘dominates’ and arms are pulled, gathering information about them
- The accuracy of the estimate $\hat{r}_i$ increases as the number of pulls to arm i increases
- As the number of pulls increase, the exporation bonus decreases and the ‘quality’ term dominates

Theorem

The **expected regret** *after n pulls, compared to optimal behavior, is bounded by $O(\log n)$.* **No algorithm achieves better regret**

What is the UCB Formula?

$$UCB(i) = \hat{r}_i + \sqrt{2 \ln n / n(i)}$$

$UCB(i)$ is an **upper bound** on a confidence interval for the **true expected reward** r_i for arm i; that is, **w.h.p.** $r_i < UCB(i)$

What is the UCB Formula?

$$UCB(i) = \hat{r}_i + \sqrt{2 \ln n / n(i)}$$

$UCB(i)$ is an **upper bound** on a confidence interval for the **true expected reward** r_i for arm i; that is, **w.h.p.** $r_i < UCB(i)$

After 'enough' pulls, $\hat{r}_i + \sqrt{2 \ln n / n(i)} < r^*$ and arm i is not pulled anymore

What is the UCB Formula?

$$UCB(i) = \hat{r}_i + \sqrt{2 \ln n / n(i)}$$

$UCB(i)$ is an **upper bound** on a confidence interval for the **true expected reward** r_i for arm i; that is, **w.h.p.** $r_i < UCB(i)$

After ‘enough’ pulls, $\hat{r}_i + \sqrt{2 \ln n / n(i)} < r^*$ and arm i is not pulled anymore

How many is enough?

What is the UCB Formula?

$$UCB(i) = \hat{r}_i + \sqrt{2 \ln n / n(i)}$$

$UCB(i)$ is an **upper bound** on a confidence interval for the **true expected reward** r_i for arm i; that is, **w.h.p.** $r_i < UCB(i)$

After ‘enough’ pulls, $\hat{r}_i + \sqrt{2 \ln n / n(i)} < r^*$ and arm i is not pulled anymore

How many is enough?

With high probability, $\hat{r}_i < r_i + \sqrt{2 \ln n / n(i)}$. Then,

$$\hat{r}_i + \sqrt{2 \ln n / n(i)}$$

What is the UCB Formula?

$$UCB(i) = \hat{r}_i + \sqrt{2 \ln n / n(i)}$$

$UCB(i)$ is an **upper bound** on a confidence interval for the **true expected reward** r_i for arm i; that is, **w.h.p.** $r_i < UCB(i)$

After 'enough' pulls, $\hat{r}_i + \sqrt{2 \ln n / n(i)} < r^*$ and arm i is not pulled anymore

How many is enough?

With high probability, $\hat{r}_i < r_i + \sqrt{2 \ln n / n(i)}$. Then,

$$\hat{r}_i + \sqrt{2 \ln n / n(i)} < r_i + 2\sqrt{2 \ln n / n(i)}$$

What is the UCB Formula?

$$UCB(i) = \hat{r}_i + \sqrt{2 \ln n / n(i)}$$

$UCB(i)$ is an **upper bound** on a confidence interval for the **true expected reward** r_i for arm i; that is, **w.h.p.** $r_i < UCB(i)$

After ‘enough’ pulls, $\hat{r}_i + \sqrt{2 \ln n / n(i)} < r^*$ and arm i is not pulled anymore

How many is enough?

With high probability, $\hat{r}_i < r_i + \sqrt{2 \ln n / n(i)}$. Then,

$$\hat{r}_i + \sqrt{2 \ln n / n(i)} < r_i + 2\sqrt{2 \ln n / n(i)} < r^*$$

if $2\sqrt{2 \ln n / n(i)} < r^* - r_i$

What is the UCB Formula?

$$UCB(i) = \hat{r}_i + \sqrt{2\ln n/n(i)}$$

$UCB(i)$ is an **upper bound** on a confidence interval for the **true expected reward** r_i for arm i; that is, **w.h.p.** $r_i < UCB(i)$

After 'enough' pulls, $\hat{r}_i + \sqrt{2\ln n/n(i)} < r^*$ and arm i is not pulled anymore

How many is enough?

