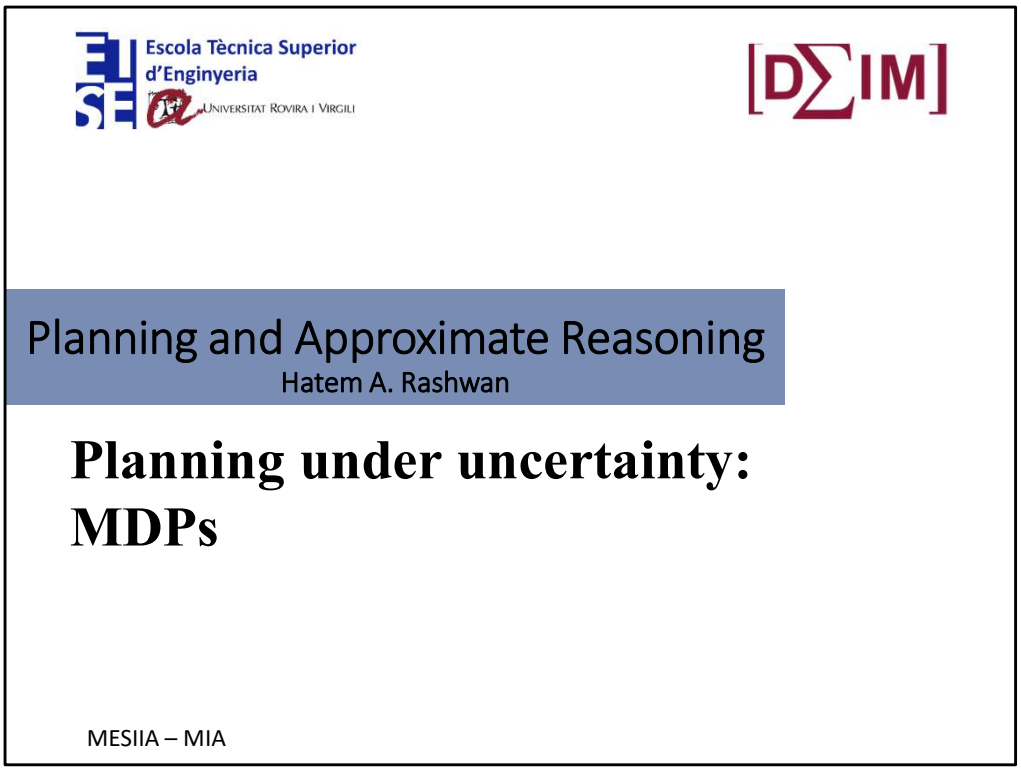


A planning problem in AI is usually specified as follows: Given a description of the current state of a system, a set of actions that can be performed on the system and a description of a goal set of states for the system, find a sequence of actions that can be performed to transform the system into one of the goal states.

In the example of the **blocks world**, the system consists of a set of named blocks, each of which may be on the ground or on top of another block. The possible actions move a block from one location to another and the goal is a particular configuration of blocks, for example a tower. When planning programs are used to **provide courses of action in real-world settings**, such as medical treatments, high-level robot control or disaster relief, they must take into account the fact that actions may have several different outcomes, some of which may be more desirable than others. They must balance the potential of some plan achieving a goal state against the risk of producing an undesirable state and against the cost of performing the plan.

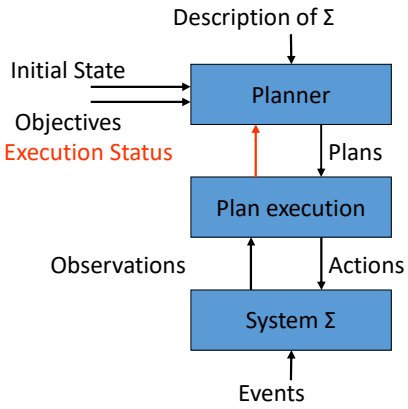
After a period where attention was focussed on efficient algorithms for planning under quite restrictive assumptions, the past twenty years the researchers start to work on **planning under uncertainty** and decision-theoretic planning. There are several reasons for this. The recent advances in Bayesian inference have made it feasible to compute (or approximate) the expected utility of a plan under conditions of uncertainty.

Uncertainties generally interfere with two aspects of planning:
1- Predictability: Due to uncertainties, it is not known what will happen in the future when certain actions are applied. This means that future states are not necessarily predictable.
2-Sensing: Due to uncertainties, the current state is not necessarily known. Information regarding the state is obtained from initial conditions, sensors, and the memory of previously applied actions.
These two kinds of uncertainty are independent in many ways. Each has a different effect on the planning problem.

The success of **Markovian approaches** in areas such as speech recognition and the closely-related reinforcement learning techniques have encouraged work in planning using Markov decision processes.

Now we're going to think about how to do planning in uncertain domains. It's an extension of decision theory, but it focused on making long-term plans of action. We'll start by laying out the basic framework, then look at Markov chains, which are a simple case. Then we'll explore what it means to have an optimal plan for an MDP, and look at an algorithm, called value iteration, for finding optimal plans. We'll finish by looking at some of the major weaknesses of this approach and seeing how they can be addressed.

Plan Execution



- **problem**: real world differs from model described by Σ
- **more realistic model**: interleaved planning and execution
  - plan supervision
  - plan revision
  - re-planning
- **dynamic planning**: closed loop between planner and controller
  - execution status

MESIIA – MIA 2

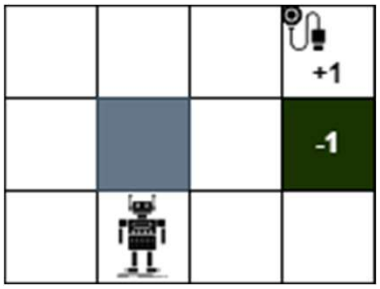
Dynamic Planning

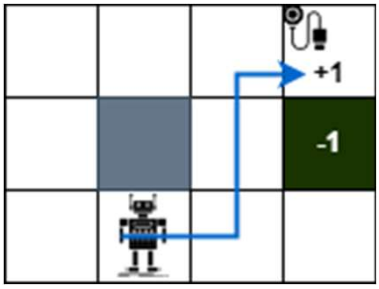
- **problem: physical system differs from model described by Σ**
  - planner only has access to model (description of Σ)
  - controller must cope with differences between Σ and real world
- **more realistic model: interleaved planning and execution**
  - **plan supervision**: detect when observations differ from expected results
  - **plan revision**: adapt existing plan to new circumstances
  - **re-planning**: generate a new plan from current (initial) state
- **dynamic planning: closed loop between planner and controller**
  - **execution status**

As shown in the figure, the execution of these actions is monitored. This way, the system is able to detect execution failures or deviations from the proposed plan and to initiate the repair of the original plan accordingly. But everything depends on the execution unit and its performance. In real cases, there is no ideal and perfect execution.

Planning Example
Robot navigation

- A robot act in the environment aiming to avoid the obstacles and reach its goal.
- The robot aims to reach the celled labelled with +1 and avoid the cell labelled with -1





MESIIA – MIA 3

As shown in the left figure there a robots aims to reach the cells labelled with +1 and avoid the cell labelled with -1

If we applied the classical planning algorithm (e.g., A*), we will concern with the short path with the minimal steps as shown the right figure. But this may be problematic, due to:

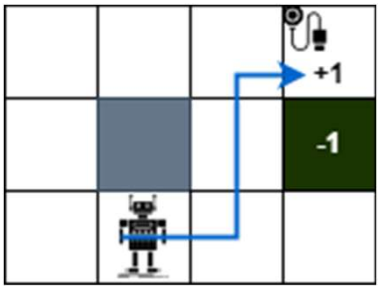
Dynamic Environments: A* assumes a static environment, but in real-world scenarios, the environment may be dynamic, with obstacles moving or appearing unexpectedly. A* may not be equipped to handle dynamic changes in the environment effectively.
Optimality Assumption: A* assumes that the shortest path with the fewest steps is always the best option. However, in certain scenarios, the robot might need to consider factors like energy efficiency, safety, or other constraints, which A* might not be able to incorporate into its decision-making process.
Uncertainty and Risk: A* does not typically account for uncertainties or risks associated with various paths. Real-world scenarios often involve uncertain or risky environments, and the robot needs to make decisions that prioritize safety and risk avoidance.

Planning Example
Why the short path is problematic

- Non-perfect execution: actions are performed with a probability < 1
- What are the best actions for a robot (an agent) under this constraint? (move right->up->up->right)
- And what is the maximum performance of the execution? (e.g., 99%)

Example

- A robot does not exactly perform the desired motions due to different reasons.
  - Uncertainty about performing actions



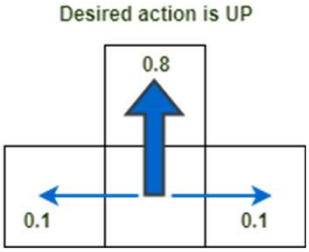
- Non-perfect execution: actions are performed with a probability < 1
- What are the best actions for a robot (an agent) under this constraint? (e.g., 99%)

Example

- A robot does not exactly perform the desired motions due to different reasons.
  - Uncertainty about performing actions

Planning Example
Non-Deterministic Transitions

- Consider non-deterministic transition model (UP/R/L)



Example

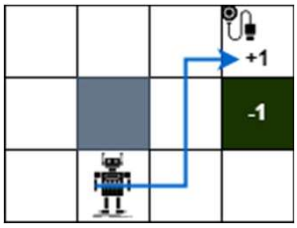
- Intended action is executed with p=0.8
- With p = 0.1, the robot can move right or left.
  - Uncertainty about performing actions

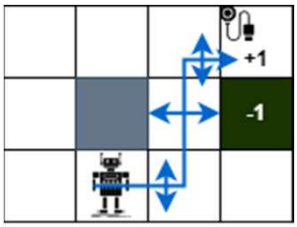
- Intended action is executed with p=0.8
- With p = 0.1, the robot can move right or left.
  - Uncertainty about performing actions will be occurred

This uncertainty might arise due to various factors, such as:
1.Lack of Information: When individuals or professionals do not have complete or accurate information, they might hesitate or feel uncertain about the appropriate steps to take or the best course of action.
2.Complexity: Some tasks or procedures might be complex or require specialized skills, leading to uncertainty or doubts about their successful execution.
3.Risk Assessment: Evaluation of potential risks and uncertainties associated with a particular action can lead to cautiousness or hesitancy, especially if the consequences of the action are unclear or potentially harmful.

Markov Decision Process (MDP) motivation

- Executing the A* plan





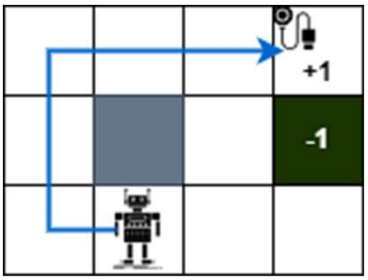
Transitions are non-deterministic

- Uncertainty about performing actions will be occurred

If we applied A* concerning on the shortest path, there is a propagability of 0.1 that the robot can go to the hole labelled with -1. so, we need to find another way to solve the planning process.

MDP motivation

- Perhaps using longer path with lower probability to not reach the cell labelled -1 is good option.



This proposed path can have the highest overall utility.

So, we need to find a longer path with a lower probability to not reach the cell labelled -1. The proposed path can have the highest overall utility. And this is exactly the MDP working on.

Axioms of Probability

- Let A be a proposition about the world
- P(A) = probability proposition A is true
- 0 <= P(A) <= 1
- P(True) = 1
- P(False) = 0
- P(A or B) = P(A) + P(B) – P(A and B)

Random Variables

- Random Variables: variables in probability to capture phenomena
- A random variable has a **domain of values** it can take on.
- Probability distribution function represents "**probability of each value**"

Firstly, we need to introduce to axioms of probability.
Chance is a slippery concept. We all know that some random events are more likely to occur than others, but how do you quantify such differences? How do you work out the probability of, say, rolling a 6 on a die? And what does it mean to say the probability is 1/6?

Mathematicians avoid these tricky questions by defining the probability of an event mathematically without going into its deeper meaning. At the heart of this definition are three conditions, called the axioms of probability theory.
Axiom 1: The probability of an event is a real number greater than or equal to 0 and smaller than or equal to 1.
Axiom 2: The probability that at least one of all the possible outcomes of a process (such as rolling a die) will occur is 1.
Axiom 3: If two events A and B are mutually exclusive, then the probability of either A or B occurring is the probability of A occurring plus the probability of B occurring.

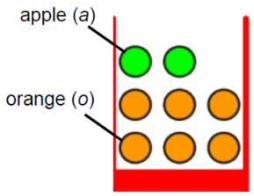
A **random variable** is a numerical description of the outcome of a statistical experiment.
A random variable that may assume only a finite number or an infinite sequence of values is said to be discrete; one that may assume any value in some interval on the real number line is said to be continuous.
For instance, a random variable representing the number of automobiles sold at a particular dealership on one day would be discrete, while a random variable representing the weight of a person in kilograms (or pounds) would be continuous.

