

# Lecture 2: Formal Models of AI Problems and Search

Viliam Lisý & **Branislav Bošanský**

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

[bosansky@fel.cvut.cz](mailto:bosansky@fel.cvut.cz)

February, 2021

# Small Steps to Great Results



How to get to the big result? What are the core AI methods?

AI methods solve difficult problems.

How can we formalize a problem (and a solution) so that an algorithm can find this solution (in an optimal way)?

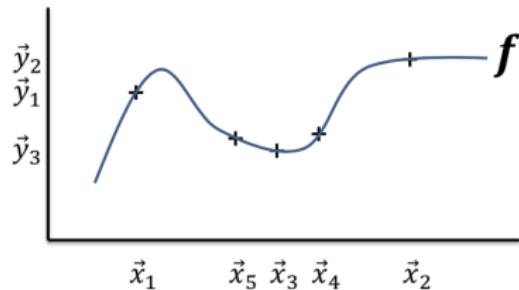
# Solving a Great AI Challenge?

... we create a matrix representation of features and train a deep neural network that will solve the problem. Done!

Is it though? Supervised learning = fitting a (high dimensional) function

For a data set  $(\vec{x}_i, \vec{y}_i)$ , find a function  $f$  that minimizes:

$$\frac{1}{n} \sum_i \|f(\vec{x}_i) - \vec{y}_i\|.$$



ML can give us quick estimates / suggestions that can be insufficient on their own.

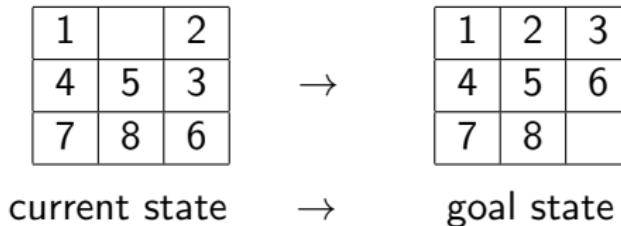
Often, the best solution of a problem comes when combining multiple AI techniques (e.g., for Alpha GO - search-based algorithm and ML-based heuristics).

We will define key building blocks of many AI methods. Let us consider any problem (scenario) we want to solve:

- determine formal representation of every possible situation in the scenario – the **states** of the problem (denoted  $S$ )
- determine how the states can be changed by the algorithm (agent) – the **actions** in the scenario (denoted  $A$ )

By applying an action  $a \in A$  to a state  $s \in S$ , the state will change to a different state  $s' \in S$ .

## Example 1 – 8 puzzle



Goal: rearrange the numbers by moving the empty square to adjacent squares so that they are ordered

Possible representations:

- values of tiles in a sequence  
 $s = [1, \_, 2, 4, 5, 3, 7, 8, 6]$
- position of numbers  
 $s = [1, 3, 6, 4, 5, 9, 7, 8]$

## Example 2 – Robotic arm



Find correct configurations of joints / parts of the arm so that the arm catches a desired object.

Possible representations:

- $s = [\theta_1, \alpha_1, \theta_2, \alpha_2, \dots]$

## Example 3 – Chess



Possible representations:

- positions of pieces on the board

$s = [[A1, B1, C1, \dots, H2], [A8, \dots, H7], \dots]$  → additional information needed besides the board itself (king has moved, rook has moved, repeated positions (!))

Alternatively, a **history of played moves** represents a state.

# Solution of a Problem

Many of the AI problems can be formulated as finding a sequence of actions that leads to a **goal state**.

We want to find the best such sequence

- minimize the number of actions
- every action can have some cost (or reward) associated with it  
→ minimization of total cost

We can reason about possible states / effects of actions  
(the rules of the environment are known (!)):

- we have a formal model
- (for the large scale) access to a simulator

# Main AI Models for (Sequential) Decision Making

There are several fundamental models when searching for optimal sequence of actions based on searching through state space (possibly uncertain effect of actions / stochastic environment):

- Markov Decision Processes (MDPs)
- Partially Observable Markov Decision Processes (POMDPs)
- (Imperfect Information) Extensive-Form Games (EFGs)
- (Partially Observable) Stochastic Games (POSGs)

We introduce selected general models now to emphasize the importance of the correct formalization of the problem. Some algorithms for solving them optimally will be introduced later but we will not be able to cover everything.

# Main AI Models for (Sequential) Decision Making

There are several fundamental models when searching for optimal sequence of actions based on searching through state space (possibly uncertain effect of actions / stochastic environment):

- Markov Decision Processes (MDPs) → perfectly observable environment, only 1 agent is acting
- Partially Observable Markov Decision Processes (POMDPs)  
→ **partially observable environment**, only 1 agent is acting
- (Imperfect Information) Extensive-Form Games (EFGs) → perfectly (partially) observable environment, finite horizon,  
***n agents can act (every agent optimizes own goal / utility)***
- (Partially Observable) Stochastic Games (POSGs) → perfectly (partially) observable environment, **infinite horizon**,  
***n agents can act (every agent optimizes own goal / utility)***

# Markov Decision Processes (MDPs)

Consider (finite) sets of states  $S$ , rewards  $R$ , and actions  $A$ . The agent interact with the environment in discrete steps

$t = 0, 1, 2, \dots$ , at each timestep the agent receives the current state  $S_t \in S$ , selects an action based on the state  $A_t \in A$ . As a consequence of taking the action, the agent receives a reward  $R_{t+1} \in R$  and find itself in a new state  $S_{t+1}$ .

Rewards and states are generated based on a *dynamics* of the MDP

$$p(s', r | s, a) \leftarrow \Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}$$

The next state depends only on the current state and the action (Markov property). **In the first lectures/labs, we assume that the environment is deterministic.**

Transition and reward dynamics can be defined separately.

# Markov Decision Processes (MDPs) – Example

#	#	#	#	#	#
#	<i>G</i>				#
#	#	#	#		#
#	↓	#	#		#
#					#
#	#	#	#	#	#

Consider a robot ( $\downarrow$ ) in a maze (# are walls), the arrow represents the direction the robot is facing,  $G$  is gold.

What are the states and actions?

- $s = (X, Y, d, G)$
- actions = (move\_forward, move\_backward, turn\_left, turn\_right)

MDP dynamics:

- $p((1, 1, \downarrow, \text{false}), 0 | (1, 2, \downarrow, \text{false}), \text{move\_forward}) = 1$
- $p((1, 1, \downarrow, \text{false}), 0 | (1, 2, \downarrow, \text{false}), \text{move\_backward}) = 0$
- ...

Why do we need some generic formal description?

- we will have a well-defined problem (inputs / outputs for the algorithm)
- formalization helps to think about the problem (e.g., formalizing the dynamics)
- we can reuse existing algorithms
- if we design and implement a brand-new algorithm for MDPs (POMDPs / EFGs / ...), we can solve (almost) all instances

# Partially Observable Markov Decision Processes (POMDPs)

States, actions, and rewards are as before, however, the agent cannot perfectly observe the current state.

The agent has a **belief** – a probability distribution over states that express the (subjective) likelihood about the current state. The agent receives **observations** from a finite set  $O$  that affect the belief. The agent starts from an **initial belief** and based on actions and observations, it updates its belief. Given the current belief  $b : S \rightarrow [0, 1]$  and some action  $a \in A$  and received observation  $o \in O$ , the new belief is defined as:

$$b(s') = \mu O(o|s', a) \cdot \sum_{s \in S} Pr(s'|s, a) \cdot b(s)$$

where  $\mu$  is a normalizing constant.

# POMDP – Example

#	#	#	#	#	#
#	<i>G</i>				#
#	#	#	#		#
#	↓	#	#		#
#					#
#	#	#	#	#	#

The robot can now perceive only its surroundings but does not know the exact position in the maze. States and actions remain the same.

- $s = (X, Y, d, G)$
- actions = (move\_forward, move\_backward, turn\_left, turn\_right)

Observations are all possible combinations of walls / free squares in the 4-neighborhood:

- $(\#, \#, \#, \#), (\#, \#, \#, -), \dots$

# Extensive-Form Games (EFGs)

Agent is not the only one that changes the environment. Every state has a player that acts in that state. EFGs are typically visualized as game-trees that:

- are finite (the game has some pre-defined horizon; note that (PO)MDPs do not have this!)
- node of the game tree corresponds to the history of actions from the beginning, edges are actions (as search trees)
- rewards (termed utilities) are defined only in terminal states (leafs of the game tree)
- agent can have imperfect information (certain states can be indistinguishable) → we will not be able to cover this in ZUI (→ A4M36MAS)

POSGs are a multi-agent extension of POMDPs → every agent can have their own actions, observations, and rewards. Every agent has its own belief (about the state, about beliefs of other agents, ...).

One of the most general formal model → algorithmically intractable in general.

# Solution of a Deterministic MDP

How can we find a solution of an MDP?

# Solution of a Deterministic MDP

How can we find a solution of an MDP?

Find the best sequence of actions leading to the goal → explore relevant states of an MDP and find the best action to be played in these states such that the trajectory (or a run)

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, \dots, S_k$$

maximizes the accumulated reward (and  $S_k$  is a goal state)<sup>1</sup>.

For now, the rewards are summed together (in case of stochastic transitions / POMDPs, a discounted sum is typically used with discount factor  $0 < \gamma < 1$ ).

---

<sup>1</sup>Maximization of rewards = minimization of costs (we will use both).

- ① Start from the initial state  $S_0$
- ② Apply actions and generate new possible states
- ③ If one of the generated states is the goal state → finish
- ④ If not, choose one state and go to step 2

Questions:

- Which state to choose to out of all generated new states in step 4?
- What if we generate a state that we have already explored?

## Q1

Which state to choose to out of all generated new states in step 4?

Goal is to find the best sequence of actions → we want to explore the ones with the highest rewards (lowest costs) first.

What if we make a mistake? → We keep a (sorted) list of reachable states that can be further explored – **open list** or **fringe**.

# Solving Deterministic MDPs – Variants of Uninformed Search

Variants of using the **fringe**:

- the fringe is sorted, new states to explore are taken from the beginning → **uniform-cost search**
- the fringe is unsorted, newly expanded states are inserted to the front, new states to explore are taken from the beginning → **depth first search (DFS)**
- the fringe is unsorted, newly expanded states are appended at the back, new states to explore are taken from the beginning → **breadth first search (BFS)**

BFS is complete, finds the shallowest solution (the sequence that requires the least number of actions while ignoring rewards). Requires exponential memory (and time).

# Solving Deterministic MDPs – Variants of Uninformed Search

DFS is not complete (the algorithm might not terminate) → we limit the maximal length of the sequence actions DFS can explore and iteratively increase this limit → **iterative deepening**.

**Uniform-cost search** is complete and optimal (in case all rewards are strictly negative). A variant of **Dijkstra's algorithm** (only the best path to a goal state not all states).

## Q2

What if we generate a state that we have already explored?

Using this algorithm, we are generating a **search tree**. Every node of the search tree corresponds to a state in the environment but multiple nodes can correspond to the same state.

We can maintain a **closed list**

# Iterative Deepening

Combining good characteristics of BFS and DFS. Let's have a limited-depth-dfs method:

- call limited-depth-dfs with depth limit 0,
- if unsuccessful, call limited-depth-dfs with depth limit 1,
- if unsuccessful, call limited-depth-dfs with depth limit 2,  
etc.

Complete, finds the shallowest solution, space requirements of a DFS. Counterintuitively, it is not that wasteful (timewise):

- the search tree grows exponentially → it is more time consuming to generate / evaluate all states in depth exactly  $d$  than repeatedly visiting states in the shallower depth

What if we want to optimize cost instead of number of actions? → limit the overall cost and increase by 1.

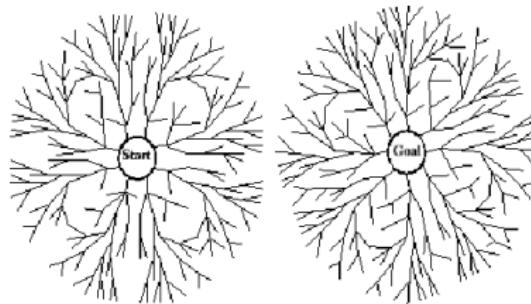
# Backward / Bidirectional Search

Do we need to search only from the initial state? → No.

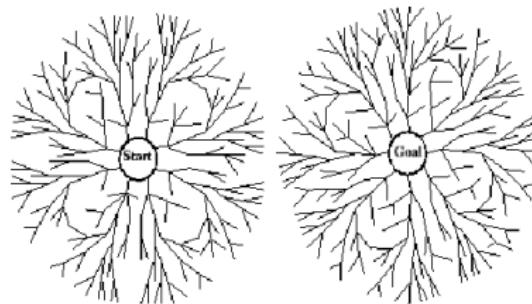
Sometimes, searching from the goal state to a starting state can be better:

- number of the actions that lead to the goal state is small (the problem is difficult at the beginning)
- we need to be able to effectively generate previous states

We can go even further → searching from the both sides.



# Bidirectional Search



It is tempting → searching from start / goal (e.g., in parallel (!)).

If the shallowest solution has depth  $d$ , we can expand only  $b^{d/2}$  nodes (where  $b$  is the branching factor (number of available actions)).

But what if the searches do not meet “in the middle”? → We'll see the next week.

# Lecture 3: Informed (Heuristic) Search

Viliam Lisý & Branislav Bošanský

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

[bosansky@fel.cvut.cz](mailto:bosansky@fel.cvut.cz)

February, 2021

# Search Algorithm

Last week – formalization and uninformed search, uniform-cost search → the state that accumulated the lowest cost is selected for the expansion

- ① Start from the initial state  $S_0$
- ② Apply actions and generate new possible states
- ③ If one of the generated states is the goal state → finish
- ④ If not, choose one state and go to step 2

## Question

For the uniform-cost search (when we optimize the costs rather the number of actions), can we terminate the search when the goal state is generated (added into the open list)?

NO → the algorithm can terminate only when the goal state is selected for expansion (it is the state with lowest costs)

# Improving Uniform-Cost Search

Last week – formalization and uninformed search, uniform-cost search → the state that accumulated the lowest cost is selected for the expansion

- ① Start from the initial state  $S_0$
- ② Apply actions and generate new possible states
- ③ Add new possible states into fringe
- ④ Select one new state from fringe
- ⑤ If it is not the goal state, go to step 2

The lowest accumulated cost does not necessarily mean that the state will be on the optimal path to the goal.

What is a good predictor of the future? → heuristics

## Heuristics

Estimate of the cost from the current state to the goal. Denoted  
 $h : S \rightarrow \mathbb{R}^+$

We can use the heuristics estimate to choose next states to expand.

Optimal heuristics  $h^*$  – the optimal cost from a state to goal.

In practice, we want to be as close to the optimal heuristics as possible. The estimate is typically a solution of a simplified problem.

We can order states in fringe according to the heuristic value.

How would that work? Will the algorithm always find an optimal solution?

# Using Heuristics

```
# # # # #
# G           #
# # # #       #
#   # #       #
#           @   #
# # # # # #
```

What can be a good heuristic for the problem of collecting the gold in a maze?

- Manhattan distance from the current position to the gold
- Euclidean distance
- ...

## Using Heuristics – Greedy Best-First Search

What if we use the Manhattan distance? The heuristic values are as follows:

#	#	#	#	#	#
#	G	1	2	3	#
#	#	#	#	4	#
#	2	#	#	5	#
#	3	@	5	6	#
#	#	#	#	#	#

Obviously, following the heuristics in a greedy manner will result in a suboptimal path.

If we do not maintain the closed list, the greedy best-first search algorithm will not terminate!

# Using Heuristics – Greedy Best-First Search

We know that uniform-cost search algorithm works here

#	#	#	#	#	#
#	G	7	6	5	#
#	#	#	#	4	#
#	2	#	#	3	#
#	1	@	1	2	#
#	#	#	#	#	#

Accumulated costs can help the algorithm to get out of parts of the state space that the heuristics incorrectly evaluates as promising.

What if we select the next state to expand based on the sum of accumulated costs and heuristic estimation?

# Using Heuristics (2)

#	#	#	#	#	#	#	#	#	#	#	#
#	<i>G</i>				#	#	<i>G</i>				
#	#	#	#		#	#	#		#		
#		#	#		#	#		2+2	#	#	
#	1+3	@	1+5		#	#	@	2+4	1+5		
#	#	#	#	#	#	#	#	#	#	#	

#	#	#	#	#	#	#	#	#	#	#	#
#	<i>G</i>				#	#	<i>G</i>				
#	#	#	#		#	#	#		#		
#	@	#	#		#	#		#	#		
#	3+3	2+4	1+5		#	#	3+3	2+4	@	2+6	#
#	#	#	#	#	#	#	#	#	#	#	

...

Let  $s \in S$  be the current state represented as a node  $n \in N$  in the search tree. Now,  $g : N \rightarrow \mathbb{R}^+$  is the cost accumulated on the path from the starting state to the current state  $s$  along the path in the search tree to node  $n$ . In the  $A^*$  algorithm, the choice of the next node to expand is the one that minimizes the function

$$f(n) = g(n) + h(n)$$

Do we have guarantees that such algorithm finds an optimal solution?

## Correct Use of Heuristics - Example

What if we have a heuristic function as follows?

#	#	#	#	#	#
#	G	0	0	0	#
#	30	#	#	0	#
#	20	#	#	0	#
#	10	@	0	0	#
#	#	#	#	#	#

The right-hand (and longer) path would be found first. → If we use an arbitrary heuristic function, the algorithm is not optimal.

How do we guarantee the optimality? → The algorithm has to use “meaningful heuristics”.

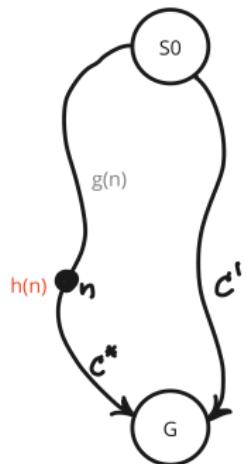
## Definition

A heuristic is **admissible** if it never overestimates the cost to the goal (the heuristic is always optimistic;  $h(n) \leq h^*(n) \forall n \in N$ ).

## Theorem

If the heuristic is **admissible**, A\* is always optimal (finds an optimal solution).

# Optimality of A\*



## proof

Let  $C^*$  be the cost of the optimal path to goal state  $G$  and let's assume that the  $A^*$  algorithm found a suboptimal path with cost  $C' > C^*$ .

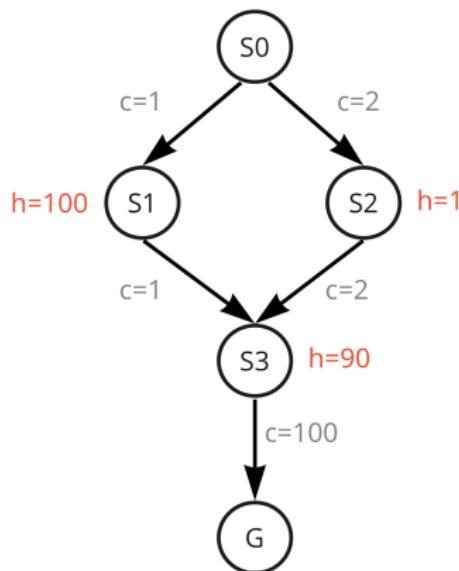
In the open list of the algorithm, there must be a node that is on the optimal path from the starting state to the goal (denote it  $n$ ).

$$f(n) = h(n) + g(n) \leq h^*(n) + g(n) \leq C^* < C'$$

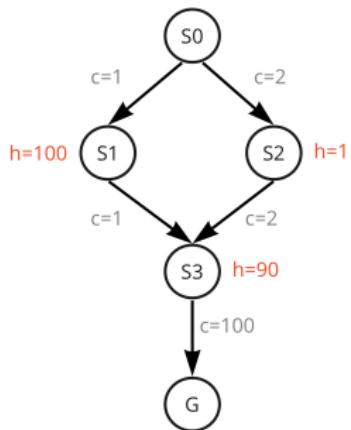
therefore, node  $n$  should have been selected for the expansion before goal state  $G$  reached via suboptimal path.

# Optimality of A\*

Is this enough? What if the algorithm reaches the same state multiple times ... can we discard the next visits or does the algorithm need to re-evaluate already closed states?



# Optimality of A\*



The algorithm explores:

- $s_2$ ;  $f(s_2) = 2 + 1 \rightarrow s_3$  added to fringe
- $s_3$ ;  $f(s_3) = 4 + 90 \rightarrow G$  added to fringe
- $s_1$ ;  $f(s_1) = 1 + 100 \rightarrow s_3?$

The algorithm cannot dismiss it as already solved since it found a better path to  $s_3$ . Otherwise, the algorithm would miss the optimal path  $s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow G$

It is safe to discard already explored state if the cost of the new path is greater or equal than the cost with which the state has been explored.

## Definition

Consider two nodes in the search tree  $n$  and  $n'$  such that  $n'$  is reached immediately by making an action in  $n$ . Denote  $c(n, n')$  the cost for this action. A heuristic is **consistent** if the following holds:

$$h(n) \leq h(n') + c(n, n')$$

$$\text{(or } g(n) + h(n) \leq g(n') + h(n')\text{)}.$$

Consistency is a stronger property than admissibility (every consistent heuristic is admissible).

Intuitions:

- similar to triangle inequality
- heuristic function is more informative deeper in the search tree

# Optimality of A\* – Efficiency

## Theorem

*Assume that the A\* algorithm uses a consistent heuristic. If a state is selected for exploration, the algorithm has already found an optimal path to this state.*

## Theorem

*A\* algorithm is **optimally efficient** in the sense no other optimal algorithm is guaranteed to expand fewer nodes than A\*.*

A\* expands all nodes in the search tree with

$f(n) = g(n) + h(n) < C^*$ . Any other algorithm has to explore these nodes as well, otherwise it can miss the optimal solution.

# Iterative Deepening A\* (IDA\*)

Despite the theoretical properties, memory requirements of the A\* algorithm can be still significant (note that  $h(n) = 0$  for all nodes  $n$  in the search tree is a consistent heuristic).

We can use **limited-cost A\*** with cost cutoff  $c$ , such that any node with  $g(n) + h(n) > c$  is not expanded. **IDA\*** gradually increases the cost cutoff.

Other variants:

- recursive best-first search (RBFS)
- (simplified) memory-bounded A\* (MA\* / SMA\*)

# Designing Heuristics

How should we design admissible heuristics?

- solve a simplified (relaxed) problem  
(e.g., there are no obstacles in a maze)
- solve only a single subproblem
- split into more subproblems
- ...

The heuristic function has to be informative (note that  $h(s) = 0$  for all states  $s \in S$  is an admissible heuristic but it is not very informative).

Consider two admissible heuristic functions  $h_1$  and  $h_2$  such that  $h_1(s) \geq h_2(s)$  for all states  $s \in S$ . We say that  $h_1$  dominates  $h_2$  and is more informative.

More informative heuristics expand fewer nodes in the search tree.

# Example Heuristics

Recall the 8-puzzle problem

1		2
4	5	3
7	8	6

What are the possible admissible heuristics?

- number of correctly placed tiles
- sum of Manhattan distances of tiles to their target location
- solving a subproblem

The diagram illustrates a state transition in a 3x3 grid. On the left, the initial state is shown with values 1, 4, and \* in the first row; 2, 3, and \* in the second row; and \*, \*, and \* in the third row. An arrow points to the right, leading to a final state where the values 1, 2, and 3 are in the first row; 4, \*, and \* are in the second row; and \*, \*, and an empty space are in the third row.

1		2
4	*	3
*	*	*

→

1	2	3
4	*	*
*	*	

all such combinations can be pre-computed and stored in a database

## Example Heuristics – Does it Matter?

YES

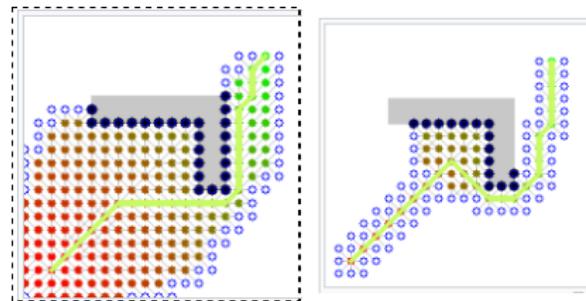
Compare the average number of expanded nodes for the number misplaced tiles ( $h_1$ ) and the sum of Manhattan distances ( $h_2$ ):

Sol. length	Iterative Deepening	$A^*(h_1)$	$A^*(h_2)$
8	6384	39	25
12	3644035	227	73
16	—	1301	211
20	—	7276	676

# Inadmissible Heuristics

In some cases, admissible heuristics can still expand too many nodes → the algorithm is slow, requires large memory (open / closed list).

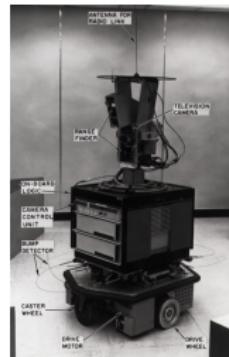
We can sacrifice optimality guarantees for performance by using inadmissible heuristics. Let  $h_a(s)$  be an admissible heuristic function. If we use a modified heuristic function  $h_w(s) = \varepsilon \cdot h_a(s)$  for some  $\varepsilon > 1$ , the found solution has a cost at most  $\varepsilon$ -times of the optimal one.



# (not so) Ancient AI

One of the best known algorithms in AI.

Created as a part of the Shakey project  
(1968)



But the research has not stopped with A\* → many variants.

AAAI 2016 Best Paper Award → **Bidirectional Search That Is Guaranteed to Meet in the Middle** by Holte et al. ([link](#))

## Abstract

We present MM, the first bidirectional heuristic search algorithm whose forward and backward searches are guaranteed to “meet in the middle”, i.e. never expand a node beyond the solution midpoint.

## Key idea

MM runs an A\*-like search in both directions, except that MM orders nodes on the Open list in a novel way. The priority of node  $n$  on  $\text{Open}_F$ ,  $pr_F(n)$ , is defined to be:

$$pr_F(n) = \max(f_F(n), 2g_F(n)).$$

# Lecture 4: Reinforcement learning

Viliam Lisý & Branislav Bošanský

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

March, 2021

Wikipedia: Reinforcement learning is “concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward”

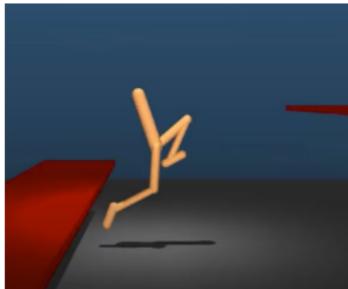
The book: “Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal.”

# Motivation

## Success stories:

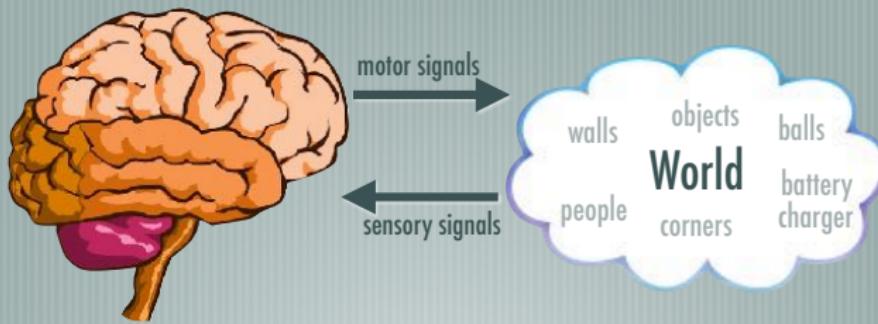


The screenshot shows the Seznam.cz homepage. At the top, there is a search bar with the placeholder "najdi tam, co nečekáš". Below the search bar are links for Internet, Piny, Mapy, Zdroj, Obrazky, Slevák, Zájmy lidí, and Video. A red "Vyhledat" button is on the far right. The main content area features several news cards. One card from "Seznam Zprávy" discusses the police search for a suspect in a shooting. Another card from "Koronec" discusses the election of a new mayor. There are also cards for "Lidl", "Očkování: rezervace a očkování", and "Email". At the bottom, there is a search bar with "seznam.cz" and a "Prohledat" button, along with a "Zadej nový e-mail" link.



