LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

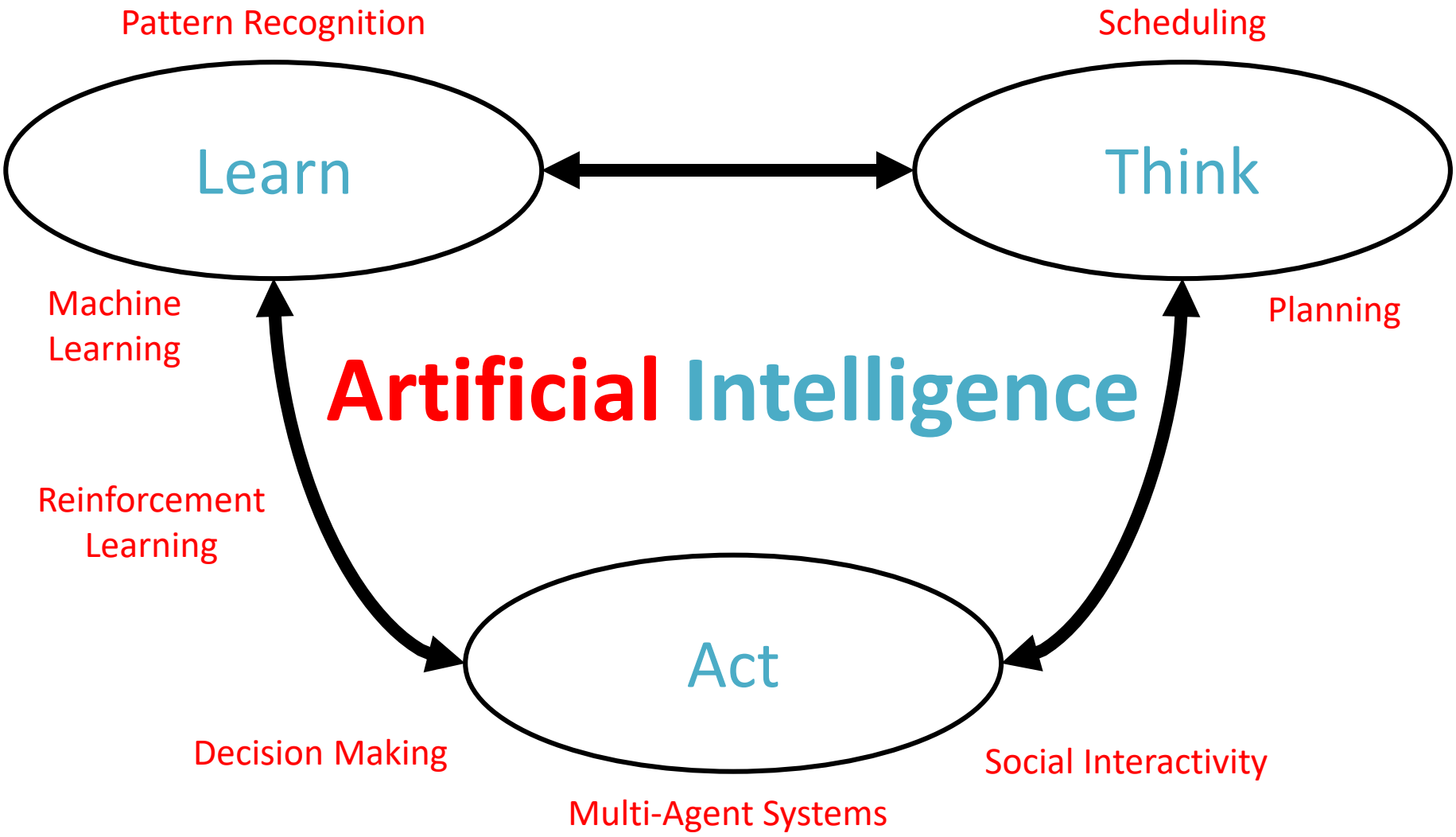
Praktikum Autonome Systeme

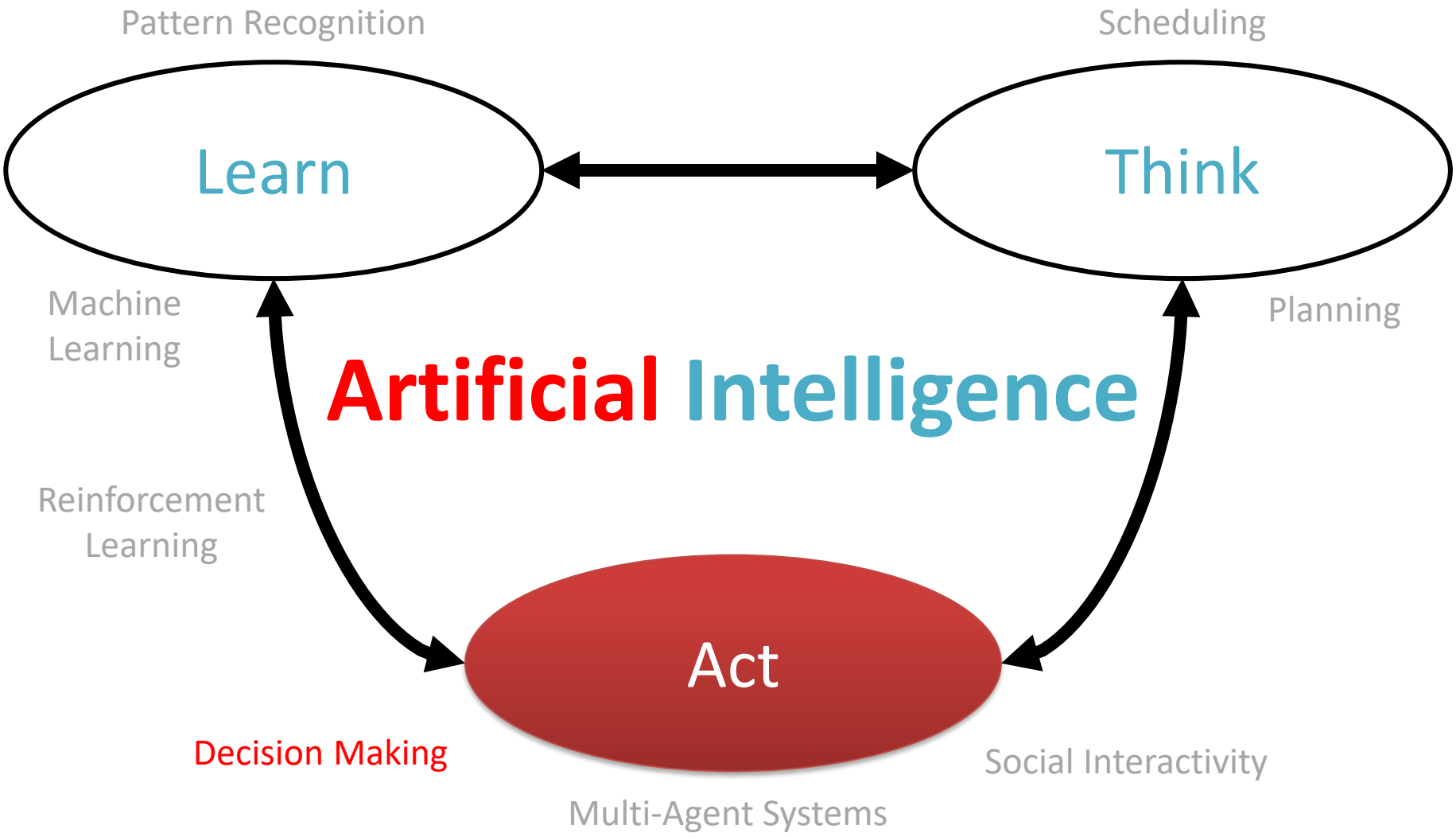
Automated Planning

Prof. Dr. Claudia Linnhoff-Popien
Thomy Phan, Andreas Sedlmeier, Fabian Ritz
http://www.mobile.ifi.lmu.de