The probability distribution for a random variable describes how the probabilities are distributed over the values of the random variable. For a discrete random variable, x, the probability distribution is defined by a probability mass function, denoted by f(x). This function provides the probability for each value of the random variable. In the development of the probability function for a discrete random variable, two conditions must be satisfied: (1) f(x)

must be nonnegative for each value of the random variable, and (2) the sum of the probabilities for each value of the random variable must equal one.

Example – Pick Fruit from Basket

- Random variable: F
- Domain: a, o
- PDF:
  - p(F = a) = ¼
  - p(F = o) = ¾



- The **expected value** of a function of a random variable **is the weighted average of the probability distribution over outcomes.**

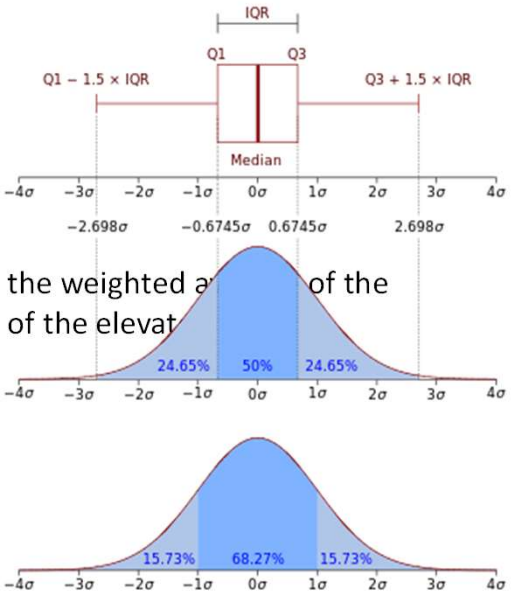
- Example: calculate the expected time of waiting for an elevator
- Time: 5ms 2ms 0.5ms
- Probability: 0.2 0.7 0.1

5 x 0.2 + 2 x 0.7+ 0.5 x 0.1 = **2.45ms**

MESIIA – MIA 9

The **probability density function** (**PDF**) is used to specify the probability of the random variable falling within a particular range of values, as opposed to taking on any one value. This probability is given by the integral of this variable's PDF over that range—that is, it is given by the area under the density function but above the horizontal axis and between the lowest and greatest values of the range. The probability density function is nonnegative everywhere, and its integral over the entire space is equal to 1. See the figure PDF of normal distribution.

In the example of fruit shown in the figure:
The probability of pick up apple is 2/8 (1/4), and the Probability of pick up orange is 6/8 (3/4). Note the The Summation of the two probabilities is 1.



The expected value of a function of a random variable is the weighted a of the probability distribution over outcomes, see the example of the elevat

Transition Model

• Given that our agent is in some state s, there is a probability to go to the first state, another probability to go to the second state, and so on for every existing state.

- This is our transition probability.
- Probability to reach the next state *s'* from state *s* by choosing action *a* is $P(s, a, s') \sim P(s|s', a)$ => It is called **Transition model**

Markov Property:

The transition probabilities from *s* to *s'* depend only on the current state *s* and not on the history of earlier states.

Reward:

- In each state s, the robot (agent) receives a reward *R(s)*.
- The reward may be positive or negative but must be bounded
- This can be generalized to be a reward function *R(s, a, s')*

Given that our agent is in some state s, there is a probability to go to the first state, another probability to go to the second state, and so on for every existing state. This is our transition probability.

Probability to reach the next state *s'* from state *s* by choosing action *a* is *P(s, a, s')*

- It is called **Transition model**

Markov Property:

The transition probabilities from *s* to *s'* depend only on the current state *s* and not on the history of earlier states.

Reward:

- In each state s, the robot (agent) receives a reward R(s).
- The reward may be positive or negative but must be bounded
- This can be generalized to be a reward function R(s, a, s')

Reward

• In our example, the reward is **-0.04** in all states (e.g**., the cost of the motion**) except the terminal states (**reward is +1/-1**)

A negative reward gives an incentive
- **to reach the goal quickly**
- **Or "living in this environment is not enjoyable"**

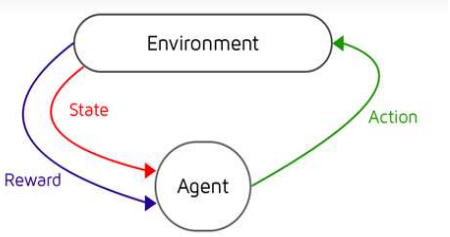
In our example, the reward is **-0.04** in all states (e.g**., the cost of the motion**) except the terminal states (**reward is +1/-1**).

Markov Decision Process-MDP

Given a **sequential decision problem** in fully observable, stochastic environment with a known Markovian transition model

Then a Markov Decision Process-MDP ***(S, A, Si, P, R)*** is defined by the components of:

- **Set of states: *S***
- **Set of actions: *A***
- **Initial sates: *Si***
- **Transition model: *P(s,a,s')***
- **Reward function: *R(s)***
  - Here: considering only R(s) (does not change the problem)



MESIIA – MIA

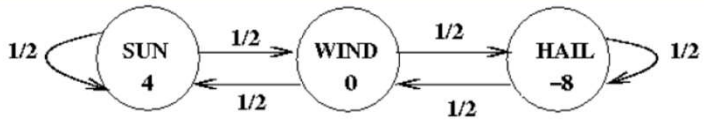
12

Given a **sequential decision problem** in fully observable, stochastic environment with a known Markovian transition model

Then a Markov Decision Process-MDP ***(S, A, Si, P, R)*** is defined by the components of:

- **Set of states: *S***
- **Set of actions: *A***
- **Initial sates: *Si***
- **Transition model: *P(s,a,s')***
- **Reward function: *R(s)***
  - Here: considering only R(s) (does not change the problem)

Example – Markov System with Reward



- Process/observation:
  - Assume start state s_i
  - Receive immediate reward r_i
  - Move, or observe a move, randomly to a new state according to the probability transition matrix
  - **Future rewards (of next state) are discounted by γ**

probability transition matrix *T* is

sun	wind	hail
1/2	1/2	0
1/2	0	1/2
0	1/2	1/2

The result of classical planning was a plan. A plan was either an ordered list of actions, or a partially ordered set of actions, meant to be executed without reference to the state of the environment. When we looked at conditional planning, we considered building plans with branches in them, that observed something about the state of the world and acted differently depending on the observation.

In an MDP, the assumption is that you could potentially go from any state to any other state in one step. And so, to be prepared, it is typical to compute a whole **policy**, rather than a simple plan. A policy is a mapping from states to actions. It says, no matter what state you happen to find yourself in, here is the action that it's best to take now. Because of the Markov property, we'll find that the choice of action only needs to depend on the current state (and possibly the current time), but not on any of the previous states.

Markov decision processes are an extension of Markov chains; the difference is the addition of actions (allowing choice) and rewards (giving motivation). Conversely, if only one action exists for each state (e.g. "wait") and all rewards are the same (e.g. "zero"), a Markov decision process reduces to a Markov chain.

So, what is our criterion for finding a good policy? In the simplest case, we'll try to find the policy that maximizes, for each state, the expected reward of executing that policy in the state. This is a particularly easy problem, because it completely decomposes into a set of decision problems: for each state, find the single action that maximizes expected reward.

In the example shown of the weather per day, you have three states S1 (sun), S2(wind), S3(hail), with three Reward functions R(s1) , R(s2) and R(s3) and with probability transition matrix is

$$\begin{matrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1/2 \\ 0 & 1/2 & 1/2 \end{matrix}$$

Example – Markov System with Reward

Solving a Markov System with Rewards

- $V^*(s_i)$ - expected discounted sum of future rewards starting in state s_i
- $V^*(s_i) = r_i + \gamma[p_{i1}V^*(s_1) + p_{i2}V^*(s_2) + \ldots p_{in}V^*(s_n)]$

- γ is a discount factor, where $\gamma \in [0, 1]$.
- It informs the agent of how much it should care about rewards now to rewards in the future.
- If ($\gamma = 0$), that means the agent is short-sighted, in other words, it only cares about the first reward.
- If ($\gamma = 1$), that means the agent is far-sighted, i.e. it cares about all future rewards.
- What we care about is the total rewards that we're going to get.

It's not usually good to be so short-sighted, though! Let's think a little bit farther ahead. We can consider "finite-horizon optimal" policies for a particular horizon k. That means that we should find a policy that, for every initial state s0, results in the maximal expected sum of rewards from times 0 to k.

So, if the horizon is 2, we're maximizing our reward today and tomorrow. If it's 300, then we're looking a lot farther ahead. We might start by doing some actions with very low rewards initially because by doing them, we are likely to be taken to states that will ultimately result in high reward. (Students are often familiar with the necessity of passing through low-reward states in order to get to states with a higher reward!). Because we will generally want to choose actions differently at the very end of our lives (when the horizon is short) than early in our lives, it will be necessary to have a non-stationary policy in the finite-horizon case. That is, we'll need a different policy for each number of time steps remaining in our life.

Because, in many cases, it's not clear how long the process will run (consider designing a robot sentry or a factory controller), it's popular to consider infinite horizon models of optimality. But if we add all the rewards into infinity, then the sums will be infinite in general. We use a discount factor to keep the math nice and put some pressure on the agent to get rewards sooner rather than later. A discount factor gamma is a number between 0 and 1, which has to be strictly less than 1. Usually, it's somewhere near 0.8 or 0.99. So, we want to maximize our sum of rewards, but rewards that happen tomorrow are only worth .9 of what they would be worth today. You can think of this as a model of the present value of money, as in economics. Or that your life will end with probability 1-gamma on each step, but you don't know when.

Assume you are an engineer in a company that takes a salary 30k per year; you can stay during your life as an engineer or after two years, you will be a senior engineer with a salary 50k per year, or you can leave the work and get nothing. However, the reward in the future is not worth quite as much as a reward now - Because of a chance of inflation or obliteration • Example: - Being promised $10000 next

year is worth only 90% as much as receiving $10000 now • Assuming payment n years in the future is worth only (0.9)n of payment now, what is the engineer's Future Discounted Sum of Rewards?

Value Iteration to Solve a Markov System with Rewards

- $V_1(s_i)$ - expected discounted sum of future rewards starting in state s_i *for one step*.
- $V_2(s_i)$ - expected discounted sum of future rewards starting in state s_i *for two steps*.
- ...
- $V_k(s_i)$ - expected discounted sum of future rewards starting in state *si for k steps*.
- As $k \rightarrow \infty V_k(s_i) \rightarrow V^*(s_i)$
- Stop when difference of $k + 1$ and k values is smaller than some $\in$.

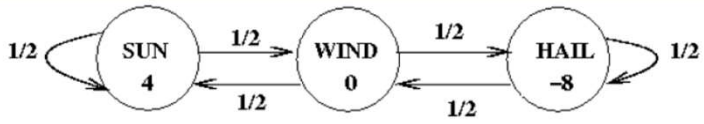
This model has the very convenient property that the optimal policy is stationary. The model is independent of how long the agent has run or will run in the future (since nobody knows that exactly). Once you've survived to live another day, in this model, the expected length of your life is the same as it was on the previous step, and so your behaviour is the same, as well.

Discounted sum of future awards using discount factor γ is - Reward now + γ(reward in 1 time step) + γ2 (reward in 2 time steps) + γ3 (reward in 3 time steps) + ...

If you have n states you are solving an n by n system of equations!
• Define
- V1(si)=Expected discounted sum of rewards over next 1 time step
- V2(si)=Expected discounted sum of rewards over next 2 time steps
- V3(si)=Expected discounted sum of rewards over next 3 time steps
- ...
-Vk(si)=Expected discounted sum of rewards over next k time steps

Stop when $(V_k - V_{k-1}) \sim \in$

Example – Markov System with Reward



3-State Example: Values $\gamma = 0.5$

Iteration	SUN	WIND	HAIL
0	0	0	0
1	4	0	-8
2	5.0	-1.0	-10.0
3	5.0	-1.25	-10.75
4	4.9375	-1.4375	-11.0
5	4.875	-1.515625	-11.109375
6	4.8398437	-1.5585937	-11.15625
7	4.8203125	-1.5791016	-11.178711
8	4.8103027	-1.5895996	-11.189453
9	4.805176	-1.5947876	-11.194763
10	4.802597	-1.5973969	-11.197388
11	4.8013	-1.5986977	-11.198696
12	4.8006506	-1.599349	-11.199348
13	4.8003254	-1.5996745	-11.199675
14	4.800163	-1.5998373	-11.199837
15	4.8000813	-1.5999185	-11.199919

The result of classical planning was a plan. A plan was either an ordered list of actions, or a partially ordered set of actions, meant to be executed without reference to the state of the environment. When we looked at conditional planning, we considered building plans with branches in them, that observed something about the state of the world and acted differently depending on the observation.