Why is all this in simulations?  
RL currently needs a huge amount of experience, which is easier to obtain in simulation

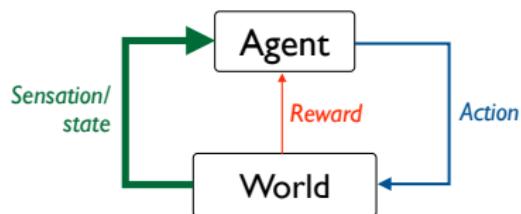
## Minds are sensori-motor information processors



the mind's job is to predict and control its sensory signals

Taken from R. Sutton's slides.

## Reinforcement learning is *more autonomous learning*



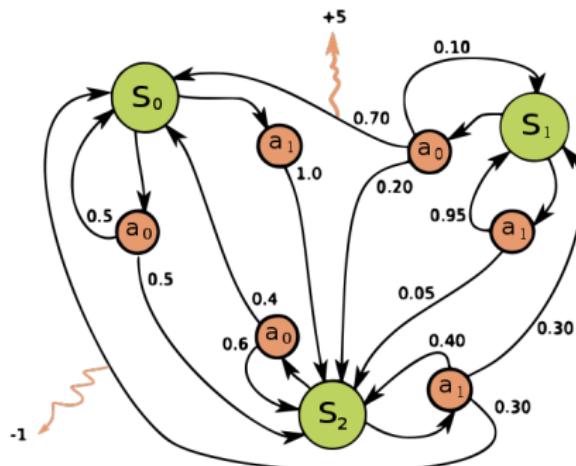
- Learning that requires less input from people
- AI that can learn for itself, during its normal operation

Taken from R. Sutton's slides (and many following are adaptations as well).

# Remember MDP

Standard model for Reinforcement Learning problems

- $S$  – states
- $R$  – rewards
- $A$  – actions
- Discrete steps  $t = 0, 1, 2, \dots$
- Environment *dynamics*



Source: Waldoalvarez © wikipedia

$$p(s', r | s, a) \leftarrow \Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}$$

# Single state MDP: Multi-armed Bandit Problem

All actions  $a_1, \dots, a_n$  lead back to the single state of MDP.

A simple case with many of the RL's fundamental problems.

# Why is it called Multi-Armed Bandit Problem



## Example problem

Action 1: Reward is always 8

$$\text{Expected reward: } q_*(1) = 8$$



Action 2: 88% chance of 0, 12% chance of 100

$$\text{Expected reward: } q_*(2) = 12$$

Action 3: Uniformly random between -10 and 35

$$\text{Expected reward: } q_*(3) = 12.5$$

Action 4: a third 0, a third 20, and a third from 8-18

$$\text{Expected reward: } q_*(4) = \frac{1}{3}(0) + \frac{1}{3}(20) + \frac{1}{3}(14) = 11$$

# Multi-armed Bandit Problem

On each of an infinite sequence of time steps,  $t = 1, 2, 3, \dots$ , you choose an action  $A_t$  from  $k$  possibilities, and receive a real-valued reward  $R_t$

The reward depends only on the action taken; it is identically, independently distributed (i.i.d.):

$$q_*(a) \doteq \mathbb{E}[R_t | A_t = a], \forall a \in \{1, \dots, k\}$$

These true values are **unknown**. The distribution is **unknown**.

Nevertheless, you must maximize your total reward

You must both try actions to learn their values (**explore**), and prefer those that appear best (**exploit**)

# The Exploration/Exploitation Dilemma

Suppose you form estimates

$$Q_t(a) \approx q_*(a), \forall a \quad \text{action-value estimates}$$

Define the **greedy action** at time  $t$  as

$$A_t^* \doteq \arg \max_a Q_t(a)$$

If  $A_t = A_t^*$  then you are *exploiting*

If  $A_t \neq A_t^*$  then you are *exploring*

You can't do both, but you need to do both

You can never stop exploring, but maybe you should explore less with time. Or maybe not.

# Action-Value Methods

Methods that learn action-value estimates and nothing else

For example, estimate action values as sample averages:

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbf{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbf{1}_{A_i=a}}$$

The sample-average estimates converge to the true values  
If the action is taken an infinite number of times

$$\lim_{N_t(a) \rightarrow \infty} Q_t(a) = q_*(a)$$

Where  $N_t(a)$  is the number of times action  $a$  has been taken by time  $t$ .

# $\epsilon$ -Greedy Action Selection

In greedy action selection, you always exploit

In  $\epsilon$ -greedy, you are usually greedy, but with probability  $\epsilon$  you instead pick an action at random (possibly the greedy action again)

This is perhaps the simplest way to balance exploration and exploitation

Algorithm  $\epsilon$ -Greedy:

Initialize, for  $a = 1$  to  $k$ :

$$Q(a) \leftarrow 0$$

$$N(a) \leftarrow 0$$

Repeat forever:

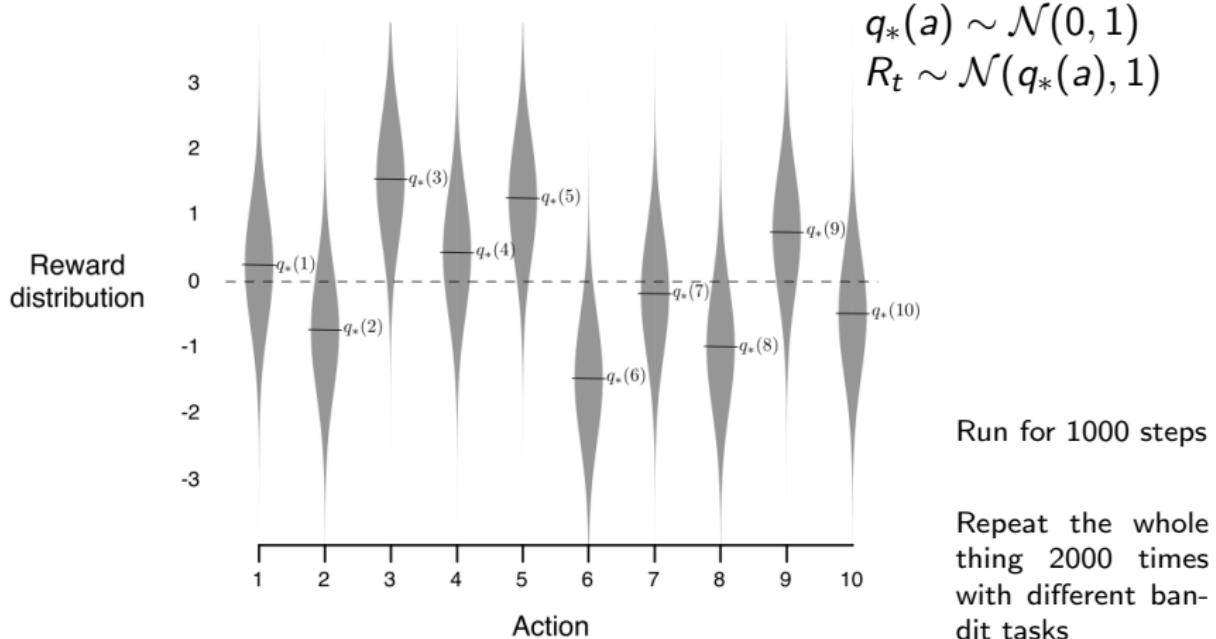
$$A \leftarrow \begin{cases} \arg \max_a Q(a) & \text{with probability } 1 - \epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases} \quad (\text{breaking ties randomly})$$

$$R \leftarrow \text{bandit}(A)$$

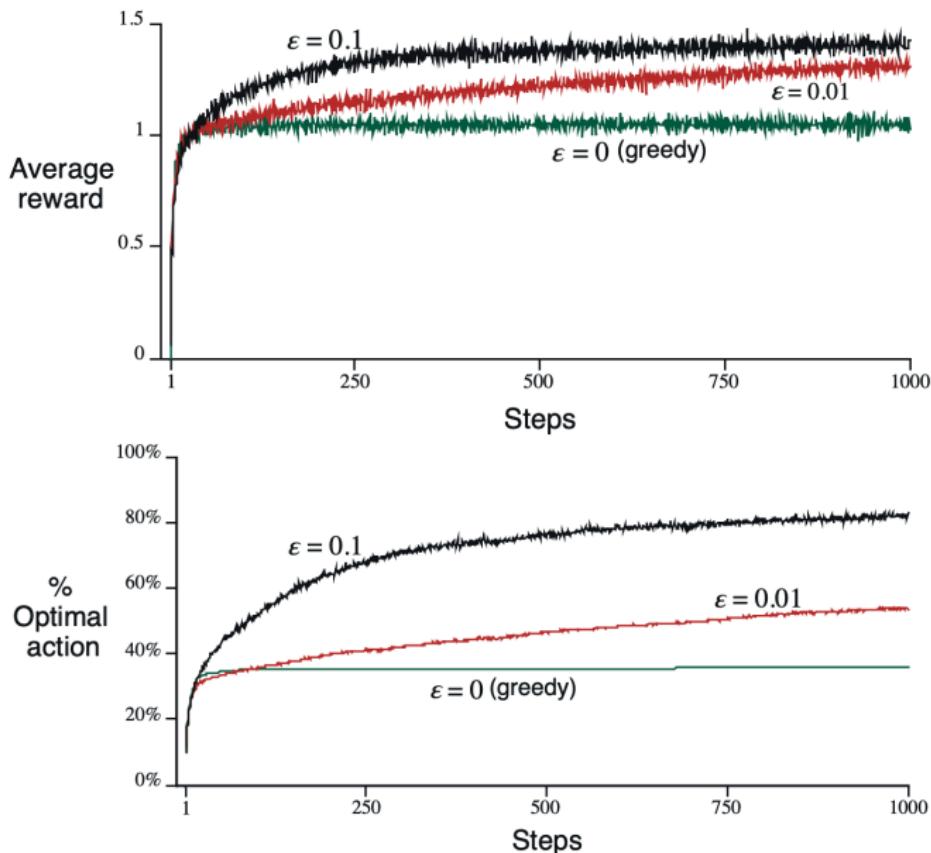
$$N(A) \leftarrow N(A) + 1$$

$$Q(A) \leftarrow Q(A) + \frac{1}{N(A)} [R - Q(A)]$$

# One Task from the 10-armed Testbed



# $\epsilon$ -Greedy Methods on the 10-Armed Testbed



# Averaging → Learning Rule

To simplify notation, let us focus on one action

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n - 1}$$

How can we do this incrementally (without storing all the rewards)?

Could store a running sum and count (and divide), or equivalently:

$$Q_{n+1} = Q_n + \frac{1}{n} [R_n - Q_n]$$

This is a standard form for learning/update rules:

$$\textit{NewEstimate} \leftarrow \textit{OldEstimate} + \textit{StepSize} [\textit{Target} - \textit{OldEstimate}]$$

# Derivation of incremental update

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n - 1}$$

$$\begin{aligned} Q_{n+1} &= \frac{1}{n} \sum_{i=1}^n R_i \\ &= \frac{1}{n} \left( R_n + \sum_{i=1}^{n-1} R_i \right) \\ &= \frac{1}{n} \left( R_n + (n-1) \frac{1}{n-1} \sum_{i=1}^{n-1} R_i \right) \\ &= \frac{1}{n} \left( R_n + (n-1)Q_n \right) \\ &= \frac{1}{n} \left( R_n + nQ_n - Q_n \right) \\ &= Q_n + \frac{1}{n} [R_n - Q_n], \end{aligned}$$

To assure convergence with probability 1:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \alpha_n^2(a) = \infty$$

e.g.,  $\alpha_n \doteq \frac{1}{n}$

not  $\alpha_n \doteq \frac{1}{n^2}$

if  $\alpha_n \doteq n^{-p}, p \in (0, 1)$

then convergence is at the  
optimal rate  $O(1/\sqrt{n})$

# Tracking a Non-stationary Problem

Suppose the true action values change (slowly) over time then we say that the problem is **nonstationary**

In this case, sample averages are not a good idea (Why?)

Better is an “exponential, recency-weighted average”:

$$\begin{aligned} Q_{n+1} &\doteq Q_n + \alpha [R_n - Q_n] \\ &= (1 - \alpha)^n Q_1 + \sum_{i=1}^n \alpha(1 - \alpha)^{n-i} R_i, \end{aligned}$$

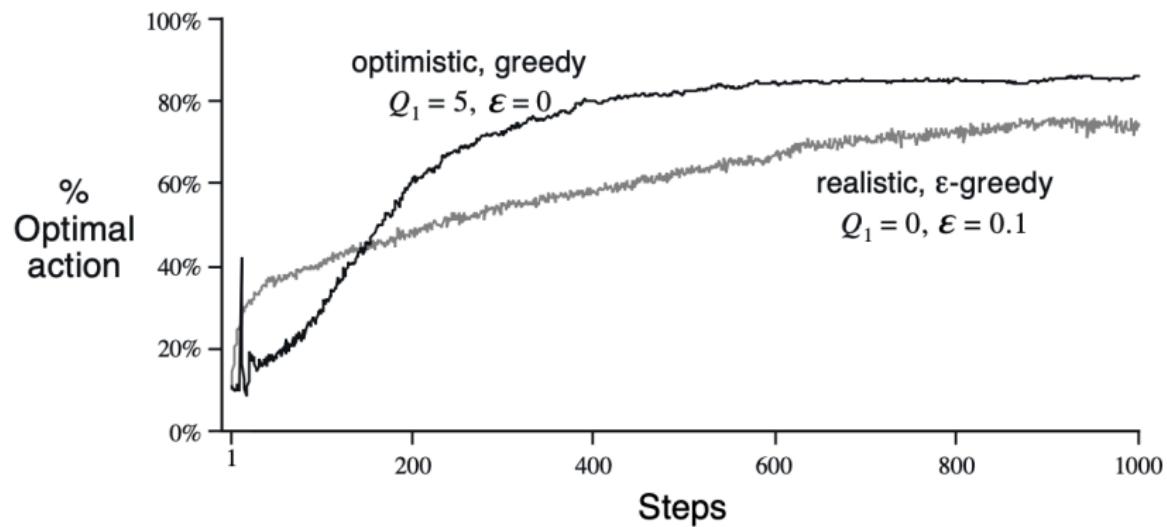
where  $\alpha$  is a constant step-size parameter,  $\alpha \in (0, 1]$

There is bias due to  $Q_1$  that becomes smaller over time

# Optimistic Initial Values

All methods so far depend on  $Q_1(a)$ , i.e., they are biased. So far we have used  $Q_1(a) = 0$

Suppose we initialize the action values **optimistically** ( $Q_1(a) = 5$ ),



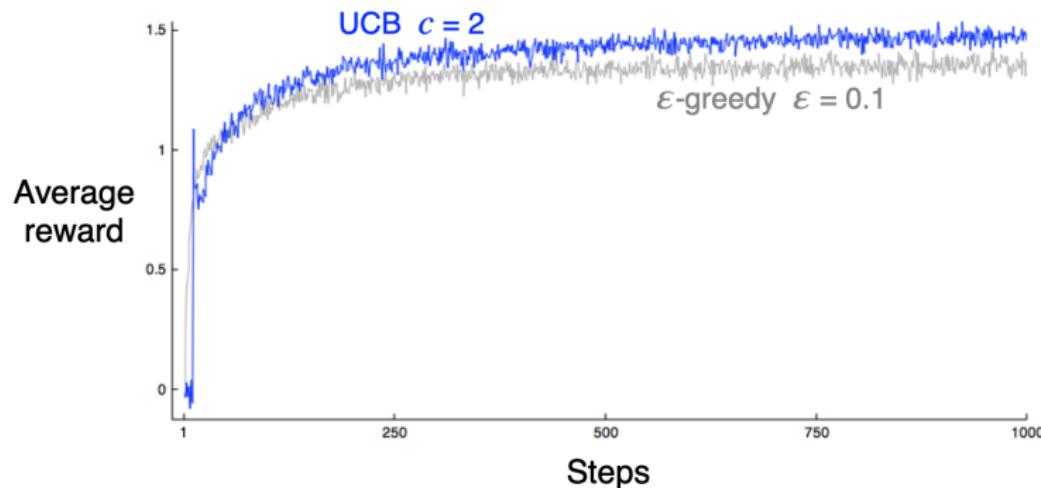
# Upper Confidence Bound (UCB) action selection

A clever way of reducing exploration over time

Estimate an upper bound on the true action values

Select the action with the largest (estimated) upper bound

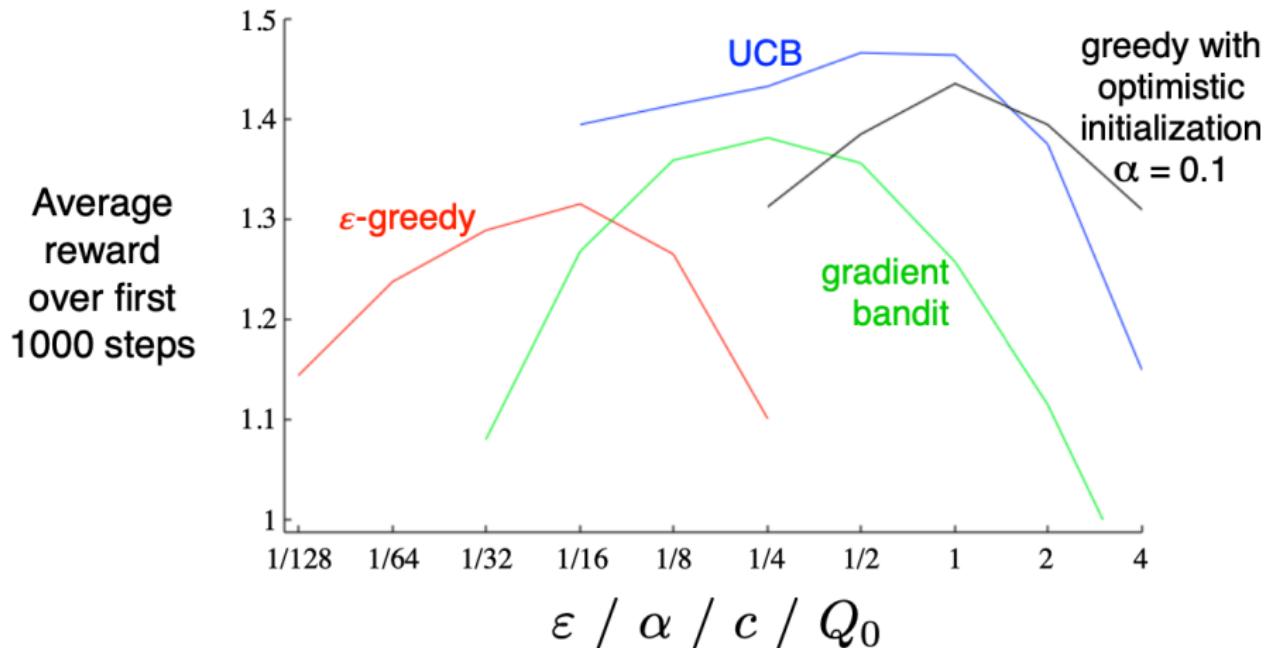
$$A_t \doteq \arg \max_a \left[ Q_t(a) + c \sqrt{\frac{\log t}{N_t(a)}} \right]$$



# Demo

<https://pavlov.tech/2019/03/02/animated-multi-armed-bandit-policies/>

# Comparison of Bandit Algorithms



These are all simple methods

- but they are complicated enough—we will build on them
- we should understand them completely
  - there is a lot of theory, e.g., upper/lower bounds
- there are still open questions

Our first algorithms that learn from evaluative feedback

- and thus must balance exploration and exploitation

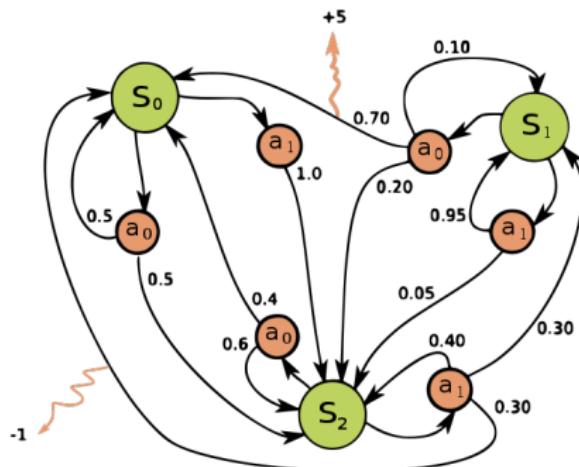
Our first algorithms that appear to have a goal

- that learn to maximize reward by trial and error



Standard model for Reinforcement Learning problems

- $S$  – states
- $R$  – rewards
- $A$  – actions
- Discrete steps  $t = 0, 1, 2, \dots$
- Environment *dynamics*



Source: Waldoalvarez © wikipedia

$$p(s', r | s, a) \leftarrow \Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}$$

# The Agent Learns a Policy

**Policy** at step  $t$ , denoted  $\pi_t$ , maps from states to actions.

$\pi_t(a|s) =$  probability that  $A_t = a$  when  $S_t = s$

Special case are **deterministic** policies.

$\pi_t(s) =$  the action taken with  $prob = 1$  when  $S_t = s$

- Reinforcement learning methods specify how the agent changes its policy as a result of experience
- Roughly, the agent's goal is to get as much reward as it can **over the long run**.

Suppose the sequence of rewards after step  $t$  is:

$$R_{t+1}, R_{t+2}, R_{t+3}, \dots$$

What do we maximize?

At least three cases, but in all of them, we seek to maximize the **expected return**,  $\mathbb{E} G_t$ , on each step  $t$ .

- **Total reward**,  $G_t = \text{sum of all future reward in the episode}$
- **Discounted reward**,  $G_t = \text{sum of all future } discounted \text{ reward}$
- **Average reward**,  $G_t = \text{average reward per time step}$

# Episodic Tasks

**Episodic tasks:** interaction breaks naturally into episodes, e.g., plays of a game, trips through a maze

In episodic tasks, we almost always use simple total reward:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T,$$

where T is a final time step at which a **terminal state** is reached, ending an episode.

# Continuing Tasks

**Continuing tasks:** interaction does not have natural episodes, but just goes on and on...

In this class, for continuing tasks we will always use *discounted return*:

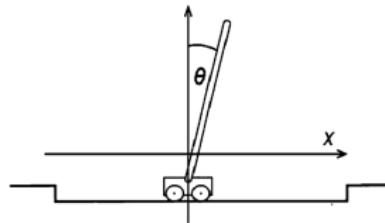
$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where  $0 \leq \gamma \leq 1$ , is the **discount rate**.

shortsighted  $0 \leftarrow \gamma \rightarrow 1$  farsighted

Typically,  $\gamma = 0.9$

# An Example: Pole Balancing



Avoid **failure**: the pole falling beyond a critical angle or the cart hitting end of track

(image from Ma&Likharev 2007)

As an **episodic task** where episode ends upon failure:

reward = +1 for each step before failure

$\Rightarrow$  return = number of steps before failure

As a **continuing task** with discounted return:

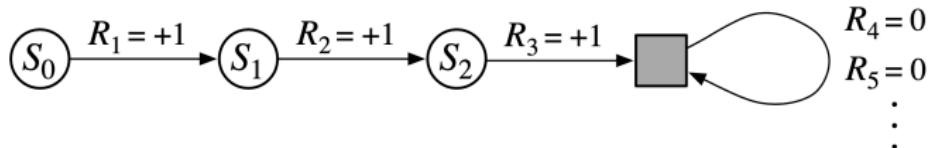
reward = -1 upon failure; 0 otherwise

$\Rightarrow$  return =  $-\gamma^k$ , for  $k$  steps before failure

In either case, return is maximized by avoiding failure for as long as possible.

# A Trick to Unify Notation for Returns

- In episodic tasks, we number the time steps of each episode starting from zero.
- We usually do not have to distinguish between episodes, so instead of writing for states in episode  $j$ , we write just  $S_t$
- Think of each episode as ending in an absorbing state that always produces reward of zero:



- We can cover **all** cases by writing  $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ , where  $\gamma$  can be 1 only if a zero rewards absorbing state is always reached.

# Summary

RL is a set of methods to learn a policy from an interaction with environment

The goal is to maximise return derived from immediate rewards

The simplest RL problem is the multi-armed bandit problem

- exploration vs. exploitation problem
- $\epsilon$ -greedy, optimistic initialisation, UCB

Canonical model of ML problems is MDP

# Lecture 5: Solving MDPs and Reinforcement Learning

Viliam Lisý & Branislav Bošanský

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

March, 2021

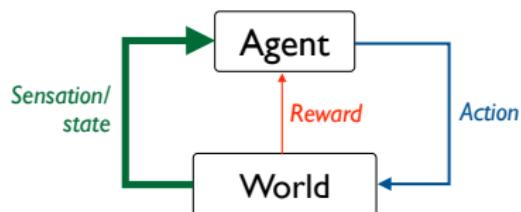
# Plan of today's lecture

- ① Value functions and Bellman equations
- ② Basic iterative solution techniques for known MDP

Next lecture

- ① RL algorithms in tabular representation for unknown MDP
- ② Scaling up with Neural Networks
- ③ DQN algorithm and its application to Atari games

## Reinforcement learning is *more autonomous learning*



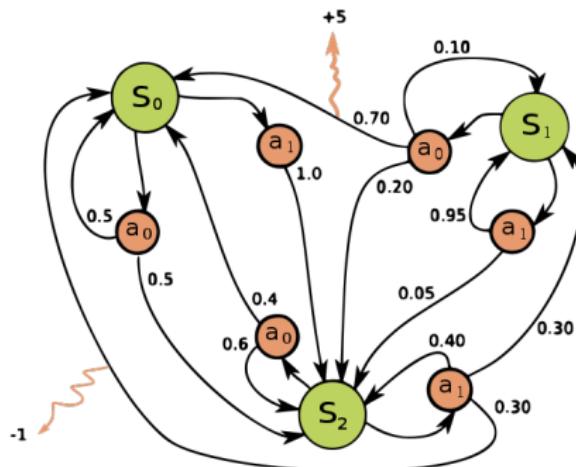
- Learning that requires less input from people
- AI that can learn for itself, during its normal operation

Taken from R. Sutton's slides (and many following are adaptations as well).

# Remember MDP

Standard model for Reinforcement Learning problems

- $S$  – states
- $R$  – rewards
- $A$  – actions
- Discrete steps  $t = 0, 1, 2, \dots$
- Environment *dynamics*



Source: Waldoalvarez © wikipedia

$$p(s', r | s, a) \leftarrow \Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}$$

# The Agent Learns a Policy

**Policy** at step  $t$ , denoted  $\pi_t$ , maps from states to actions.

$\pi_t(a|s) =$  probability that  $A_t = a$  when  $S_t = s$

Special case are **deterministic** policies.

$\pi_t(s) =$  the action taken with  $prob = 1$  when  $S_t = s$

- Reinforcement learning methods specify how the agent changes its policy as a result of experience
- Roughly, the agent's goal is to get as much reward as it can **over the long run**.

# Value Functions

One of the most fundamental concepts of RL!

A **value function** for an MDP and a policy  $\pi$

$$v_\pi : \mathcal{S} \rightarrow \mathbb{R}$$

is a function assigning each state  $s$  the expected return  
 $v_\pi(s) = \mathbb{E}_\pi G_0$  obtained by following policy  $\pi$  from state  $s$ .