With high probability, $\hat{r}_i < r_i + \sqrt{2\ln n/n(i)}$. Then,

$$\hat{r}_i + \sqrt{2\ln n/n(i)} < r_i + 2\sqrt{2\ln n/n(i)} < r^*$$

if $2\sqrt{2\ln n/n(i)} < r^* - r_i$

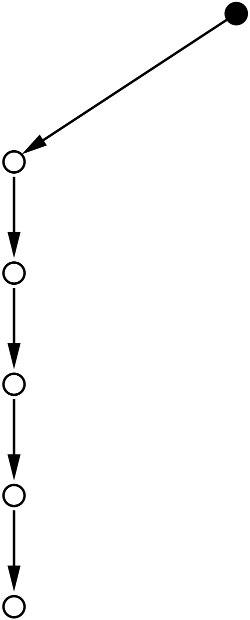
Solving for $n(i)$, $n(i) > \frac{8\ln n}{(r^*-r_i)^2}$ **(max. pulls of suboptimal arm i)**

UCT: Upper Confidence Bounds Applied to Trees

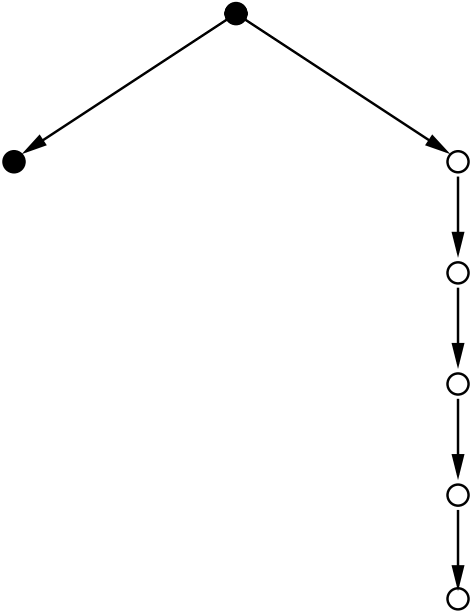
- Generates an **sparse tree of depth** h, one node at a time by stochastic simulation (Monte-Carlo Tree Search)
- Each stochastic simulation starts at root of tree and finishes in the first node that is not in the tree
- The tree is grown to include such node and its value initialized
- The value is propagated upwards towards the root updating sampled averages $\hat{Q}(s,a)$ along the way
- The stochastic simulation descends the tree selecting actions that maximizes

$$\hat{Q}(s,a) + C\sqrt{2\ln n(s)/n(s,a)}$$

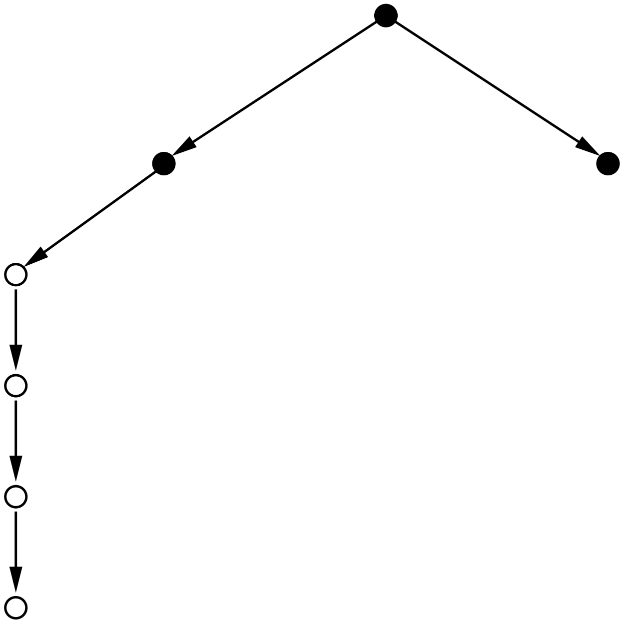
UCT: Example


"