In an MDP, the assumption is that you could potentially go from any state to any other state in one step. And so, to be prepared, it is typical to compute a whole **policy**, rather than a simple plan. A policy is a mapping from states to actions. It says, no matter what state you happen to find yourself in, here is the action that it's best to take now. Because of the Markov property, we'll find that the choice of action only needs to depend on the current state (and possibly the current time), but not on any of the previous states.

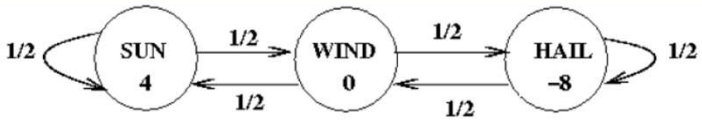
Markov decision processes are an extension of Markov chains; the difference is the addition of actions (allowing choice) and rewards (giving motivation). Conversely, if only one action exists for each state (e.g. "wait") and all rewards are the same (e.g. "zero"), a Markov decision process reduces to a Markov chain.

So, what is our criterion for finding a good policy? In the simplest case, we'll try to find the policy that maximizes, for each state, the expected reward of executing that policy in the state. This is a particularly easy problem, because it completely decomposes into a set of decision problems: for each state, find the single action that maximizes expected reward.

In the example shown of the weather per day, you have three states S1 (sun), S2(wind), S3(hail), with three Reward functions R(s1) , R(s2) and R(s3) and with probability transition matrix is

$$\begin{matrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1/2 \\ 0 & 1/2 & 1/2 \end{matrix}$$

Example – Markov System with Reward



3-State Example: Values $\gamma = 0.9$

Iteration	SUN	WIND	HAIL
0	0	0	0
1	4	0	-8
2	5.8	-1.8	-11.6
3	5.8	-2.6100001	-14.030001
4	5.4355	-3.7035	-15.488001
5	4.7794	-4.5236254	-16.636175
6	4.1150985	-5.335549	-17.521912
7	3.4507973	-6.0330653	-18.285858
8	2.8379793	-6.6757774	-18.943516
9	2.272991	-7.247492	-19.528683
...	...	...	...
50	-2.8152928	-12.345073	-24.633476
51	-2.8221645	-12.351946	-24.640347
52	-2.8283496	-12.3581295	-24.646532
...	...	...	...
86	-2.882461	-12.412242	-24.700644
87	-2.882616	-12.412397	-24.700798
88	-2.8827558	-12.412536	-24.70094

The result of classical planning was a plan. A plan was either an ordered list of actions, or a partially ordered set of actions, meant to be executed without reference to the state of the environment. When we looked at conditional planning, we considered building plans with branches in them, that observed something about the state of the world and acted differently depending on the observation.

In an MDP, the assumption is that you could potentially go from any state to any other state in one step. And so, to be prepared, it is typical to compute a whole **policy**, rather than a simple plan. A policy is a mapping from states to actions. It says, no matter what state you happen to find yourself in, here is the action that it's best to take now. Because of the Markov property, we'll find that the choice of action only needs to depend on the current state (and possibly the current time), but not on any of the previous states.

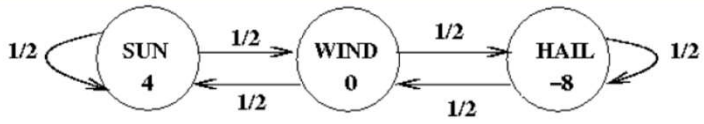
Markov decision processes are an extension of Markov chains; the difference is the addition of actions (allowing choice) and rewards (giving motivation). Conversely, if only one action exists for each state (e.g. "wait") and all rewards are the same (e.g. "zero"), a Markov decision process reduces to a Markov chain.

So, what is our criterion for finding a good policy? In the simplest case, we'll try to find the policy that maximizes, for each state, the expected reward of executing that policy in the state. This is a particularly easy problem, because it completely decomposes into a set of decision problems: for each state, find the single action that maximizes expected reward.

In the example shown of the weather per day, you have three states S1 (sun), S2(wind), S3(hail), with three Reward functions R(s1) , R(s2) and R(s3) and with probability transition matrix is

$$\begin{matrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1/2 \\ 0 & 1/2 & 1/2 \end{matrix}$$

Example – Markov System with Reward



3-State Example: Values $\gamma = 0.2$

Iteration	SUN	WIND	HAIL
0	0	0	0
1	4	0	-8
2	4.4	-0.4	-8.8
3	4.4	-0.44000003	-8.92
4	4.396	-0.452	-8.936
5	4.3944	-0.454	-8.9388
6	4.39404	-0.45443997	-8.93928
7	4.39396	-0.45452395	-8.939372
8	4.393944	-0.4545412	-8.939389
9	4.3939404	-0.45454454	-8.939393
10	4.3939395	-0.45454526	-8.939394
11	4.3939395	-0.45454547	-8.939394
12	4.3939395	-0.45454547	-8.939394

The result of classical planning was a plan. A plan was either an ordered list of actions, or a partially ordered set of actions, meant to be executed without reference to the state of the environment. When we looked at conditional planning, we considered building plans with branches in them, that observed something about the state of the world and acted differently depending on the observation.

In an MDP, the assumption is that you could potentially go from any state to any other state in one step. And so, to be prepared, it is typical to compute a whole **policy**, rather than a simple plan. A policy is a mapping from states to actions. It says, no matter what state you happen to find yourself in, here is the action that it's best to take now. Because of the Markov property, we'll find that the choice of action only needs to depend on the current state (and possibly the current time), but not on any of the previous states.

Markov decision processes are an extension of Markov chains; the difference is the addition of actions (allowing choice) and rewards (giving motivation). Conversely, if only one action exists for each state (e.g. "wait") and all rewards are the same (e.g. "zero"), a Markov decision process reduces to a Markov chain.

So, what is our criterion for finding a good policy? In the simplest case, we'll try to find the policy that maximizes, for each state, the expected reward of executing that policy in the state. This is a particularly easy problem, because it completely decomposes into a set of decision problems: for each state, find the single action that maximizes expected reward.

In the example shown of the weather per day, you have three states S1 (sun), S2(wind), S3(hail), with three Reward functions R(s1) , R(s2) and R(s3) and with probability transition matrix is

$$\begin{matrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1/2 \\ 0 & 1/2 & 1/2 \end{matrix}$$

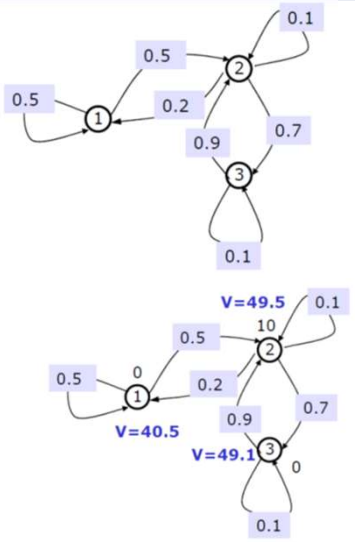
Markov Chain - Example

- Markov Chain
  - states
  - transitions
  - rewards
  - no actions
- Value of a state, using infinite discounted horizon

$$V^*(s_i) = r_i + \gamma[p_{i1}V^*(s_1) + p_{i2}V^*(s_2) + \ldots p_{in}V^*(s_n)]$$

- Assume γ=0.9

V(1)= 0+ .9(.5 V(1)+.5 V(2))
V(2)=10+ .9(.2 V(1)+.1 V(2)+.7 V(3))
V(3)= 0+ .9(.9 V(2)+.1 V(3))



Here's a tiny example of a Markov chain. It has three states. The transition probabilities are shown on the arcs between states. Note that the probabilities on all the outgoing arcs of each state sum to 1. Markov chains don't always have reward values associated with them, but we're going to add rewards to ours. We'll make states 1 and 3 have an immediate reward of 0, and state 2 have immediate reward of 10.

Now, we can define the infinite horizon expected discounted reward as a function of the starting state. We'll abbreviate this as the **value** of a state. So, how much total reward do we expect to get if we start in state s? Well, we know that we'll immediately get a reward of R(s).

But then, we'll get some reward in the future. The reward we get in the future is not worth as much to us as reward in the present, so we multiply by discount factor gamma.

Now, we consider what the future might be like. We'll compute the expected long-term value of the next state by summing over all possible next states, s', the product of the probability of making a transition from s to s' and the infinite horizon expected discounted reward, or **value** of s'. Since we know R and P (those are given in the specification of the Markov chain), we'd like to compute V. If n is the number of states in the domain, then we have a set of n equations in n unknowns (the values of each state). Luckily, they're easy to solve. So, here are the equations for the values of the states in our example, assuming a discount factor of 0.9.
Now, if we solve for the values of the states, we get that V(1) is 40.5, V(2) is 49.5, and V(3) is 49.1. This seems at least intuitively plausible. State 1 is worth the least, because it's kind of hard to get from there to state 2, where the reward is high. State 2 is worth the most; it has a large reward and it usually goes to state 3, which usually comes right back again for another large reward. State 3 is close to state 2 in value, because it usually takes only one step to get from 3 back to 2.
If we set gamma to 0, then the values of the nodes would be the same as their rewards. If gamma were small but non-zero, then the values would be smaller than in this case and their differences more pronounced.

Markov Decision Processes

- Finite set of states, s_1,..., s_n
- **Finite set of actions, a_1,..., a_m**
- Probabilistic state, action transitions:
- $P_{ij}^{K} = prob\ (next = s_j \mid current = s_i \mid take\ action = a_k)$
- ***Markov property***: State transition function only dependent on current state, not on the "history" of how the state was reached.
- Reward for each state, $r1$,..., r_n
- Process:
  - Start in state s_i
  - Receive immediate reward r_i
  - Choose action $a_k \in A$
  - Change to state s_j with probability P_{ij}^{k}
  - Discount future rewards

The input, outcomes and process of MDP.
The difference between MDP from MC is adding the actions to the process.

Solving an MDP

- Find an action to apply to each state.
- A policy is a mapping from states to actions.
- Optimal policy - for every state, there is no other action that gets a higher sum of discounted future rewards.
- For every MDP there exists an optimal policy.
- Solving an MDP is finding an optimal policy.
- A specific policy converts an MDP into a plain Markov system with rewards.

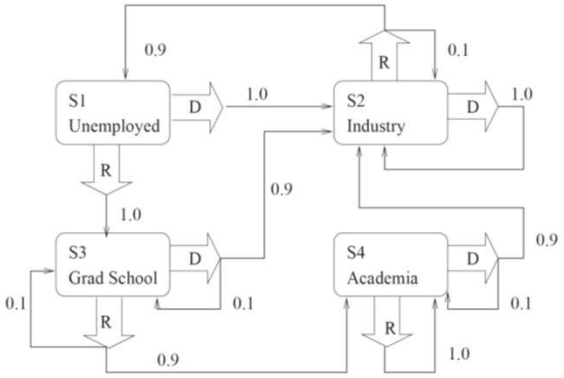
- MDP + Policy(action) = Markov Chain
- MDP = the way the world works
- Policy = the way the agent works.

Now, we'll go back to thinking about Markov Decision Processes. If you take an MDP and fix the policy, then all the actions are chosen and what you have left is a Markov chain. So, given a policy, it's easy to evaluate it, in the sense that you can compute what value the agent can expect to get from each possible starting state, if it executes that policy.

A **Markov Decision Process (MDP)** model contains:
A set of possible world states S.
A set of Models.
A set of possible actions A.
A real valued reward function R(s',s,a).
A policy the solution of **Markov Decision Process**.

Now the difference between MDP and Marchov chain is the set of actions.

Solving an MDP - Example



- Note the need to have a finite set of states and actions (R or D). R=> Research, and D=> Development
- Note the need to have all transition probabilities.

Value Iteration

- $V^*(s_i)$ - expected discounted future rewards, if we start from state s_i, and we follow the optimal policy.
- Compute V^* with value iteration:
  - $V_k(s_i)$ = maximum possible future sum of rewards starting from state s_i for k steps.
- Bellman's Equation:

$$V^{n+1}(s_i) = max_k\{r_i + \gamma \sum_{j=1}^{N} P_{ij}^k V^n(s_j)\}$$

- Dynamic programming

MESIIA – MIA

23

We want to find the best possible policy. We'll approach this problem by thinking about V*, the optimal value function. V* is defined using the following set of recursive equations. The optimal value of a state s is the reward that we get in s, plus the maximum over all actions we could take in s, of the discounted expected optimal value of the next state.

