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

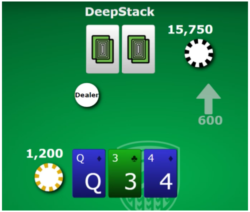
bosansky@fel.cvut.cz

February, 2024

Key Points of the Lecture

- Why we are talking about search at all?
- Why it is good to have a formal representation of a problem?
- Provide a more unifying perspective on different algorithms.

Many (even recent) great AI breakthroughs use search as one of the components.



Search and AI

Search is one of the most fundamental and universal method for solving problems.

Having a **formal representation** of the problem, search algorithms allow us to look for a solution in a systematic way:

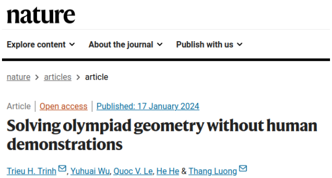
- formal representation of every possible situation in the scenario – the **states** of the problem (denoted S)
- how the states can be changed by the algorithm (agent) – the **actions** in the scenario (denoted A)

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

States must contain all information necessary to determine applicable actions, transformation of the environment, or whether the goal has been reached.

Search and Symbolic Reasoning in 2024

Search is used even in today's most recent results with large language models (LLMs):



nature

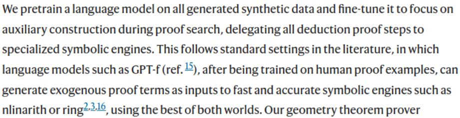
Explore content ⌄ About the journal ⌄ Publish with us ⌄

nature > articles > article

Article | Open access | Published: 17 January 2024

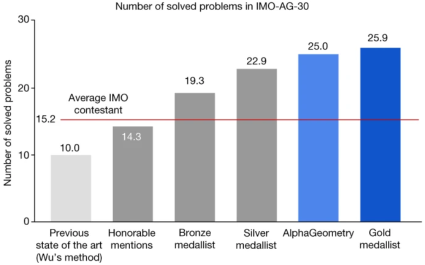
Solving olympiad geometry without human demonstrations

Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He & Thang Luong



We pretrain a language model on all generated synthetic data and fine-tune it to focus on auxiliary construction during proof search, delegating all deduction proof steps to specialized symbolic engines. This follows standard settings in the literature, in which language models such as GPT-f (ref. [15]), after being trained on human proof examples, can generate exogenous proof terms as inputs to fast and accurate symbolic engines such as nlinarith or ring[2,3,16], using the best of both worlds. Our geometry theorem prover

Fig. 2: AlphaGeometry advances the current state of geometry theorem prover from below human level to near gold-medallist level.



Example 1 – 8 puzzle

1		2
4	5	3
7	8	6

→

1	2	3
4	5	6
7	8	

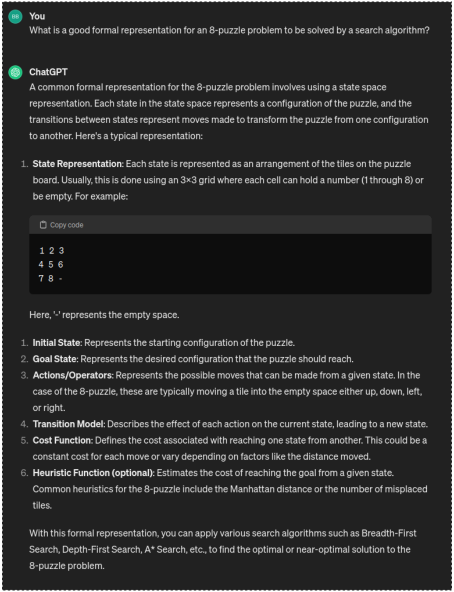
current state → goal state

Goal: rearrange the numbers by moving the empty square to adjacent squares so that the numbers 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 1 – 8 puzzle



You

What is a good formal representation for an 8-puzzle problem to be solved by a search algorithm?