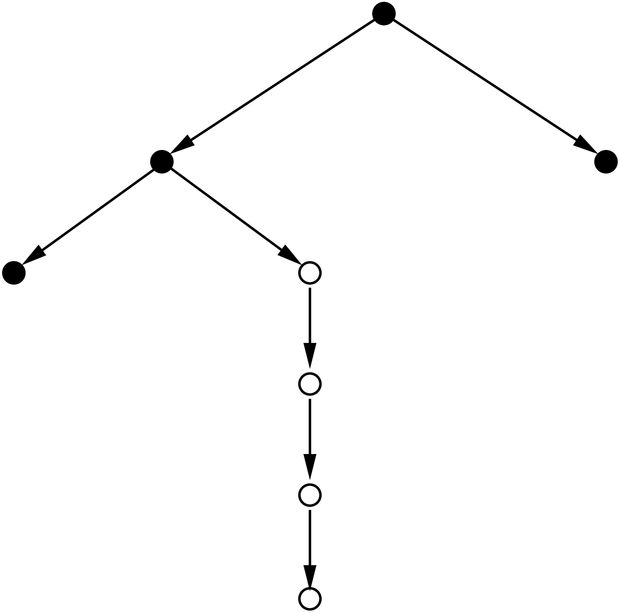
UCT: Example



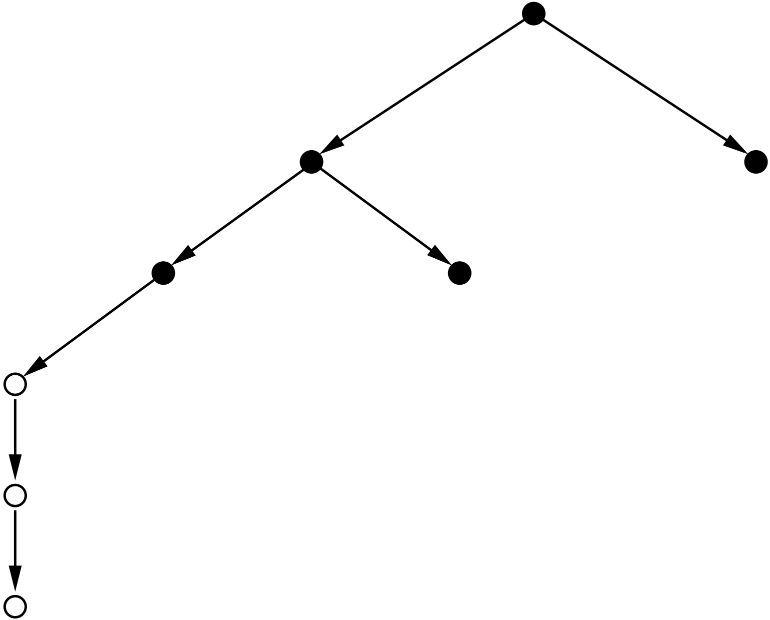
UCT: Example



UCT: Example



UCT: Example



UCT: Success Histories

- Game of Go (GrandMaster level achieved in 9×9 Go)
- Klondike Solitaire (wins 40% of games; human expert 36.6%))
- General Game Playing Competition
- Real-Time Strategy Games
- Canadian Traveller Problem
- Combinatorial Optimization

Summary

- Sometimes the problem is just too big to spply a traditional algorithm or a search-based algorithm
- Monte-Carlo methods designed to work only with a simulator of the problem
- These a are **model-free** algorithms for autonomous behaviour, yet the model is used implictily through simulator
- Important theoretical results for the Multi-Armed Bandit Problem that have far reaching consequences
- UCT algorithm applies the ideas of UCB to MDPs
- Big success of UCT in some applications
- UCT may require a great deal of **tunning** in some cases

References and Related Work I

Introduction:

- H. Geffner. *Tutorial on Advanced Introduction to Planning*. IJCAI 2011.

General MDPs and Stochastic Shortest-Path Problems:

- D. Bertsekas. *Dynamic Programming and Optimal Control*. Vols 1-2. Athena Scientific.
- D. Bertsekas, J. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific.
- M. Puterman. *Markov Decision Processes – Discrete Stochastic Dynamic Programming*. Wiley.

Algorithms for MDPs:

- D. Bertsekas. *Dynamic Programming and Optimal Control*. Vols 1-2. Athena Scientific.
- M. Puterman. *Markov Decision Processes – Discrete Stochastic Dynamic Programming*. Wiley.