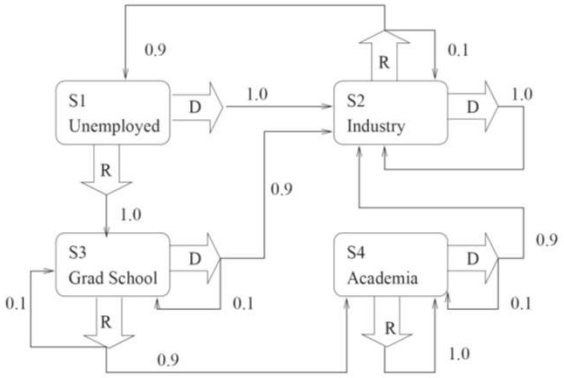
The idea is that in every state, we want to choose the action that maximizes the value of the future.

This is sort of like the set of equations we had for the Markov chain. We have n equations in n unknowns. The problem is that now we have these annoying "max" operators in our equations, which makes them non-linear, and therefore non-trivial to solve. Nonetheless, there is a theorem that says, given R and P, there is a unique function V* that satisfies these equations.

If we know V*, then it's easy to find the optimal policy. The optimal policy, pi*, guarantees that we'll get the biggest possible infinite-horizon expected discounted reward in every state. So, if we find ourselves in a state s, we can pick the best action by considering, for each action, the average the V* value of the next state according to how likely it is to occur given the current state s and the action under consideration. Then, we pick the action that maximizes the expected V* on the next step.

So, we've seen that if we know V*, then we know how to act optimally. So, can we compute V*? It turns out that it's not too hard to compute V*. There are three fairly standard approaches to it. We'll just cover the first one, value iteration.

Nondeterministic Example



- Reward and discount factor to be decided.
- Note the need to have a finite set of states and actions.
- Note the need to have all transition probabilities.

Value Iteration-Bellman equation

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

$$\pi_{k+1}(s_i) = r_i + \arg max_a\{\gamma \sum_{j=1}^{N} P_{ij}^a V^{\pi_n}(s_j)\}$$

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

Here's the value iteration algorithm. We are going to compute V*(s) for all s, by doing an iterative procedure, in which our current estimate for V* gets closer to the true value over time. We start by initializing V(s) to 0, for all states. We could actually initialize to any values we wanted to, but it's easiest to just start at 0.

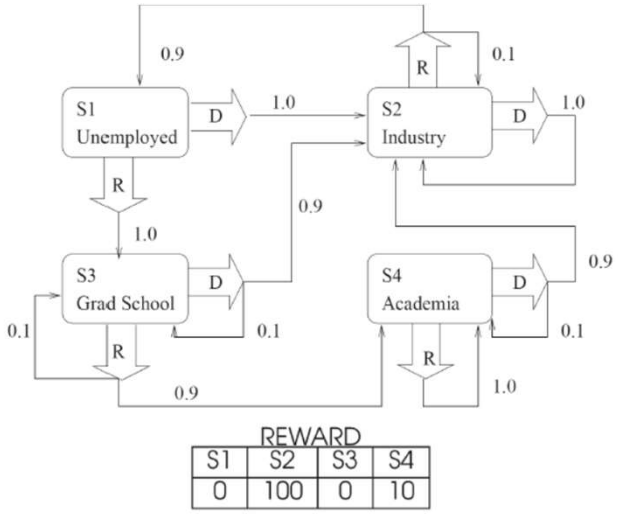
Now, we loop for a while (we'll come back to the question of how long).
Now, for every s, we compute a new value, V_t+1 (s) using the equation that defines V* as an assignment, based on the previous values V_t of V. So, on each iteration, we compute a whole new value function given the previous one.

This algorithm is guaranteed to converge to V*. It might seem sort of surprising. We're starting with completely bogus estimates of V, and using them to make new estimates of V. So why does this work? It's possible to show that the influence of R and P, which we know, drives the successive Vs to get closer and closer to V*.

Not only does this algorithm converge to V*, we can simplify it a lot and still retain convergence. First of all, we can get rid of the different V_t functions in the algorithm and just use a single V, in both the left and right hand sides of the update statement.

In addition, we can execute this algorithm asynchronously. That is, we can do these state-update assignments to the states in any order we want to. In fact, we can even do them by picking states at random to update, as long as we update all the states sufficiently often.

Nondeterministic Example



This is an example of a system with 4 states (s1, s2, s3 and s4) and with each state two actions (R and D).
With four reward R(s1) = 0, R(s2) = 100, R(s3) = 0 and R(s4) = 10.

Now, let's go back to the "loop for a while" statement, and make it a little bit more precise. Let's say we want to guarantee that when we terminate, the current value function differs from V* by at most epsilon (the double bars indicate the max norm, which is just the biggest difference between the two functions, at any argument s).

In order to guarantee this condition, it is sufficient to examine the maximum difference between V_t and Vt+1. As soon as it is below epsilon times (1 – gamma) / gamma, we know that V_t is within epsilon of V*.

Now, although we may never converge to the exact, analytically correct V* in a finite number of iterations, it's guaranteed that we can get close enough to V* so that using V to choose our actions will yield the optimal policy within a finite number of iterations. In fact, using value iteration, we can find the optimal policy in time that is polynomial in the size of the state and action spaces, the magnitude of the largest reward, and in 1/1-gamma.

Nondeterministic Example

$\pi^*(s) = D$, for any s= S1, S2, S3, and S4, $\gamma = 0.9$.

```
V*(S2) = r(S2,D) + 0.9 (1.0 V*(S2))
V*(S2) = 100 + 0.9 V*(S2)
V*(S2) = 1000.

V*(S1) = r(S1,D) + 0.9 (1.0 V*(S2))
V*(S1) = 0 + 0.9 x 1000
V*(S1) = 900.

V*(S3) = r(S3,D) + 0.9 (0.9 V*(S2) + 0.1 V*(S3))
V*(S3) = 0 + 0.9 (0.9 x 1000 + 0.1 V*(S3))
V*(S3) = 81000/91.

V*(S4) = r(S4,D) + 0.9 (0.9 V*(S2) + 0.1 V*(S4))
V*(S4) = 40 + 0.9 (0.9 x 1000 + 0.1 V*(S4))
V*(S4) = 85000/91.
```


For a policy of the action D, we can compute V* for each state as shown above.
We have four states, so we will have four equations in four unknown V* (S1), V* (S2), V* (S3) and V* (S4) of a policy (D).
Simialry applied the second policy, action (R), and compute V* (S1), V* (S2), V* (S3) and V* (S4) of a policy (R).

Then the new value will be the máximum value of each state.

Repeat the calculation until the V of each state is fixed.

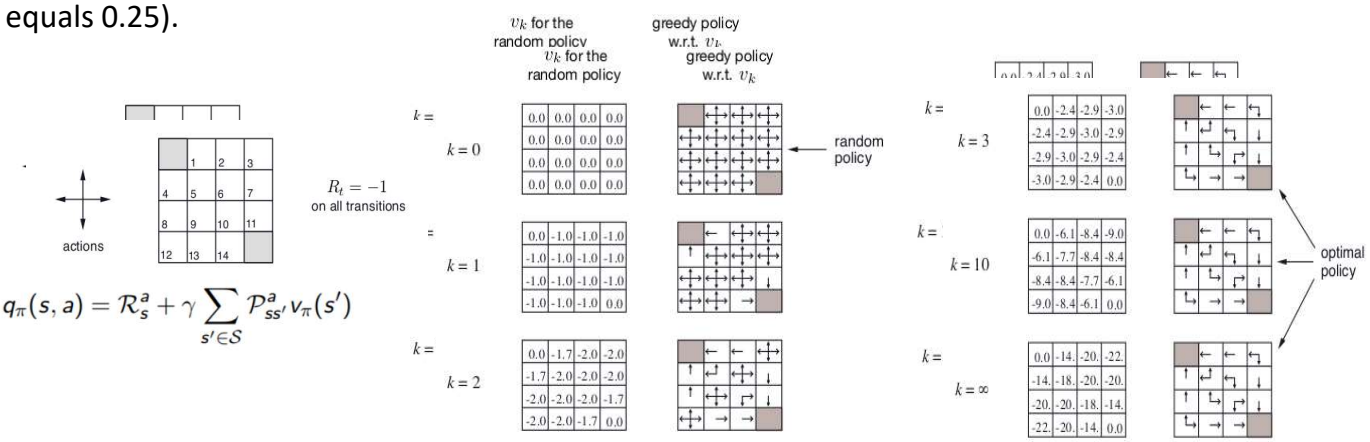
Solve it for a policy of R

Solve the MDP

Markov Decision Processes satisfy both mentioned properties.

- Bellman equation gives us recursive decomposition (the first property).
- Bellman equation tells us how to break down the optimal value function into two pieces,
  - the optimal behaviour for one step followed by
  - the optimal behaviour after that step.
- We can do prediction, i.e., evaluate the given policy to get the value function on that policy (Dynamic Programming).
- ***Evaluating a random policy***
- ***Policy Update***

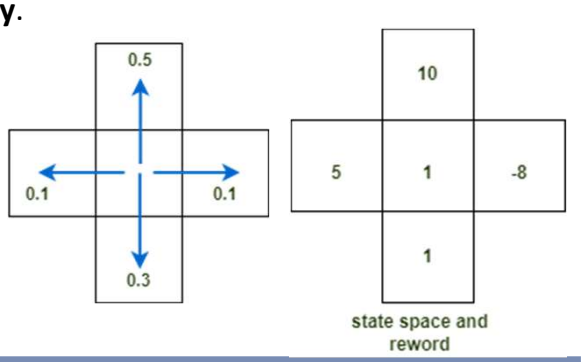
- Dynamic programming assumes full knowledge of the MDP. It's used in planning. There are two main ideas we tackle in a given MDP. If someone tells us the MDP, where ***M*** = (***S, A***, ***P***, ***R***, ***γ***), and a policy ***π*** where ***M*** = (***S***, ***P***, ***R***, ***γ***), we can do prediction, i.e. evaluate the given policy to get the value function on that policy.
- The other main idea is, we are given an MDP, ***M*** = (***S, A***, ***P***, ***R***, ***γ***) and we are asked for the optimal value function and optimal policy. This is full optimization, this is what we are after and it will allow us to do control in a given MDP. What we are trying to figure out is the best thing to do, and this is what we think of when we're trying to solve an MDP.
- ***Policy Evaluation:*** In policy evaluation, we're given an MDP and a policy ***π***. We try to evaluate the given policy. We can evaluate the given policy through iterative application of Bellman expectation equation, and later use the Bellman optimality equation to do control. We are going to start off with an initial value function, e.g. zero for all states. We are going to do that one step look ahead using Bellman expectation equation to figure out a new value function, iterate this process many times for all states, and at the end we'll end up with the true value function for the given policy.
- See the next example with value iteration (Assuming follows uniform random policy for all actions equals 0.25).



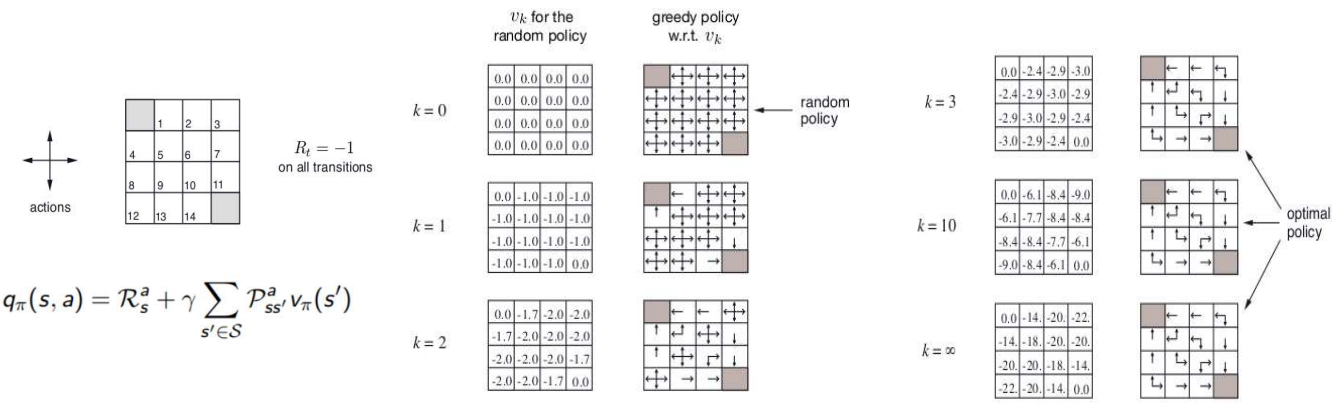
MDP solving - Value Iteration Example

- In this Grid World, we get a reward of as shown in the figure for each transition we make (actions we take). And the actions that we can take are **north, south, east and west** .
- Our agent is following some random policy π **with a weight of 0.5, 0.3, 0.1 and 0.1 for moving to north, south, east and west directions, respectively**.