# Optimal Value Functions

- For finite MDPs, policies can be **partially ordered**:

$$\pi \leq \pi' \text{ if and only if } v_\pi(s) \leq v_{\pi'}(s) \text{ for all } s \in \mathcal{S}$$

- There are always one or more policies that are better than or equal to all the others. These are the **optimal policies**. We denote them all  $\pi_*$ .
- Optimal policies share the same **optimal state-value function**:

$$v_*(s) = \max_{\pi} v_{\pi}(s) \text{ for all } s \in \mathcal{S}$$

- Optimal policies also share the same **optimal action-value function**:

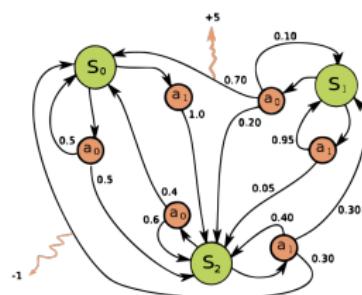
$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a) \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}.$$

This is the expected return for taking action  $a$  in state  $s$  and thereafter following an optimal policy.

# Why Are Optimal (Action-) Value Functions Useful

Any policy that is greedy with respect to  $v_*$  is an optimal policy.

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



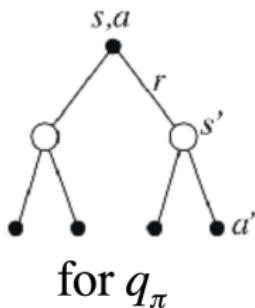
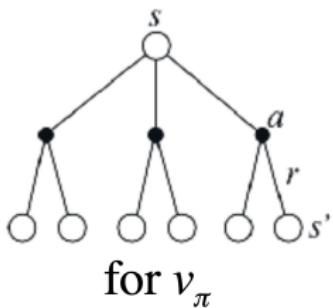
Given  $q_*$ , the agent does not even have to do a one-step-ahead search:

$$\pi_*(s) = \arg \max_a q_*(s, a)$$

# Bellman Equation for a Policy

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

This is a set of equations (in fact, linear), one for each state. The value function for  $\pi$  is its unique solution.

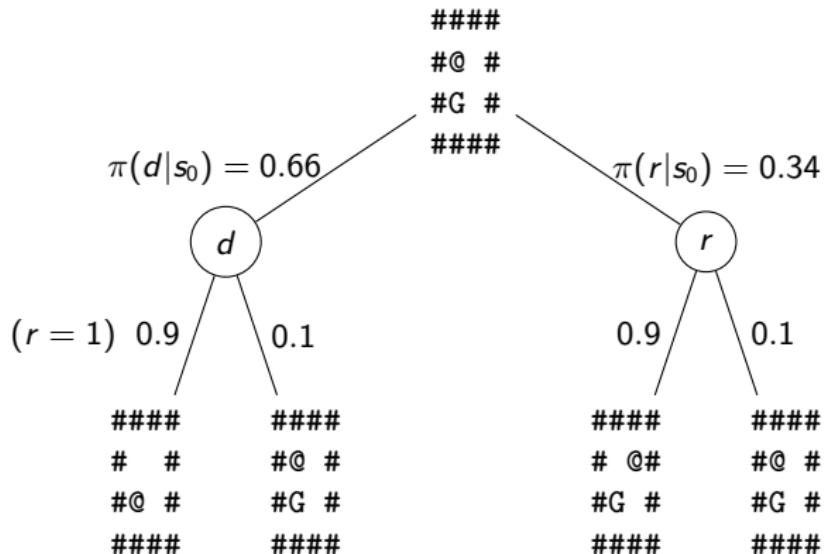


Allows simple computation of values for a policy.

Can we also compute policy for values?

# Backup example

A robot on a slippery floor successfully moves with probability 0.9.

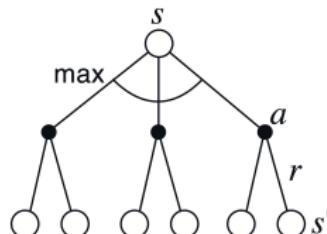


# Bellman Optimality Equation for $v_*$

The value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned}v_*(s) &= \max_a q_{\pi_*}(s, a) \\&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\&= \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_*(s')].\end{aligned}$$

The relevant backup diagram:

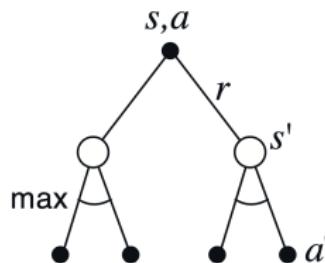


$v_*$  is the **unique** solution of this system of nonlinear equations.

# Bellman Optimality Equation for $q_*$

$$\begin{aligned} q_*(s, a) &= \mathbb{E} \left[ R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a \right] \\ &= \sum_{s', r} p(s', r | s, a) \left[ r + \gamma \max_{a'} q_*(s', a') \right]. \end{aligned}$$

The relevant backup diagram:



$q_*$  is the unique solution of this system of nonlinear equations.

## Relation to algorithm A

Is there any relation between  $v, q$  used in RL and  $f, g, h$  we defined for deterministic MDPs for algorithm A?

$$v_*(s) = h^*(s)$$

# Solving the Bellman Optimality Equation

Finding an optimal policy by solving the Bellman Optimality Equation **requires** the following:

- accurate knowledge of environment dynamics;
- we have enough space and time to do the computation;
- the Markov Property.

How much space and time do we need?

- polynomial in number of states,
- BUT, number of states is often huge (e.g., backgammon has about  $10^{20}$  states).

We usually have to settle for approximations.

Many RL methods can be understood as approximately solving the Bellman Optimality Equation.

# Policy Evaluation (Prediction)

---

**Policy Evaluation:** for a given policy  $\pi$ , compute the state-value function  $v_\pi$

Recall: **State-value function for policy  $\pi$**

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]$$

Recall: **Bellman equation for  $v_\pi$**

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

—a system of  $|S|$  simultaneous equations

# Iterative Policy Evaluation (Prediction)

---

$$v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k \rightarrow v_{k+1} \rightarrow \cdots \rightarrow v_\pi$$

a “sweep” 

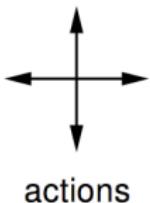
A sweep consists of applying a **backup operation** to each state.

A **full policy-evaluation backup**:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')] \quad \forall s \in \mathcal{S}$$

# A Small Gridworld Example

---



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

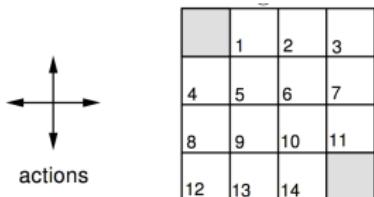
$R = -1$   
on all transitions

$$\gamma = 1$$

- An undiscounted episodic task
- Nonterminal states: 1, 2, . . . , 14;
- One terminal state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state unchanged
- Reward is  $-1$  until the terminal state is reached

# Iterative Policy Eval for the Small Gridworld

$\pi$  = equiprobable random action choices



$$R = -1 \text{ on all transitions}$$

$$\gamma = 1$$

$$k = 0$$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$$k = 1$$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$$k = 2$$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$$k = 3$$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$$k = 10$$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$$k = \infty$$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma v_k(s')] \quad \forall s \in \mathcal{S}$$

# Iterative Policy Evaluation – One array version

---

Input  $\pi$ , the policy to be evaluated

Initialize an array  $V(s) = 0$ , for all  $s \in \mathcal{S}^+$

Repeat

$$\Delta \leftarrow 0$$

For each  $s \in \mathcal{S}$ :

$$v \leftarrow V(s)$$

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

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until  $\Delta < \theta$  (a small positive number)

Output  $V \approx v_\pi$

# Why Does Iterative Policy Evaluation Work?

Many other RL algorithms use the same proof technique.

## Definition ( $\gamma$ -contraction)

Any function  $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$  is a  $\gamma$ -contraction for  $0 < \gamma < 1$  if and only if for some norm  $\| \cdot \|$  and all  $x, y \in \mathbb{R}^n$

$$\|F(x) - F(y)\| \leq \|x - y\|$$

## Theorem (Contraction mapping)

For a  $\gamma$ -contraction  $F$

- *Iterative application of  $F$  converges to a **unique** fixed point independently of the starting point*
- *at a linear convergence rate determined by  $\gamma$ .*

(Based on Tom Mitchell's [slides](#) )

Suppose we have computed a  $v_\pi$  for policy  $\pi$ . Can we easily improve it?

If there is a state  $s$  and action  $a$  such that  $q_\pi(s, a) > v_\pi(s)$  than setting  $\pi(s) = a$  improves the strategy.

Can it break the strategy somewhere else?

No, because the value at  $s$  improves.

The values in other states that eventually lead to  $s$  improve.

There is **no** state for which the value can decrease.

Can anything break if we modify more states at once?

No, for a similar reason, the value in each state only increases.

# Policy Iteration – One array version (+ policy)

---

1. Initialization

$V(s) \in \mathbb{R}$  and  $\pi(s) \in \mathcal{A}(s)$  arbitrarily for all  $s \in \mathcal{S}$

2. Policy Evaluation

Repeat

$$\Delta \leftarrow 0$$

For each  $s \in \mathcal{S}$ :

$$v \leftarrow V(s)$$

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

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until  $\Delta < \theta$  (a small positive number)

3. Policy Improvement

*policy-stable*  $\leftarrow$  true

For each  $s \in \mathcal{S}$ :

$$a \leftarrow \pi(s)$$

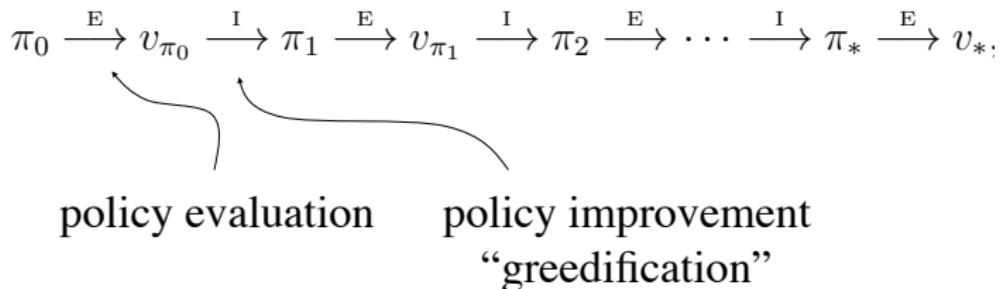
$$\pi(s) \leftarrow \arg \max_a \sum_{s',r} p(s', r | s, a) [r + \gamma V(s')]$$

If  $a \neq \pi(s)$ , then *policy-stable*  $\leftarrow$  false

If *policy-stable*, then stop and return  $V$  and  $\pi$ ; else go to 2

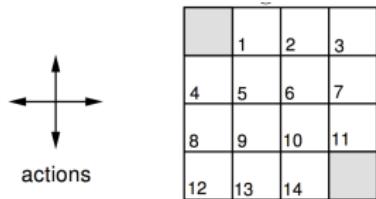
# Policy Iteration

---



# Iterative Policy Eval for the Small Gridworld

$\pi$  = equiprobable random action choices



$$R = -1 \text{ on all transitions}$$

$$\gamma = 1$$

	$V_k$ for the Random Policy				Greedy Policy w.r.t. $V_k$
$k = 0$	0.0	0.0	0.0	0.0	

	$V_k$ for the Random Policy				Greedy Policy w.r.t. $V_k$
$k = 1$	0.0	-1.0	-1.0	-1.0	
	-1.0	-1.0	-1.0	-1.0	

	$V_k$ for the Random Policy				Greedy Policy w.r.t. $V_k$
$k = 2$	0.0	-1.7	-2.0	-2.0	
	-1.7	-2.0	-2.0	-2.0	
	-2.0	-2.0	-2.0	-1.7	
	-2.0	-2.0	-1.7	0.0	

	$V_k$ for the Random Policy				Greedy Policy w.r.t. $V_k$
$k = 3$	0.0	-2.4	-2.9	-3.0	
	-2.4	-2.9	-3.0	-2.9	
	-2.9	-3.0	-2.9	-2.4	
	-3.0	-2.9	-2.4	0.0	

	$V_k$ for the Random Policy				Greedy Policy w.r.t. $V_k$
$k = 10$	0.0	-6.1	-8.4	-9.0	
	-6.1	-7.7	-8.4	-8.4	
	-8.4	-8.4	-7.7	-6.1	
	-9.0	-8.4	-6.1	0.0	

	$V_k$ for the Random Policy				Greedy Policy w.r.t. $V_k$
$k = \infty$	0.0	-14.	-20.	-22.	
	-14.	-18.	-20.	-20.	
	-20.	-20.	-18.	-14.	
	-22.	-20.	-14.	0.0	

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

for all  $s \in \mathcal{S}$

- An undiscounted episodic task
- Nonterminal states: 1, 2, ..., 14;
- One terminal state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state unchanged
- Reward is  $-1$  until the terminal state is reached

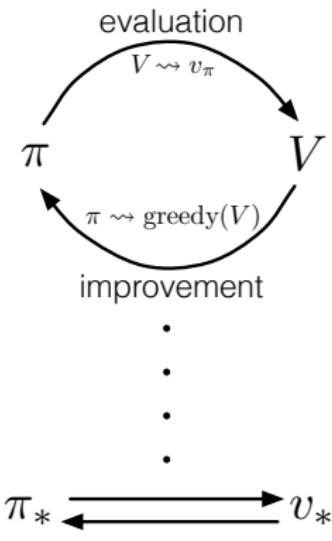
random policy

optimal policy

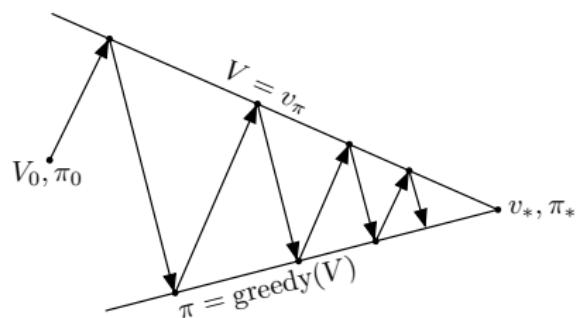
# Generalized Policy Iteration

**Generalized Policy Iteration (GPI):**

any interaction of policy evaluation and policy improvement, independent of their granularity.



A geometric metaphor for convergence of GPI:



# Generalised Policy Iteration

It is sufficient to co combine **any** consistent improvement in value estimate with **any** consistent improvement of the policy based on the value.

- Subset of states (even one)
- Improvement only in expectation
- Policy improvement only based on one action
- Small value improvement in the right direction

# Value Iteration

---

Recall the **full policy-evaluation backup**:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) [r + \gamma v_k(s')] \quad \forall s \in \mathcal{S}$$

Here is the **full value-iteration backup**:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a) [r + \gamma v_k(s')] \quad \forall s \in \mathcal{S}$$

# Value Iteration – One array version

---

Initialize array  $V$  arbitrarily (e.g.,  $V(s) = 0$  for all  $s \in \mathcal{S}^+$ )

Repeat

$$\Delta \leftarrow 0$$

For each  $s \in \mathcal{S}$ :

$$v \leftarrow V(s)$$

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

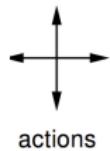
$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until  $\Delta < \theta$  (a small positive number)

Output a deterministic policy,  $\pi$ , such that

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

# Example



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$R = -1$   
on all transitions

$\gamma = 1$

$$V_0 =$$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$$V_1 =$$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$$V_2 =$$

0.0	-1.0	-2.0	-2.0
-1.0	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.0
-2.0	-2.0	-1.0	0.0

$$V_3 =$$

0.0	-1.0	-2.0	-3.0
-1.0	-2.0	-3.0	-2.0
-2.0	-3.0	-2.0	-1.0
-3.0	-2.0	-1.0	0.0

State and action value functions are key concepts in RL

Their values in different states are tied by Bellman equations

Bellman equations used as operators are contractions and hence their iterative application converges to unique solutions

(Generalized) policy iteration and value iterations are simple algorithms to solve MDPs

However, there are algorithms that require full knowledge of MDP, which is not necessary in RL methods, which generally approximate the full Bellman operator

# Lecture 6: Q-Learning and DQN

Viliam Lisý & Branislav Bošanský

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

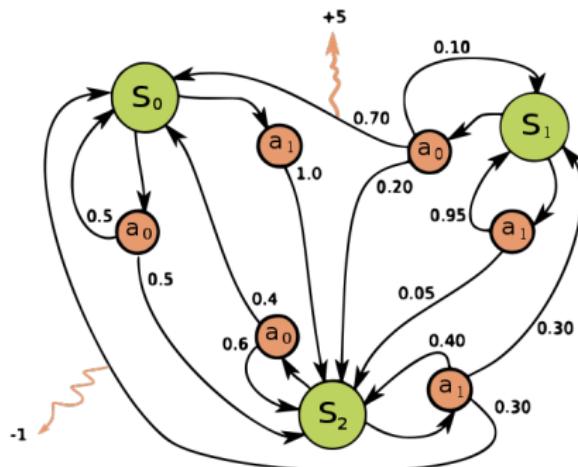
March, 2021

# Plan of today's lecture

- ① RL algorithms in tabular representation for unknown MDP
- ② Scaling up with Neural Networks
- ③ DQN algorithm and its application to Atari games

# Remember MDP

Standard model for Reinforcement Learning problems



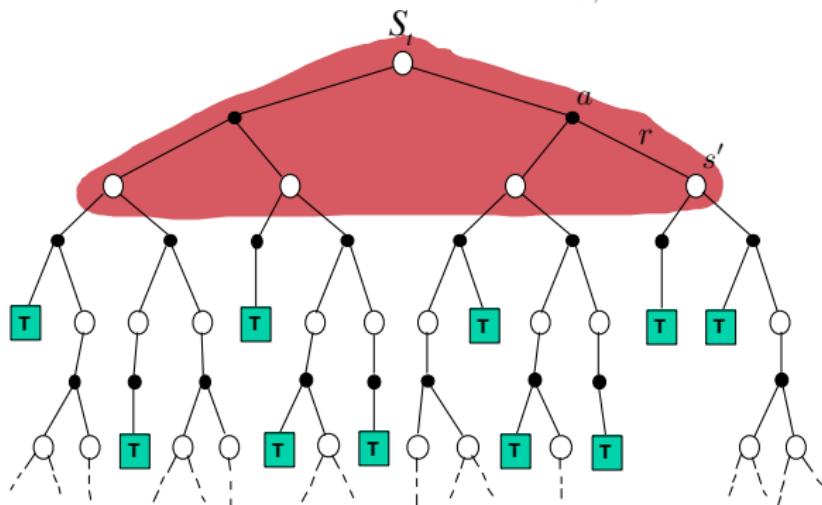
- $S$  – states
- $R$  – rewards
- $A$  – actions
- Discrete steps  $t = 0, 1, 2, \dots$
- Environment *dynamics*

Source: Waldoalvarez © wikipedia

$$p(s', r | s, a) \leftarrow \Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}$$

# Dynamic Programming

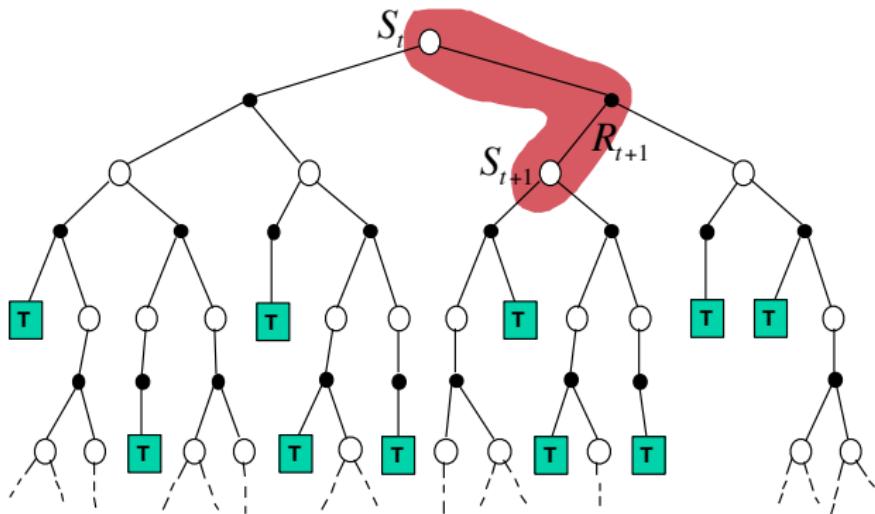
$$V(S_t) \leftarrow E_{\pi} \left[ R_{t+1} + \gamma V(S_{t+1}) \right] = \sum_a \pi(a|S_t) \sum_{s',r} p(s',r|S_t,a) [r + \gamma V(s')]$$



(Based on slides shared by R. Sutton)

# Simplest Temporal Difference Method

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$



(Based on slides shared by R. Sutton)

# Learning An Action-Value Function

---

Estimate  $q_\pi$  for the current policy  $\pi$



After every transition from a nonterminal state,  $S_t$ , do this:

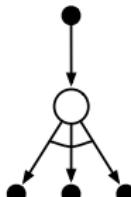
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

If  $S_{t+1}$  is terminal, then define  $Q(S_{t+1}, A_{t+1}) = 0$

# Q-Learning: Off-Policy TD Control

One-step Q-learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$



Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$ ;

    until  $S$  is terminal

# Q-Learning Example

$(2, \rightarrow) \Rightarrow -0.1$

$(3, \leftarrow) \Rightarrow -0.1$

$(2, \leftarrow) \Rightarrow -0.1$

$(1, \rightarrow) \Rightarrow -0.11$



$$\alpha = 0.1$$

Default:  $(*, *) \Rightarrow 0$

$(2, \leftarrow) \Rightarrow -0.09$

$(1, \leftarrow) \Rightarrow -0.1$

$(2, \leftarrow) \Rightarrow -0.191$

$(1, \leftarrow) \Rightarrow -0.19$

Repeat (for each step of episode):

Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)

Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$ ;

until  $S$  is terminal

# Q-Learning vs. $\epsilon$ -Greedy Bandit

## Q-Learning: Off-Policy TD Control

One-step Q-learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$



Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$   
Repeat (for each episode):

  Initialize  $S$

  Repeat (for each step of episode):

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$ ;

  until  $S$  is terminal

## $\epsilon$ -Greedy Action Selection

In greedy action selection, you always exploit

In  $\epsilon$ -greedy, you are usually greedy, but with probability  $\epsilon$  you instead pick an action at random (possibly the greedy action again)

This is perhaps the simplest way to balance exploration and exploitation

Algorithm  $\epsilon$ -Greedy:

  Initialize, for  $a = 1$  to  $k$ :

$$Q(a) \leftarrow 0$$

$$N(a) \leftarrow 0$$

  Repeat forever:

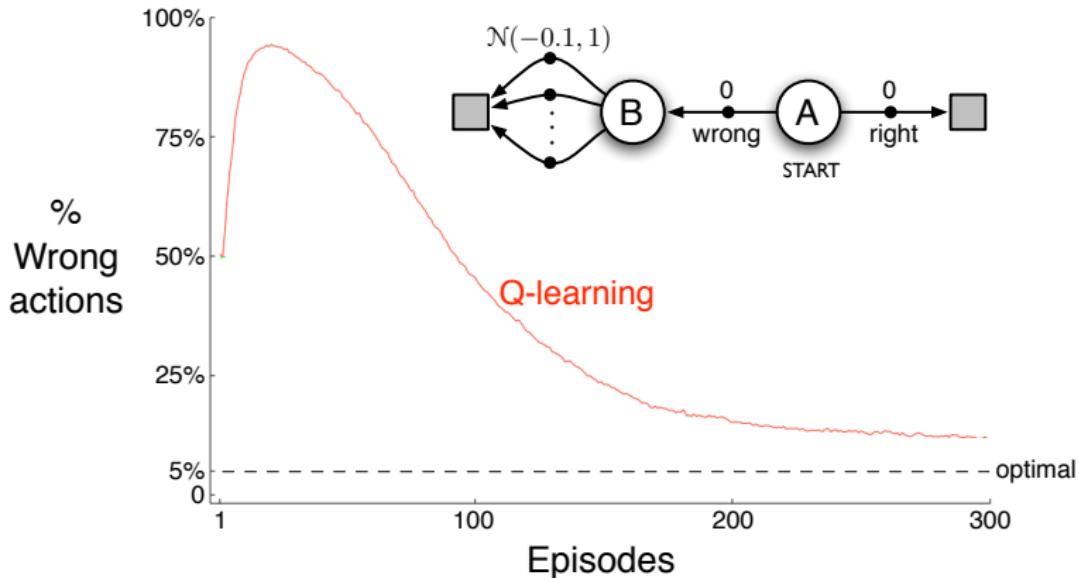
$A \leftarrow \begin{cases} \arg \max_a Q(a) & \text{with probability } 1 - \epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases}$  (breaking ties randomly)

$$R \leftarrow \text{bandit}(A)$$

$$N(A) \leftarrow N(A) + 1$$

$$Q(A) \leftarrow Q(A) + \frac{1}{N(A)} [R - Q(A)]$$

# Maximization Bias



**Tabular Q-learning:** 
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

# Double Q-Learning

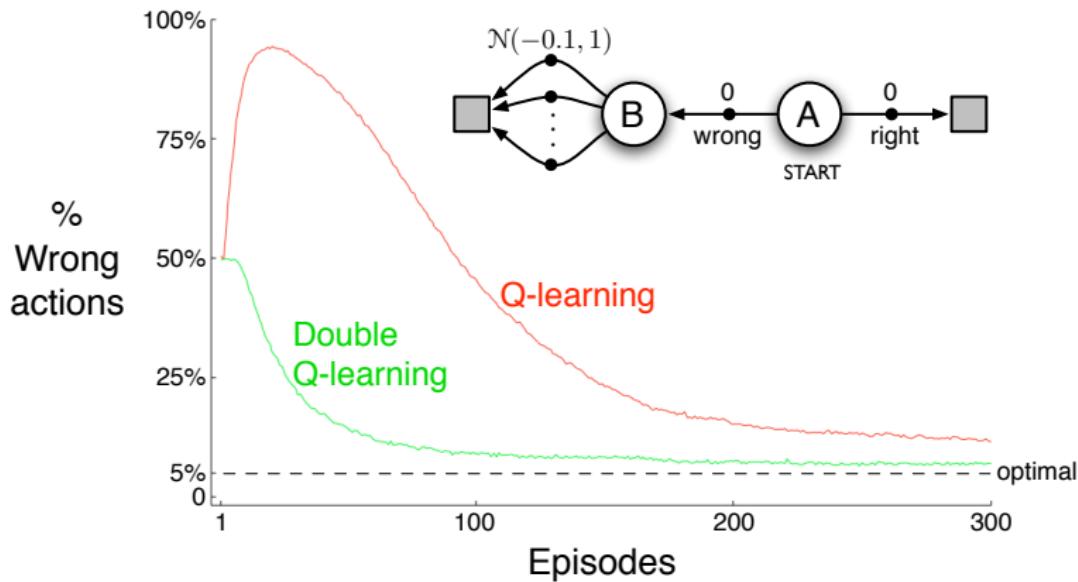
A solution to mitigate the maximization bias by van Hasselt [2010]

- Train two action-value functions  $Q_1$  and  $Q_2$
- Do Q-learning on both, but
  - never on the same time steps ( $Q_1$  and  $Q_2$  are independent)
  - pick  $Q_1$  or  $Q_2$  at random to be updated on each step
- If updating  $Q_1$  use  $Q_2$  for the value of the next state:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left( R_{t+1} + Q_2(S_{t+1}, \arg \max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right)$$

- Action selection can use a combination of  $Q_1$  and  $Q_2$

# Maximization Bias Mitigated



**Double Q-learning:**

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma Q_2(S_{t+1}, \arg \max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right]$$

# Atari problem

Create a program that would learn to play any Atari game



Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47, 253-279.