SoSe 2019

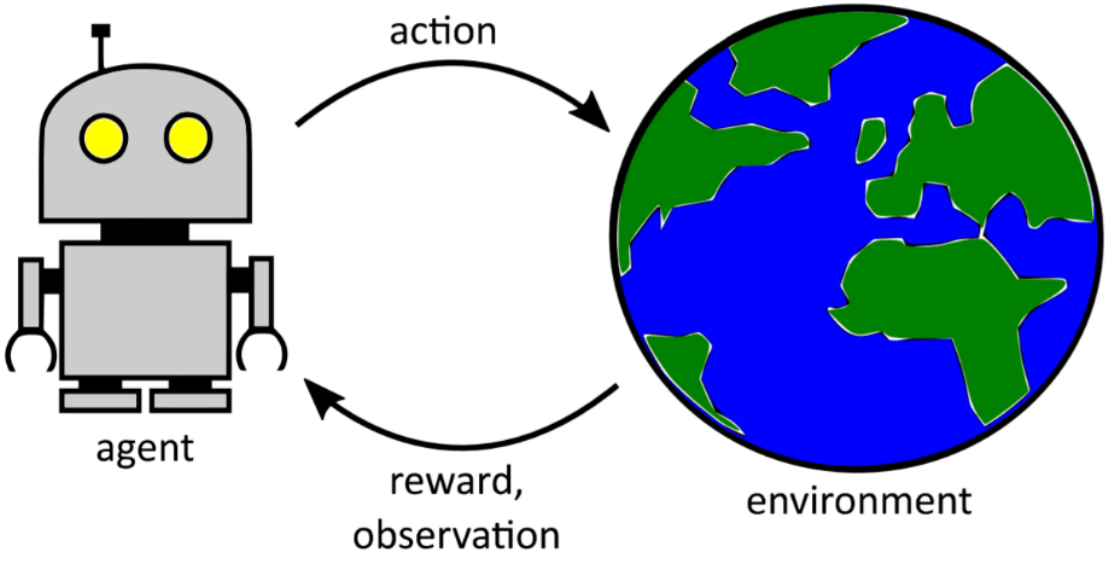
→ Recap: Decision Making


Pattern Recognition
Scheduling
Learn
Think
Machine Learning
Planning
Artificial Intelligence
Reinforcement Learning
Act
Decision Making
Social Interactivity
Multi-Agent Systems


Pattern Recognition
Scheduling
Learn
Think
Machine Learning
Planning
Artificial Intelligence
Reinforcement Learning
Act
Decision Making
Social Interactivity
Multi-Agent Systems

Decision Making

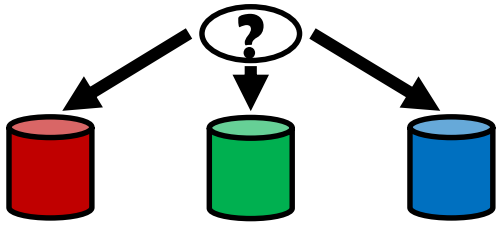
- **Goal:** Autonomously select actions to solve a (complex) task
  - time could be important (but not necessarily)
  - maximize the **expected reward** for each state



Multi-Armed Bandits

- **Multi-Armed Bandit**: situation, where you have to learn to make a good (long-term) choice

- **Explore** choices to gather information (= Exploration)
  - Example: random choice



- **Prefer** promising choices (= Exploitation)
  - Example: greedy choice (e.g., using `argmax`)

- A good Multi-Armed Bandit solution should **always** balance between Exploration and Exploitation

Multi-Armed Bandits Example

- ϵ-greedy ($\epsilon > 0$):

With probability $\begin{cases} \epsilon, \text{ select randomly} \\ 1 - \epsilon, \text{ select choice highest average reward} \end{cases}$

  - Many approaches use ϵ-greedy with annealing ϵ

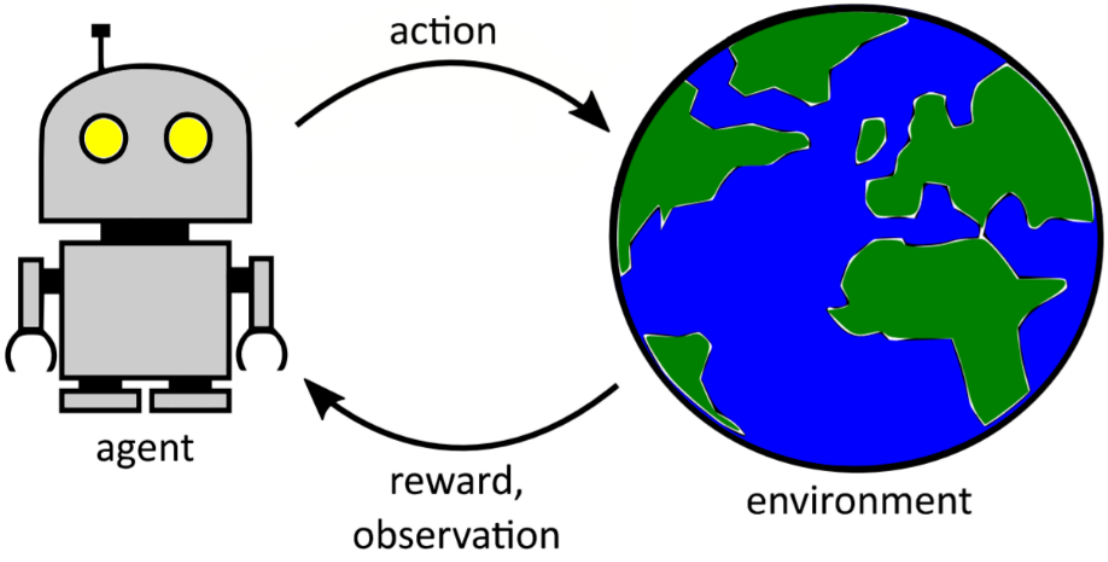
- UCB1 (Upper Confidence Bound):
  - Select by maximizing:

$$\overline{R_i} + c\sqrt{\frac{2logN}{N_i}}$$

→ Sequential Decision Making

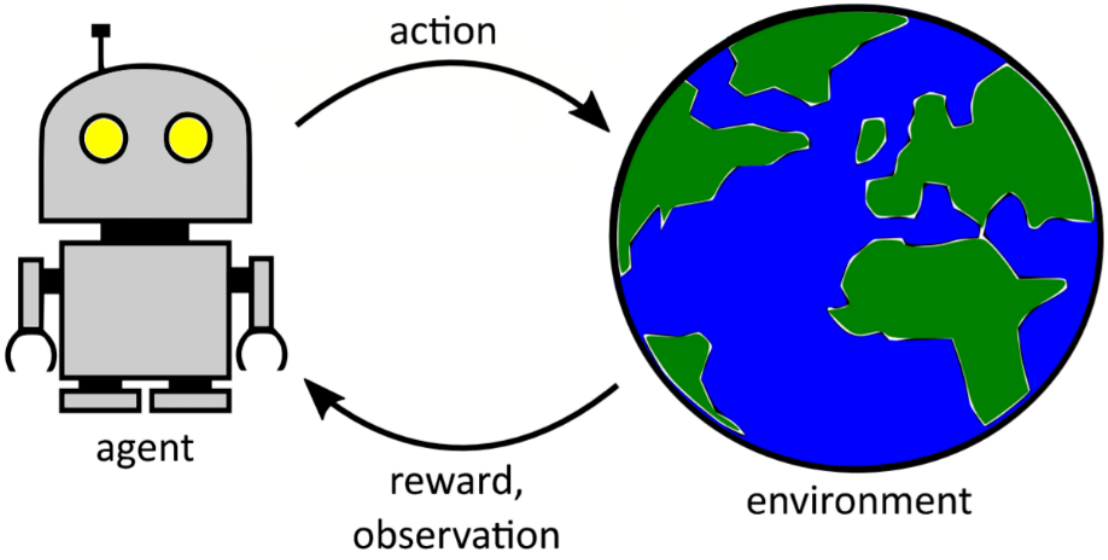
Decision Making

- **Goal:** Autonomously select actions to solve a (complex) task
  - time could be important (but not necessarily)
  - maximize the **expected reward** for each state



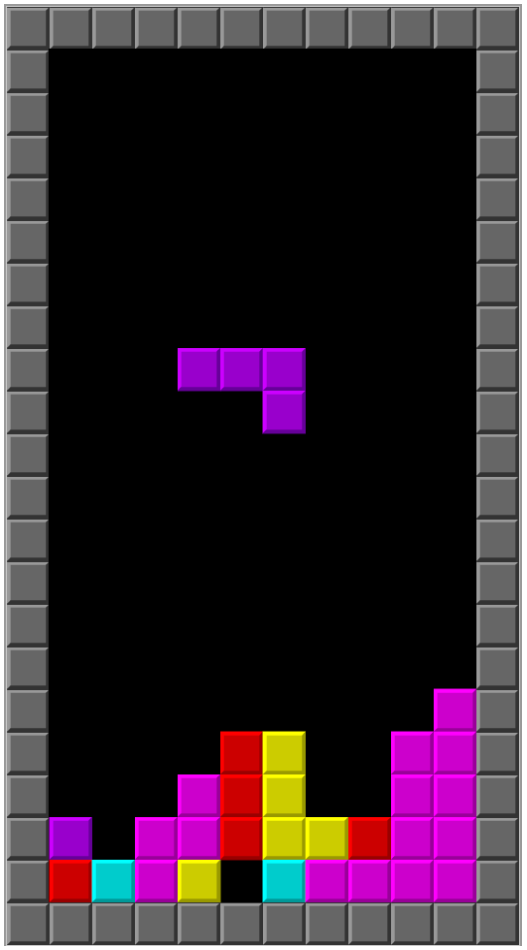
Sequential Decision Making

- **Goal:** Autonomously select actions to solve a (complex) task
  - time is important (actions might have **long term** consequences)
  - maximize the **expected cumulative reward** for each state



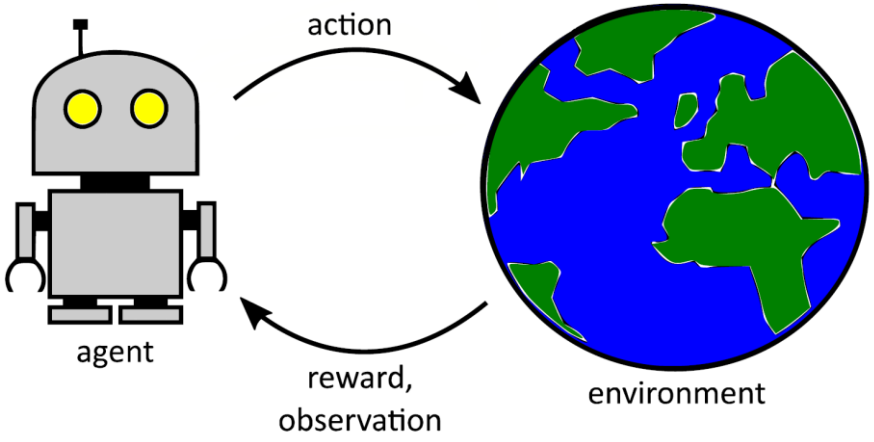
Sequential Decision Making Example

- **Tetris**
  - Actions:
    - Move Left, Right, Down
    - Rotate clockwise
    - Do Nothing
  - Goals:
    - Per time step: Whatever …
    - Short-term: Fill the gaps
    - Long-term: Don‘t let stack size exceed board height



Markov Decision Processes

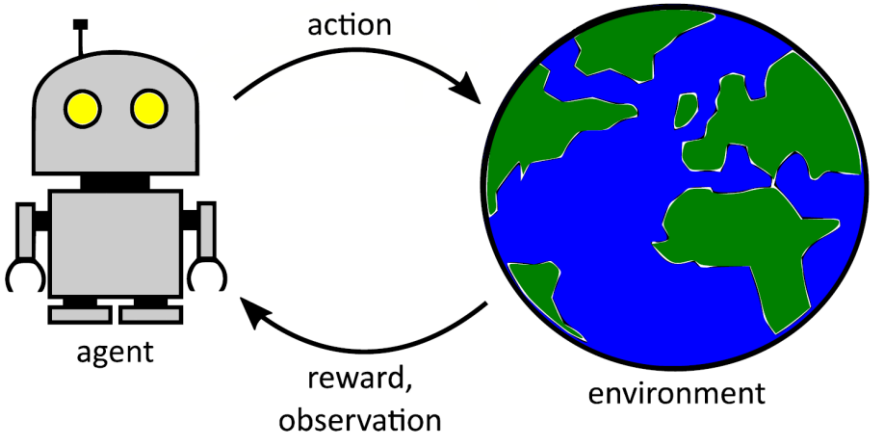
- A Markov Decision Process (MDP) is defined as $\mathrm{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$:
  - $\mathcal{S}$ is a (finite) set of states
  - $\mathcal{A}$ is a (finite) set of actions
  - $\mathcal{P}(s_{t+1}|s_t, a_t) \in [0, 1]$ is the probability for reaching $s_{t+1} \in \mathcal{S}$ when executing $a_t \in \mathcal{A}$ in $s_t \in \mathcal{S}$
  - $\mathcal{R}(s_t, a_t) \in \mathbb{R}$ is a reward function