We need to calculate the policy for four actions, **up, left, right and down with the center node with a reward of 1**

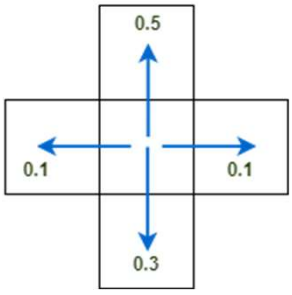


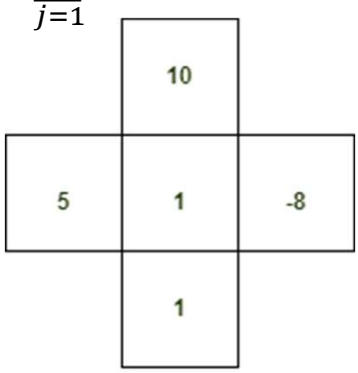
- In this Grid World, we get a reward of as shown in the figure for each transition we make (actions we take). And the actions that we can take are **north, south, east and west** .
- Our agent is following some random policy π **with a weight of 0.5, 0.3, 0.1 and 0.1 for moving to north, south, east and west directions, respectively**.
- **We need to calculate the policy for four actions, up, left, right and down with the center node with a reward of 1**



MDP solving – Value Iteration Example

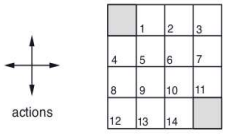
$$\pi_{k+1}(s_i) = r_i + \arg max_a\{\gamma \sum_{j=1}^{N} P_{ij}^a V^{\pi_n}(s_j)\}$$





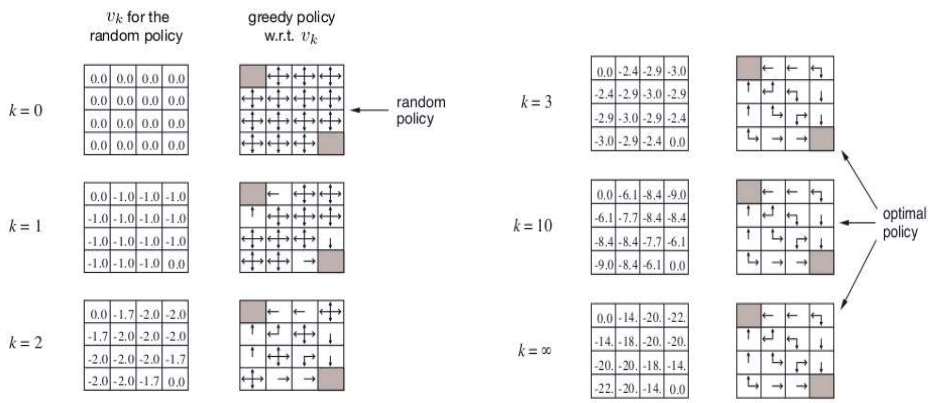
state space and reword

- See the next example with value iteration (Assuming follows uniform random policy for all the four actions: **up, left, right and down**).



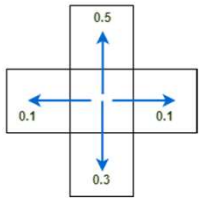
$R_t = -1$ on all transitions

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$

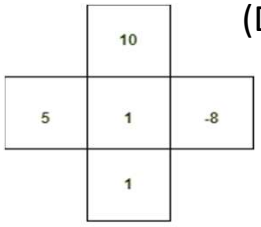


MDP solving – Value Iteration Example

$$\pi_{k+1}(s_i) = r_i + \arg max_a\{\gamma \sum_{j=1}^{N} P_{ij}^a V^{\pi_n}(s_j)\}$$



(left←) 0.5*5+0.1*10+0.1*1+0.3*-8 = 1.2
(up ↑) 0.5*10+0.1*5+0.1*-8+0.3*1 = 5.0
(right→) 0.5*-8+0.1*10+0.1*1+0.3*5 = -1.4
(Down ↓) 0.5*1+0.1*5+0.1*-8+0.3*10 = 3.2



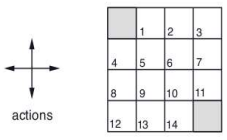
state space and reword

$\pi_{1=1+\max\{1.2,5.0,-1.4,3.2\}=6.0}$ ↑

MESIIA – MIA

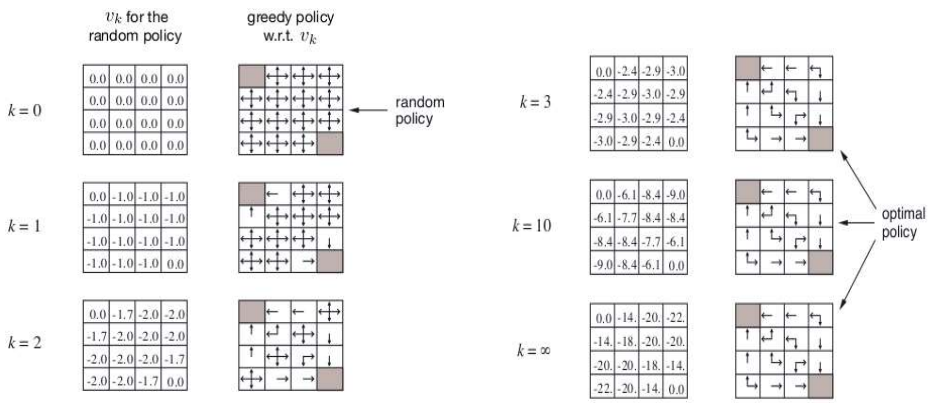
3
1

- We have to use the Bellman equation to calculate the utility V for the centre node with the four actions and then find the maximum value to update the reward of the node and the policy.
- The policy will be go to UP with a reward of 6.0.



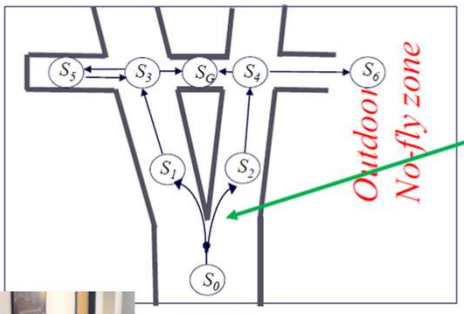
$R_t = -1$ on all transitions

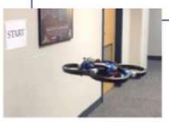
$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$



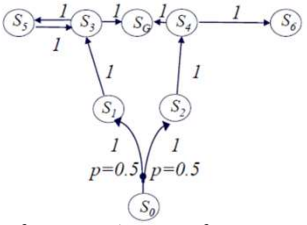
MDP vs. POMDP

- Consider a path planning example
- Assume **imperfect action execution** and full knowledge of the state (i.e., perfect localization)





Let's assume
50% chance of ending up on the left and
50% ending up on the right



MDP:

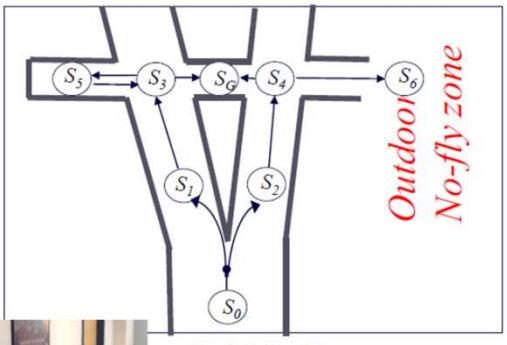
Defined as {S, A, T, C}, where S –set of states, A –set of actions, ***T(s,a,s') -Prob(s'|s, a)****, C –costs of all (s,a) pairs*

In a standard MDP, the state captures all relevant variables. There is nothing that the agent is not aware of. The agent might be unsure about the transition function or the reward function, but the agent always knows everything there is to know about the current state of the entire system. As shown in the figure above, all possibilities are provided to the system.

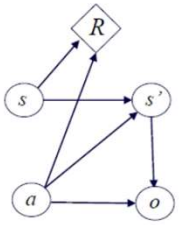
Example: Boiling water.
State 1: Water is at 23 C
Action 1: Agent sets stove top at 200 C
Reward (30 seconds after, when water started to boil): +1 for the fact that water boiled in the end, but -0.1 reward for each second it took for the water to start boiling. So total reward was -1.9 (-2.9 because the water did not boil for 29 seconds, then +1 for water boiling on 30th second)

MDP vs. POMDP

- Consider a path planning example
- Assume imperfect action execution and **partial observability of the state**(i.e., **imperfect localization**)



Causal relationship



POMDP: *{S, A, T, R, Ω, O}, where S, A, T(s,a,s'), R(s,a) – all as in MDP, Ω – set of all possible observation vectors o,* ***O(s',a,o) – Prob(o\|s',a) probability of seeing o after executing action a and ending up at state s'***

In a partially observable MDP, the agent has severe restrictions. There is an underlying MDP, but the agent does not know exactly what state the environment is currently in. All the agent gets is an *observation*. Typically, the state is described by several variables and the agent only observes a limited set of these variables, but there can be several other situations: the agent has noisy information, the agent has to pay to make observations, …

Dealing with a POMDP is very hard. You could try to use regular Q-learning using the observations instead of states, but this is not guaranteed to give a good policy (the values the algorithm converges to depend on the exploration policy). Other approaches use some form of memory (using the last n observations and actions as "state") but these require a lot more samples than regular MDP.

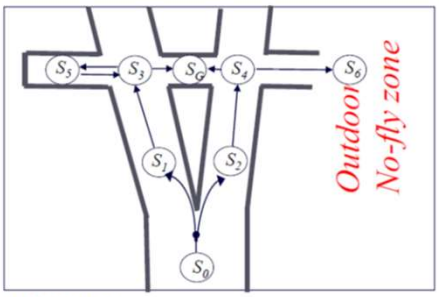
This system is described by a **partially observable Markov decision process (POMDP).** This is a computationally intensive problem to solve (P-SPACE-complete for a finite horizon). Partially observable because the agent only has access to limited amount of information about the current state of the system.

The agent does not have all information regarding the current state, and has only an observation, which is a subset of all the information in a given state. Therefore, it is impossible for the agent to truly behave optimally because of lack of information. One way to solve this is to use belief states to try to remember previous states to make better judgements on future actions (i.e., we may need states from the previous 10 time steps to know exactly what's going on currently).

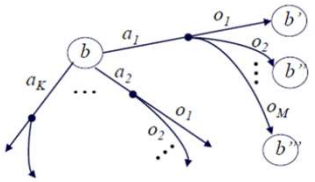
Example: We are in a room that is pitch black. At any time instant, we do not know exactly where we are, and if we only take our current state, we would have no idea. However, if we remember that we took 3 steps forward already, we have a much better idea of where we are.

Belief State Space

- **Belief state *b***: Probability distribution over the states the robot believes it is currently in



Belief State Space
(for K actions, M possible observations)



POMDP: *{S, A, T, R, Ω, O}, where T(s,a,s') = P(s'|s,a), R(s,a), O(s',a,o) = Prob(o|s',a)*

It is MDP!
We just need to compute transition probabilities
$\tau(b,a,b') = P(b'|b,a)$ and reward function $\rho(b,a)$

What is Belief State Space?

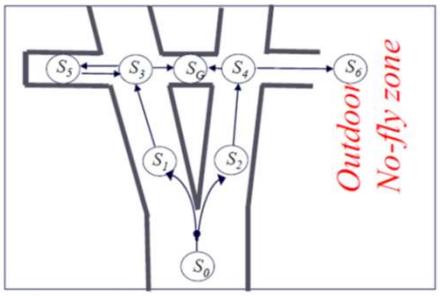
Another big issue is partial observability. MDPs assume that any crazy thing can happen to you, but when it does, you'll know exactly what state you're in. This is an incredibly strong assumption that only very rarely holds. Generally, you can just see a small part of the whole state of the world, and you don't necessarily even observe that reliably.

In order to make an informed decisions based on this partial data, the agent has to maintain **a list of belief-states** in which it could be at all time, **with their respective probability given the sequence of choice and observations.**

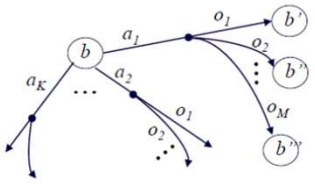
In terms of **learning,** the POMDP can use the same framework than in classic Reinforcement learning algorithms like Q-learning, where at each action or after a series of action you get a reward, but using belief-states instead of states. If you were learning "on-line" as the robot evolves the reward would have to be propagated to all the beliefs states that could have led to the reward. With enough trials, the agent learns not only the transition probabilities for each action but also the noise in its instruments.