# Atari problem solution

"This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks."

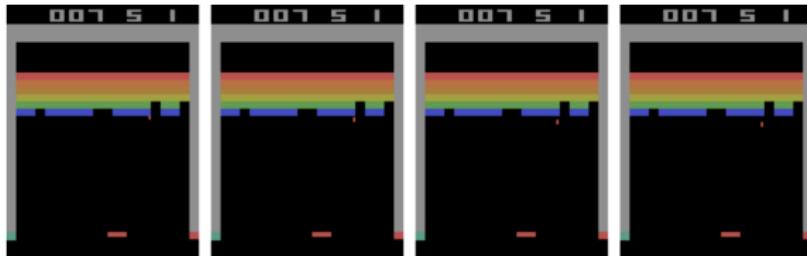


Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature* 518, 529-533.

# Atari Games MDP Representation

## States:

- Using four consecutive frames as state:



(Source: Greg Surma @ medium.com)

- Reduction of image size:  $210 \times 160 \times 3 \rightarrow 84 \times 84 \times 1$

## Actions:

- $2 \times 8$  directions of the joystick + button

Transitions are taken directly from a game emulator

## Rewards:

- Based on the game score
- Any score increase  $\rightarrow +1$ , any score decrease  $\rightarrow -1$

## How big is the MDP?

Assume we would quantise the colours to just black and white.  
The number of possible states of the MDP is then:

$$2^{84 \times 84 \times 4} = 28224 \approx 10^{8496}.$$

There are estimated  $10^{80}$  atoms in the observable universe.  
Hence, a tabular representation of the  $q$  function may not work.

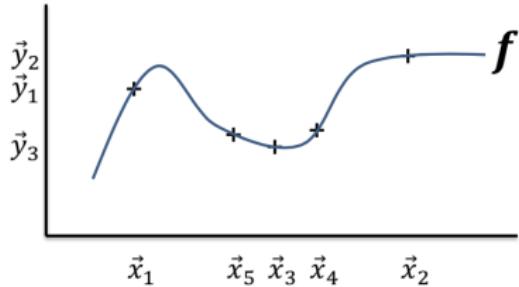
# Supervised Machine Learning

A useful tool for AI, which is **not** a focus of this course

Supervised learning = fitting a (high dimensional) function

For a data set  $(\vec{x}_i, \vec{y}_i)$ , find a function  $f$  that minimizes:

$$\frac{1}{n} \sum_i \|f(\vec{x}_i) - \vec{y}_i\|.$$



For example,  $f(2) = 2, f(3) = 3, f(4) = 4, f(5) = 5$ .

Q function is just a high-dimensional function approximable by a Neural network.

$$q(s, a) : \mathbb{R}^{28224} \times \mathcal{A} \rightarrow \mathbb{R}$$

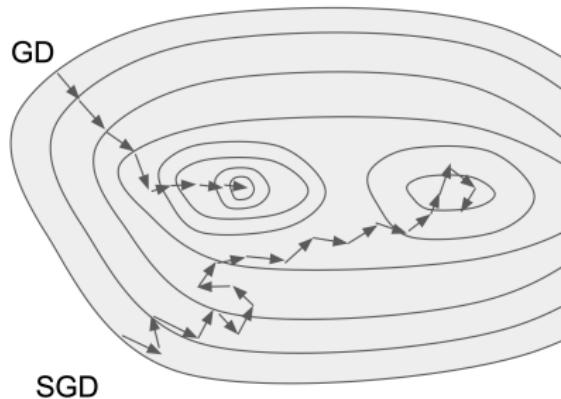
# Stochastic Gradient Descent

Dataset  $D = (\vec{x}_i, \vec{y}_i)$

Neural network  $f_w$  with weights  $w \in \mathbb{R}^m$

Loss:  $I(D, w) = \frac{1}{|D|} \sum_i \|f_w(\vec{x}_i) - \vec{y}_i\|$ .

Gradient descent:  $w' = w - \alpha \frac{\partial I(D, w)}{\partial w}$



Mini-batched version of the loss function:

For a uniformly selected subset of data  $\tilde{D} \subset D$  called a minibatch

define the approximate loss:  $\hat{I}(\tilde{D}, w) = \frac{1}{|\tilde{D}|} \sum_i \|f_w(\vec{x}_i) - \vec{y}_i\|$

and update:  $w' = w - \alpha \frac{\partial \hat{I}(\tilde{D}, w)}{\partial w}$ .

It works, because  $\mathbb{E} \hat{I}(\tilde{D}, w) = I(D, w)$ .

# DQN algorithm

## Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

        With probability  $\varepsilon$  select a random action  $a_t$

        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$

        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$

        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$

        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

        Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

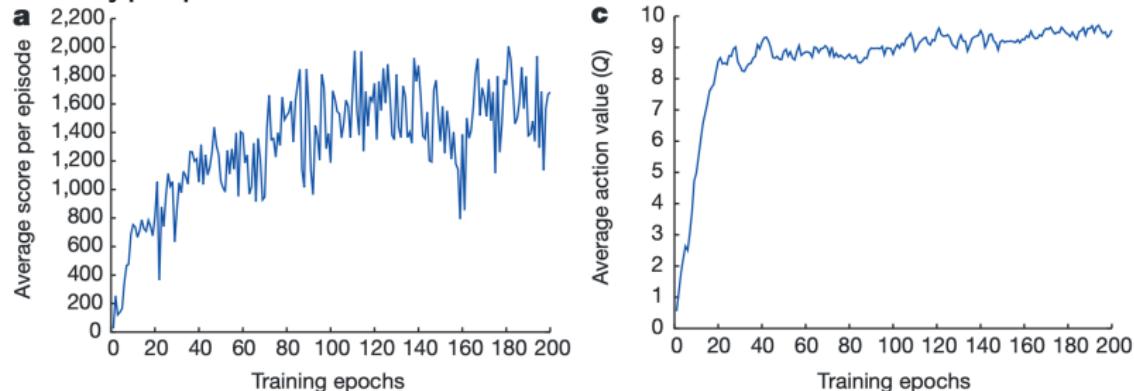
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

    Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

# DQN Training

Trained for each game separately, but using the same architecture and hyperparameters.

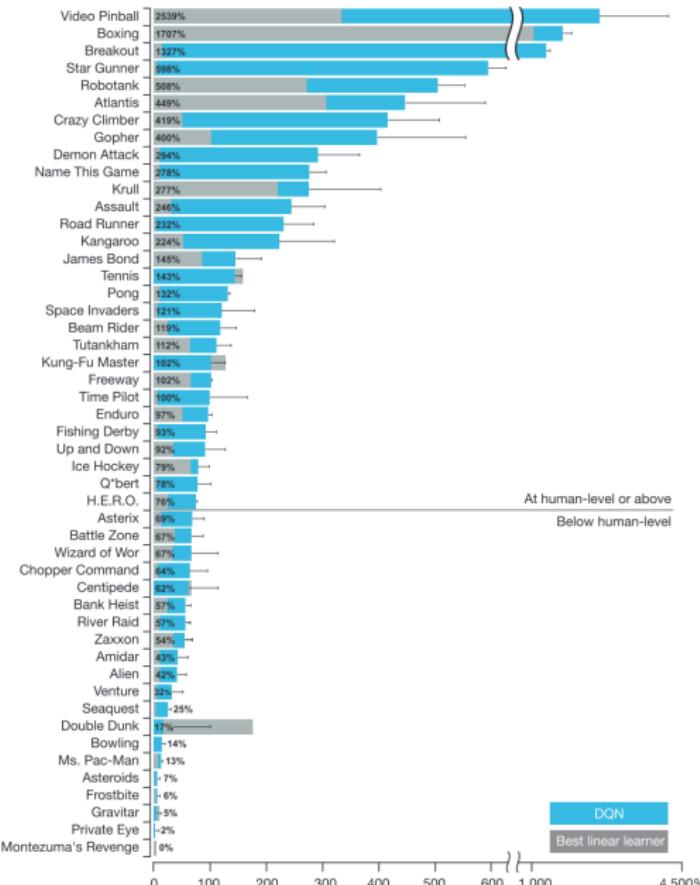


Convergence curve for "Space Invaders". One epoch is 520k frames.  $\epsilon = 0.05$ .

Training details: Minibatch size 32; exploration scaled from 1.0 to 0.1 over 1M frames and then fixed; overall 50M frames of training (38 days); replay buffer for 1M most recent frames. Probably 10 days of training per game and agent (not reported).

Breakout  
Space Invaders

# Results Relative to an Expert Human



# Summary

RL can solve huge MDPs without their explicit knowledge.

Key components of RL algorithms are policy evaluation and policy improvement.

Just using these steps on whole state space leads to

- policy iteration
- value iteration.

These algorithms are not super fast, but extremely versatile.

- Updates of just selected states
- Minimal / stochastic updates of policy and values
- Function approximation
- Endless modifications explored in RL literature

# Lecture 7: Two-Player Games

Viliam Lisý & **Branislav Bošanský**

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

[bosansky@fel.cvut.cz](mailto:bosansky@fel.cvut.cz)

February, 2021

# Moving to Two-Player Setting

Up to this point → finding optimal plan / best actions to be played in an environment

The agent was the only one changing the environment  
**(deterministic environment)** or there were **stochastic events**.

The stochastic events happen according to a known probability  
(probability of a box slipping out of the crane, etc.)

What if the environment (or another agent) is deliberately choosing the actions? → What is the “optimal plan” and how do we find it?

## Game Theory

An agent explicitly reasons about the possible actions of the other agents, their goals, and seeks own actions to be played w.r.t. to what other agents are going to play → such optimal behavior is defined by **game theory**.

Game theory is a broad scientific field covering parts of computer science, mathematics, economy.

We will only scratch the surface (see B4M36MAS Multiagent Systems or XEP36AGT for more in-depth topics regarding game theory).



# Games, Game Theory, and Artificial Intelligence

What does this have to do with AI?

Playing games well has been a challenge from the beginning of AI and computer science.

John von Neumann, one of the founders of computer science, also established game theory (von Neumann minmax theorem) and was interested in poker and bluffing.



Games are very good benchmarks for algorithms (popular, known, well-defined rules), they are challenging (the state space is huge).

# What are we going to cover?

We need to restrict to one of the most simple class of games:

- two players
- strictly competitive (or *zero-sum*) – win of one player is the loss of the opponent
- perfect information (chess, go, tic-tac-toe, ...)

What are we going to learn (this week)?

- how to find the optimal solution (optimal strategy)
- classical algorithms (pre 2006)
  - variants of branch-and-bound algorithm
  - useful heuristics

Next week → scaling-up and Monte Carlo sampling.

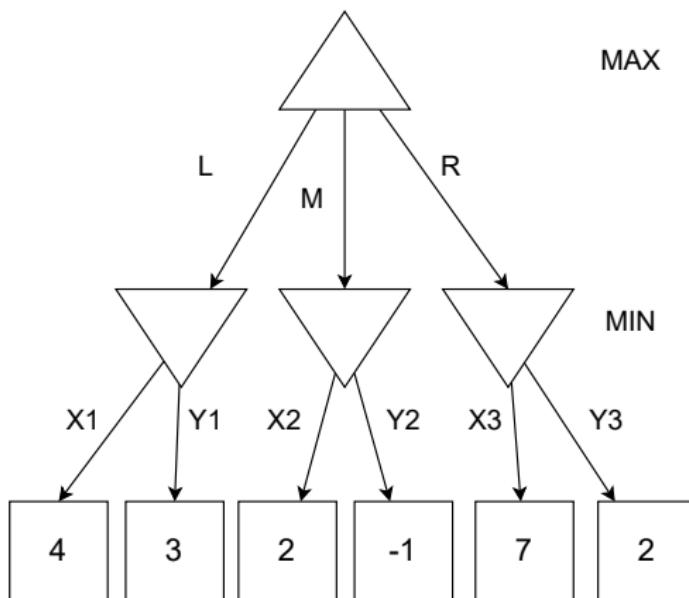
# Extensive-Form Games with Perfect Information

Sequential games with finite horizon use **extensive-form representation** that generalize search trees:

- $P$  is a set of players  $P = \{1, 2\}$  (or MAX and MIN)
- $H$  is a finite set of histories of actions from the initial positions where some player makes a decision.  $H_i$  are decision points of player  $i \in P$ .
- $A$  is a finite set of actions,  $A(h)$  denotes a set of actions applicable in a decision node  $h \in H$
- $Z \subseteq H$  is a set of terminal histories where the game ends
- $u$  is the utility function that assigns an outcome of the game to each terminal history,  $u : Z \rightarrow \mathbb{R}$

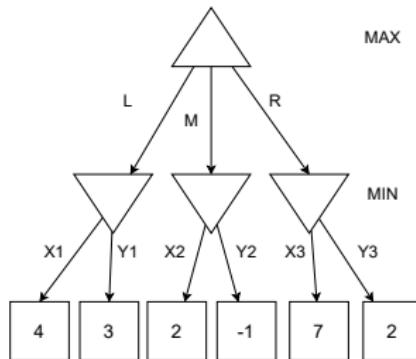
Technically, in game theory, every player maximizes their own utility function. In zero-sum games, utility of player 1 equals negative utility of player 2. Hence minimizing the utility of player 1 is the same as maximizing its negative value.

# Solving the Two-Player Games



# Solving the Two-Player Games

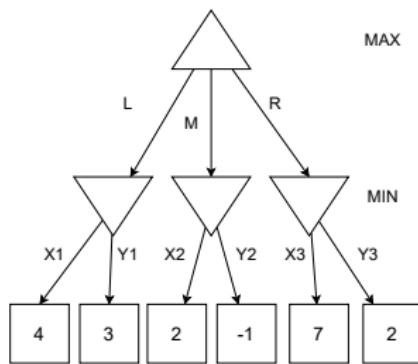
How to solve two-player games?



In MDPs, we have sought for an optimal plan or the best action for each state.

In games, we can also select the best action to be played in each decision point. Best action can either maximize (player 1) or minimize (player 2) the utility.

# Solving the Two-Player Games



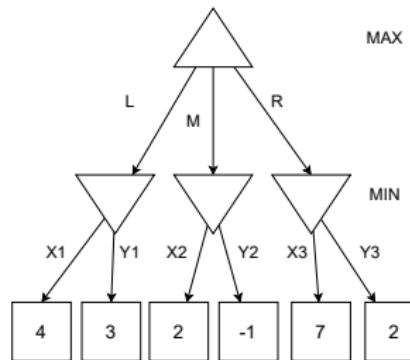
Similarly to deterministic uninformed search, we can use a depth-first search algorithm. For a history  $h$ :

- ① if  $h$  is a terminal history ( $h \in Z$ ), then return  $u(z)$ ,
- ② if  $h$  is a decision node, evaluate all children  
 $v_a = \text{search}(ha)$ ,  $a \in A(h)$  and
  - ① if  $h \in H_1$ , return  $\max_{a \in A(h)} v_a$
  - ② if  $h \in H_2$ , return  $\min_{a \in A(h)} v_a$

This baseline algorithm is known as **minimax** algorithm or simply a **backward induction** in two-player perfect information games.

The utility of player 1 when both players play optimally is called **the value of the game**.

# We Do Not Need To Consider Everything



Search through the complete game tree can be impractical and unnecessary. Consider how the algorithm advances through the search tree:

- after fully evaluating action  $L$  in the root node,  $v_L = 3$ ,
- when evaluating  $M$ , the algorithm first visits  $X_2$  and determines that  $v_{X_2} = 2$

## Obsevation

Regardless of the utility action after playing  $Y_2$ , action  $M$  is never going to be selected in the root node as the best action!

# Alpha-Beta Pruning

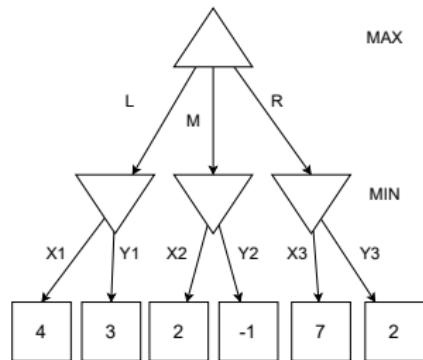
Minimax search with **alpha-beta pruning**:

- variant of a branch-and-bound algorithm
- extend the search with lower ( $\alpha$ ) and upper-bound ( $\beta$ ) estimates on the value of the game.

For a decision point  $h$  (initial values for bounds  $\alpha_h$  and  $\beta_h$  are passed as parameters):

- ① if  $h$  is a leaf ( $h \in Z$ ), then return  $u(z)$ ,
- ② if  $h$  is a decision node and  $h \in H_1$  then for each  $a_h \in A(h)$ :
  - ①  $v_h = \max(v_h, \text{search}(a_h, \alpha_h, \beta_h))$      $\alpha_h = \max(\alpha_h, v_h)$
  - ② if  $\beta_h \leq \alpha_h$  then break
- ③ if  $h$  is a decision node and  $h \in H_2$  then for each  $a_h \in A(h)$ :
  - ①  $v_h = \min(v_h, \text{search}(a_h, \alpha_h, \beta_h))$      $\beta_h = \min(\beta_h, v_h)$
  - ② if  $\beta_h \leq \alpha_h$  then break
- ④ return  $v_h$

# Example with Alpha-Beta Pruning



Alpha-beta pruning starts with  $\alpha_\emptyset = -\infty$  and  $\beta_\emptyset = \infty$ . After evaluating  $X_1$ ,  $\beta_L$  is set to 4. The value is updated to 3 after evaluating  $Y_1$ .

The solution of  $v_L$ , value 3, is then propagated to  $\alpha_\emptyset$  and thus the next recursive call uses updated lower bound (hence,  $\alpha_M$  is initialized to 3).

After evaluating  $X_2$ ,  $\beta_M = 2$ . Therefore, we have  $\beta_M < \alpha_M$  and we can stop exploring node  $M$ .

# Negamax

In many standard board games, players are alternating (MAX player moves, then MIN, etc.).

We can simplify the pseudocode of the algorithm by reverting the rewards and bounds:

- ① if  $h$  is a leaf ( $h \in Z$ ), then return  $u(z)$ ,
- ② for each  $a_h \in A(h)$ :
  - ①  $v_h = \max(v_h, -\text{search}(a_h, -\beta_h, -\alpha_h))$        $\alpha_h = \max(\alpha_h, v_h)$
  - ② if  $\beta_h \leq \alpha_h$  then break
- ③ return  $v_h$

This algorithm is known as **Negamax**. It is just a more compact way of describing the same behavior.

For educational purposes (learning and understanding the algorithm), I recommend using the full version (2 players, MAX, MIN; this holds also for follow-up algorithms, such as Negascout).

# Changing the Bounds

Recall, what we stated before: Alpha-beta pruning starts with  $\alpha_\emptyset = -\infty$  and  $\beta_\emptyset = \infty$ .

Setting the bounds this way is definitely correct. But what happens if we run Alpha-Beta pruning with some other initial values?

Imagine that we have a position in a game and we estimate (using heuristic evaluation) that the value of the game should be around 10.

What if we run Alpha-Beta pruning with interval  $[\alpha_\emptyset, \beta_\emptyset] = [9, 11]$ ?

## **Positive Impact:**

We can evaluate significantly fewer nodes!

## **Negative Impact:**

We can miss the correct solution!

# Changing the Bounds

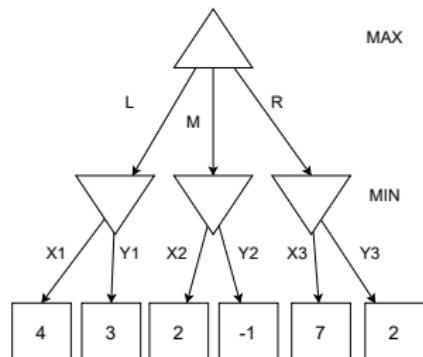
What if we run Alpha-Beta pruning with interval  $[\alpha_\emptyset, \beta_\emptyset] = [9, 11]$ ?

## Positive Impact:

We can evaluate significantly fewer nodes!

## Negative Impact:

We can miss the correct solution!



Consider our example: After evaluating  $X_1$ ,  $\beta_L$  is set to 4 and if  $\alpha_\emptyset$  (and thus initial value of  $\alpha_L$ ) is set to 9, then  $\beta_L < \alpha_L$ . Therefore,  $Y_1$  is never evaluated and the correct solution for this subtree (and also the whole game) is not found.

## Question

What is the value returned if we continue with the algorithm?

# Changing the Bounds

## Question

What is the value returned if we continue with the algorithm?

The algorithm returns value 7. This is not the value of the game (we know it is 3).

What can we conclude from return value 7?

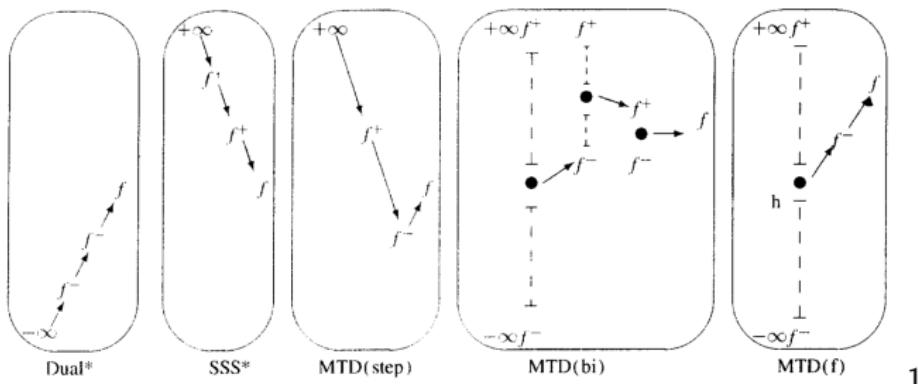
- the value of the game is not in the interval  $[9, 11]$  → our initial estimation was incorrect
- since the return value is **lower** than the estimated lower bound, we know that some actions of MIN player were not properly explored
- hence, the actual value of the game can only be lower → we can rerun the search with interval  $[-\infty, 7]$

If the return would be higher than the estimated upper bound, we know that the true value is higher.

# Changing the Bounds

We can systematically search for a correct value of the game (thus the optimal strategy) by:

- (e.g., binary) search over the interval of values
- repeated calls to the alpha-beta algorithm with modified bound interval (called **window**)



1

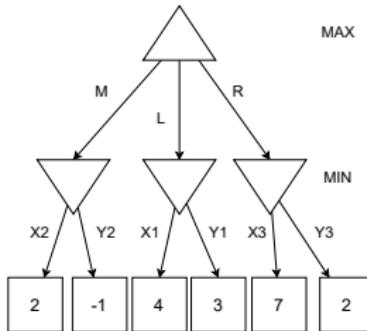
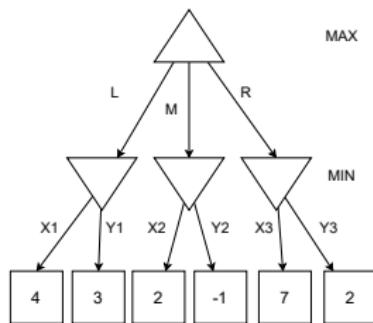
---

<sup>1</sup>Best-first fixed-depth minimax algorithms. Plaat et. al. , In Artificial Intelligence, Volume 87, Issues 1-2, November 1996, Pages 255-293

# Action Ordering

Can we use the idea of quick check **during** the search?

How does the ordering in which the actions are evaluated affect the number of searched nodes?



Assuming the actions are evaluated from left to right, Alpha-Beta pruning will not prune out anything in the second game.

The estimated searched space of Alpha-Beta pruning is  $O(b^{d/2})$  in case actions are evaluated in the optimal order (as opposed to  $O(b^d)$  for minimax).

# NegaScout (or Principal variation search)

What if we assume have the optimal ordering of actions?

We can, for example, have a good heuristic that sorts the actions prior to an evaluation in each node.

Idea:

- fully evaluate the first action (i.e., with the full-sized interval),
- evaluate subsequent actions with a minimal-sized interval (**null window**),
- based on the null-window evaluation:
  - if the returned value indicates that the value of the subtree is the same or worse, the algorithm proceeds with next action
  - if the returned value indicates that the value of the subtree can be better, the algorithm must evaluate this action again properly

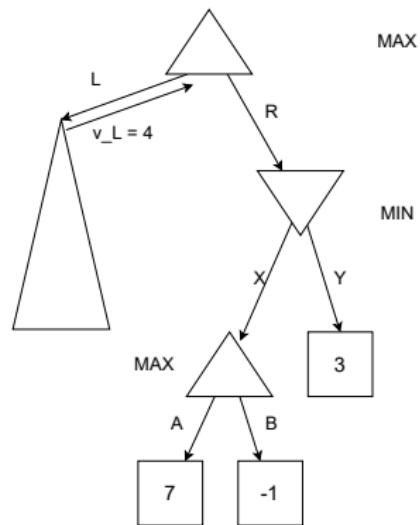
# NegaScout (or Principal variation search)

For a decision point  $h$  (initial values for bounds  $\alpha_h$  and  $\beta_h$  are passed as parameters):

- ① if  $h$  is a leaf ( $h \in Z$ ), then return  $u(z)$ ,
- ②  $b_h = \beta_h$
- ③ if  $h$  is a decision node and  $h \in H_1$  then for each  $a_h \in A(h)$ :
  - ①  $v_h = \text{search}(a_h, \alpha_h, b_h)$
  - ② if  $((\alpha_h < v_h < \beta_h) \text{ and } (h \text{ is not the first child}))$ 
    - $v_h = \text{search}(a_h, v_h, \beta_h)$
  - ③  $\alpha_h = \max(\alpha_h, v_h)$
  - ④ if  $\beta_h \leq \alpha_h$  then break
  - ⑤  $b_h = \alpha_h + 1$
- ④ ... (similarly for player 2) ...
- ⑤ return  $\alpha_h$

# NegaScout vs. Alpha-Beta

NegaScout can evaluate some nodes multiple times. At the same time, it can never evaluate more different histories and can be 10 – 20% faster than Alpha-Beta pruning.



Assume that after evaluating subtree  $L$ , the value  $v_L$  equals to 4. The evaluation of  $R$  starts with  $[\alpha_R, \beta_R] = [4, 5] = [\alpha_{RX}, \beta_{RX}]$ . When the algorithm evaluates  $A$ , it updates  $\alpha_{RX}$  to 7. Since  $\beta_{RX}$  has been manually set to 5, it holds that  $\beta_{RX} < \alpha_{RX}$  and the algorithm prunes out evaluation of action  $B$ .

Note that this would not happen in Alpha-Beta pruning.

# Game Solving vs. Game Playing

The algorithms that we have described so far are **offline (equilibrium computation) algorithms**.

Given a game, they compute an optimal strategy (for both players) and value of the game.

However, this is not tractable for most practical games due to the size of the game tree:

- chess has branching factor (number of applicable actions)  
≈ 35, Go up to 360, etc.; games can take tens (hundreds) of moves of both players to terminate.

Game-playing algorithms are searching only to a limited depth. Instead of a utility function applied on terminal states, they return a value of a **heuristic evaluation function**.

# Game Solving vs. Game Playing

Unfortunately, designing a well-informed heuristic evaluation function is much more challenging than in the single-player case.

Optimizing a strategy in a depth-limited game does not have to correspond to truly optimal strategy:

- consider chess and assume the algorithm searches to depth  $d$  actions for both players
- player 1 has to sacrifice a queen to avoid getting checkmate and losing
- however, the checkmate is beyond the horizon of  $d$  actions
- hence, from the perspective of evaluation function, it can be better to sacrifice a pawn or a knight instead of the queen – the algorithm does not know that sacrificing a different piece does not prevent the problem

There are heuristics tackling these issues (e.g., at certain dynamic positions, the search is not strictly terminated at the horizon).