Markov Decision Processes

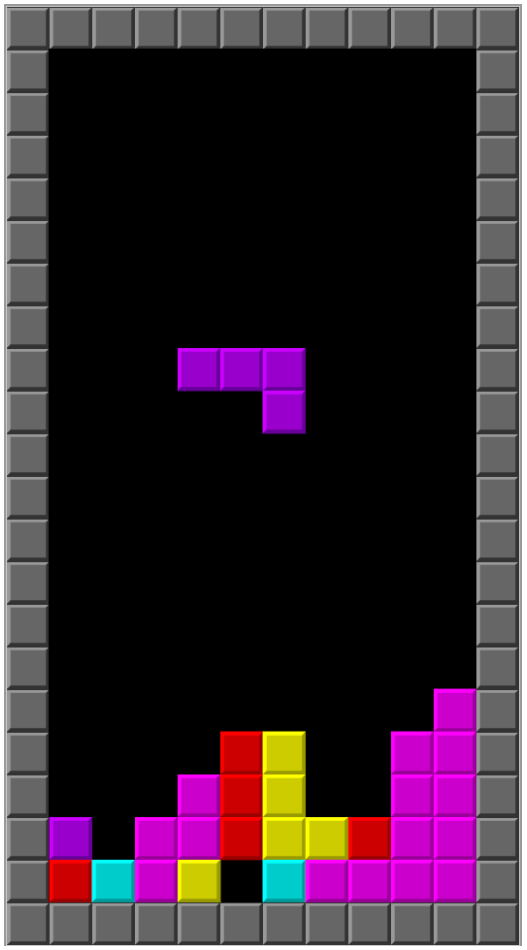
- MDPs formally describe environments for Sequential Decision Making
- All states $s_t \in \mathcal{S}$ are **Markov** such that

  $\mathbb{P}(s_{t+1}|s_t) = \mathbb{P}(s_{t+1}|s_1, \ldots, s_t)$ (no history of past states required)
- Assumes **full observability** of the state
- States and actions may be **discrete** or **continuous**
- Many problems can be formulated as MDPs!
  - E.g., multi-armed bandits are MDPs with a single state



Tetris as MDP

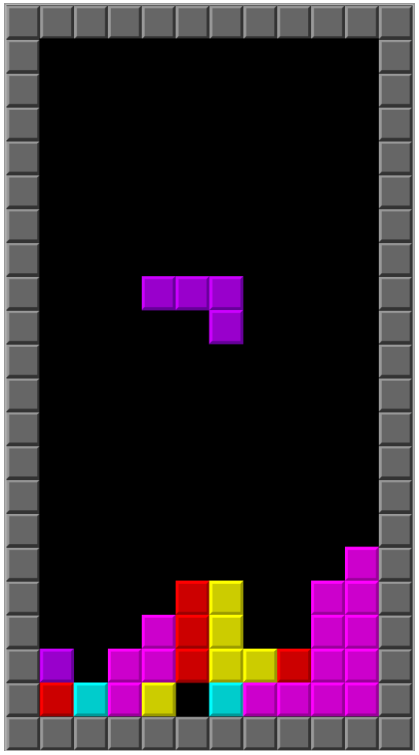
- Define Tetris as MDP $\mathrm{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$:
  - **States $\mathcal{S}$**: describe board size, stone position, and current stack
  - **Actions $\mathcal{A}$**: move left/right/down, rotate, nothing
  - **Transitions $\mathcal{P}$**: deterministic movement of stone, random initialization of next stone
  - **Rewards $\mathcal{R}$**: depend on actual goal
    - Short-term: penalize each gap in stack with -1
    - Long-term: if stack size exceeds board height, penalize agent with -1



Policies

- A **policy** $\pi: \mathcal{S} \rightarrow \mathcal{A}$ represents the behavioural strategy of an agent
  - Policies may also be stochastic $\pi(a_t|s_t) \in [0,1]$

- Tetris Examples:
  - π_0: maps each state $s_t \in \mathcal{S}$ to a random action $a_t \in \mathcal{A}$
  - π_1 : maps each state $s_t \in \mathcal{S}$ to action $a_t = MoveDown \in \mathcal{A}$
  - π_2 : maps each state $s_t \in \mathcal{S}$ with even time steps t to action $a_t = MoveDown \in \mathcal{A}$ and $a_t = MoveLeft \in \mathcal{A}$ otherwise



- **How do we know which policy is better?**

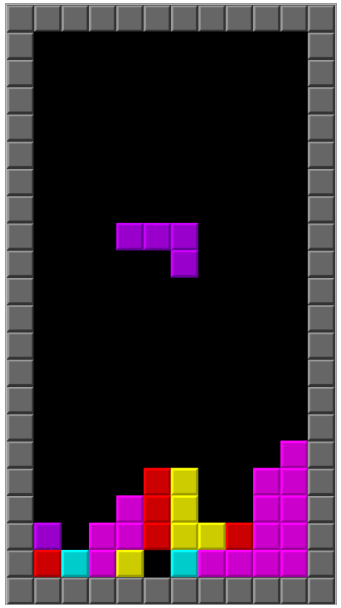
Returns

- The **return** of a state $s_t \in \mathcal{S}$ for a horizon h given a policy π is the cumulative (discounted) future reward (h may be infinite!):

$$G_t = \sum_{k=0}^{h-1} \gamma^k \, \mathcal{R}\big(s_{t+k}, \pi(s_{t+k})\big), \gamma \in [0,1]$$

- Tetris Example:
  - Play a Tetris game given a policy π
  - Record all rewards $r_t = \mathcal{R}(s_t, a_t)$ and their resp. time steps t
  - Compute return $G_t = \sum_{k=0}^{h-1} \gamma^k \, r_t, \gamma \in [0,1]$

- Discount factor γ is used to weight future reward



Value Functions

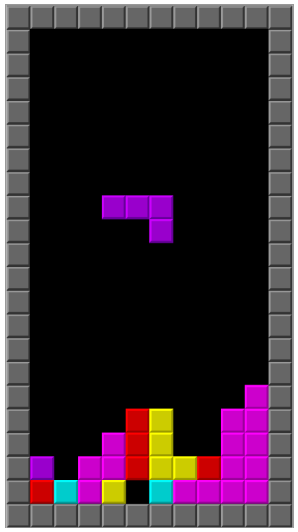
- The **value** of a state $s_t \in \mathcal{S}$ is the expected return of s_t for a horizon $h \in \mathbb{N}$ given a policy π:

$$V^{\pi}(s_t) = \mathbb{E}[G_t | s_t]$$

- The **action value** of a state $s_t \in \mathcal{S}$ and action $a_t \in \mathcal{A}$ is the expected return of executing a_t in s_t for a horizon $h \in \mathbb{N}$ given a policy π:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}[G_t | s_t, a_t]$$

- Tetris Example:
  - V^{π} and/or Q^{π} can be computed by **averaging** over **several returns** G_t observed by playing with a (fixed) policy π



- **Value functions (V^{π} and/or Q^{π}) can be used to evaluate policies π**

Optimal Policies and Value Functions

- **Goal:** Find an *optimal policy* π^* which maximizes the expected return $\mathbb{E}[G_t|s_t]$ for each state:

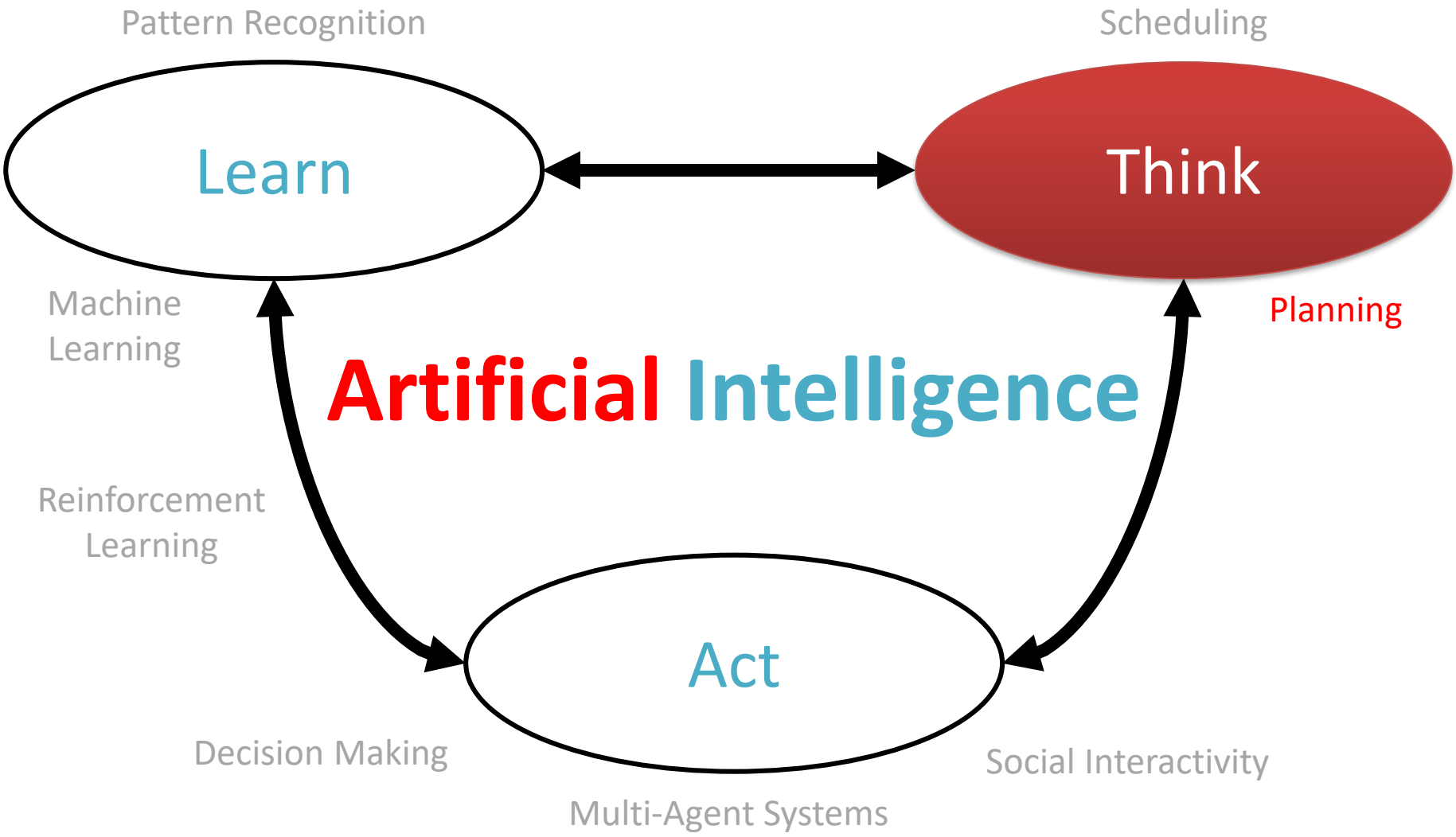
$$\pi^* = argmax_\pi V^\pi(s_t), \forall s_t \in \mathcal{S}$$

- The *optimal value function* is defined by:

$$V^*(s_t) = V^{\pi^*}(s_t) = max_\pi V^\pi(s_t)$$
$$Q^*(s_t, a_t) = Q^{\pi^*}(s_t, a_t) = max_\pi Q^\pi(s_t, a_t)$$

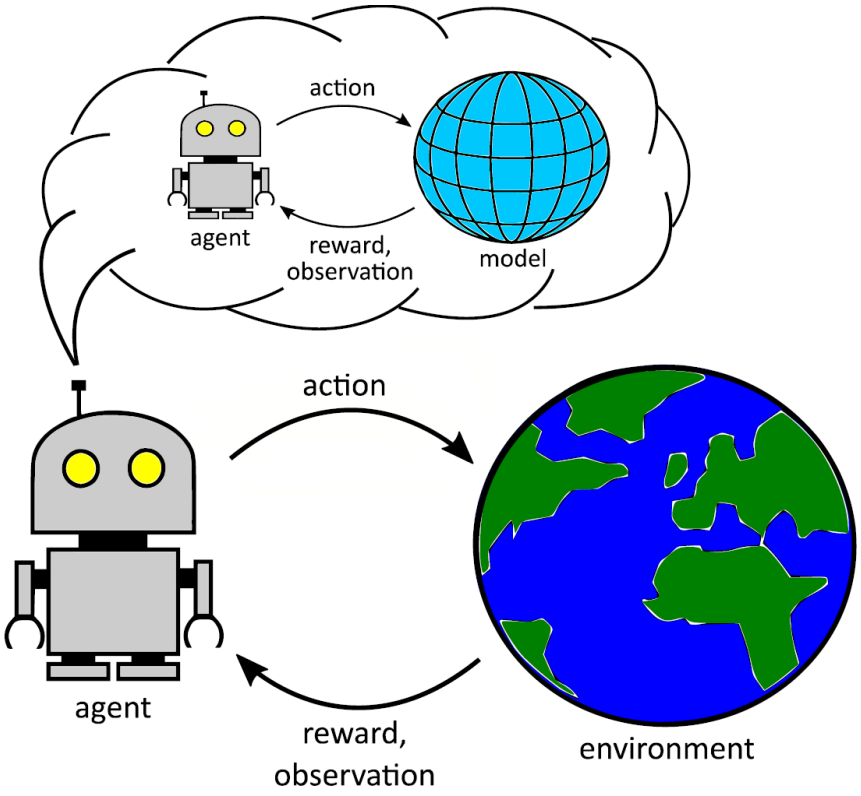
- When V^* or Q^* is known, π^*can be derived.