Belief State Space

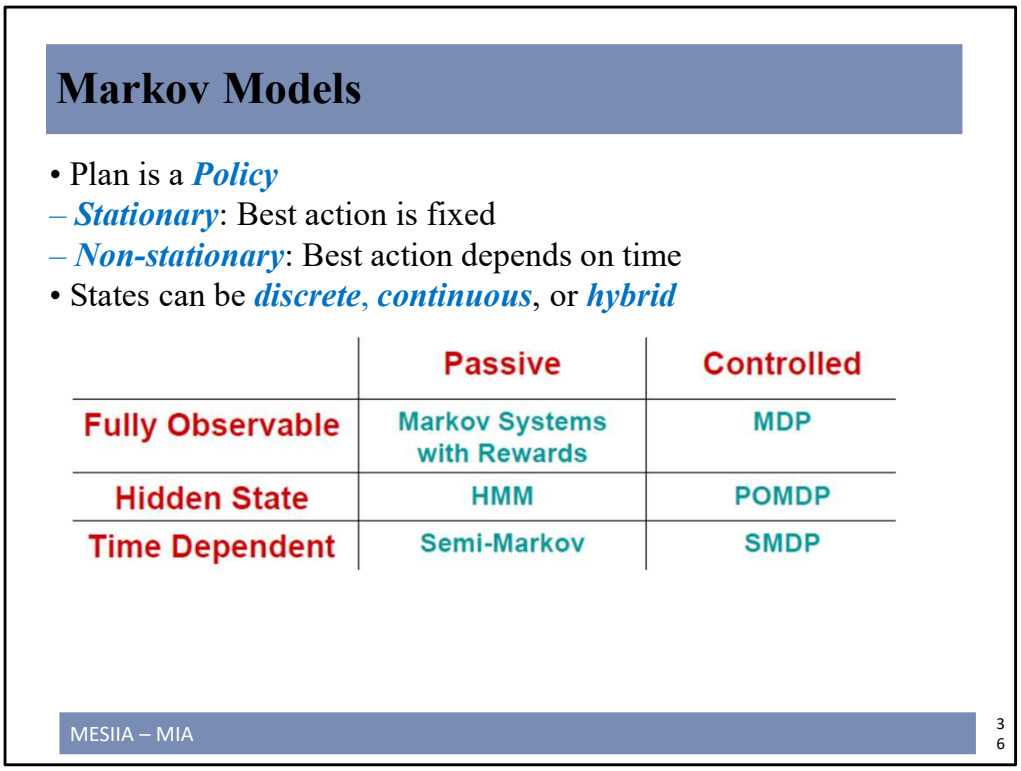
- **Belief state *b***: Probability distribution over the states the robot believes it is currently in
- Popular techniques for solving POMDPs
  - –by discretizing belief state space into a finite # of states [Lovejoy, '91]
  - –by taking advantage of the geometric nature of value function [Kaelbing, '98]
  - –by sampling-based approximations [Pineau, Gordon & Thrun, '03]



Belief State Space
(for K actions, M possible observations)



POMDP: *{S, A, T, R, Ω, O}, where T(s,a,s') = P(s'|s,a), R(s,a), O(s',a,o) = Prob(o|s',a)*



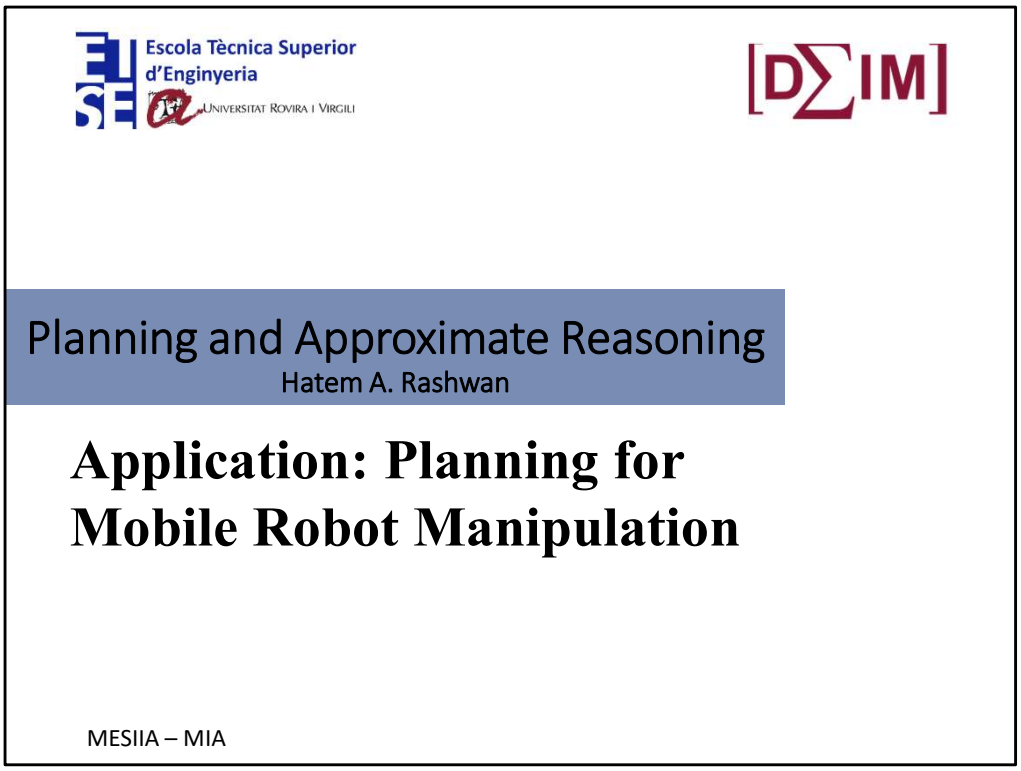
A **Markov model** is a stochastic model used to model randomly changing systems where it is assumed that **future states depend only on the current state not on the events that occurred before it** (that is, it assumes the Markov property). Generally, this assumption enables reasoning and computation with the model that would otherwise be intractable.

There are six common Markov models used in different situations, depending on whether every sequential state is observable or not, and whether the system is to be adjusted on the basis of observations made.

	System state is fully observable	System state is partially observable
System is autonomous	Markov chain (Markov system with reward)	Hidden Markov odel (HMM)
System is controlled	Markov decision process (MDP)	Partially observable Markov decision process (POMDP)

Summary

- MDP generalizes Graph representation
- POMDP generalizes MDP representation
- POMDP –representation of problems where the state of relevant variables is NOT fully known
- Solving POMDP can be represented as solving a Belief MDP (whose size is infinite though)
- Approximation techniques exist but intractability is still a huge issue for using POMDP planning in real world



Robotic manipulation?

People and animals can effectively manipulate objects of many shapes, sizes, weights, and materials using a variety of primitives such as grasping, pushing, sliding, tipping, rolling, and throwing. In contrast, most robots manipulate objects by pick-and-place. There is good reason for this: once a firm grasp is established, the robot can reliably control the motion of the part without needing to continuously sense the state of the part or correct for modeling uncertainties. Most manipulation primitives mentioned above are more sensitive to uncertainties in part state, geometry, mass, friction, and restitution, and to the robot's own control errors. Nonetheless, restricting robots to only grasp objects artificially limits the set of tasks that they can accomplish. Leveraging a larger set of manipulation primitives is crucial for robots to reach their full potential in industrial automation, exploration, home care, military, and space applications.

Different methods and planning used for manipulation, such as:

In-hand Sliding Manipulation

This work focuses on planning manipulation tasks that involve an object sliding in a manipulator's grasp. Sliding can allow for error-corrective motions when performing robot assembly tasks, which can decrease the chance of jamming and improve the robustness of planned motions.

Hybrid Manipulation Planning and Control

This work focuses on motion planning and control for robotic manipulation tasks in which the manipulator, object, and the environment transition between different contact modes.

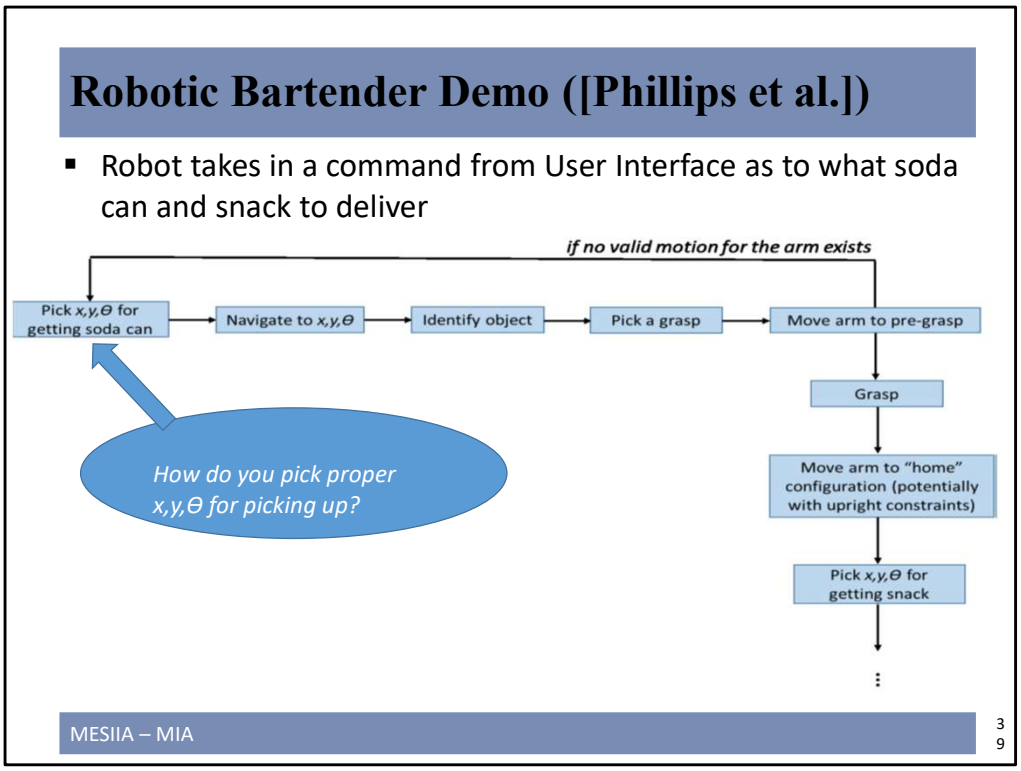
Vibratory Manipulation

This project examines a very simple and versatile robotic manipulator with surprising capabilities: a six-degree-of-freedom (6-dof) rigid vibrating plate whose motion can be programmed.

Rolling Manipulation

A typical manipulation plan consists of a sequence of manipulation primitives chosen from a library of

primitives, with each primitive equipped with its own feedback controller.



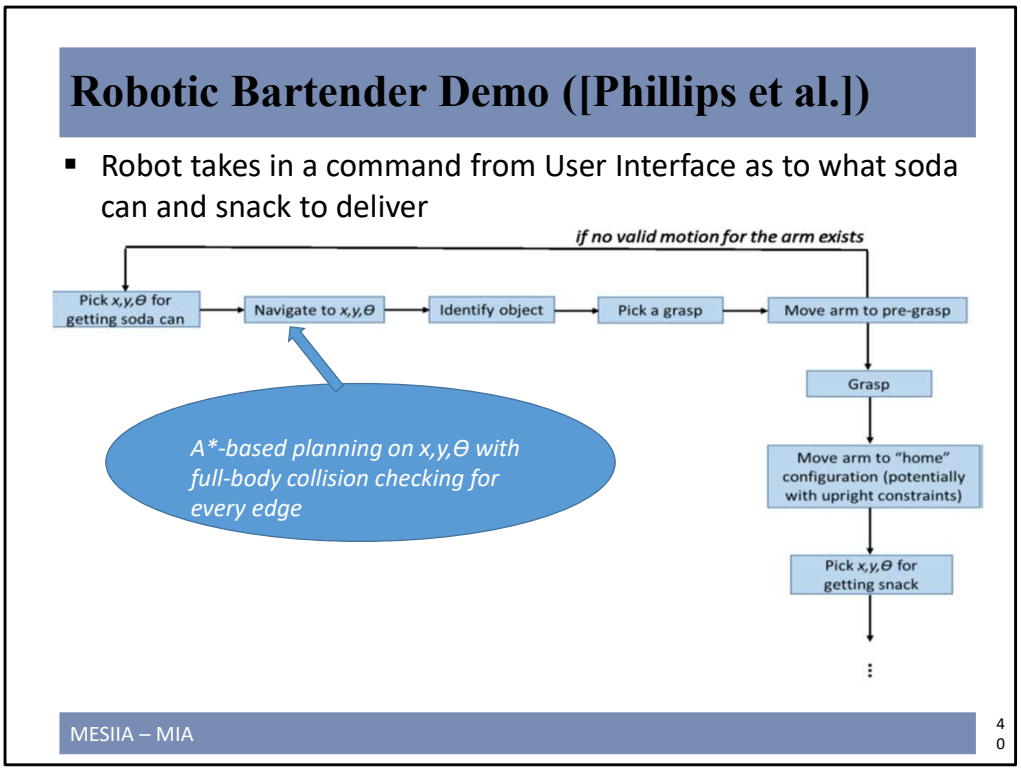
There are different applications for robotic manipulation.
One of the most important application is for mobile robot that can take order, navigate to reach a stationary object and then grasp and pick the target object.