There are many useful heuristics developed for games, that can be useful in other problems.

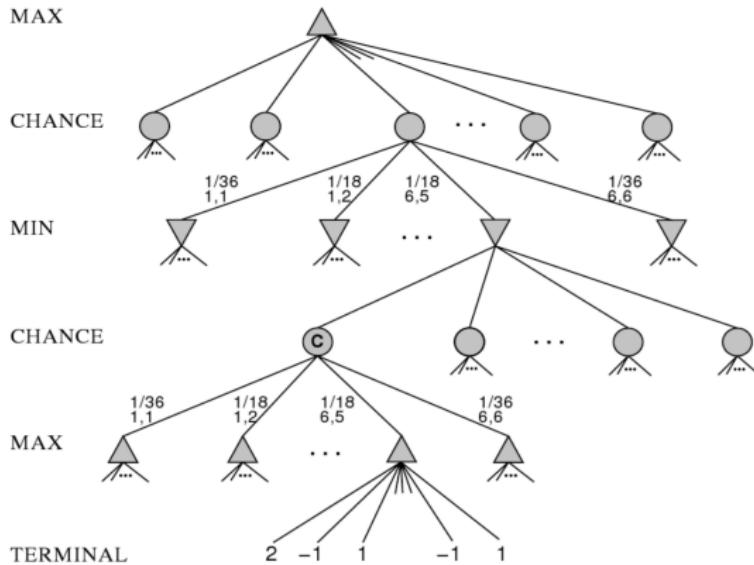
One of the best known are **transposition tables**:

- a cache table with previously evaluated positions,
- minimize negative effects of game tree (if the same position is reached with a different sequence of actions),
- for a depth-limited alpha-beta (negascout) algorithm, the depth limit and bounds can affect the computed (and stored) value

# Games with Chance

Two-player games can easily be extended with stochastic events.

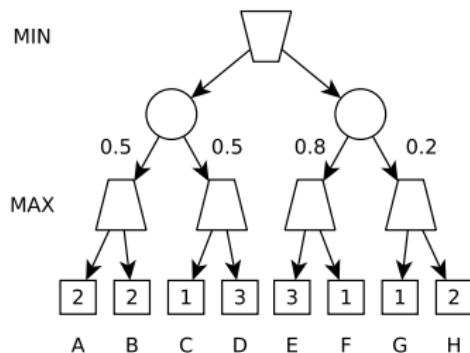
We can introduce another (Nature or chance) player that chooses actions according to a known distribution.



## Question

How does the Alpha-Beta pruning translates to a game with chance nodes?

Example:



Can Alpha-Beta pruning algorithm prune-out anything? Assume that the utility values can be  $\in \mathbb{R}$ .

What happens if the possible utility values are from interval  $[1, \infty]$ ?

You should be able to figure this out! A similar task **can be** in the exam.

# What is it all good for?

The first algorithm that beat a chess professional, Deep Blue, was built on these algorithms ([link to a paper](#)).



*Deep Blue relies on many of the ideas developed in earlier chess programs, including quiescence search, iterative deepening, transposition tables (all described in [24]), and NegaScout [23].*

Besides that, Deep Blue was heavily using parallel search and special hardware “chess chips”.

# What is it all good for?

The techniques are applicable for other problems / search-based algorithms:

- use of bounds and pruning out not-perspective branches
- problems with the horizon (if a monotonic heuristic is not possible)
- use of cached values

# Lecture 8: MCTS and AlphaGo

Viliam Lisý & Branislav Bošanský

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

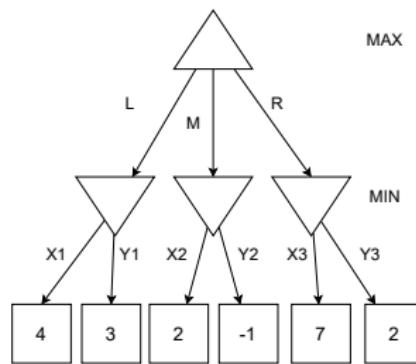
viliam.lisy@fel.cvut.cz

April, 2021

# Plan of today's lecture

- ① Monte Carlo Tree Search
- ② Overview of basic improvements
- ③ The challenge of computer Go
- ④ AlphaGo

## Recap.: Solving the Two-Player Games



Similarly to deterministic uninformed search, we can use a depth-first search algorithm. For a history  $h$ :

- ① if  $h$  is a terminal history ( $h \in Z$ ), then return  $u(z)$ ,
- ② if  $h$  is a decision node, evaluate all children  $v_a = \text{search}(A(h))$  and
  - ① if  $h \in H_1$ , return  $\max_{a \in A(h)} v_a$
  - ② if  $h \in H_2$ , return  $\min_{a \in A(h)} v_a$

This baseline algorithm is known as **minimax** algorithm or simply a **backward induction** in two-player perfect information games.

The utility of player 1 when both players play optimally is called **the value of the game**.

# Games are BIG

The number of reachable states:

- Chess:  $\approx 10^{45}$   ~~$10^{45}$~~   $10^{23}$
- Go:  $\approx 10^{170}$   ~~$10^{170}$~~   $10^{85}$

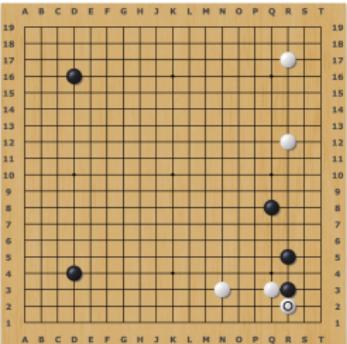
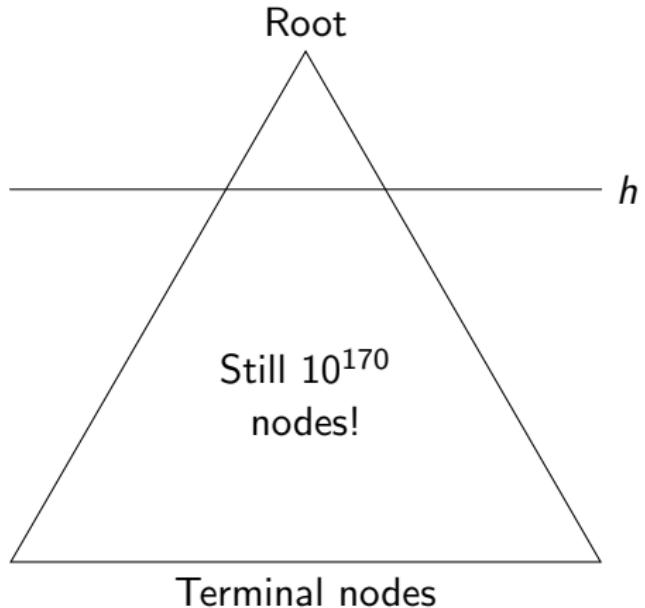
Can't we just prune most of the states out?

## Theorem

*For a game with branching factor  $b$  and depth  $d$ ,  $\alpha\beta$ -search will evaluate at least  $b^{d/2} = \sqrt{b^d}$  nodes.*

RCI cluster (most powerful Czech AI cluster for 2M EUR in 2018)  
 $1.5\text{TB} = 10^{12}$  of RAM,  $226\text{TB}$  of drives  $= 2.2 \times 10^{14}$   
 $10^9$  RCI clusters necessary to store the strategy with 1B per state

# Depth-limited game solving



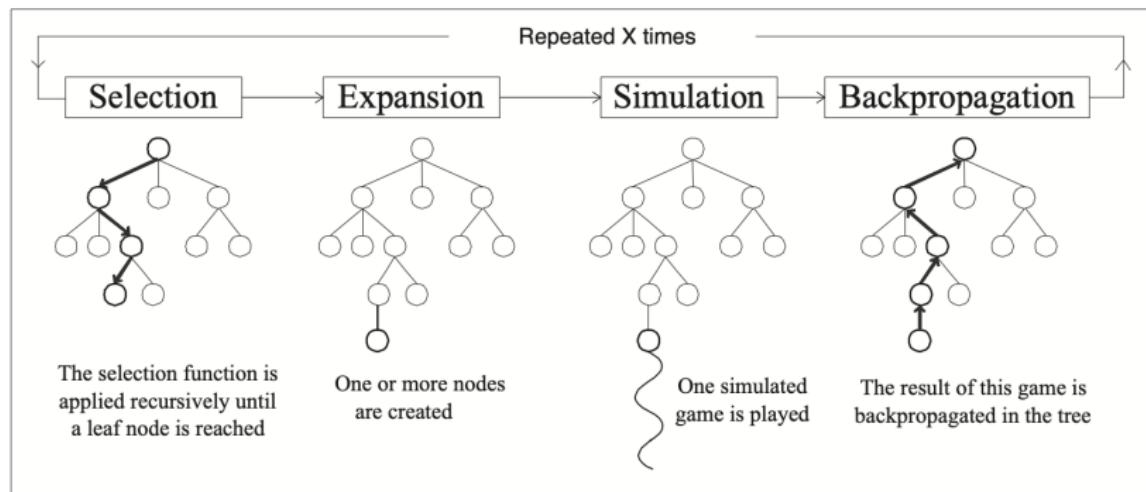
Who will win?

Sometimes very hard to make a good heuristic evaluation.

# Monte Carlo Tree Search

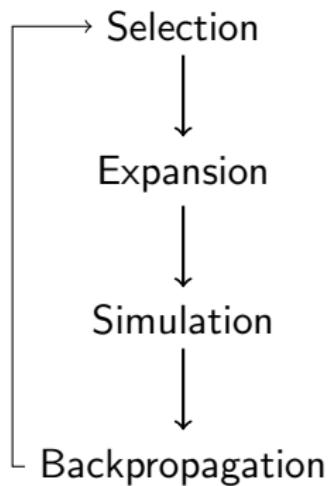
Idea:

- ① Instead of evaluation function, use random roll-outs (simulations) of the rest of the game
- ② Store detailed statistics only in relevant parts of the game tree



(Image from Chaslot et al. 2007)

# Monte Carlo Tree Search - Demonstration



# MCTS Selection

We want to explore the more promising actions more often  
We want to learn which actions are the most promising

Does it sound familiar?

Exploration vs. exploitation dilemma

Any algorithm for the multi-arm bandit problem can be used

MCTS + UCB = UCT – the most popular MCTS variant

$$A_t(s) \doteq \arg \max_a \left[ Q_t(s, a) + c \sqrt{\frac{\log N_t(s)}{N_t(s, a)}} \right]$$

Where  $s$  is the node in the tree where we perform the selection.

The part of the search space storing the statistics is expanded

- all actions may be added
- a single state-action may be added
- a node may be expanded only after visited multiple times

Progressive widening

- games may have many actions – Go ( $19^2$ ), Arrimaa ( $\approx 20k$ )
- a single state-action may be added at a time
- PW:
  - start with few (heuristically chosen?) actions initially
  - add more once the previously added are explored sufficiently
  - works even in with infinite number of actions
  - keep  $k = \lceil C \cdot N(s)^\alpha \rceil$  actions with  $\alpha < 0.5$
  - studied in bandit literature on infinitely many armed bandits

Simulation: choose actions based on fast policies until game ends

- purely random surprisingly effective
- hand-coded knowledge
- learned knowledge

Backpropagation: update statistics used by the selection

- $N(s)$ ,  $N(s, a)$
- $Q(s, a)$
- whatever – rewards range, variance,  $Q(a)$ , etc.
- each player stores his perspective vs. min / max

# MCTS Is useful even in non-game setting

First developed and popularised in games

Everything works as well with single player

PROST, POMCP, etc.

More on it in B(E)4M36PUI – Planning for Artificial Intelligence

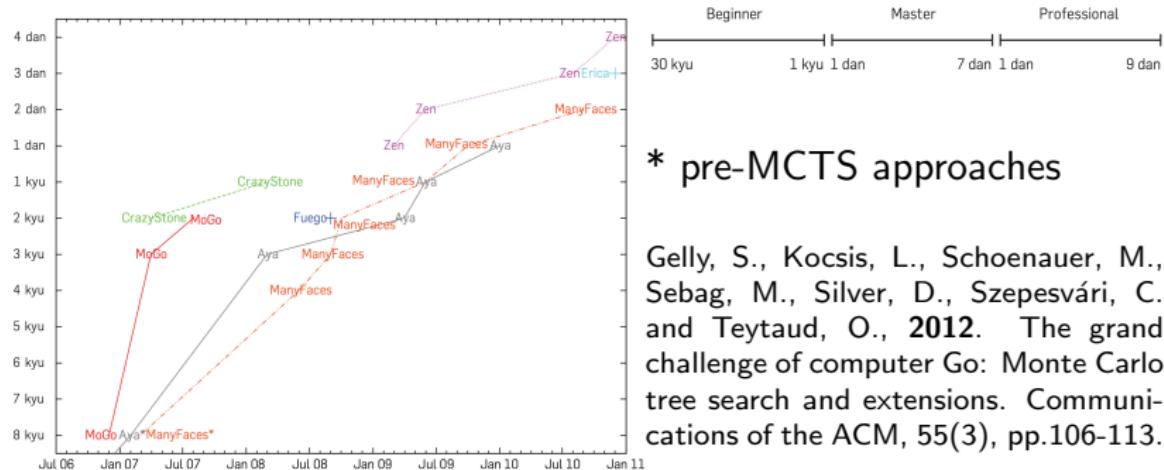
Further reading on MCTS

- RL Introduction (Book) – Section 8.11
- Browne, C., Powley, E., Whitehouse, D., et al. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1), pp.1-43.

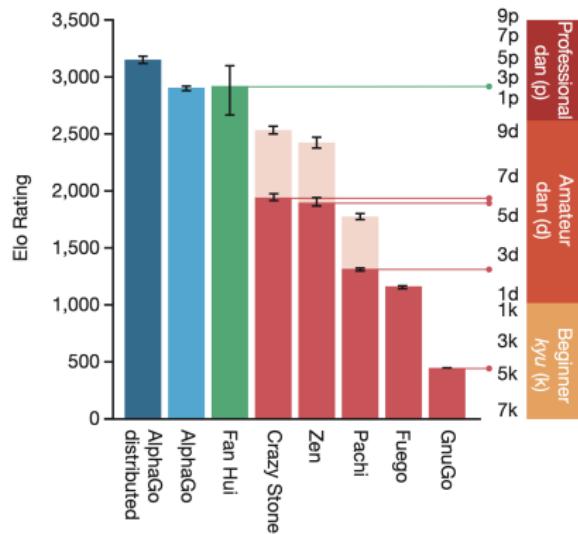
# The challenge of Go

Following DeepBlue's victory in 1997, Go was the next challenge

- branching  $\approx 35 \rightarrow \approx 350$
- game length  $\approx 57$  moves  $\rightarrow \approx 300$  moves
- popular: 4000+ years old and  $\approx 27M$  players worldwide



# AlphaGo



Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. and Dieleman, S., 2016. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587), pp.484-489.

Idea:

- Use MCTS as the base algorithm
- Capture the existing human knowledge in a policy  $p_\sigma(a|s)$
- Learn a fast simulation policy  $p_\pi(a|s)$  for rollouts
- Use RL techniques to optimize policy  $p_\rho(a|s)$  in self-play
- Use RL to learn a value function  $v(s)$
- Guide MCTS by policies and combine simulations with  $v$

## Supervised learning of human policy

- Data:  $(s_i, a_i)$  for 30 million positions from KGS Go Server
- Stochastic Gradient Ascent maximizing  $\mathbb{E}_i \log p_\sigma(a_i|s_i)$
- Final prediction accuracy was 57%
- 1000x faster roll-out policy  $p_\pi$  trained the same achieved 24% accuracy

## Improving policy in self-play

- Initialise  $p_\rho$  by  $p_\sigma$
- Play one match  $s_1, \dots, s_T$  and receive outcome  $z \in \{-1, 1\}$
- Use SGA to maximize  $\mathbb{E}_{t < T} \log p_\sigma(a_t|s_t)z$
- Eventually  $p_\rho$  wins over  $p_\sigma$  in 80% of games

The goal is to estimate state value under policy  $p_\rho$ :

$$v^{p_\rho}(s) = \mathbb{E}[z|s_t = s, a_{t\dots T} \sim p_\rho]$$

- Data:  $(s_i, z_i)$  for 30 million self-play games (only per game)
- Use Stochastic Gradient Descent to minimize  $\mathbb{E}_i(v(s_i) - z)^2$
- Resulting  $v$  consistently more accurate than  $p_\pi$  rollouts

# AlphaGo – search

Selection:

$$a_t = \arg \max_a \left( Q(s_t, a) + c \frac{p_\sigma(a|s_t) \sqrt{N(s_t)}}{1 + N(s_t, a)} \right)$$

Expansion:

“leaf node may be expanded” hence, likely not always

Simulation:

The result of the value function and simulation  $z \sim p_\pi$  is combined

$$V(s_L) = (1 - \lambda)v(s_L) + \lambda z$$

Backpropagation:

For all visited  $(s_t, a_t)$

$$N(s_t, a_t) += 1$$

$$Q(s_t, a_t) += \frac{1}{N(s_t, a_t)} V(s_L)$$

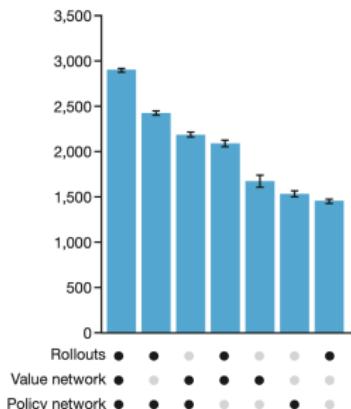
# AlphaGo – results

AlphaGo won 494/496 matches against the existing programs  
AlphaGo won 5 – 0 against professional European champion

Larger distributed version on 1202 CPUs and 176 GPUs

## Observations:

- All components are important →
- AlphaGo evaluated thousands times less positions than DeepBlue
- 10 years earlier than expected
- Human policies still helped in 2015



## AlphaGo Zero (2017)

- No human knowledge

## AlphaZero (2018)

- No simulation
- Chess: 9 hours, shogi: 12 hours, Go: 13 days

## MuZero (2020)

- Not even game rules are necessary

Imperfect information games?

Common games are large

If you can create a good evaluation function, use  $\alpha\beta$  variants

If it is hard to provide evaluation function, use MCTS

If you do not mind a lot of training, combine MCTS with learned policy and value functions

Playing perfect information games is mostly a solved problem

# Lecture 9: Constraint Satisfaction Programming and Scheduling

Viliam Lisý & Branislav Bošanský

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

bosansky@fel.cvut.cz

February, 2021

What we have covered so far:

- (un)informed search
- reinforcement learning
- two-player games

In all these problems, we have not assumed that states of the world have some specific structure.

## Question

What if we restrict the structure of the states?

## Question

What if we restrict the structure of the states?

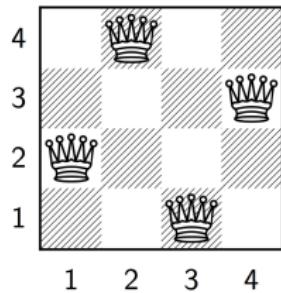
- we lose generality (not every problem could be represented)
- + we gain performance (we will be able to solve much larger problems)

We can identify and solve (exactly!) instances of a subclass of problems and improve scalability by several orders of magnitude compared to standard search algorithms.

# N-Queens Example

Consider an N-Queens problem: Place on a chessboard of size  $N \times N$  squares  $N$  queens so that no two queens threaten each other. For  $N = 4$ :

What would be the state representation?



- $N$  coordinates (one tuple of coordinates for each queen)
- $N$  numbers (every queen has to be in a different column, we can only represent rows)

Action changes position of one (or more) queen.

Differences from previous (general) problems:

- there is no start state (we can start from any state), hence
- the path to the goal state is not interesting, only the goal state itself

# Constraint Satisfaction Problems (CSPs)

The class of problems that include the N-Queens problems are known as CSPs (subclass of NP-complete problems).

CSPs are defined by 3 finite sets:

- **variables** ( $x_1, x_2, \dots, x_n$ )
- **domains** ( $D_i$  for each variable  $x_i$ )
- **constraints** ( $c_1, c_2, \dots, c_m$ )

A constraint is specified as a tuple of

- subset of variables  $x_{j_1}, \dots, x_{j_l}$
- all allowed joint assignments ( $l$ -tuples from  $D_{j_1}, \dots, D_{j_l}$ )

**Goal:** find such an assignment values to variables that satisfy all the constraints

# Constraint Satisfaction Problems – Examples

Many problems can be represented as CSPs. These include known puzzles:

- Sudoku,
- Cryptarithmic,

essential NP problems:

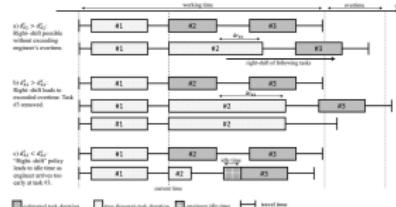
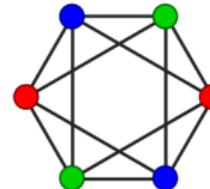
- SAT,
- Graph Coloring,

and many practical problems:

- Scheduling

A 9x9 grid puzzle with some numbers filled in. To its right is a cryptarithmic addition problem:

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$



# N-Queens Example as a CSP

We can formulate N-Queens problem as a CSP:

- **variables:**  $x_1, \dots, x_N$  (one variable for each queen, queen  $i$  is placed in the  $i$ -th column)
- **domains:**  $D_i = \{1, \dots, N\}$  (the row in which the queen is placed)
- **constraints:**
  - $x_i \neq x_j \quad \forall i, j \in \{1, \dots, N\}, i \neq j$   
(some solvers support global constraint **alldifferent**( $x_1, \dots, x_N$ ))
  - $|x_i - x_j| \neq |i - j| \quad \forall i, j \in \{1, \dots, N\}, i \neq j$

## Question

How do we search for a solution?

# Search Tree for CSPs

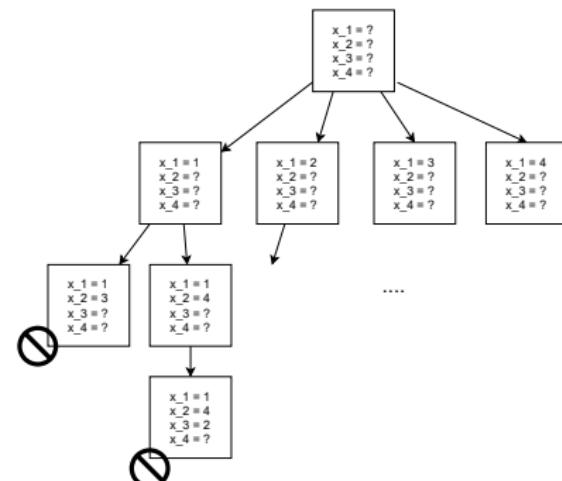
We use (uninformed) search as we know it (for now) and represent the search space as a search tree.

What are the nodes and actions in the search tree for a CSP?

- **Nodes** in the search tree – (partial) assignment of values to variables,
- **Edges** – choosing an unassigned variable and assign a value to this variable.

During the assignment, the algorithm must check whether the assignment does not violate constraints.

If there is no satisfying assignment, the algorithm backtracks.



We now move to specific CSP algorithms. Many of them assume only **binary constraints**.

## Question

Is it a problem? Is it a subclass of CSP problems?

Not really, we can reformulate any  $k$ -ary constraint as a set of binary constraints:

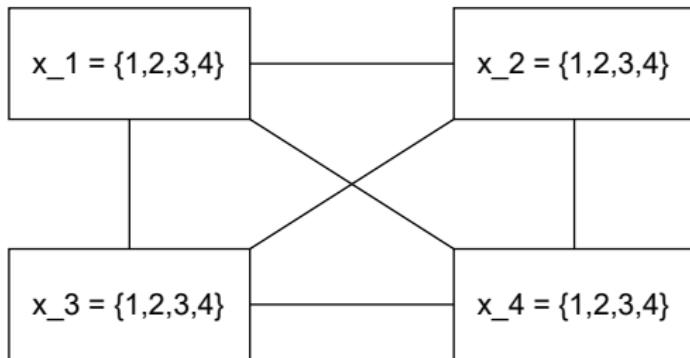
- Assume there is a constraint  $c$  involving  $k$  variables. Let  $\Gamma$  be the set of all  $k$ -tuples that satisfy this constraint.
- Create a new variable  $x_c$  with the domain  $\Gamma$  and create  $k$  binary constraints with involved  $k$  variables, such that  $i$ -th item of the value of  $x_c$  equals to value of the variable  $i$ .

# Standard Representation of CSPs – Visualization

Having only binary constraints, we can visualize CSPs as graphs:

- variables are vertices in the graph,
- constraints are edges in the graph.

There is an edge connecting two vertices if there is a constraint between these variables.

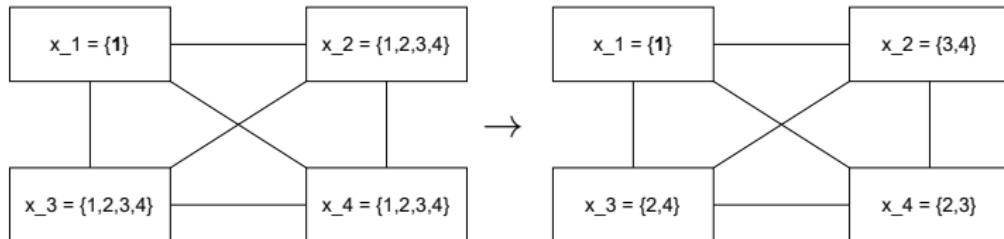


# CSP Search and Propagating Constraints

Can we utilize the fact that we have a specific structure of the problem?

The simple search checks the constraints only in a passive way.

We can propagate the values to other variables. Every time we set a value for some variable, we can filter out values of other variables that do not satisfy constraints → **forward checking**:



# CSP Search and Propagating Constraints

Assume the search algorithm selects a value for variable  $x_i$ . Now:

- for every other variable  $x_j$  such that there is a constraint  $c_{ij}$  between  $x_i$  and  $x_j$ , we evaluate all available values from  $D_j$  and keep only those that satisfy  $c_{ij}$

How is the forward checking integrated into the search algorithm?

- the algorithm keeps available values for every variable
- if for any variable its domain is empty after the forward checking, the algorithm immediately backtracks

First heuristic → **minimal remaining value (MRV)**.

So far, there was no rule which variable to choose next in the search tree. MRV heuristics is a fail-fast heuristic that can quickly prune out dead-ends.

# Search with Forward Checking

pseudocode of the search algorithm:

- **if** all variables are assigned **then return** current assignment (solution)
- $x_i \leftarrow \text{ChooseVariable}(X, D)$
- for each  $v \in D_i$ 
  - assign  $x_i = v$
  - valid = ForwardChecking( $X, D, i, v$ )
  - **if** valid **then** search( $X, D$ )
  - undo local assignments
- **return** false

# Towards a Better Use of Constraints

Forward checking ensures that there are supporting values in domains of other involved constraints.

The algorithm removes those values that do not satisfy the constraints.

But this can violate some other constraints ... Is there a way we can ensure that every constraint **can be satisfied** (termed **consistent**)?

Yes! We can have an algorithm that makes every edge (constraint) consistent.

# Arc Consistency

Making one edge (arc)  $c_{ij}$  consistent:

- deleted = false
- **for each**  $v \in D_i$ 
  - supported = false
  - **for each**  $v' \in D_j$ 
    - if  $c_{ij}(v, v')$  **then** supported = true
  - **if not** supported **then**
    - remove  $v$  from  $D_i$
    - deleted = true
- **return** deleted

The procedure checks one constraint (in a directed manner) and returns true if some value was removed from domain  $D_i$ .