References and Related Work II

- B. Bonet, E. Hansen. *Heuristic Search for Planning under Uncertainty*. In *Heuristics, Probability and Causality: A Tribute to Judea Pearl*. College Publications.
- R. Korf. *Real-Time Heuristic Search*. Artificial Intelligence 42, 189–211.
- A. Barto, S. Bradtke, S. Singh. *Learning to Act Using Real-Time Dynamic Programming*. Artificial Intelligence 72, 81–138.
- E. Hansen, S. Zilberstein. *LAO*: A Heuristic Search Algorithm that Finds Solutions with Loops*. Artificial Intelligece 129, 35–62.
- B. Bonet, H. Geffner. *Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming*. ICAPS 2003, 12–21.
- B. Bonet, H. Geffner. *Faster Heuristic Search Algorithms for Planning with Uncertainty and Full Feedback*. IJCAI 2003, 1233–1238.
- B. Bonet, H. Geffner. *Learning Depth-First Search: A Unified Approach to Heuristic Search in Deterministic and Non-Deterministic Settings, and its Applications to MDPs*. ICAPS 2006, 142–151.
- H. McMahan, M. Likhachev, G. Gordon. *Bounded Real-Time Dynamic Programming: RTDP with Monotone Upper Bounds and Performance Guarantees*. ICML 2005, 569–576.

References and Related Work III

- T. Smith, G. Simmons. *Focused Real-Time Dynamic Programming for MDPs: Squeezing More Out of a Heuristic*. AAAI 2006, 1227–1232.
- J. Hoey, R. St-Aubin, A. Hu, C. Boutilier. *SPUDD: Stochastic Planning Using Decision Diagrams*. UAI 1999, 279–288.
- Z. Feng, E. Hansen. *Symbolic Heuristic Search for Factored Markov Decision Processes*. AAAI 2002, 455-460.
- Z. Feng, E. Hansen, S. Zilberstein. *Symbolic Generalization for On-Line Planning*. UAI 2003, 209–216.
- C. Boutilier, R. Reiter, B. Price. *Symbolic Dynamic Programming for First-Order MDPs*. IJCAI 2001, 690–697.
- H. Warnquist, J. Kvarnstrom, P. Doherty. *Iterative Bounding LAO**. ECAI 2010, 341–346.

Heuristics:

- B. Bonet, H. Geffner. *Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming*. ICAPS 2003, 12–21.
- B. Bonet, H. Geffner. *mGPT: A Probabilistic Planner Based on Heuristic Search*. JAIR 24, 933–944.

References and Related Work IV

- R. Dearden, C. Boutilier. *Abstraction and Approximate Decision-Theoretic Planning*. Artificial Intelligence 89, 219–283.
- T. Keller, P. Eyerich. *A Polynomial All Outcome Determinization for Probabilistic Planning*. ICAPS 2011.

Monte-Carlo Planning:

- A. Fern. *Tutorial on Monte-Carlo Planning*. ICAPS 2010.
- = D.P. Bertsekas, J.N. Tsitsiklis, C. Wu. *Rollout algorithms for combinatorial optimization*. Journal of Heuristics 3: 245–262. 1997.
- M. Kearns, Y. Mansour, A.Y. Ng. *A sparse sampling algorithm for near-optimal planning in large MDPs*. IJCAI 99, 1324–1331.
- P. Auer, N. Cesa-Bianchi, P. Fischer. *Finite-time analysis of the multiarmed bandit problem*. Machine Learning 47: 235–256. 2002.
- Success UCT: various: CTP, Sylver's POMDPs, Go, others
- G.M.J. Chaslot, M.H.M. Winands, H. Herik, J. Uiterwijk, B. Bouzy. *Progressive strategies for Monte-Carlo tree search*. New Mathematics and Natural Computation 4. 2008.

References and Related Work V

- L. Kocsis, C. Szepesvari. *Bandit based Monte-Carlo planning*. ECML 2006, 282–293.
- S. Gelly, D. Silver. *Combining online and offline knowledge in UCT*. ICML 2007, 273–280.
- H. Finnsson, Y. Björnsson. *Simulation-based approach to general game playing*. AAAI 2008, 259–264.
- P. Eyerich, T. Keller, M. Helmert. *High-Quality Policies for the Canadian Traveler's Problem*. AAAI 2010.
- R. Munos, P.A. Coquelin. *Bandit Algorithms for Tree Search*. UAI 2007.
- R. Ramanujan, A. Sabharwal, B. Selman. *On Adversarial Search Spaces and Sampling-based Planning*. ICAPS 2010, 242–245.
- D. Silver, J. Veness. *Monte-Carlo Planning in Large POMDPs*. NIPS 2010.
- R.K. Balla, A. Fern. *UCT for Tactical Assault Planning in Real-Time Strategy Games*. IJCAI 2009, 40–45.

International Planning Competition:

- 2004

References and Related Work VI

- 2006
- 2008
- 2011