To catch an object, the child first reaches the object and then he tries to catch it. Similarly, the robot has two separate subsystems responsible for fetching a stationary object: one for how the robot is approaching the target and another for how it is grasping it. Thus, any mobile robot has to be capable of answering two pragmatic principles:

For **an approaching system**: a) how can the robot reach the object? which objects should the system grasp? b) How can the robot identify them? c) How can the robot estimate the 3D pose of these objects? Here you need to give your order to the robot to pick an object defined **in x,y and Θ**.
For **a grasping system**: a) what are the hazards related to that object (hot or cold, or it is a glass)? b) When can the system command the arm to interact with the object? c) Which suitable grasping method can be used to grip it?

See the next video as an example of robot works as bartender,
https://www.youtube.com/watch?v=lpNbt7btxgg

Another example of bartender robotic system https://www.youtube.com/watch?v=EivBxP3kqWs



A motion planning algorithm is said to be complete if the planner determines in finite time either a solution or correctly reports that there is none. Most complete algorithms are geometry-based.
Resolution completeness is the property that the planner is guaranteed to find a path if the resolution of an underlying grid is fine enough. Most resolution complete planners are grid-based or interval-based.

Probabilistic completeness states that the planner fails to find a path (if one exists) asymptotically approaches zero. The performance of a probabilistically complete planner is measured by the rate of convergence.

Incomplete planners do not always produce a feasible path when one exists.

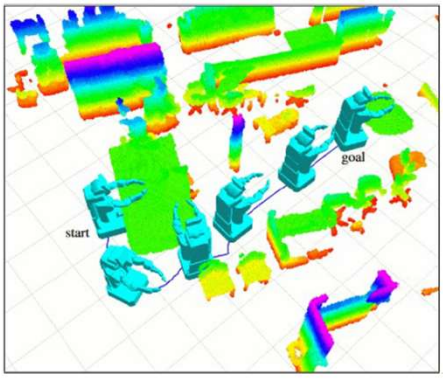
The performance of a complete planner is assessed by its computational complexity.

In this example, efficient and complete motion planning of a three wheeled mobile robot (**x,y and Θ**) by implementing an algorithm (the A∗ search algorithm ARA) for a path planning with the real Pioneer 2DX robot.

The A ∗ algorithm tries to combine the advantages offered by the Dijkstra algorithm and the **Breadth-first search** (BFS) algorithm. The A∗ algorithm tries during each new movement to take the shortest step and tries to determine if the step lies on the direction from source to target. The disadvantage of the A∗ algorithm is that it uses large amounts of memory in order to store the path planning environment.

Graph for Navigation with Complex 3D Body [Hornung et al., '12]

- 3D (x,y,θ) lattice-based graph representation for full-body collision checking
  - –takes set of motion primitives as input
  - –takes *N*footprints of the robot defined as polygons as input
  - –each footprint corresponds to the projection of a part of the body onto x,y plane
  - –collision checking/cost computation is done for each footprint at the corresponding projection of the 3D map



MESIIA – MIA

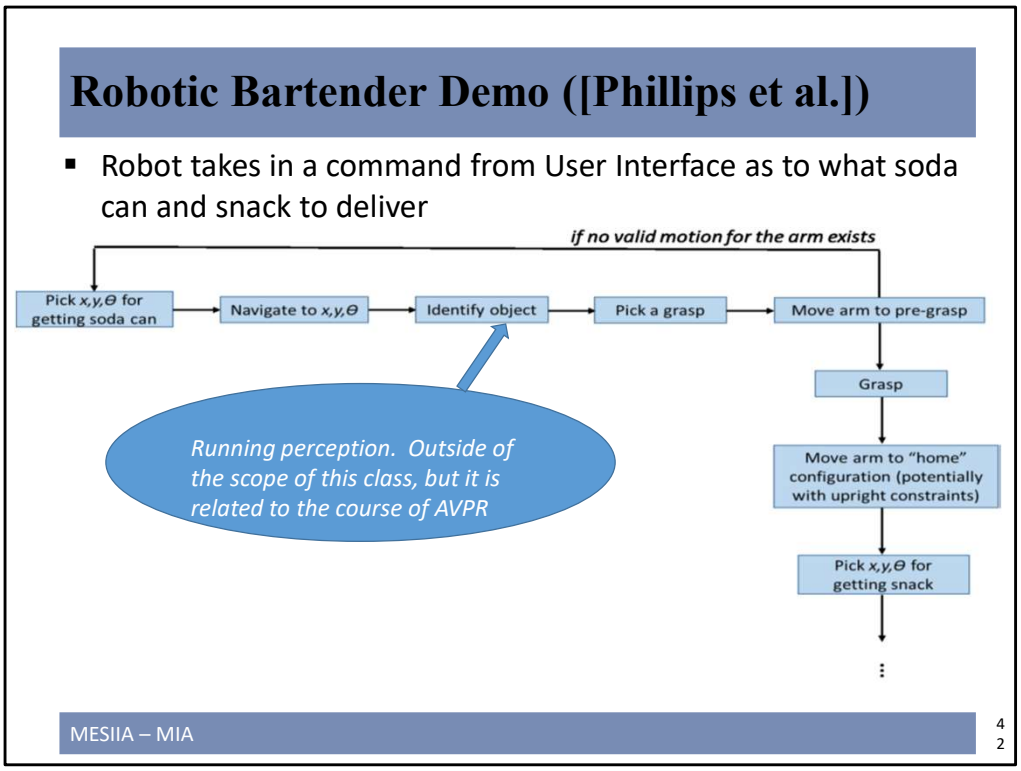
41

An example of the navigation of mobile robot that "**Navigation in three-dimensional cluttered environments for mobile manipulation" proposed by Hornung 2012**. In this work, collision-free navigation in cluttered environments is essential for any mobile manipulation system. This approach prevents navigation close to objects in situations where projected 3D configurations are in collision within the 2D grid map even if actually no collision occurs in the 3D environment. Accordingly, when using such a 2D for planning paths of a mobile manipulation robot, the number of planning problems which can be solved is limited and suboptimal robot paths may result. They utilized a combination of multi-layered 2D and 3D representations to improve planning speed, allowing the generation of almost real-time plans with bounded sub-optimality.

The robot footprint model approximates the robots' 2D contour for optimization purposes. The model is crucial for the complexity of distance calculations and hence for the computation time. Therefore, the robot footprint model constitutes a dedicated parameter instead of loading the footprint from the common 2D cost-map parameters. The optimization footprint model might differ from the cost-map footprint model. All different types are described in the following paragraphs:

"Obstacle Avoidance and Robot Footprint Model"

1. Footprint Point: The robot is modelled as a single point. For this type the least computation time is required.
2. Footprint Circular: The robot is modelled as a simple circle with a given radius.
3. Footprint Line: The line robot is useful for robots that exhibit different expansions/lengths in the longitudinal and lateral directions.
4. Footprint Two Circles: Another possibility to approximate the robot's contour consists of defining two circles. Each circle is described by an offset along the robots' x-axis and a radius.
5. Footprint Polygon: A complex model can be incorporated by defining a closed polygon. The polygon is defined in terms of a list of vertices (provide x and y coordinates for each vertex). The robots axis of rotation is assumed to be located at [0,0].

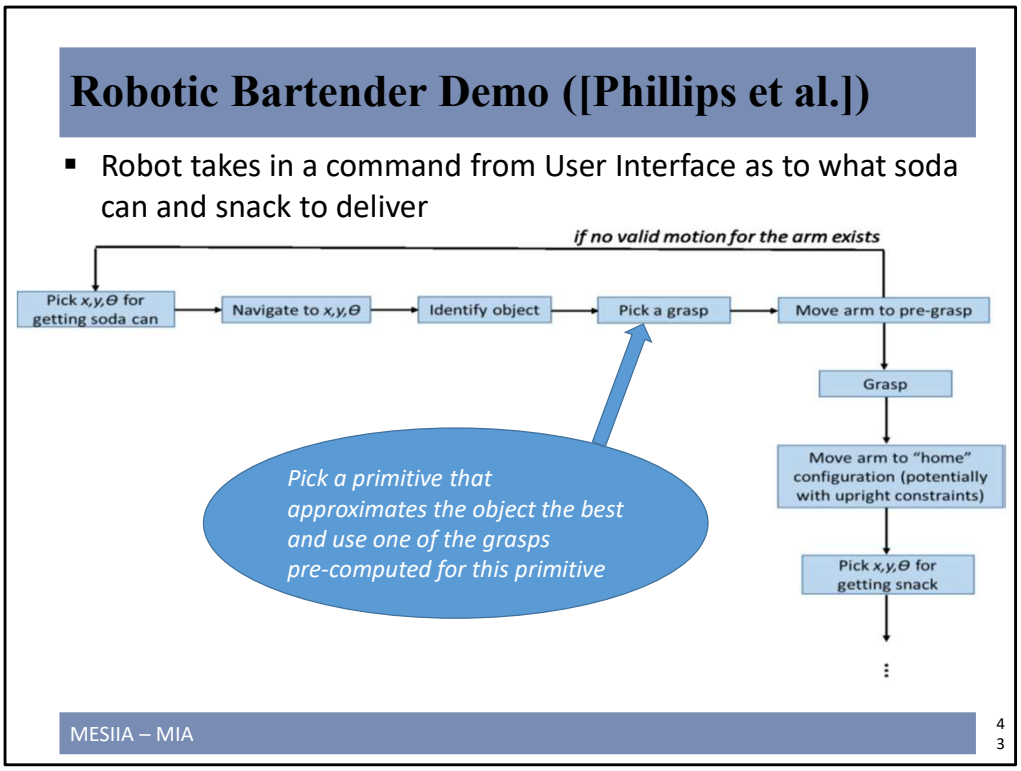


How can robots perceive the world and their own movements so that they accomplish navigation and manipulation tasks? In this module, you need to study how images and videos acquired by cameras mounted on robots are transformed into representations like features and optical flow. Such 2D representations allow us then to extract 3D information about where the camera is and in which direction the robot moves. You will come to understand how grasping objects is facilitated by the computation of 3D posing of objects and navigation can be accomplished by visual odometry and landmark-based localization.

Robotic perception is related to many applications in robotics where sensory data and artificial intelligence/machine learning (AI/ML) techniques are involved. Examples of such applications are object detection, environment representation, scene understanding, human/pedestrian detection, activity recognition, semantic place classification, object modelling, among others.

Robotic perception encompasses the ML algorithms and techniques that empower robots to learn from sensory data (i.e., cameras, RGB, RGB-D cameras) and, based on learned models, to react and take decisions accordingly.

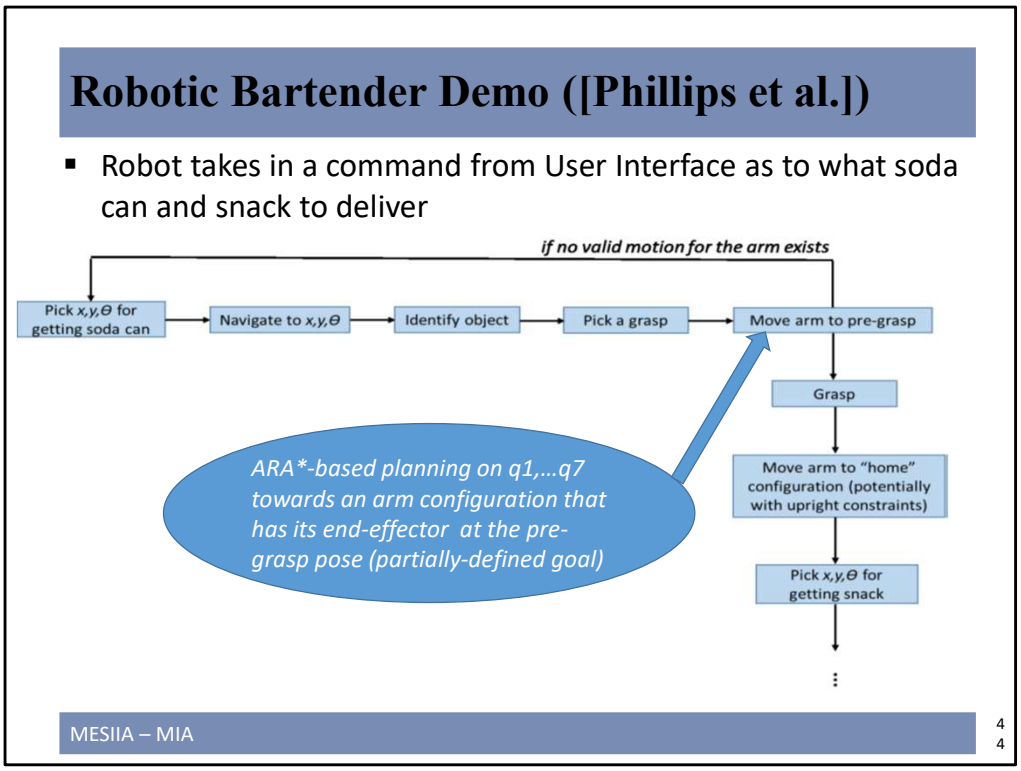
The recent developments in machine learning, namely deep-learning approaches, are evident and, consequently, robotic perception systems are evolving in a way that new applications and tasks are becoming a reality.