→ Automated Planning

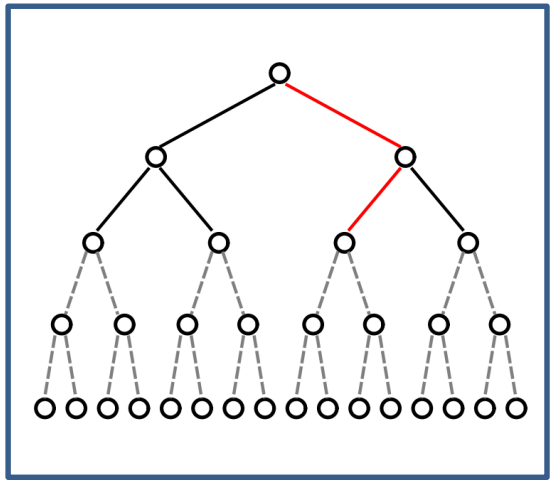

Pattern Recognition
Scheduling
Learn
Think
Machine Learning
Planning
Artificial Intelligence
Reinforcement Learning
Act
Decision Making
Social Interactivity
Multi-Agent Systems

Automated Planning

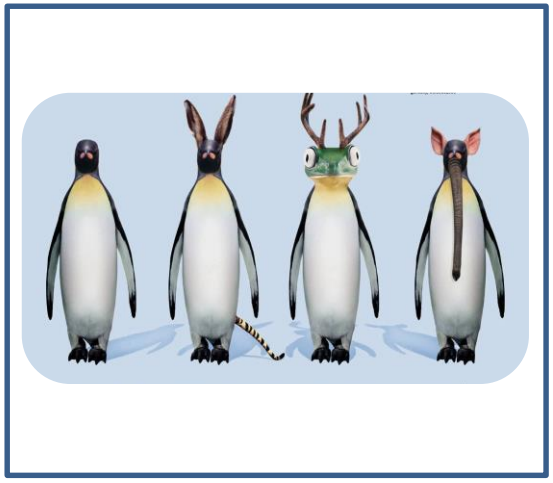
- **Goal:** Find (near-)**optimal policies** π^* to solve complex problems
- Use (heuristic) lookahead search on a **given model** $\widehat{M} \approx M$ of the problem



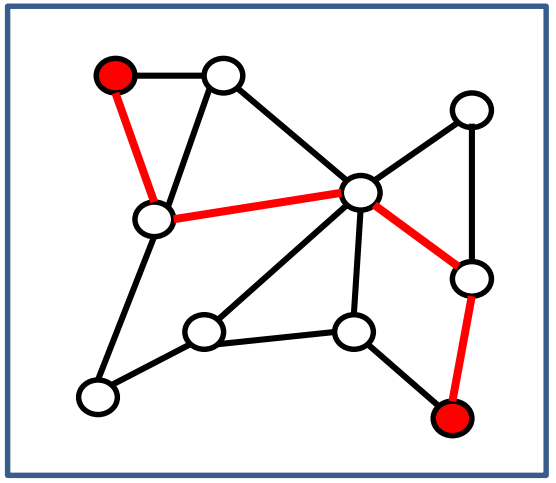
Planning Approaches (Examples)



Tree Search



Evolutionary Computation



Dynamic Programming

→ Dynamic Programming

Dynamic Programming

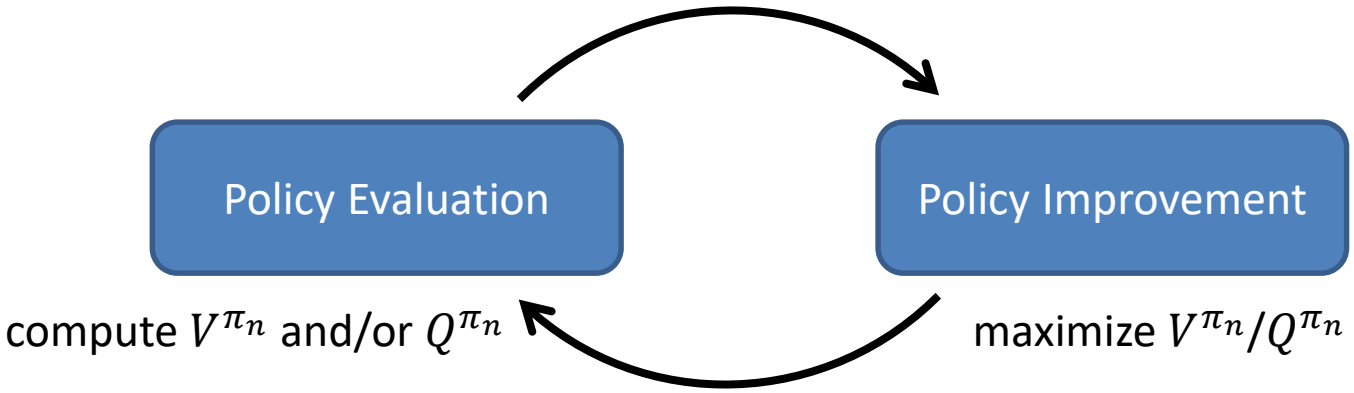
- **Dynamic** refers to sequential / temporal component of a problem
- **Programming** refers to optimization of a program

- We want to solve Markov Decision Processes (MDPs):
  - MDPs are **sequential** decision making problems
  - To find a solution, we need to optimize a **program** (policy π)

Policy Iteration

- Dynamic Programming approach to find an optimal policy π^*

- Starts with a (random) guess π_0
- Consists of two alternating steps given π_n:



- Terminates when $\pi_{i+1} = \pi_i$ or when a time budget runs out

- **Policy Iteration forms the basis for most Planning and Reinforcement Learning algorithms!**

Value Iteration

- Dynamic Programming approach to find the optimal value function V^*

- Starts with a (random) guess V^0
- Iteratively updates the value estimate $V^n(s_t)$ for each state $s_t \in \mathcal{S}$

$$V^{n+1}(s_t) = \max_{a_t \in \mathcal{A}} \{\mathcal{R}(s_t, a_t) + \gamma \sum_{s_{t+1} \in \mathcal{S}} \mathcal{P}(s_{t+1}|s_t, a_t) V^n(s_{t+1})\}$$

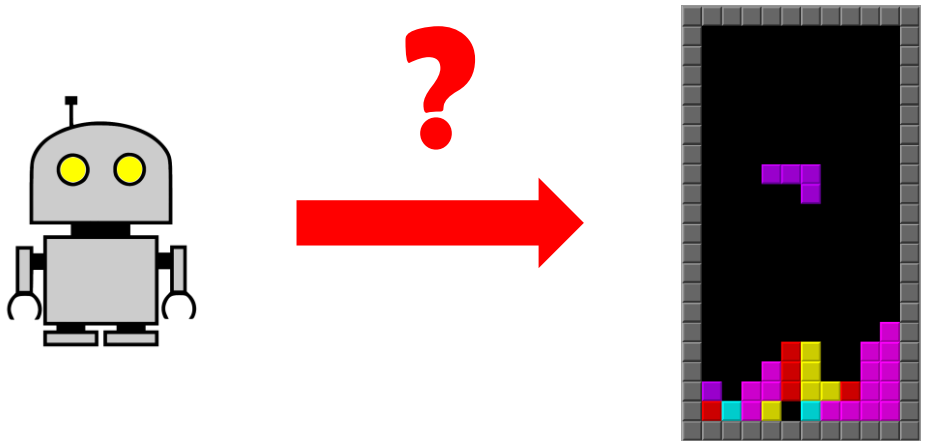
- Terminates when $V^{n+1} = V^n$ or when a time budget runs out
- The optimal action-value function Q^* is computed analogously

- V^* and/or Q^* can be used to derive an optimal policy π^*

- **Do you see the link to Policy Iteration?**

Advantages and Disadvantages of DP

- General approach (does not require explicit domain knowledge)
- Converges to optimal solution
- Does not require exploration-exploitation (all states are visited anyway)
- Computational costs
- Memory costs
- Availability of an explicit model $\mathrm{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$



Intermediate Summary

- What we know so far:
  - Multi-armed bandits
  - Markov Decision Processes (MDPs)
  - Policies and Value Functions
  - Optimally solve MDPs with Dynamic Programming

- What we don‘t know (yet):
  - How to find solutions in a more scalable way?
  - How to find solutions without a model?
  - How to react to unexpected events?