ChatGPT

A common formal representation for the 8-puzzle problem involves using a state space representation. Each state in the state space represents a configuration of the puzzle, and the transitions between states represent moves made to transform the puzzle from one configuration to another. Here's a typical representation:

1. **State Representation**: Each state is represented as an arrangement of the tiles on the puzzle board. Usually, this is done using an 3×3 grid where each cell can hold a number (1 through 8) or be empty. For example:

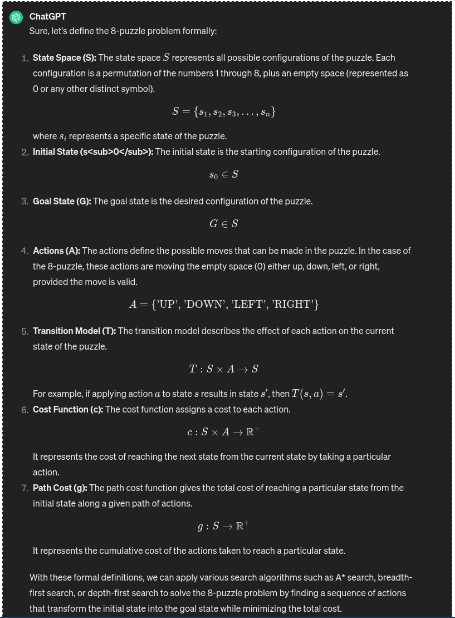
```
1 2 3
4 5 6
7 8 -
```

Here, '-' represents the empty space.

1. **Initial State**: Represents the starting configuration of the puzzle.
2. **Goal State**: Represents the desired configuration that the puzzle should reach.
3. **Actions/Operators**: Represents the possible moves that can be made from a given state. In the case of the 8-puzzle, these are typically moving a tile into the empty space either up, down, left, or right.
4. **Transition Model**: Describes the effect of each action on the current state, leading to a new state.
5. **Cost Function**: Defines the cost associated with reaching one state from another. This could be a constant cost for each move or vary depending on factors like the distance moved.
6. **Heuristic Function (optional)**: Estimates the cost of reaching the goal from a given state. Common heuristics for the 8-puzzle include the Manhattan distance or the number of misplaced tiles.

With this formal representation, you can apply various search algorithms such as Breadth-First Search, Depth-First Search, A* Search, etc., to find the optimal or near-optimal solution to the 8-puzzle problem.

Example 1 – 8 puzzle



ChatGPT

Sure, let's define the 8-puzzle problem formally:

1. **State Space (S):** The state space S represents all possible configurations of the puzzle. Each configuration is a permutation of the numbers 1 through 8, plus an empty space (represented as 0 or any other distinct symbol).

$$S = \{s_1, s_2, s_3, \ldots, s_n\}$$

   where s_i represents a specific state of the puzzle.

2. **Initial State (s₀):** The initial state is the starting configuration of the puzzle.

$$s_0 \in S$$

3. **Goal State (G):** The goal state is the desired configuration of the puzzle.

$$G \in S$$

4. **Actions (A):** The actions define the possible moves that can be made in the puzzle. In the case of the 8-puzzle, these actions are moving the empty space (0) either up, down, left, or right, provided the move is valid.

$$A = \{\text{'UP'}, \text{'DOWN'}, \text{'LEFT'}, \text{'RIGHT'}\}$$

5. **Transition Model (T):** The transition model describes the effect of each action on the current state of the puzzle.

$$T : S \times A \to S$$

   For example, if applying action a to state s results in state s', then $T(s, a) = s'$.

6. **Cost Function (c):** The cost function assigns a cost to each action.

$$c : S \times A \to \mathbb{R}^+$$

   It represents the cost of reaching the next state from the current state by taking a particular action.

7. **Path Cost (g):** The path cost function gives the total cost of reaching a particular state from the initial state along a given path of actions.

$$g : S \to \mathbb{R}^+$$

   It represents the cumulative cost of the actions taken to reach a particular state.

With these formal definitions, we can apply various search algorithms such as A* search, breadth-first search, or depth-first search to solve the 8-puzzle problem by finding a sequence of actions that transform the initial state into the goal state while minimizing the total cost.