## Arc Consistency – AC-3

Assume the algorithm has set value for variable  $x_i$ . We need to make consistent all incoming edges to node  $i$  (constraints that depend on this selected value). Next, if some value is removed from any variable  $x_j$ , we need to do the same for node  $j$ .

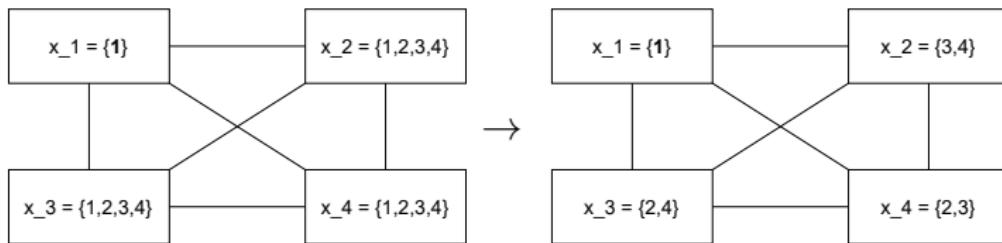
We will have a queue  $Q$  of all edges to make consistent:

- $Q = \{(j, i) \mid c_{ji} \in C, i \neq j\}$
- **while**  $Q$  is **not empty**
  - $(a, b) = \text{pop}(Q)$
  - **if**  $\text{MakeConsistent}(a, b)$  **then**
    - append( $Q, \{(k, a) \mid c_{ka} \in C, k \neq a\}$ )

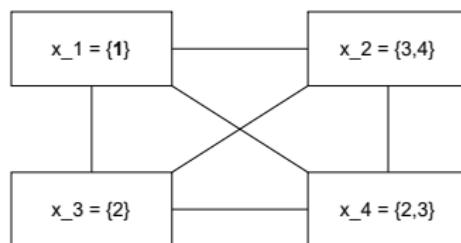
This algorithm is known as **AC-3**.

# AC3 Algorithm – Example

Step 1: making consistent all edges  $(n, 1)$  for  $n = \{2, 3, 4\}$



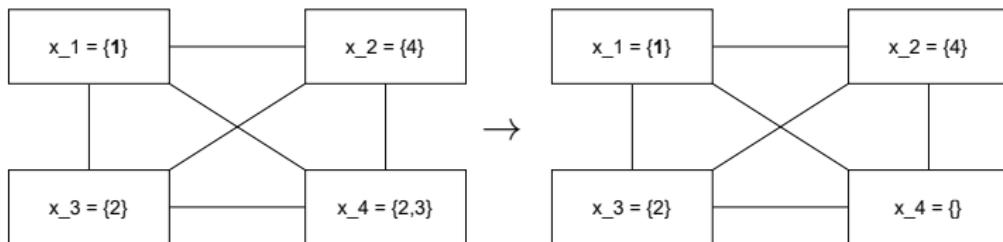
Step 2: making consistent all edges  $(n, 2)$  (AC3 deleted from  $D_2$ )



Value 4 is removed from  $D_3$  since it is not supported by any value in  $D_2$ .

# AC3 Algorithm – Example

Step 3: making consistent all edges ( $n, 3$ )



Value 3 is removed from  $D_2$  since it is not consistent with  $x_3 = 2$ .

Next, all values in  $D_4$  are removed since neither of them is consistent with  $x_3 = 2 \rightarrow$  no solution!

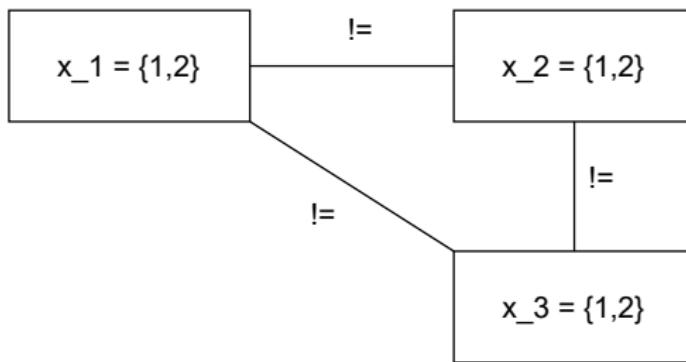
The AC3 algorithm can determine after the first assignment  $x_1 = 1$  that this action does not lead to goal.

# AC3 Algorithm

## Question

Does AC3 solve everything? Do we still need search?

Unfortunately, AC3 is not able to guarantee there exists a solution. If AC3 prunes out some domain, the search algorithm can safely backtrack. Otherwise, the search needs to continue.



## Least Constraining Value

Another heuristic for CSPs – among all the values to be assigned to a variable, choose such that supports the most other values.

## Backjumping

Inability of choosing valid value for one variable can be caused by a choice of a variable up in the search tree. → The algorithm can identify which variables cause the conflict and can backtrack immediately to this conflicting variable (jumping back).

## Dynamic Backtracking

In backjumping, the assignment between two conflicting variables is lost if we jump (even if it was a good one) → dynamic backtracking can dynamically choose which variable to assign (or re-assign) so that partially valid solutions are not lost.

Constraints in CSPs are **hard constraints** – they need to be satisfied to find a solution.

Often, not all constraints have to be hard – we can combine CSPs with an objective function representing **soft constraints**.

For example in scheduling – we cannot plan execution of two jobs on a single machine (hard constrain) but we want to minimize time required to finish all the jobs (objective function).

After finding a solution, we can keep it and continue searching – current solution is a lower bound on the optimal value (if we maximize), hence we can add additional pruning – **branch and bound** (similar to alpha beta pruning).

There are many additional modifications and improvements regarding CSP.

There are many solvers that you can use.

Again, the ideas from CSP algorithms can be used also elsewhere (using specific structure to prune out not perspective branches, exploiting locality of partial solution in dynamic backtracking, etc.)

A good source for a quick reference – [CSP course at MFF CUNI](#).

# Lecture 10: Logical Agents and Planning

Viliam Lisý & Branislav Bošanský

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

April, 2021

# Plan of today's lecture

- ① Logic in AI in the past and now
- ② Logical problem representations
- ③ Situation calculus
- ④ Intelligent planning

# Acknowledgements

Slides are heavily based on J. Klema's slides. For more details on logical agents see his [video](#) from the last year.

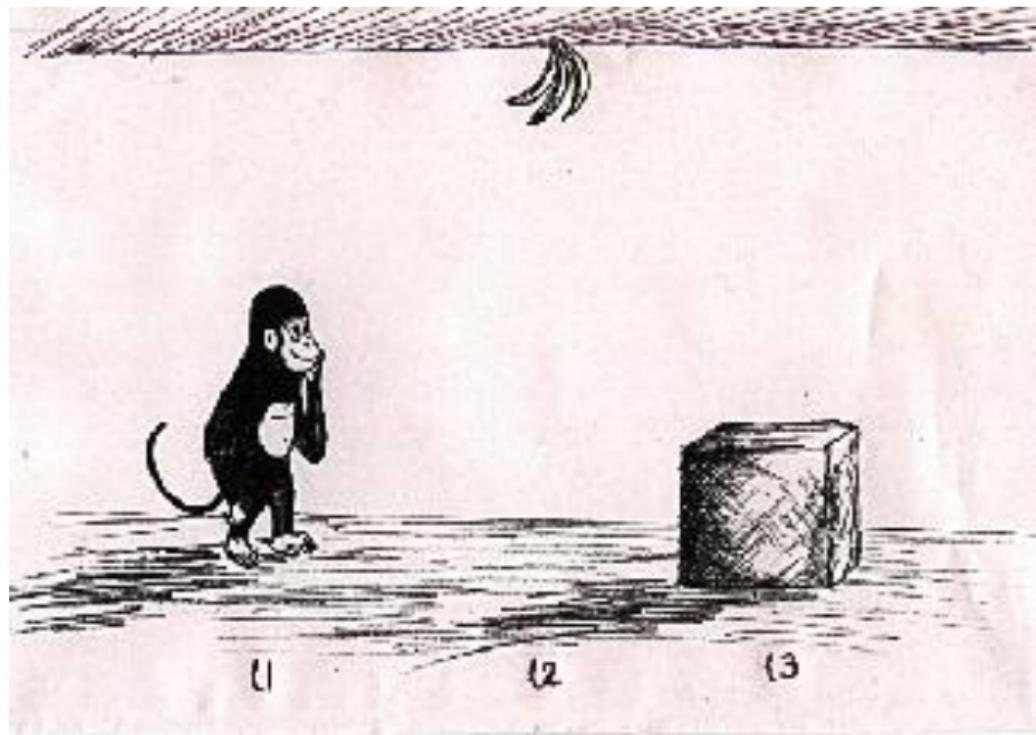
There has been a big hype of logical agents in 60s and 70s.

- + It can represent knowledge about the world
- + It can represent intelligent reasoning
- It is not very convenient for working with uncertainty
- It is usually extremely computationally expensive
  - ( expressivity vs. completeness vs. effectivity )

Logic in AI 2020s

- Interpretable AI
- Relational ML/RL
- Theorem proving
- Model checking
- Knowledge graphs
- Automated planning

# Motivation example – monkey and banana



Vladimir Lifschitz: Planning course, The University of Texas at Austin.

# Motivation example – monkey and banana

## Problem description

- a monkey is in a room, a banana hangs from the ceiling,
- the banana is beyond the monkey's reach,
- the monkey is able to walk, move and climb objects, grasp banana,
- the room is just the right height so that the monkey can move a box, climb it and grasp the banana,
- the goal is to generate this plan (sequence of actions) automatically.

## Key characteristics

- a deterministic task
- a general description available
  - all the necessary knowledge is provided
  - we need to **represent it** in some **language**
  - and perform certain **reasoning / inference**
- a planning task

Remember B0B01LGR: Logic and Graphs

## Jazyk

*Jazyk predikátové logiky* obsahuje tyto symboly:

① logické symboly

- proměnné; Var je množina všech proměnných
- logické spojky:  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\Rightarrow$ ,  $\Leftrightarrow$ , popř. též  $\text{tt}$ ,  $\text{ff}$ ,  $|$ ,  $\downarrow$ ,  $\oplus$
- kvantifikátory  $\forall$  (obecný) a  $\exists$  (existenční)
- symbol rovnosti:  $=$

② speciální symboly

- predikátové, kde každý má svou aritu  $n \geq 0$ ;  
Pred je množina predikátových symbolů
- funkční, kde každý má svou aritu  $n > 0$ ;  
Func je množina funkčních symbolů
- konstantní; Kons je množina konstantních symbolů

③ pomocné symboly, jako jsou závorky (, ) a čárka ,

The following slides would, in principle, work with stronger logic!  
Modal Logic, epistemic logic, temporal logic, ATL

# Planning problem representation in FOL

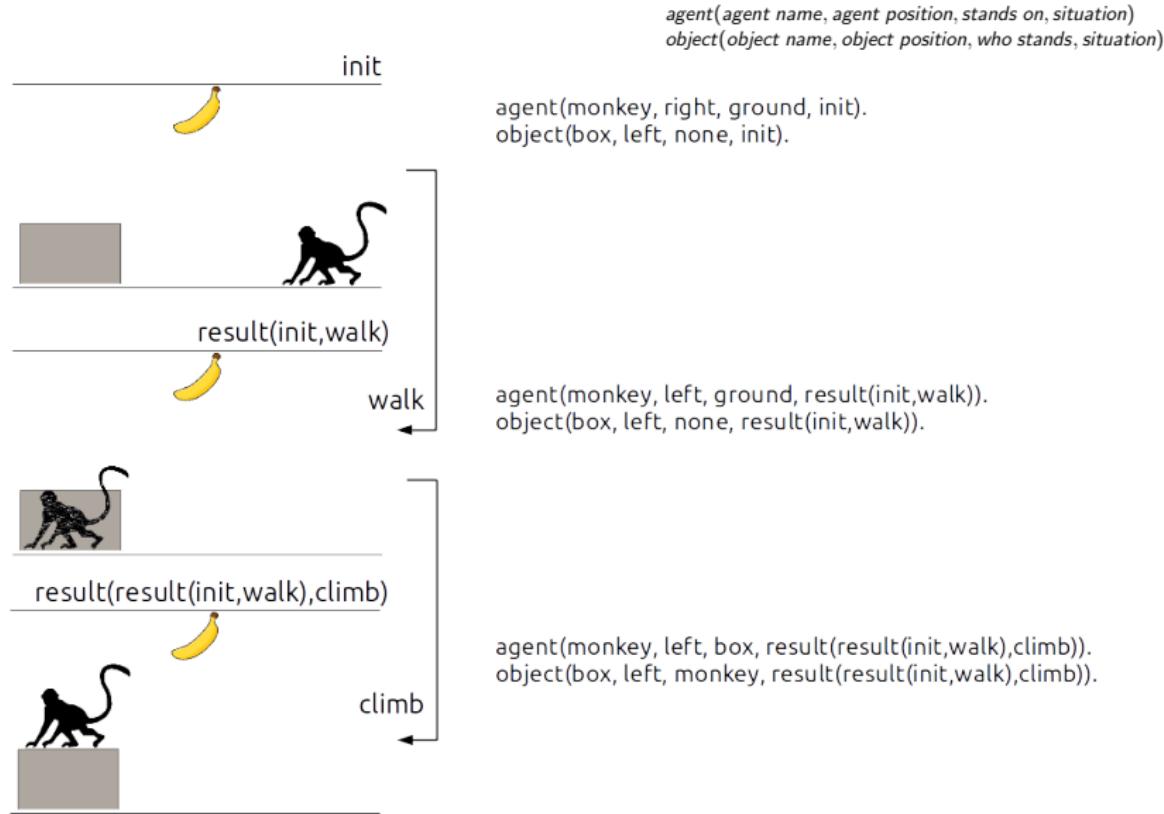
**Situation calculus** is one way to represent changing world in FOL

- facts hold in particular situations ( $\approx$  world state histories)
- predicates either rigid (eternal) or fluent (changing)
- fluent predicates include a situation argument
  - e.g.,  $agent(monkey, at\_ban, now)$ , term *now* denotes a situation
- rigid predicates hold regardless of a situation
  - e.g.,  $walks(monkey)$ ,  $moveable(box)$
- situations are connected by the *result* function
  - if *s* is a situation then  $result(s, a)$  is also a situation

The monkey problem state can be represented using two predicates

- $agent(agent\ name, agent\ position, stands\ on, situation)$
- $object(object\ name, object\ position, who\ stands, situation)$

# Keeping track of evolving situations



# Description and application of actions

*agent(agent name, agent position, stands on, situation)  
object(object name, object position, who stands, situation)*

Action “effect” axiom for  $walk(X, P_1, P_2)$ :

$$\begin{aligned} \forall X, P_1, P_2, Z \ (\textit{agent}(X, P_1, \textit{ground}, Z) \wedge \textit{walks}(X) \\ \rightarrow \textit{agent}(X, P_2, \textit{ground}, \textit{result}(Z, \textit{walk}(X, P_1, P_2))) \end{aligned}$$

Action “effect” axiom for  $climb(X)$ :

$$\begin{aligned} \forall X, P, Z \ (\textit{agent}(X, P, \textit{ground}, Z) \wedge \textit{object}(box, P, \textit{none}, Z) \\ \rightarrow \textit{agent}(X, P, \textit{box}, \textit{result}(Z, \textit{climb}(X))) \\ \wedge \textit{object}(box, P, X, \textit{result}(Z, \textit{climb}(X))) \end{aligned}$$

# Frame problem

Action axioms describe how fulents change between situations

What happens to fluents, which are not used in the actions?

e.g., the objects while the agent walks

**Frame problem:** how to cope with the unchanged facts smartly

- many “frame” axioms may be necessary to express them in FOL

$$\forall X, V, W, Z, P_1, P_2$$
$$(object(X, V, Y, Z) \rightarrow object(X, V, Y, result(Z, walk(P_1, P_2))))$$

- $f$  fluent predicates,  $a$  actions requires  $O(f \cdot a)$  frame axioms
- many applications of axioms each step is computationally expensive
- some tricks diminish the problem, but it never goes away

# Logical planning

We already have representations of **states** and **actions**

Goal of planning: logical representation of the desired state

$$\mathcal{G} \equiv \exists Z \text{ agent}(monkey, middle, box, Z)$$

**Reasoning** checks whether the goal formula follows from KB

$$KB \models \mathcal{G}$$

- knowledge base (KB) are the inference rules and the initial state
- reasoning finds a suitable  $Z$  or proves it does not exist
- desirable properties: **soundness**, **completeness**, **efficiency**
- reasoning procedures: **resolution**, deductive inference, etc.
  - see B0B01LGR
  - generally extremely computationally hard, possibly undecidable
  - the solution is correct, if reasoning successfully finishes
  - can be efficient and useful with **additional restrictions**

Subfield of AI dealing (mainly) with

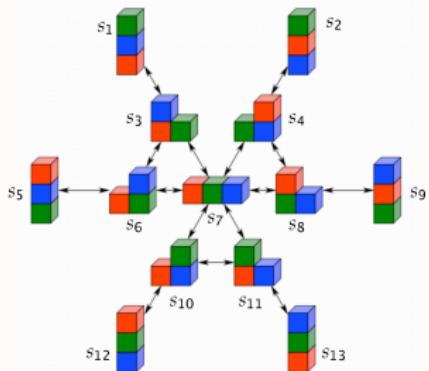
- representation languages with reasonable tradeoffs of expressivity and efficiency
- algorithms for finding plans for problems expressed in these languages

(The following slides are heavily based on Carmel Domshlak's slides)

# Planning problems

What is in common?

- All these problems deal with **action selection** or **control**
- Some notion of problem **state**
- (Often) specification of **initial state** and/or **goal state**
- Legal moves or **actions** that transform states into other state



# Planning task

For now focus on:

- Plans (aka **solutions**) are sequences of moves that transform the initial state into the goal state
- Intuitively, not all solutions are equally desirable

What is our task?

- ① Find out whether there is a solution
- ② Find any solution
- ③ Find an optimal (or near-optimal) solution
- ④ Fixed amount of time, find best solution possible
- ⑤ Find solution that satisfy property  $\aleph$  (what is  $\aleph$ ? you choose!)

# Three Key Ingredients of Planning

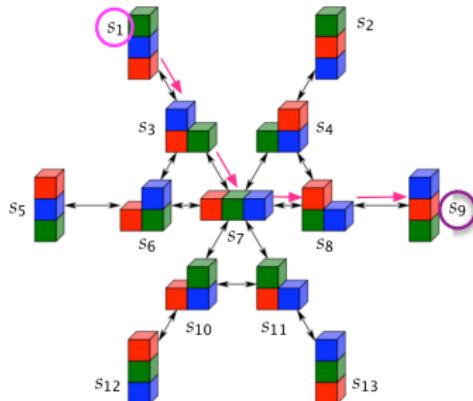
Planning is a form of **general problem solving**

Problem  $\Rightarrow$  Language  $\Rightarrow$  **Planner**  $\Rightarrow$  Solution

- ① **models** for defining, classifying, and understanding problems
  - what is a *planning problem*
  - what is a *solution (plan)*, and
  - what is an *optimal solution*
- ② **languages** for representing problems
- ③ **algorithms** for solving them

# Why planning is difficult?

- Solutions to planning problems are paths from an initial state to a goal state in the transition graph
- Dijkstra's algorithm solves this problem in  $O(|V| \log (|V|) + |E|)$
- Can we go home??



- Solutions to planning

# What is “classical” planning?

- dynamics: **deterministic**, nondeterministic or probabilistic
  - observability: full, partial or **none**
  - horizon: **finite** or infinite
  - ...
- ① **classical planning**
  - ② conditional planning with full observability
  - ③ conditional planning with partial observability
  - ④ conformant planning
  - ⑤ Markov decision processes (MDP)
  - ⑥ partially observable MDPs (POMDP)

- More **compact** representation of actions than as relations is often
  - **possible** because of symmetries and other regularities,
  - **unavoidable** because the relations are too big.
- Represent different aspects of the world in terms of different **state variables**.  $\rightsquigarrow$  A state is a **valuation of state variables**.
- Represent actions in terms of changes to the state variables.

## Key issue

Models represented **implicitly** in a **declarative language**

Play two roles

- **specification:** concise model description
- **computation:** reveal useful info about problem's *structure*

# The STRIPS language

A problem in **STRIPS** is a tuple  $\langle P, A, I, G \rangle$

- $P$  stands for a finite set of **atoms** (boolean vars)
- $I \subseteq P$  stands for **initial situation**
- $G \subseteq P$  stands for **goal situation**
- $A$  is a finite set of **actions**  $a$  specified via  $\text{pre}(a)$ ,  $\text{add}(a)$ , and  $\text{del}(a)$ , all subsets of  $P$

- States are **collections of atoms**
- An action  $a$  is applicable in a state  $s$  iff  $\text{pre}(a) \subseteq s$
- Applying an applicable action  $a$  at  $s$  results in  
$$s' = (s \setminus \text{del}(a)) \cup \text{add}(a)$$

# Why STRIPS is interesting?

- STRIPS operators are **particularly simple**, yet expressive enough to capture general planning problems.
- In particular, STRIPS planning is **no easier** than general planning problems.
- Many algorithms in the planning literature are **easier to present in terms of STRIPS**.

(The following example is based on Antonin Komanda's slides)

# Sokoban - Example planning domain

State representation:

```
positions: a1, ... a6,...  
          f1, ..., f2  
box_at(P), free(P)  
player_at(P)  
adjacent(P1,P2)  
adjacent2(P1,P2)
```

Operators (Actions):

```
move(X,Y):  
    pre: player_at(X)  
          adjacent(X,Y)  
          free(Y)  
    add: player_at(Y)  
    del: player_at(X)  
  
push(X, Y, Z):  
    pre: player_at(X)  
          box_at(Y)  
          free(Z)  
          adjacent(X,Y)  
          adjacent(Y,Z)  
          adjacent2(X,Z)  
  
...
```



# Grounding of Actions

## Operators (Actions):

```
move(X, Y):  
    pre: player_at(X)  
          adjacent(X, Y)  
          free(Y)  
    add: player_at(Y)  
    del: player_at(X)  
  
push(X, Y, Z):  
    pre: player_at(X)  
          box_at(Y)  
          free(Z)  
          adjacent(X, Y)  
          adjacent(Y, Z)  
          adjacent2(X, Z)  
    add: player_at(Y)  
          box_at(Z)  
          free(Y)  
    del: player_at(X)  
          box_at(Y)  
          free(Z)
```

## Grounding:

```
move_a1_a2  
    pre: player_at_a1, adjacent_a1_a2, free_a2  
    add: player_at_a2  
    del: player_at_a1  
  
move_a2_a3  
    pre: player_at_a2, adjacent_a2_a3, free_a3  
    add: player_at_a3  
    del: player_at_a2  
  
...  
  
push_a1_a2_a3  
    pre: player_at_a1, box_at_a2, free_a3  
          adjacent_a1_a2, adjacent_a2_a3,  
          adjacent_a1_a3  
    add: player_at_a2, box_at_a3, free_a2  
    del: player_at_a1, box_at_a2, free_a3  
...
```

# STRIPS Representation of Sokoban

A problem in **STRIPS** is a tuple  $\langle P, A, I, G \rangle$

- $P$  stands for a finite set of **atoms** (boolean vars)
- $I \subseteq P$  stands for **initial situation**
- $G \subseteq P$  stands for **goal situation**
- $A$  is a finite set of **actions**  $a$  specified via  $\text{pre}(a)$ ,  $\text{add}(a)$ , and  $\text{del}(a)$ , all subsets of  $P$

```
P = {player_at_a2, ..., player_at_d3,  
     box_at_a2, ..., box_at_d3,  
     free_a2, ..., free_d3,  
     adjacent_a2_b2, ..., adjacent_d2_d3,  
     adjacent2_a2_c2, ..., adjacent2_d1_d3 }
```

```
I = {player_at_b2, box_at_c1, box_at_c2,  
     free_a2, free_b1, ..., free_d3,  
     adjacent_a2_b2, ..., adjacent_d2_d3, adjacent2_a2_c2, ..., adjacent2_d1_d3}
```

```
G = {box_at_a2, box_at_d1}
```



We can just use A\*:

- State: a set of true atoms
- Applicable actions: based on preconditions
- Action application: add the “add” atoms and delete the “del” atoms  
(No need for separate simulator implementation)

Problem structure allows **automated** construction of **heuristics**!

- Allows exploring general heuristics domain independently
- Simple heuristic:  $h(s) = |G \setminus s|$
- Solve a suitable **simpler** version of the problem
- Abstraction: solve a smaller problem
  - e.g., completely remove a predicate from the problem
- **Relaxation**: solve a less constraint problem
- Landmarks

# Relaxation heuristics

Whole sub-field of planning in STRIPs and beyond

- Relaxation is a general technique for heuristic design:
  - **Straight-line heuristic** (route planning): Ignore the fact that one must stay on roads.
  - **Manhattan heuristic** (15-puzzle): Ignore the fact that one cannot move through occupied tiles.
- We want to apply the idea of relaxations to planning.
- Informally, we want to ignore **bad side effects** of applying actions.

## Example (8-puzzle)

If we move a tile from  $x$  to  $y$ , then the **good effect** is (in particular) that  $x$  is now free.

The **bad effect** is that  $y$  is not free anymore, preventing us from moving tiles through it.

In STRIPS, good and bad effects are easy to distinguish:

- Effects that make atoms true are good  
*(add effects)*.
- Effects that make atoms false are bad  
*(delete effects)*.

Idea for the heuristic: *Ignore all delete effects.*

# Relaxed planning tasks in STRIPS

## Definition (relaxation of actions)

The **relaxation**  $a^+$  of a STRIPS action

$a = \langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle$  is the action

$a^+ = \langle \text{pre}(a), \text{add}(a), \emptyset \rangle$ .

## Definition (relaxation of planning tasks)

The **relaxation**  $\Pi^+$  of a STRIPS planning task  $\Pi = \langle P, A, I, G \rangle$

is the planning task  $\Pi^+ := \langle P, \{a^+ \mid a \in A\}, I, G \rangle$ .

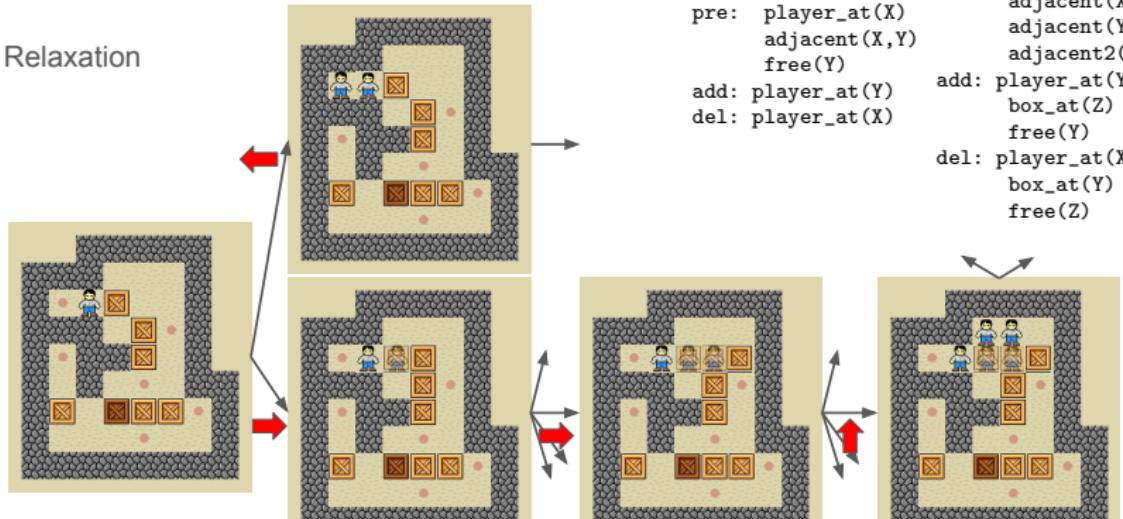
## Definition (relaxation of action sequences)

The **relaxation** of an action sequence  $\pi = a_1 \dots a_n$  is the action

sequence  $\pi^+ := a_1^+ \dots a_n^+$ .