In order to perform grasping and manipulation tasks in the unstructured environments of the real world, a robot must be able to compute grasps for the almost unlimited number of objects it might encounter. In addition, it needs to be able to act in dynamic environments, whether that be changes in the robot's workspace, noise and errors in perception, inaccuracies in the robot's control, or perturbations to the robot itself.

To grasp the objects, these objects could be known or unknown. For known objects, each object is represented by a model consisting of a set of points with corresponding surface normal. A perception method simultaneously recognizes multiple model instances and estimates their pose in the scene. A variety of tests shows that the proposed method performs well on noisy, cluttered and unsegmented range scans in which only small parts of the objects are visible.

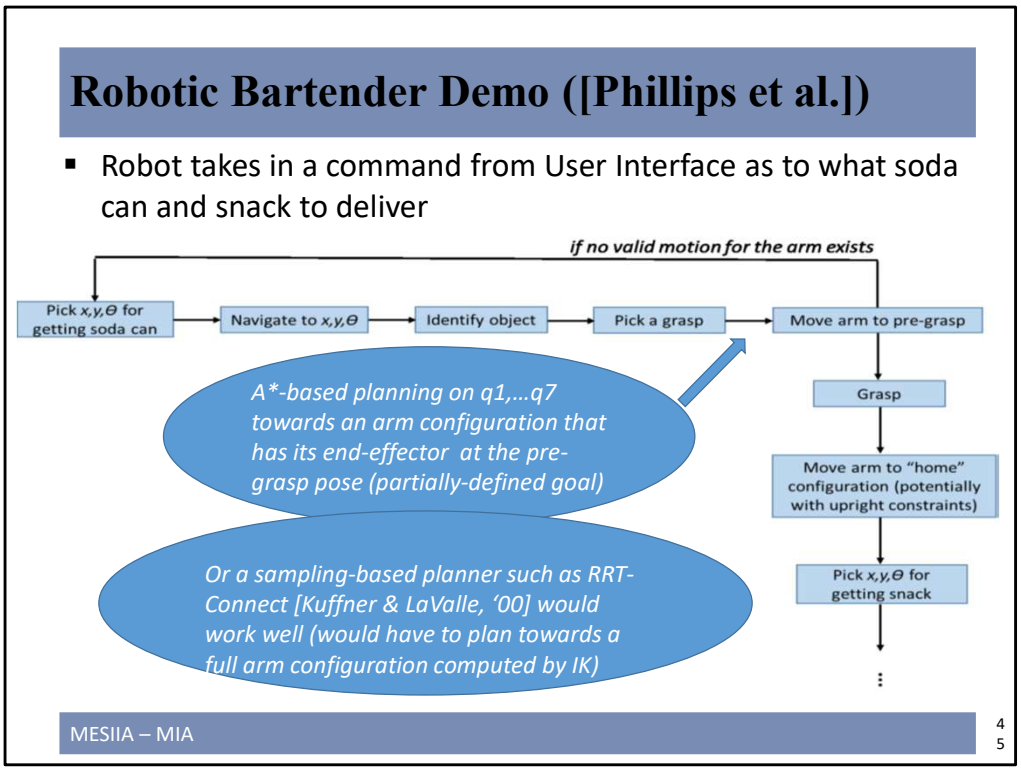
For unknown objects, reinforcement learning can take a rule in the grasping problem. This can be implemented by a two-stage model of grasping: stage one is an orientation of the hand and wrist and a ballistic reach toward the object; stage two is hand pre-shaping and adjustment. Visual features are first extracted from the unmodeled object. These features and their relations are used by an expert system (e.g., RL) to generate a set of valid reach/grasps for the object. These grasps are then used in driving the robot hand and arm to bring the fingers into contact with the object in the desired configuration.



Planning for the base is done using elastic bands to dynamically avoid unknown obstacles, e.g. humans walking around, in addition to known obstacles like walls or tables that have been modelled by hand. In contrast, the planner for the robot arm currently neither uses sensor data to detect unknown obstacles nor is fast enough to be able to react to a dynamic environment. Planning for the robot base (3 DOF) and the arm (7 DOF) are currently independent of each other. While the arm is moving the base is not, and otherwise round.

It only checks for collision with a hand-crafted static model. No sensors are used to detect unknown obstacles. The planning process takes about 2 seconds. The collision checker uses bounding boxes of the robot and the environment. The current path planners only returns whether the given configuration is in collision but no information on the distance to the closest obstacle. Checking one robot configuration for collision takes about 0.3 milliseconds in this example.

The currently used path planner for the robot arm ignores dynamic obstacles, and even if they would be included in the planning process, the planner is too slow to react to a changing environment. The main problem of researchers nowadays to be solved here is that of developing a motion planner for the robot arm that is fast enough to react to a dynamic environment and, thus, is able to adapt the movement of the arm online while it is moving



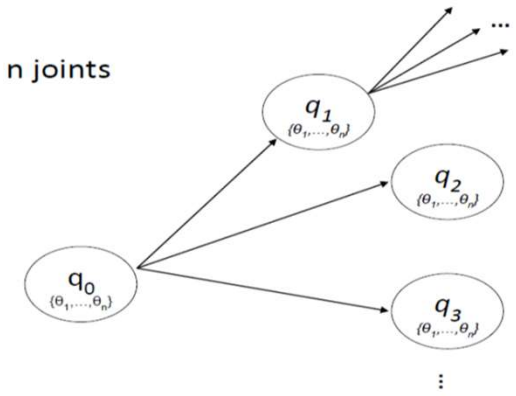
Sampling based planning (SBP) is unique in the fact that planning occurs by sampling the configuration space (C-space). In a sense SBP attempt to capture the connectivity of the C-space by sampling it. This arbitrary approach has its advantages in terms of providing fast solutions for difficult problems. The downside is that the solutions are widely regarded as suboptimal. Sampling based planners are not guaranteed to find a solution if one exists, a property that is referred to as *completeness*. They ensure a weaker notion of completeness that is *probabilistic completeness*. A solution will be provided, if one exists, given sufficient runtime of the algorithm (in some cases infinite runtime).

Perhaps the most commonly used algorithms are Probabilistic Roadmap Method (PRM), and Rapidly-exploring Random Trees (RRT). The intuitive implementation of both RRT and PRM, and the quality of the solutions, lead to their widespread adoption in robotics and many other fields. RRT represents a category of sampling based planners, which are single-query planners. A tree is incrementally grown from the start configuration to the goal configuration, or vice versa.

A configuration is randomly selected in the configuration space. If it lies in the free space, a connection is attempted to the nearest vertex in the tree. For single query problems, RRT is faster compared to PRM. It does not need to sample the configuration space and construct a roadmap i.e. learning phase. RRT was shown to be probabilistically complete

Manipulation Lattice for Arm Planning [Cohen et al., '13]

•Representation
–ex. Single arm with n joints {θ1,...,θn}



MESIIA – MIA

46

In this paper proposed by Cohen et al., 2013, the authors presented a heuristic search-based approach to motion planning for manipulation that does deal effectively with the high dimensionality of the problem. This approach achieves the necessary efficiency by exploiting the following three key principles:

(a) representation of the planning problem with what we call a manipulation lattice graph;
(b) use of the ARA* search which is an anytime heuristic search with provable bounds on solution sub-optimality; and
(c) use of informative yet fast-to-compute heuristics.

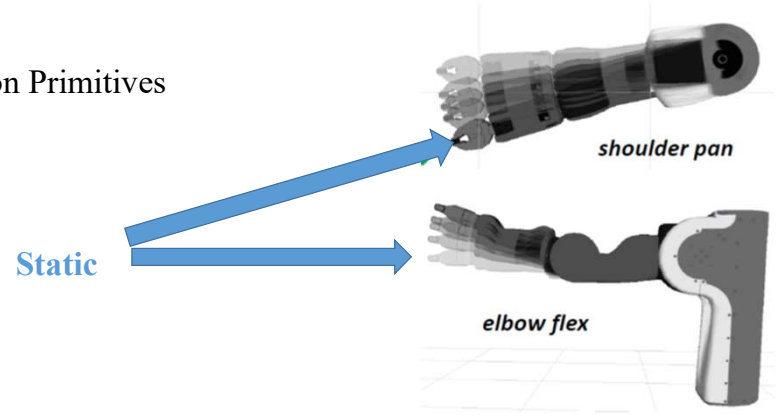
The paper proposed by Cohen et al., 13 showed how to apply it to single-arm and dual-arm motion planning with upright constraints on a PR2 robot operating in non-trivial cluttered spaces. This approach is comparable with other most common sampling-based approaches despite the high dimensionality of the problems. Each arm is defined with n joints. Every joint motion leads to the motion of the next joint.

In this paper, the authors presented a method of planning for single-arm manipulation in which the configuration space of the arm in joint space is represented. Thus, if you need to plan motions for a robot arm with seven joints, it would result in a graph with seven

dimensions.

Manipulation Lattice for Arm Planning [Cohen et al., '13]

- Representation
  - –ex. Single arm with n joints {θ1,...,θn}
- Motion Primitives



In this paper, the authors construct an n dimensional state space when planning for an arm with n joints. Each state is represented by an n element vector whose elements correlate to actual joint positions. Each motion primitive is an n element vector of velocities.

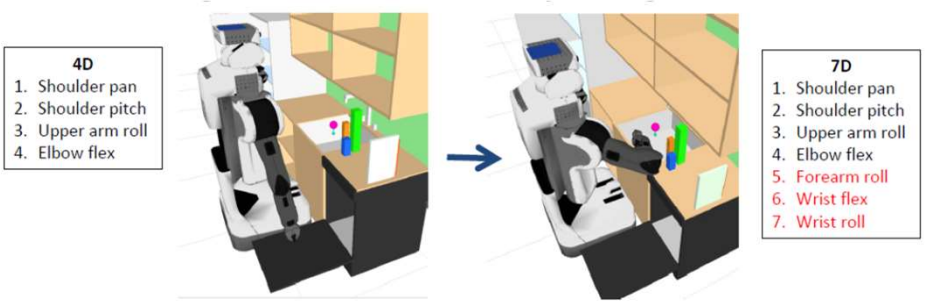
This representation was designed to specifically handle the complexities of manipulation through the use of static and adaptive motion primitives and by decoupling the problem when appropriate by varying the dimensionality of the lattice.

They define a state s as an n-tuple (θ0, θ1, θ2, ..., θn) for a manipulator with n joints. It is important to note that the graph is constructed dynamically by the graph search as it expands states since pre-allocation of memory for the entire graph would be infeasible for an n DOF manipulator with any reasonable n. Each motion primitive is a single vector of joint velocities, (v0, v1, v2, ..., vn) for all of the joints in the manipulator. The set of primitives is the set of the smallest possible motions that can be performed at any given state.

They use a pre-defined set of actions that can be performed at any given state, as static motion primitives. These actions are chosen before the search begins and their purpose is to uniformly explore the space for a valid path. Given that this set is the majority of all motion primitives used, it has a major impact on the branching factor of the graph search.

Manipulation Lattice for Arm Planning [Cohen et al., '13]

- Non-uniform Dimensionality
  - –far from goal: only 4 DoFs active
  - –around goal: all 7 DoFs are active
- Non-uniform Resolution
  - –far from goal: larger discretization of joint angles
  - –around goal: finer discretization of joint angles



As you see in the figure, the planning motion is different when arm is before approaching the object and then after approaching it. Far from the target (goal), only 4 DoF will be adaptive as shown in the figure. In turn, around the goal, 7 DoF will be active. Also for the resolution of motion will be larger discretization when the arm is far from the target, however it will be finer when the arm is around the target to get more accurate position.

Summary

- Multiple planners used for both domains
- Start and goal configurations are often most constrained –can be exploited by the planners
- Planning is higher-dimensional but can take longer than on ground and aerial vehicles
- Design of proper heuristics is a key to efficiency

End