Example 2 – Robotic arm

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

Possible representations:

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

Example 3 – Chess

Possible representations:

- positions of pieces on the board $s = [[A1, B1, C1, \dots, H2], [A8, \dots, H7], \dots] \rightarrow$ 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 a sequence

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

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. Selected algorithms for solving them optimally will be introduced later.

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 interacts with the environment in discrete steps $t = 0, 1, 2, \ldots$. 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

#	#	#	#	#	#
#	G				#
#	#	#	#		#
#	↓	#	#		#
#					#
#	#	#	#	#	#

Consider a robot (↓) 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:

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

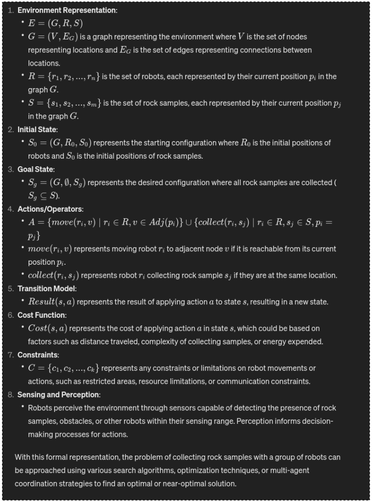
Markov Decision Processes (MDPs)

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

Markov Decision Processes (MDPs)

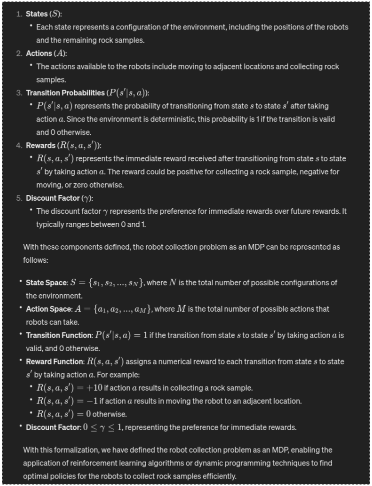
What if we have a problem, where there is a group of robots that need to collect rock samples. How the formalization can look like?



1. **Environment Representation:**
   - $E = (G, R, S)$
   - $G = (V, E_G)$ is a graph representing the environment where V is the set of nodes representing locations and E_G is the set of edges representing connections between locations.
   - $R = \{r_1, r_2, ..., r_n\}$ is the set of robots, each represented by their current position p_i in the graph G.
   - $S = \{s_1, s_2, ..., s_m\}$ is the set of rock samples, each represented by their current position p_j in the graph G.
2. **Initial State:**
   - $S_0 = (G, R_0, S_0)$ represents the starting configuration where R_0 is the initial positions of robots and S_0 is the initial positions of rock samples.
3. **Goal State:**
   - $S_g = (G, \emptyset, S_g)$ represents the desired configuration where all rock samples are collected ($S_g \subseteq S$).
4. **Actions/Operators:**
   - $A = \{move(r_i, v) \mid r_i \in R, v \in Adj(p_i)\} \cup \{collect(r_i, s_j) \mid r_i \in R, s_j \in S, p_i = p_j\}$
   - $move(r_i, v)$ represents moving robot r_i to adjacent node v if it is reachable from its current position p_i.
   - $collect(r_i, s_j)$ represents robot r_i collecting rock sample s_j if they are at the same location.
5. **Transition Model:**
   - $Result(s, a)$ represents the result of applying action a to state s, resulting in a new state.
6. **Cost Function:**
   - $Cost(s, a)$ represents the cost of applying action a in state s, which could be based on factors such as distance traveled, complexity of collecting samples, or energy expended.
7. **Constraints:**
   - $C = \{c_1, c_2, ..., c_k\}$ represents any constraints or limitations on robot movements or actions, such as restricted areas, resource limitations, or communication constraints.
8. **Sensing and Perception:**
   - Robots perceive the environment through sensors capable of detecting the presence of rock samples, obstacles, or other robots within their sensing range. Perception informs decision-making processes for actions.