# Relaxation of actions in Sokoban

Relaxation



```
push(X, Y, Z):  
    pre: player_at(X)  
         box_at(Y)  
         free(Z)  
  
move(X, Y):  
    pre: player_at(X)  
         adjacent(X, Y)  
         free(Y)  
    add: player_at(Y)  
    del: player_at(X)  
  
    add: player_at(Y)  
         box_at(Z)  
         free(Y)  
    del: player_at(X)  
         box_at(Y)  
         free(Z)
```

# Building Relaxed Planning Graph

Computing the optimal relaxed plan is still NP hard

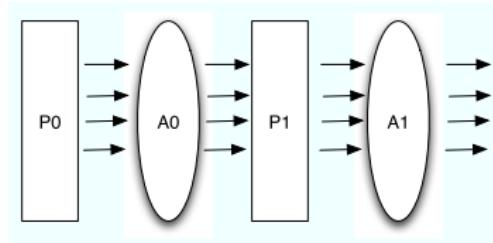
But we can do something simpler

- Build a layered **reachability graph**  $P_0, A_0, P_1, A_1, \dots$

$$P_0 = \{p \in I\}$$

$$A_i = \{a \in A \mid \text{pre}(a) \subseteq P_i\}$$

$$P_{i+1} = P_i \cup \{p \in \text{add}(a) \mid a \in A_i\}$$



- Terminate when  $G \subseteq P_i$

## Example

$$I = \{a = 1, b = 0, c = 0, d = 0, e = 0, f = 0, g = 0, h = 0\}$$

$$a_1 = \langle \{a\}, \{b, c\}, \emptyset \rangle$$

$$a_2 = \langle \{a, c\}, \{d\}, \emptyset \rangle$$

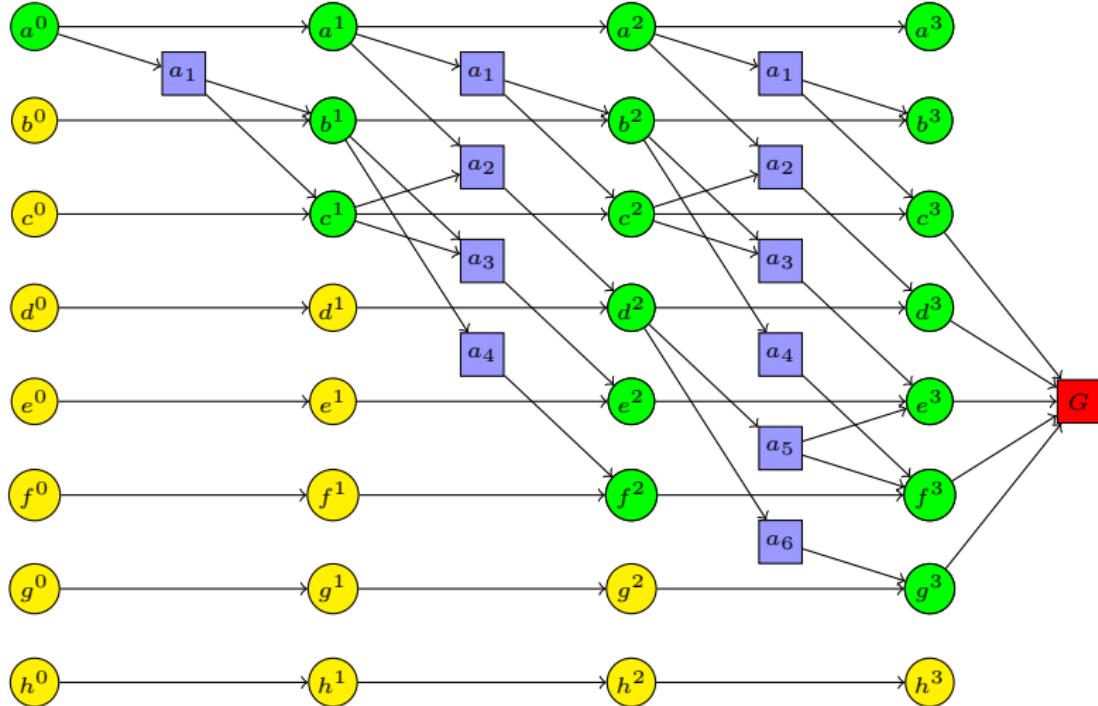
$$a_3 = \langle \{b, c\}, \{e\}, \emptyset \rangle$$

$$a_4 = \langle \{b\}, \{f\}, \emptyset \rangle$$

$$a_5 = \langle \{d\}, \{g\}, \emptyset \rangle$$

$$G = \{c = 1, d = 1, e = 1, f = 1, g = 1\}$$

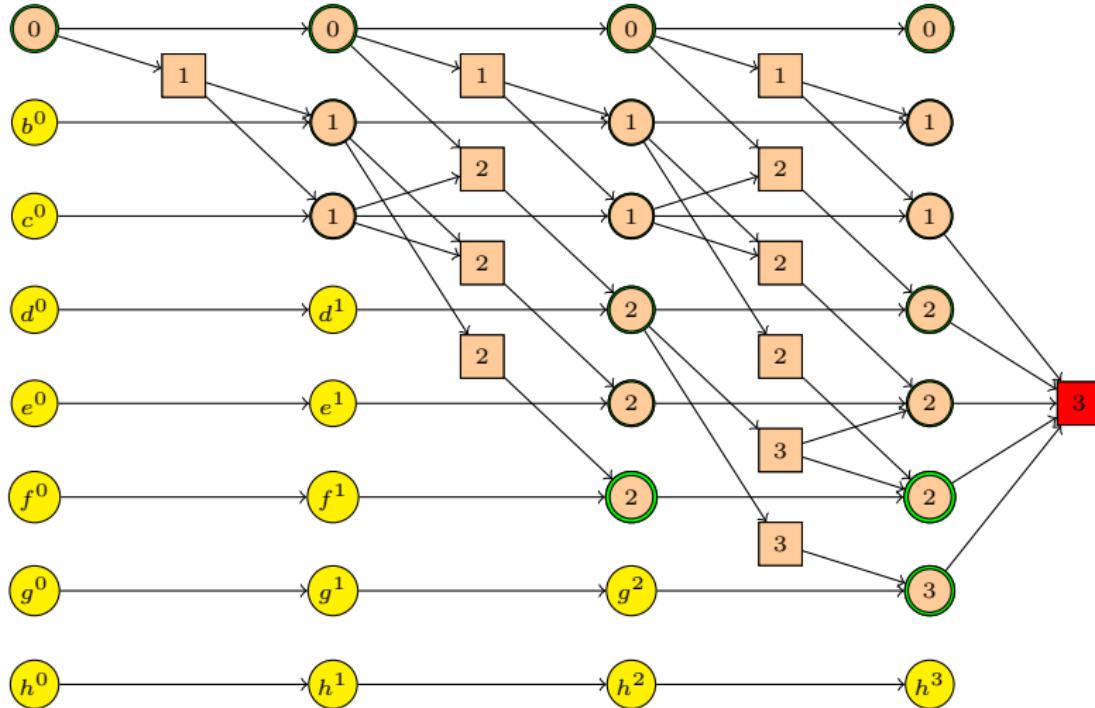
# Relaxed Planning Graph



## Forward cost heuristic $h_{max}$

- Propagate cost layer by layer from start to goal
- At actions, take maximum cost of achieving preconditions +1
- At propositions, take the cheapest action to achieve it

# Computing heuristic $h_{max}$



Logic is a powerful language for describing AI problems

Situation calculus is a logical formalism for reasoning about situations developing in time

Of-the-shelf logical reasoning methods are usable for planning

However, expressivity goes against efficiency

AI planning creates logical representations and algorithms specially designed for planning

STRIPS is a simple, but powerful language for representing planning problems

Logical representation of problems allows automated construction of A\* heuristics

# Lecture 11: Rational Decision Making with Uncertainty

Viliam Lisý & Branislav Bošanský

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

April, 2021

# Plan of today's lecture

- ① Rationality as expected utility maximization
- ② Reasoning with joint probability distribution
- ③ Bayesian networks
- ④ Decision networks

# Acknowledgements

Slides are closely following AIMA 3rd edition (mainly Ch. 13, 14)

Further based on:

- Percy Liang's [lecture](#)
- Patrick Winston's [lecture](#)
- Roman Bartak's [lecture](#)

# Rational agent

Rational agent chooses the actions that maximise its expected utility over all possible outcomes.

$$\begin{aligned} \textit{rational decisions} &= \textit{decision theory} \\ &= \mathbf{utility\ theory} + \mathbf{probability\ theory} \end{aligned}$$

Rationality and its limitation is often studied in the form of lotteries, e.g., Would you rather have:

- 20% chance of winning \$100 or
- 50% chance of winning \$20?

To build intelligent (rational) agents, we need to assess the utility and the probability of various events.

## Diagnosis support

- Medical, IT support, machinery service, etc.

## Robotic localisation: what is the position of the robot given

- Noisy actuators
- Multiple noisy sensors

## Natural language processing

- What is the topic of a text given its words?

# Basic (discrete) probability recapitulation

**Random variable:** sunshine  $S \in \{0, 1\}$ , rain  $R \in \{0, 1\}$ ,  
dice  $D \in \{1, 2, 3, 4, 5, 6\}$ .

**Joint distribution:**

$s$	$r$	$P(S = s, R = r)$
0	0	0.20
0	1	0.08
1	0	0.70
1	1	0.02

$$P(S, R) =$$

**Marginal distribution:**

$s$	$P(S = s)$
0	0.28
1	0.72

(sum rows)

**Conditional distribution:**

$s$	$P(S = s R = 1)$
0	0.8
1	0.2

(select rows + normalize)

# Basic probability statements

Random variables are exhaustive and mutually exclusive:

$$\sum_{a \in A} P(A = a) = 1$$

Inclusion-exclusion principle:

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b)$$

Product rule:

$$P(A, B) = P(A|B)P(B)$$

Bayes' rule:

$$P(A, B) = P(B, A)$$

$$P(A|B)P(B) = P(B|A)P(A)$$

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

# Inference Using Full Joint Distribution

Knowledge base:

	toothache		¬toothache	
	catch	¬catch	catch	¬catch
cavity	0.108	0.012	0.072	0.008
¬cavity	0.016	0.064	0.144	0.576

Updating belief based on evidence:

$$P(\text{cavity}|\text{toothache}) = \frac{P(\text{cavity} \wedge \text{toothache})}{P(\text{toothache})} = \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = 0.6$$

$$P(\neg\text{cavity}|\text{toothache}) = \frac{P(\neg\text{cavity} \wedge \text{toothache})}{P(\text{toothache})} = \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4$$

# The size of full joint distribution grows exponentially

Slightly more realistic example:

$Cavity \in \{true, false\}$	$Sex \in \{male, female\}$	$Hygiene \in$
$Toothache \in \{true, false\}$	$Age \in \{child, teen, adult, senior\}$	$History \in$
$Catch \in \{true, false\}$	$Diet \in \{omnivore, vegetarian, vegan\}$	...

The size of the joint probability table for problem with variables  $X_1, \dots, X_n$  is:

$$\prod_{i=1}^n |X_i| \geq 2^n$$

We need to:

- Store the data in memory
- Iterate over large portions of them to answer queries
- Obtain a probability for each cell!

# Absolute Independence

Assume variables:

$Cavity, Toothache, Catch, Weather \in \{cloudy, sunny, rain, snow\}$

The size of  $P(Cavity, Toothache, Catch, Weather)$  is  $2 \times 2 \times 2 \times 4 = 32$ .

We know that

$$P(Cavity, Toothache, Catch, Weather) =$$

$$P(Weather|Cavity, Toothache, Catch)P(Cavity, Toothache, Catch)$$

Dental problems do not influence the weather, hence:

$$P(Weather|Cavity, Toothache, Catch) = P(Weather)$$

Therefore without loss of precision, we can represent

$$P(Cavity, Toothache, Catch, Weather) =$$

$$P(Cavity, Toothache, Catch) \quad \text{of size } 2 \times 2 \times 2 = 8$$

$$\cdot P(Weather) \quad \text{of size 4}$$

The overall size of the representation is  $8 + 4 = 12$ .

# Conditional Independence

Absolute independence is quite rare. We can use conditional independence to reduce the representation size further.  
When one has cavity, does catch depend on toothache?

$$P(\text{Catch} | \text{Toothache}, \text{Cavity}) = P(\text{Catch} | \text{Cavity})$$

Variables  $X$  and  $Y$  are independent given  $Z$ , if we any of the following holds:

$$P(X|Y, Z) = P(X|Z), P(Y|X, Z) = P(Y|Z), P(X, Y|Z) = P(X|Z)P(Y|Z)$$

$$\begin{aligned} P(\text{Toothache}, \text{Catch}, \text{Cavity}) &= \\ &P(\text{Toothache} | \text{Catch}, \text{Cavity})P(\text{Catch} | \text{Cavity})P(\text{Cavity}) = \\ &= P(\text{Toothache} | \text{Cavity}) \quad \text{of size } 2 \times 2 = 4 \\ &\quad *P(\text{Catch} | \text{Cavity}) \quad \text{of size } 2 \times 2 = 4 \\ &\quad *P(\text{Cavity}) \quad \text{of size } 2 \end{aligned}$$

It does not lead to savings here, but often does in large problems.

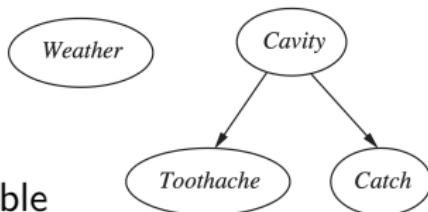
# Bayesian Network

Formal framework for compact representation and inference in large joint distributions.

It specifies the **conditional independence relationships among random variables** and the corresponding necessary joint distributions.

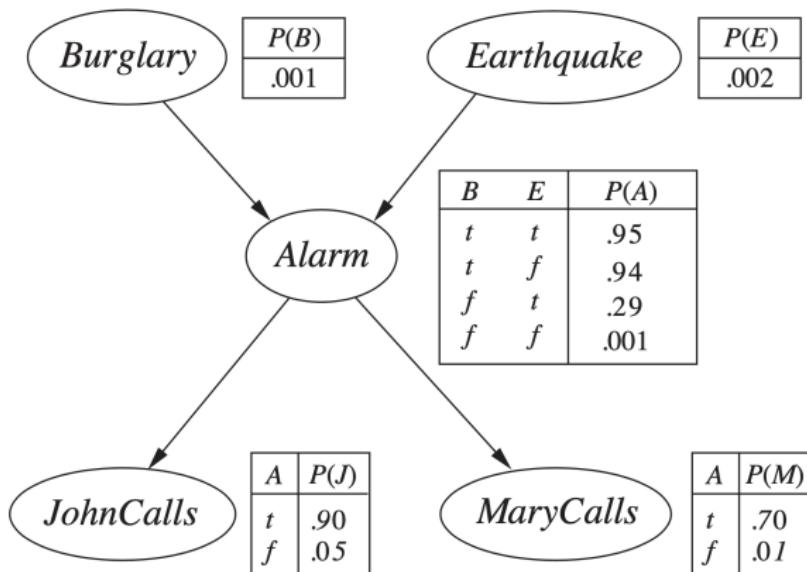
Bayesian network consists of:

- A graph node for each random variable
- Directed edges from parents to children represented direct influence of the children's values by the parent values. The edges form a Directed Acyclic Graph (DAG).
- For each node  $X_i$ , a conditional probability table  $P(X_i | \text{Parents}(X_i))$



# Bayesian Network Example

An **alarm** usually sounds when a **burglary** is in progress, but sometimes it is started by a minor **earthquake**. There are two neighbours which may here the alarm and call us and **John** is more likely to call than **Mary**.



# How does a BN represent the joint distribution?

From the chain rule (iterative application of the product rule) we know:

$$P(J, M, A, B, E) = P(J|M, A, B, E) * P(M|A, B, E) * P(A|B, E) * P(B|E) * P(E)$$

From conditional independence of individual variables

$$= P(J|A) * P(M|A) * P(A|B, E) * P(B) * P(E)$$

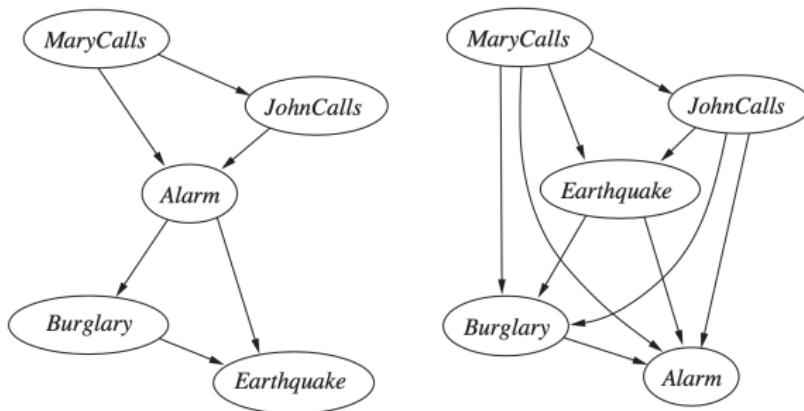
And these are exactly the tables included in the BN

$$= \prod_i P(X_i | Parents(X_i))$$

Since BN is a DAG, the topological ordering will always provide a correct ordering of the variables.

## Other structures may also represent the distribution

While the edges are easy to think about as **causality**, it is not necessarily the case.



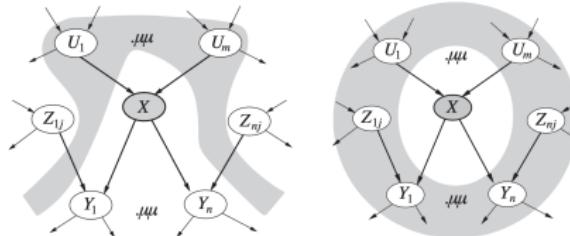
# Conditional independence in BNs

We required that only node's parents influence its value; hence

- (1) a node is conditionally independent of its predecessors, given its parents.

The BN's topology captures other independence relationships:

- (2) a node is conditionally independent of its non-descendants, given its parents;
- (3) a node is conditionally independent of all other nodes, given its parents, children, and children's parents.



Understanding independence speeds up the inference algorithms.

Task: Compute the **posterior** probability distribution over a set of **query variables ( $\mathbf{X}$ )**, given some observed assignments for **evidence variables ( $\mathbf{E} = \mathbf{e}$ )**.

Let  $\mathbf{Y}$  be the set of non-query and non-evidence variables, the task is

$$P(\mathbf{X}|\mathbf{e}) = \frac{P(\mathbf{X}, \mathbf{e})}{P(\mathbf{e})} = \alpha P(\mathbf{X}, \mathbf{e}) = \alpha \sum_{\mathbf{y} \in \mathbf{Y}} P(\mathbf{X}, \mathbf{e}, \mathbf{y}),$$

where the last joint distribution is represented by the BN as the product

$$\prod_i P(X_i | Parents(X_i)).$$

## Inference in BNs by enumeration

In the burglary example, assume that both John and Marry called and we are interested in the probability of the burglary.

$$\mathbf{X} = \{\text{Burglary}\}, \mathbf{E} = \{\text{MaryCalls}, \text{JohnCalls}\}, \mathbf{Y} = \{\text{Alarm}, \text{Earthquake}\}$$

Than from the previous

$$P(B|j, m) = \alpha P(B, j, m) = \alpha \sum_{e \in E} \sum_{a \in A} P(B, j, m, e, a)$$

For  $\text{Burglary} = \text{true}$

$$P(b|j, m) = \alpha \sum_{e \in E} \sum_{a \in A} P(b)P(e)P(a|b, e)P(j|a)P(m|a)$$

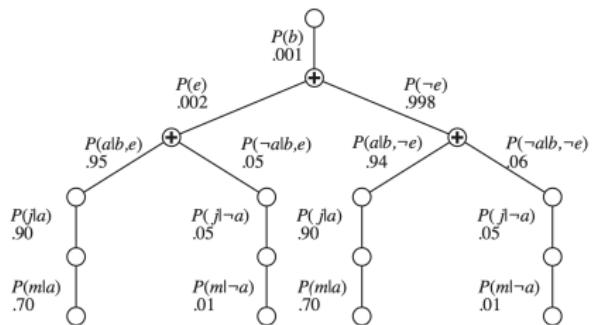
It works, but requires iterating all combinations of variables in  $\mathbf{Y}$ .

# Inference in BNs by enumeration

$$\begin{aligned} P(b|j, m) &= \alpha \sum_{e \in E} \sum_{a \in A} P(b)P(e)P(a|b, e)P(j|a)P(m|a) \\ &= \alpha P(b) \sum_{e \in E} P(e) \sum_{a \in A} P(a|b, e)P(j|a)P(m|a) \end{aligned}$$

The computation can be visualised in a tree.

- Multiplication by  $P(b)$  only once
- Identical subtrees can be re-used



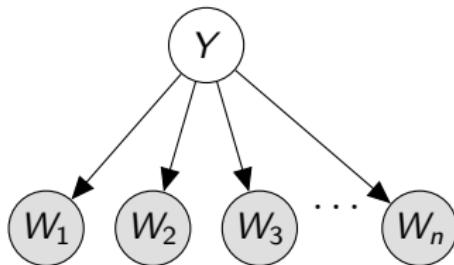
If the BN is a **polytree** (a tree disregarding the edge orientation), then the time and space complexity is linear in the overall number of conditional probability table entries  $O(n.d^k)$ .

For general BNs, the problem is **#P-hard** (harder than NP-complete).

## Special types of BNs: Naïve Bayes classifier

Simple classifier, used for example for spam filtering.

An unobservable class  $Y \in \{ham, spam\}$  influences the probability of occurrence of individual words, e.g.,  $W_1 = bargain$ ,  $W_2 = socrates$ , etc.



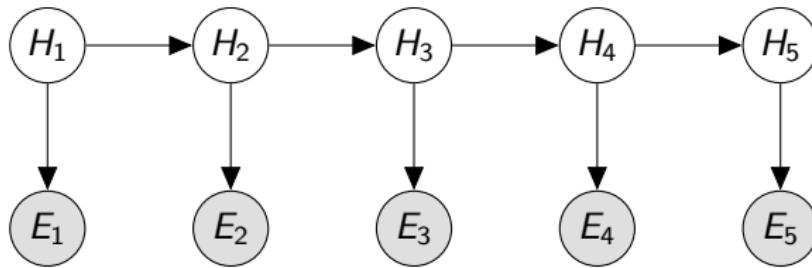
Given an email, what is the probability that it is spam?

$$P(spam|\mathbf{w}) = \alpha P(spam) \prod_{i=1}^n P(w_i|spam)$$

## Special types of BNs: Hidden Markov model

Special case of **dynamic** Bayesian network, used for example for robotic localisation or speech recognition.

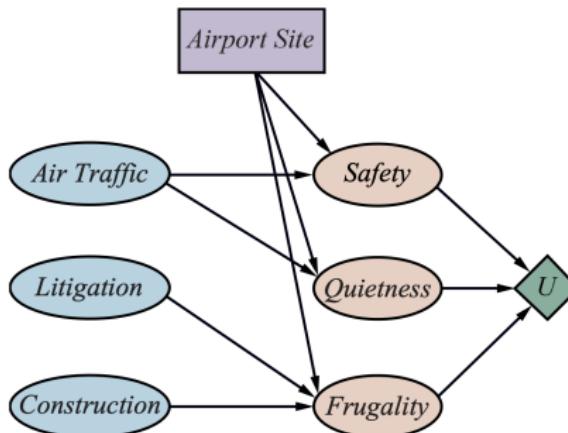
An unobservable true position  $H_t$  in time step  $t$ . Noisy sensor reading about the position  $E_t$ .



Given last sensor readings, what is the probability of some position?

# Decision networks

Influence diagrams (aka. decision networks) combine Bayesian networks with additional nodes for decisions (rectangles) and utilities (diamonds).



In the simplest form, each decision is evaluated in the same way as a chance node with evidence and the decision that maximises the expected utility is selected.

# Summary

Many AI problems require reasoning with uncertainty

Joint probability distribution is a powerful knowledge representation

- But they are growing exponentially, which causes problems

Bayesian networks compactly represent joint distributions

- Represent exactly the joint distribution in smaller space
- Inference is still exponential in general
- Inference is polynomial with special structure

Bayesian networks are a general framework with many specializations

- Naive Bayes, HMM

Influence diagrams introduce decisions and utilities and allow expected utility maximization

# Lecture 12: Sequential Decisions with Partial Information (POMDPs)

Viliam Lisý & Branislav Bošanský

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

[viliam.lisy@fel.cvut.cz](mailto:viliam.lisy@fel.cvut.cz), [bosansky@fel.cvut.cz](mailto:bosansky@fel.cvut.cz)

May, 2021

# What we already know?

What we already covered:

- finding optimal plan
- search-based (A\*) / learning-based (RL) / sampling-based (MCTS) approaches
- uncertainty

The main formal model for us was Markov Decision Process (MDP).

Unfortunately, the world is not perfect – agents often do not have perfect information about the true state of the environment  
→ Partially Observable MDPs (POMDPs).

# Motivation for POMDPs

Many practical applications naturally fit to the POMDP class:

- more realistic
  - agents often receive partial information about the true state (observations) rather than complete states
- in robotics, the exact location of the robot in the environment is typically not known
  - sensors are imperfect (there is always some level of noise/uncertainty)
  - actions are imperfect
- security scenarios (assuming fixed strategy of the opponent)
  - agents typically do not know the effects of the actions of the opponent (which computer has been infiltrated by an attacker)

# Definition POMDPs

Recall the definition of POMDPs – We have a finite sets of states  $\mathcal{S}$ , rewards  $\mathcal{R}$ , and actions  $\mathcal{A}$ . The agent interacts with the environment in discrete steps  $t = 0, 1, 2, \dots$ . At each timestep, the agent has a **belief** – a probability distribution over states that expresses the (subjective) likelihood about the current states.

The agent receives **observations** from a finite set  $\mathcal{O}$  that affect the belief. The agent starts from an **initial belief** and based on actions and observations, it updates its belief. Given the current belief  $b : \mathcal{S} \rightarrow [0, 1]$  and some action  $a \in \mathcal{A}$  and received observation  $o \in \mathcal{O}$ , the new belief is defined as:

$$b(s') = \mu O(o|s', a) \cdot \sum_{s \in \mathcal{S}} Pr(s'|s, a) \cdot b(s)$$

where  $\mu$  is a normalizing constant.

# POMDP – Example

#	#	#	#	#	#
#	<i>G</i>				#
#	#	#	#		#
#	↓	#	#		#
#					#
#	#	#	#	#	#

The robot can now perceive only its surroundings but does not know the exact position in the maze. States and actions remain the same.