→ Monte Carlo Planning

Global Planning and Local Planning

- Global Planning
  - considers the **entire state space** $\mathcal{S}$ to approximate π^*
  - produces for each state $s_t \in \mathcal{S}$ a mapping to actions $a_t \in \mathcal{A}$
  - typically performed **offline** (before deploying the agent)
  - Examples: Policy and Value Iteration

- Local Planning
  - only considers the **current state** $s_t \in \mathcal{S}$ (and possible future states) to approximate $\pi^*(s_t)$
  - recommends an action $a_t \in \mathcal{A}$ for current state $s_t \in \mathcal{S}$
  - can be performed **online** (interleaving planning and execution)
  - Examples: Monte Carlo Tree Search

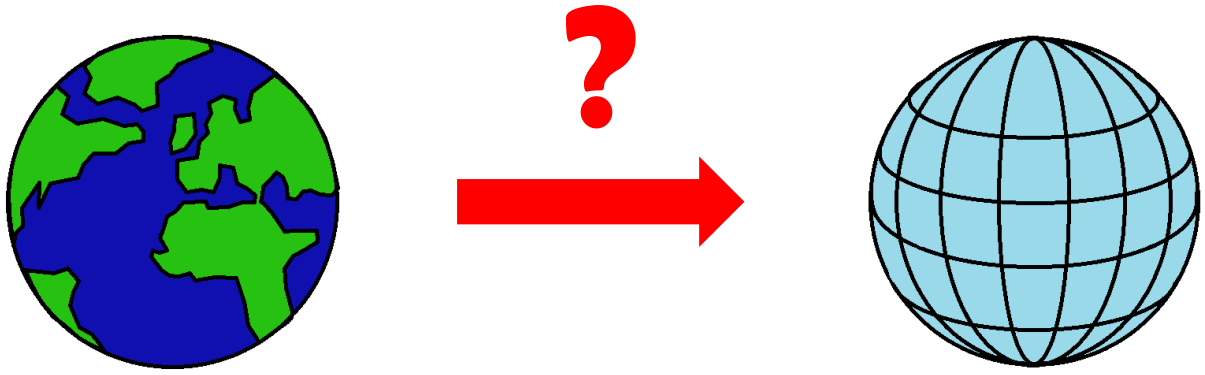
Monte Carlo Planning

- Dynamic Programming always assumes **full knowledge** of the underlying MDP $\mathrm{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$
  - Most real-world applications have extremely large state spaces
  - Especially $\mathcal{P}(s_{t+1}|s_t, a_t)$ is hard to pre-specify in practice!

- Monte Carlo Planning only requires a generative model as **blackbox simulator** $\widehat{M} \approx \mathrm{M}$
  - Given some state $s_t \in \mathcal{S}$ and action $a_t \in \mathcal{A}$, the generative model provides a sample $s_{t+1} \in \mathcal{S}$ and $r_t = \mathcal{R}(s_t, a_t)$
  - Can be used to approximate V^* or Q^* via statistical sampling
  - Requires minimal domain knowledge ($\widehat{M}$ can be easily replaced)

Explicit Model vs. Generative Model

- Generative model can be easier implemented than explicit probability distributions!



Explicit Model vs. Generative Model

- Generative model can be easier implemented than explicit probability distributions!

- Example: Throwing two dices

  - Explicit Model:

Total	2	3	4	5	6	7	8	9	10	11	12
Prob.	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

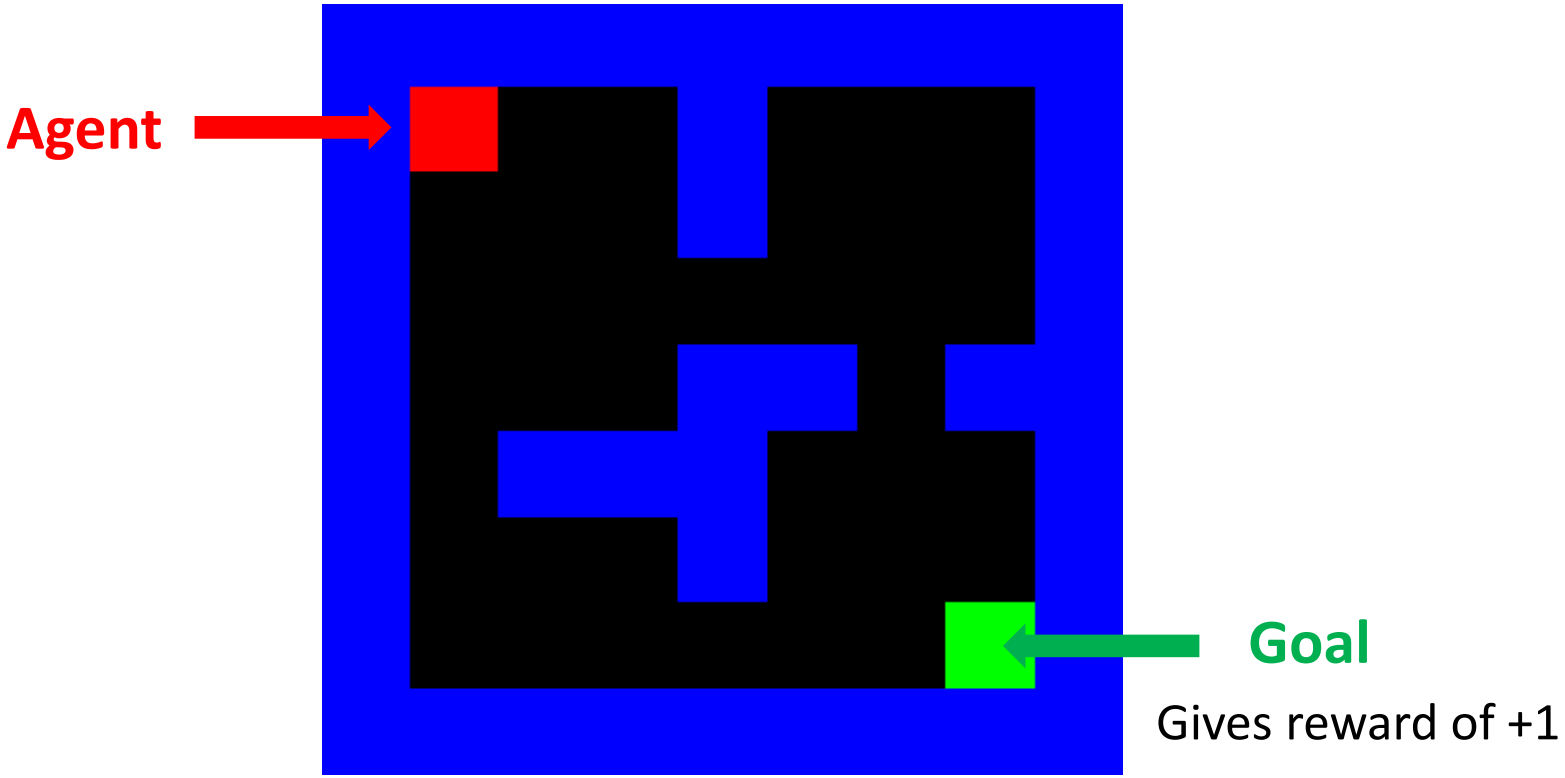
  - Generative Model:
    - Generate two random numbers ranging from 1 to 6 