With this formal representation, the problem of collecting rock samples with a group of robots can be approached using various search algorithms, optimization techniques, or multi-agent coordination strategies to find an optimal or near-optimal solution.

$\longrightarrow$



1. **States** (S):
   - Each state represents a configuration of the environment, including the positions of the robots and the remaining rock samples.
2. **Actions** (A):
   - The actions available to the robots include moving to adjacent locations and collecting rock samples.
3. **Transition Probabilities** ($P(s'|s, a)$):
   - $P(s'|s, a)$ represents the probability of transitioning from state s to state s' after taking action a. Since the environment is deterministic, this probability is 1 if the transition is valid and 0 otherwise.
4. **Rewards** ($R(s, a, s')$):
   - $R(s, a, s')$ represents the immediate reward received after transitioning from state s to state s' by taking action a. The reward could be positive for collecting a rock sample, negative for moving, or zero otherwise.
5. **Discount Factor** (γ):
   - The discount factor γ represents the preference for immediate rewards over future rewards. It typically ranges between 0 and 1.

With these components defined, the robot collection problem as an MDP can be represented as follows:

- **State Space**: $S = \{s_1, s_2, ..., s_N\}$, where N is the total number of possible configurations of the environment.
- **Action Space**: $A = \{a_1, a_2, ..., a_M\}$, where M is the total number of possible actions that robots can take.
- **Transition Function**: $P(s'|s, a) = 1$ if the transition from state s to state s' by taking action a is valid, and 0 otherwise.
- **Reward Function**: $R(s, a, s')$ assigns a numerical reward to each transition from state s to state s' by taking action a. For example:
  - $R(s, a, s') = +10$ if action a results in collecting a rock sample.
  - $R(s, a, s') = -1$ if action a results in moving the robot to an adjacent location.
  - $R(s, a, s') = 0$ otherwise.
- **Discount Factor**: $0 \leq \gamma \leq 1$, representing the preference for immediate rewards.

With this formalization, we have defined the robot collection problem as an MDP, enabling the application of reinforcement learning algorithms or dynamic programming techniques to find optimal policies for the robots to collect rock samples efficiently.

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 μ is a normalizing constant.

POMDP – Example

#	#	#	#	#	#
#	*G*				#
#	**#**	#	#		#
#	↓	**#**	#		#
#					#
#	#	#	#	#	#

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:

- $(\#,\#,\#,\#),(\#,\#,\#,\_),\ldots$

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 (→ B4M36MAS)

Partially Observable Stochastic Games (POSGs)

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 a deterministic 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)

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

maximizes the accumulated reward (and S_k is a goal state)[1].

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

---

[1] Maximization of rewards = minimization of costs (we will use both).

Solving Deterministic MDPs

1. Start from the initial state S_0
2. Apply available actions to the current state and generate new possible states
3. Select one of the newly generated states as the current one
4. If the current state is the goal state → finish
5. If not, go to step 2

Questions:

- Which state should we select out of all generated new states in step 3?
- What if we generate a state that we have already explored?

Solving Deterministic MDPs – Choosing Next State

Q1

Which state should we select out of all generated new states in step 3?

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 cannot forget which other states are reachable → 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).

Solving Deterministic MDPs

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** of already evaluated states.

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 the cost iteratively by 1.

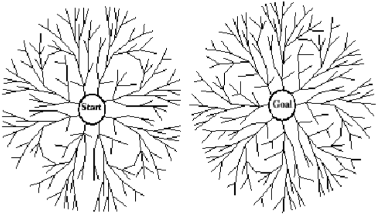
Backward / Bidirectional Search

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

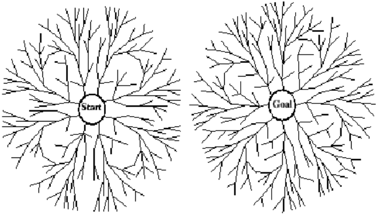
Sometimes, searching from the goal state to a starting state can be better:

- 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.