- $s = (X, Y, d, G)$
- actions = (move\_forward, move\_backward, turn\_left, turn\_right)

Observations are all possible combinations of walls / free squares in the 4-neighborhood (in front, right, behind, left ):

- $(\#, \#, \#, \#), (\#, \#, \#, -), \dots$

# Beliefs in POMDPs

So how exactly we compute the beliefs<sup>1</sup>:

current beliefs $b_t$							new beliefs $b_{t+1}$					
#	#	#	#	#	#	#	#	#	#	#	#	#
#	G	0.25	0.25		#	#	#	G		0.5		#
#	#	#	#	#	#	#	#	#	#		#	#
#		#	#	#	#	#	#		#		#	#
#			0.25	0.25		#	#	0.5				#
#	#	#	#	#	#	#	#	#	#	#	#	#

for  $s' = (1, 1, <, -)$ , it holds

$$b'_{t+1}(s') = O(o|s', a) \cdot Pr(s'|a, (2, 1, <, -)) \cdot b_t((2, 1, <, -))$$

$$b'_{t+1}(s') = 1 \cdot 1 \cdot 0.25$$

and then  $b_{t+1}(s') = \mu b'_{t+1}(s')$  where  $\mu = \frac{1}{b'_{t+1}((1, 1, <, -)) + b'_{t+1}((4, 4, >, -))}$

---

<sup>1</sup>Coordinates (0,0) are in the bottom left corner.

# How to act optimally in MDPs

Recall a value function for an MDP and a policy  $\pi$

$$v_\pi : \mathcal{S} \rightarrow \mathbb{R}$$

is a function assigning each state  $s$  the expected return

$v_\pi(s) = \mathbb{E}_\pi G_0$  obtained by following policy  $\pi$  from state  $s$ .

Optimal policies share the same **optimal state-value function**:

$$v_*(s) = \max_{\pi} v_{\pi}(s) \text{ for all } s \in \mathcal{S}$$

Any policy that is greedy with respect to  $v_*$  is an optimal policy.

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

# How things change for POMDPs?

Which action is optimal depends on the **belief** over states:

#	#	#	#	#	#
#	G			> (0.5)	#
#		#	#		#
#		#	#		#
#	< (0.5)				#
#	#	#	#	#	#

Consider 2 actions – **move backward** and **turn right**

- **move backward** is better for the state  $(4, 4, >, -)$
- **turn right** is better for the state  $(1, 1, <, -)$

The value of each action depends on the exact belief  $\rightarrow$  value function also depends on beliefs.

# Value function for POMDPs

A value function for a POMDP and a policy  $\pi$

$$v_\pi : \Delta(\mathcal{S}) \rightarrow \mathbb{R}$$

Can we update Bellman equation to use beliefs? Yes!

$$v_*(b) = \max_a \int p(b', r|b, a) [r + \gamma v_*(b')] db'$$

... the “only problem” is that  $b$  is a continuous variable  
→ computing optimal value function in this form is not practical.

# Representation of Value Function

Using beliefs, we have formulated an **MDP with a continuous set of states**.

Discretization of beliefs is not very practical due to high dimension ( $|\mathcal{S}|$ ).

Consider the Bellman equation again – what is our goal?

$$v_*(b) = \max_a \int p(b', r|b, a) [r + \gamma v_*(b')] db'$$

Find the best action (and value) for each belief point.

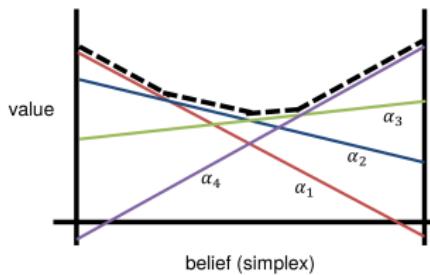
There is infinitely many belief points but set of actions  $\mathcal{A}$  is finite!

## Representation of Value Function – $\alpha$ vectors

If we fix an action  $a \in \mathcal{A}$ , the value function (for that action) is a **linear function** in the current belief. These linear functions are called  **$\alpha$ -vectors**.

For each belief point, we take the best action hence we maximize over all  $\alpha$ -vectors:

$$v(b) = \max_{\alpha} \sum_{s \in \mathcal{S}} \alpha(s) \cdot b(s)$$



$\alpha$ -vectors are in fact more general → they represent expected value for a **policy** (contingency plan consisting of multiple steps).

# Using $\alpha$ -vectors in value iteration

Using  $\alpha$ -vectors corresponding to the value functions of currently considered policies, we can compute new value (next iteration):

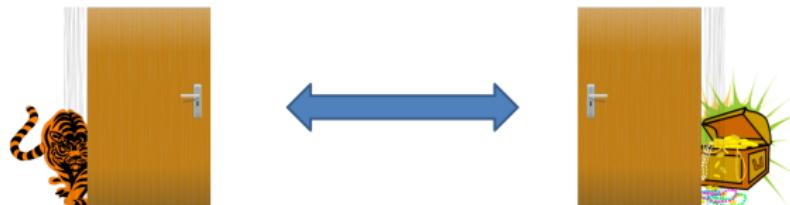
$$v_{t+1}(b) = \max_a \left\{ \sum_{o \in O} \max_{\alpha' \in v_t} \left[ \sum_{r, s, s'} \mu p(s', r | s, a) b(s) O(o | s', a) (r + \gamma \alpha'(s')) \right] \right\}$$

... but how do we construct  $\alpha$ -vectors from  $v_{t+1}$ ?

- ➊ assume there are  $\alpha$ -vectors  $\alpha'$  representing values of policies in step  $t$
- ➋ in step  $t + 1$ , we choose some action and then, **based on the observation**, we follow with some of the policy corresponding to  $\alpha'$  from  $v_t$  (different observation leads to a different belief)
- ➌ for example, choose action  $a_3$  and then
  - if  $o_2$  is received, use value of  $\alpha'_4$  (i.e., this value is achievable via some policy corresponding to this  $\alpha$ -vector)
  - if  $o_1$  is received, use value of  $\alpha'_2$

## Tiger example

Let's consider the best-known POMDP example – a tiger problem:  
There are 2 doors hiding a treasure or a tiger. The agent does not know where is the tiger and where is the treasure. The agent can gather observations (listen) or open one of the doors.



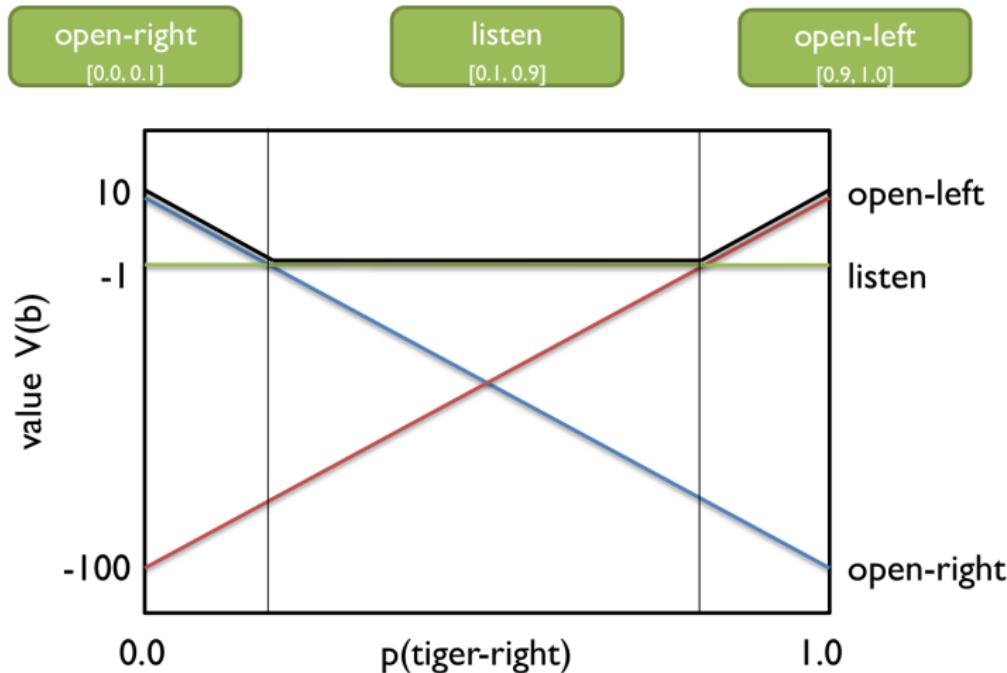
- states – { $\text{tiger\_left}(TL)$ ,  $\text{tiger\_right}(TR)$ }
- actions – { $\text{open\_left}$ ,  $\text{open\_right}$ ,  $\text{listen}$ }
- observations – { $\text{hearTL}$ ,  $\text{hearTR}$ }
- rewards –
  - $-1$  for any listening action (in all states)
  - $+10$  for opening the door with treasure
  - $-100$  for opening the door with tiger

## Tiger example

- states –  $\{\text{tiger\_left}(TL), \text{tiger\_right}(TR)\}$
- actions –  $\{\text{open\_left}, \text{open\_right}, \text{listen}\}$
- observations –  $\{\text{hearTL}, \text{hearTR}\}$
- rewards –
  - $-1$  for any listening action (in all states)
  - $+10$  for opening the door with treasure
  - $-100$  for opening the door with tiger
- initial belief is uniform –  $b_0(TL) = b_0(TR) = 0.5$
- transition dynamics –
  - performing action **listen** does not change the state
  - opening a door “restarts” the problem (i.e.,  $p(s'|s, a) = 0.5$  for both states  $s' \in \{TL, TR\}$ ).
- observation probabilities –
  - listening action generates observation **hearTL/TR** with a 15% error – i.e., agent chooses action  $a = \text{listen}$ , then  $O(\text{hearTR}|a, TR) = 0.85$  and  $O(\text{hearTR}|a, TL) = 0.15$ .

# Tiger example

What are the optimal actions (1-step policy)?



## Tiger example

Choosing action **listen** is not sufficient → what should we do next?

Depending on the observation, the belief will change:

- assume  $b_0(TR) = b_0(TL) = 0.5$ ,  $a = \text{listen}$ , and  $o = \text{hearTR}$
- now  $b_1(TR) = \frac{0.5 \cdot 0.85}{0.5 \cdot 0.85 + 0.5 \cdot 0.15} = 0.85$

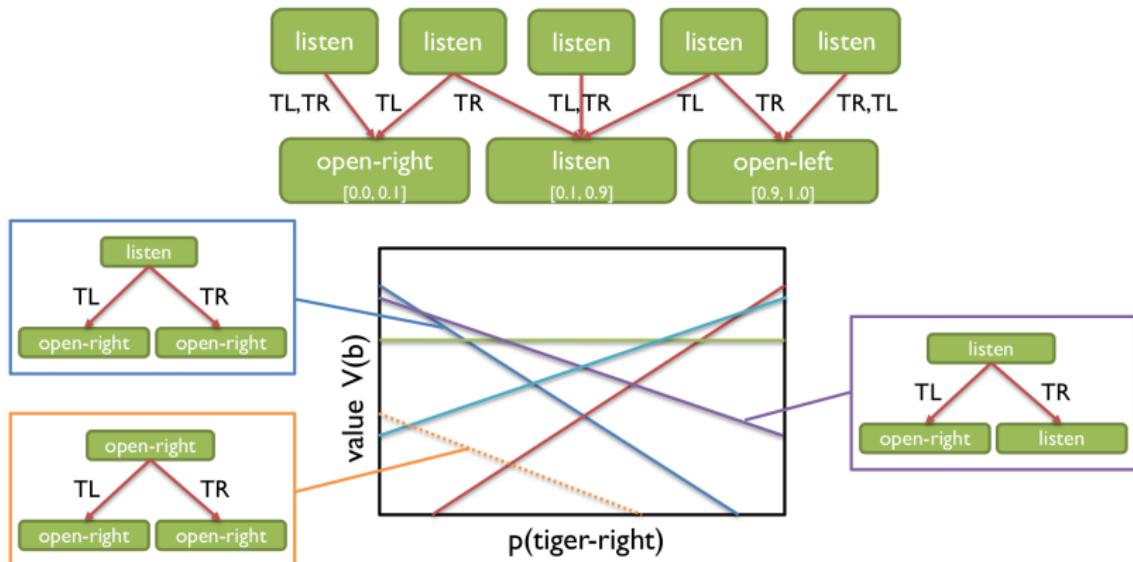
Since  $0.85 \in [0.1, 0.9]$ , after one observation the next optimal action is still **listen**.

In general, the chosen actions in policies depend on received observation, for example (a 2-step policy):

- **listen**
  - if (observation is hearTR → open\_left)
  - else if (observation is hearTL → listen)

# Tiger example

What do the  $\alpha$ -vectors corresponding to 2-step policies look like?



## Exact value iteration in POMDPs

In exact (full) value iteration in POMDPs,  $|\mathcal{A}| \cdot |\mathcal{O}|$  new  $\alpha$ -vectors are generated for every already existing  $\alpha$ -vector in each step of the algorithm.

It is clear that such approach will not scale well. Pruning dominated  $\alpha$ -vectors is possible but does not solve the issue.

### Observation

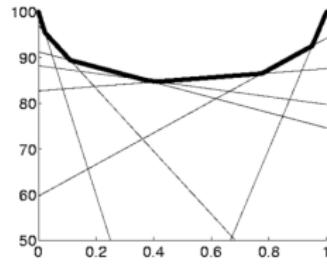
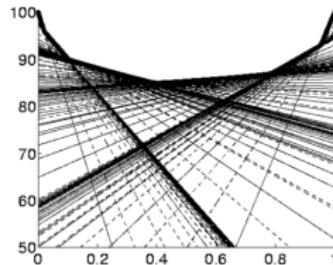
We do not need to compute all  $\alpha$ -vectors – large portion of belief space is (often) not reached hence not relevant for solving the problem.

We can keep only a bounded number of belief points and for each belief point we keep 1 (the best)  $\alpha$ -vector.

# Point-based updates and point-based value iteration (PBVI)

Let  $\mathcal{B} = \{b^1, b^2, \dots\}$  be a set of  $|\mathcal{B}|$  belief points. **Point-based value iteration** performs Bellman update only for this limited set of belief points:

- instead of adding all  $\alpha$ -vectors, only the  $\alpha$ -vectors that are optimal in some of the belief points from  $\mathcal{B}$  are kept,

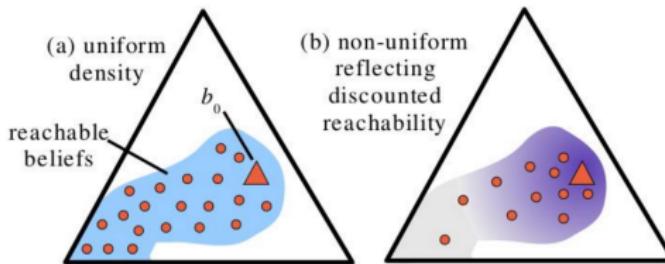


Comparison of generated  $\alpha$ -vectors for full VI and PBVI for tiger example after 30 iterations (from slides of M. Herrmann, RL 13).

# Point-based updates and point-based value iteration (PBVI)

Let  $\mathcal{B} = \{b^1, b^2, \dots\}$  be a set of  $|\mathcal{B}|$  belief points. Point-based value iteration performs Bellman update only for this limited set of belief points:

- the set of belief points  $\mathcal{B}$  can correspond to a uniform coverage of the belief space or the points can focus on more relevant parts of the belief space



## Scaling-up solving POMDPs

- more scalable VI-based algorithms
- using MCTS-like algorithm for solving POMDPs
- from POMDPs to II games and DeepStack (poker)

# Lecture 13: Sequential Decisions with Partial Information (POMDPs) 2

Viliam Lisý & Branislav Bošanský

Artificial Intelligence Center  
Department of Computer Science, Faculty of Electrical Eng.  
Czech Technical University in Prague

[viliam.lisy@fel.cvut.cz](mailto:viliam.lisy@fel.cvut.cz), [bosansky@fel.cvut.cz](mailto:bosansky@fel.cvut.cz)

May, 2021

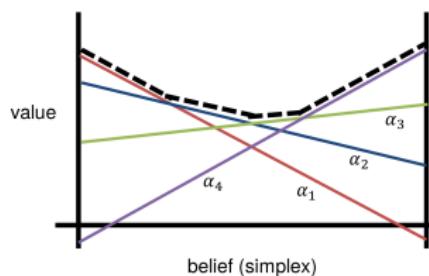
## Reminder: Representation of Value Function with $\alpha$ vectors

If we fix an action  $a \in \mathcal{A}$ , the value function (the expected reward after playing that action) is a **linear function** in the current belief. These linear functions are called  **$\alpha$ -vectors**.

For each belief point, we take the best action hence we maximize over all  $\alpha$ -vectors:

$$v(b) = \max_{\alpha \in V} \sum_{s \in S} \alpha(s) \cdot b(s)$$

where  $\alpha(s)$  corresponds to the value of the linear function  $\alpha$  in state  $s$ .



In general,  $\alpha$ -vectors represent expected value for a **policy** (contingency plan consisting of multiple steps).

## Exact value iteration in POMDPs

In exact (full) value iteration in POMDPs,  $|V_t| = |\mathcal{A}| \cdot |V_{t-1}|^{|\mathcal{O}|}$  new  $\alpha$ -vectors are generated in each step of the algorithm.

$V_t$  converges to optimal value function (the algorithm incrementally constructs all possible  $t$ -step policies).

It is clear that such approach will not scale well. Pruning dominated  $\alpha$ -vectors is possible but does not solve the issue.

### Observation

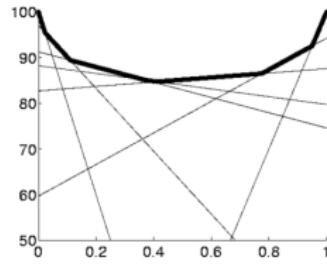
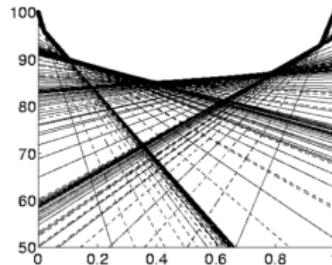
We do not need to compute all  $\alpha$ -vectors – large portion of belief space is (often) not reached hence not relevant for solving the problem.

We can keep only a bounded number of belief points and for each belief point we keep 1 (the best)  $\alpha$ -vector.

# Point-based updates and point-based value iteration (PBVI)

Let  $\mathcal{B} = \{b^1, b^2, \dots\}$  be a set of  $|\mathcal{B}|$  belief points. **Point-based value iteration** performs Bellman update only for this limited set of belief points:

- instead of adding all  $\alpha$ -vectors, only the  $\alpha$ -vectors that are optimal in some of the belief points from  $\mathcal{B}$  are kept,

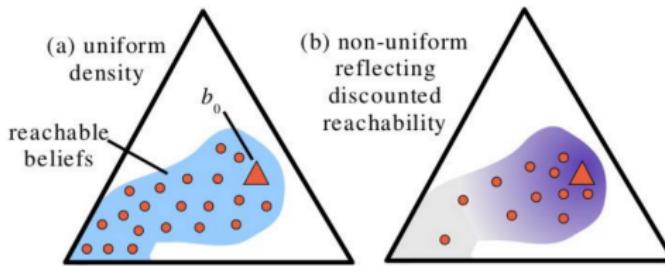


Comparison of generated  $\alpha$ -vectors for full VI and PBVI for tiger example after 30 iterations (from slides of M. Herrmann, RL 13).

# Point-based updates and point-based value iteration (PBVI)

Let  $\mathcal{B} = \{b^1, b^2, \dots\}$  be a set of  $|\mathcal{B}|$  belief points. Point-based value iteration performs Bellman update only for this limited set of belief points:

- the set of belief points  $\mathcal{B}$  can correspond to a uniform coverage of the belief space or the points can focus on more relevant parts of the belief space



## Advantages of PBVI:

- removes exponential complexity (the number of alpha vectors is bounded)
- a practical algorithm for solving POMDPs

## Disadvantages of PBVI:

- it is not clear how far from the optimum is the current solution
- the set of belief points needs to be updated / maintained
- it is not clear which part of the belief space to explore

# Heuristic Search Value Iteration (HSV1)

Approximates the value function with 2 approximate value functions:

- lower bound – a set of alpha vectors corresponding to infinite-step policies
- upper bound – a set of points overestimating values for each belief point

Steps of the algorithm:

- ① initialization of lower bound and upper bound approximate functions
- ② selecting belief points to update using a forward search  
(selecting the best action to explore most promising space of belief points)
- ③ performing point-based updates for both approximate functions

## Question

How to initialize lower / upper bound value function approximations?

- lower bound – choosing some action in all belief points all the time is clearly a lower bound on the expected reward
- upper bound – solving a simplified problem → solving an MDP for each state  $s \in \mathcal{S}$

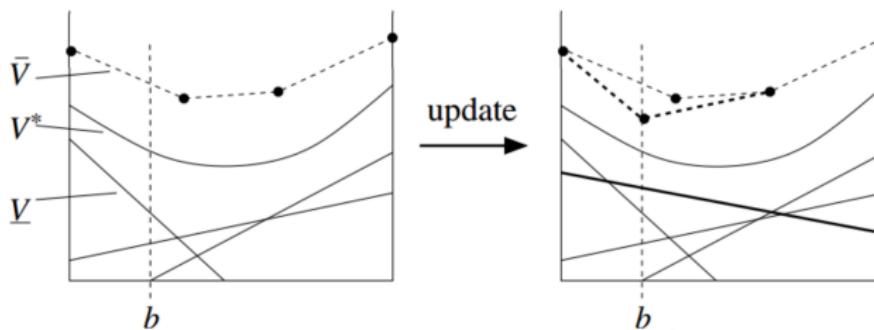
Updates:

- lower bound – point-based value updates (only the set of belief points is not bounded)
- upper bound – compute a lower convex envelope of a set of points in the upper bound and then use point-based value updates

# Heuristic Search Value Iteration (HSV1)

Updates:

- lower bound – point-based value updates (only the set of belief points is not bounded)
- upper bound – compute a lower convex envelope of a set of points in the upper bound and then use point-based value updates



After an update, a new  $\alpha$  vector is added into the lower bound and/or a new point is added into the upper bound.

# Heuristic Search Value Iteration (HSVI)

Selection of the belief points to explore:

- the algorithm explores the most promising actions →
  - the algorithm selects the action based on the upper bound approximation
  - see the connection with search-based methods → the upper bound is an optimistic evaluation of each belief point
  - the idea is either to (1) prove that the most-promising action actually leads to this reward (thus increase the lower bound) or (2) prove that the reward was overestimated and thus decrease the upper bound for relevant belief points
- the updates for the same action is performed for lower bound approximation

This is a very common structure of AI algorithms – upper bound drives the search, lower bounds maintains the best-found solution.

# Scaling up – Monte Carlo for POMDPs

Monte Carlo Tree Search (MCTS) methods were discussed in the context of two player games.

However, we can use the same ideas for solving MDPs and also POMDPs → **POMCP** algorithm.

## Question

How to construct a Monte Carlo tree?

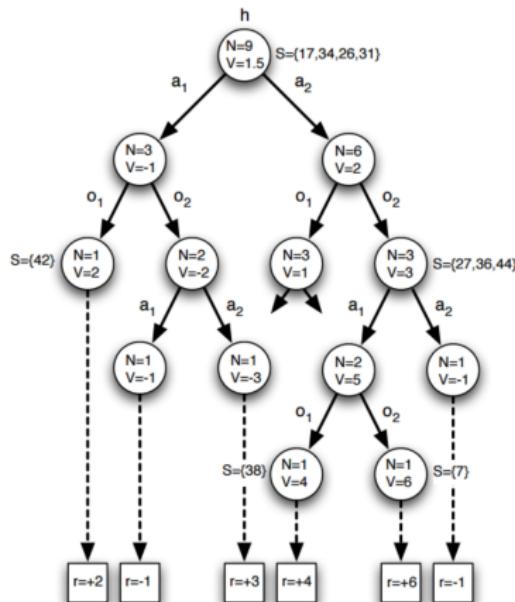
In games (and also for MDPs), there are perfectly observable states (histories of actions) where a bandit algorithm (UCB) is used. But states are not observable in POMDPs ...

Instead, we can use **action-observation histories**.

- the agent is starting in some initial belief
- executing some action generates possible observations (according to a known probability distribution)
- receiving observation updates belief in a well-defined manner (computing a belief update)

## Why action-observation histories are sufficient?

- the agent is starting in some initial belief
- executing some action generates possible observations (according to a known probability distribution)
- receiving observation updates belief in a well-defined manner (computing a belief update)



Where is the catch?

Bayes update of belief points can be too computationally expensive for large domains (with many states and/or observations).

The belief update can be done using **particle filtering**:

- execute K random trials (randomly choosing a true world state based on current belief, executing an action, determining the next state)
- this way we can approximate the next belief points (with probabilities of receiving an observation)

## Question

What are the advantages / disadvantages of exact/approximate algorithm compared to a sampling-based approach?

- exact/approximate algorithms are better (in terms of the quality of found solution) for small cases
- exact/approximate algorithms do not scale for larger domains (HSV will suffer from increasing number of  $\alpha$ -vectors, maintaining the upper bound approximate function)
- sampling-based methods are capable of producing reasonable solution even for large problems
- often, additional improvements are necessary (as the particle filtering for POMCP)

We have seen that MDPs with huge state space can be tackled by (deep) reinforcement learning algorithms.

RL-based methods can be also used for POMDPs (recall that a POMDP is “only” an MDP with infinitely large (continuous) state space).

Atari games that require memory (i.e., it is not sufficient to choose the best action only from the single (or a small number of) image(s) correspond to POMDPs rather than MDPs (there is some hidden state).

DQNs have been successfully used also for POMDPs (for example with NNs that use notion of memory – Deep Recurrent Q-Learning for Partially Observable MDPs [link](#)).

# From POMDPs to Games with Imperfect Information and DeepStack

Why are imperfect information games different from single-agent (or perfect-info) problems?

- ① to play optimally, the players may need to randomly choose an action from some probability distribution over available actions
  - consider, for example, rock-paper-scissors game → when playing, you randomly choose each action with probability 1/3
  - MDPs, POMDPs, 2 player perfect info games → it is sufficient to consider only a single best action
- ② in POMDPs, only the actions of the player and the well-defined environment affect the belief point → in games, there are also (possibly unobservable) actions of the opponent
  - consider, for example, a network-security problem where the attacker infects some hosts → the network admin does not directly observe these actions

# From POMDPs to Games with Imperfect Information and DeepStack

What are the consequences? Why cannot we simply use RL for imperfect information games?

We can, but it is substantially more difficult to learn optimal probability distribution than to identify the best action for a decision point.

Consequently, for imperfect information games that do not require that much randomization, RL-based approach can lead to superhuman performance (for example, in [Dota 2](#)).

For other imperfect information games where randomization is crucial, simple RL-based methods do not work well (e.g., poker).

For poker, however, a variant of limited lookahead algorithm was designed ([link](#)) that beat professional human poker players.



Key components:

- instead of a belief point (possible states) of a single player, we need to consider a set of states that some of the players consider possible
- instead of evaluating a single state using a heuristic eval function (e.g., a NN), we need to evaluate a set of possible states (with probabilities)
- formulating and solving a valid limited-lookahead game requires additional steps (it is not only a simple subtree of a game tree) and a more complex algorithm

In poker, the amount of unknown information is constant (opponent's cards). In real-world games, the amount of unknown information grows exponentially.

Designing an algorithm for online game playing of imperfect information is an ongoing challenge.

For games with very long horizon (or even unbounded/infinite), the history cannot be used for identifying decision points → for some games where only one player has imperfect information, HSVI algorithm can be adopted ([link](#)).

# Overview of the course

We have covered basic areas of AI:

- formal representation – MDPs/POMDPs, CSP, Logic
- search – (un)informed, branch-and-bound pruning (alpha-beta), sampled-based approaches
- reinforcement learning
- dealing with uncertainty – Bayesian networks, sequential decision making under uncertainty

Described methods are general (domain-independent), the ideas can be used for solving many types of problems.

We have highlighted that typically a (novel) combination of these techniques can lead to great (groundbreaking) results.

There are new challenges ahead.