Monte Carlo Rollouts

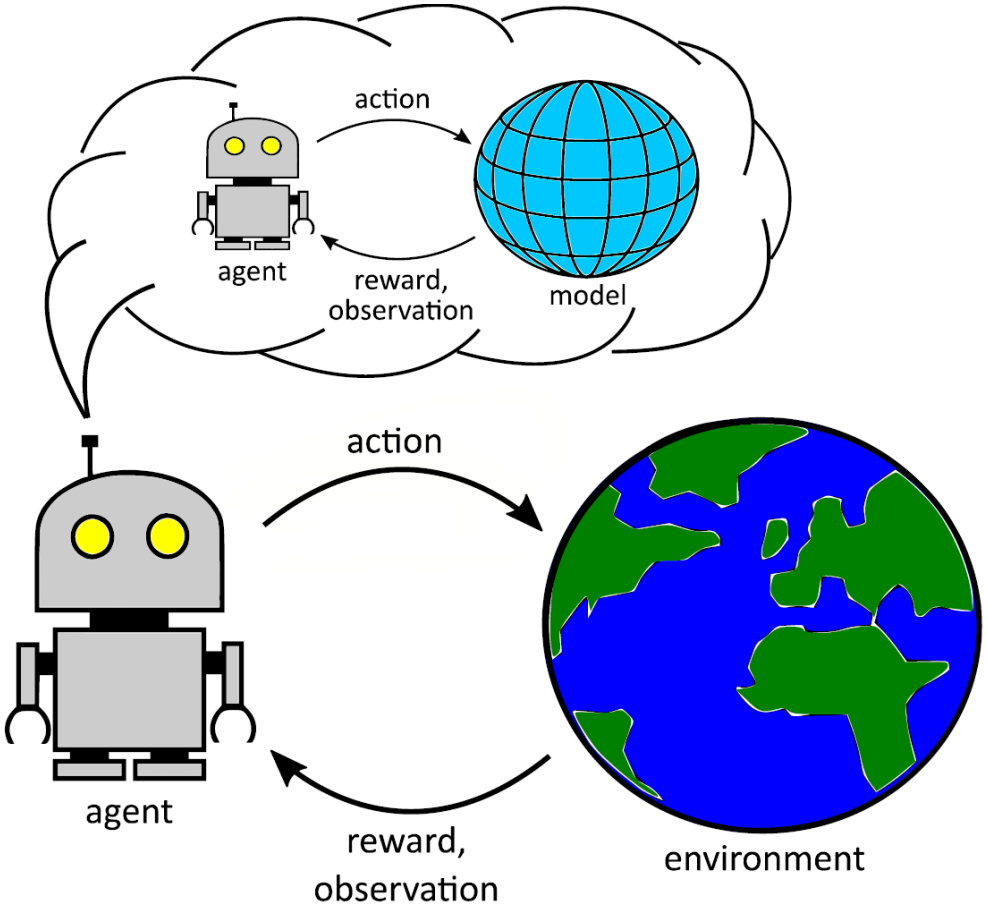
- **Goal:** Given a state $s_t \in \mathcal{S}$ and a policy $\pi_{rollout}$, we want to find the action $a_t \in \mathcal{S}$ which maximizes $Q^{\pi_{rollout}}(s_t, a_t) = \mathbb{E}[G_t | s_t, a_t]$

- **Approach**: Given a computation budget of K simulations and a horizon h
  - Sample K action sequences (= plans) of length h from $\pi_{rollout}$
  - Simulate all plans with generative model $\widehat{\mathrm{M}}$ and compute the return G_t for each plan
  - Update estimate of $Q^{\pi_{rollout}}(s_t, a_t) = \mathbb{E}[G_t | s_t, a_t]$
  - **Finally:** Select action $a_t \in \mathcal{S}$ with highest $Q^{\pi_{rollout}}(s_t, a_t)$

Monte Carlo Rollouts Demonstration



Automated Planning: Recap



Monte Carlo Rollouts and Policy Iteration

- **Goal:** Given a state $s_t \in \mathcal{S}$ and a policy $\pi_{rollout}$, we want to find the action $a_t \in \mathcal{A}$ which maximizes $Q^{\pi_{rollout}}(s_t, a_t) = \mathbb{E}[G_t | s_t, a_t]$

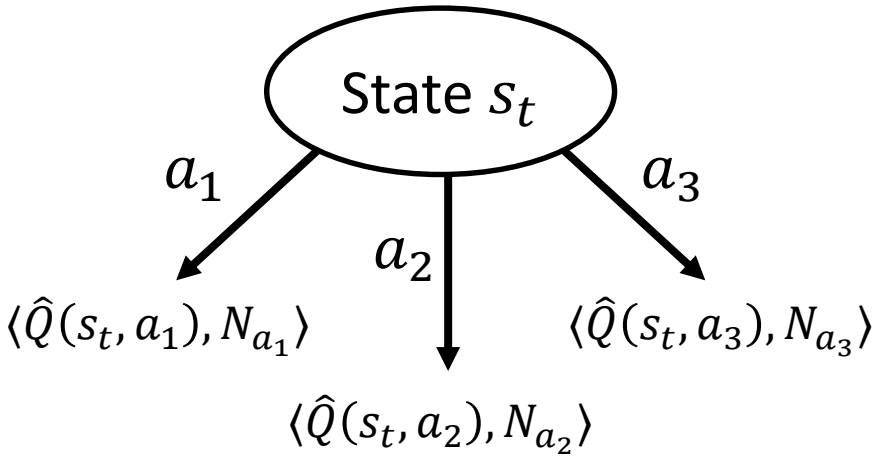
- **Approach**: Given a computation budget of K simulations and a horizon h
  - Sample K action sequences (= plans) of length h from $\pi_{rollout}$
  - Simulate all plans with generative model $\widehat{\mathrm{M}}$ and compute the return G_t for each plan
  - Update estimate of $Q^{\pi_{rollout}}(s_t, a_t) = \mathbb{E}[G_t | s_t, a_t]$ ➡ Evaluation
  - **Finally:** Select action $a_t \in \mathcal{S}$ with highest $Q^{\pi_{rollout}}(s_t, a_t)$ ➡ Improvement

- **Monte Carlo Rollouts lead to a better policy than $\pi_{rollout}$**

Monte Carlo Tree Search (MCTS)

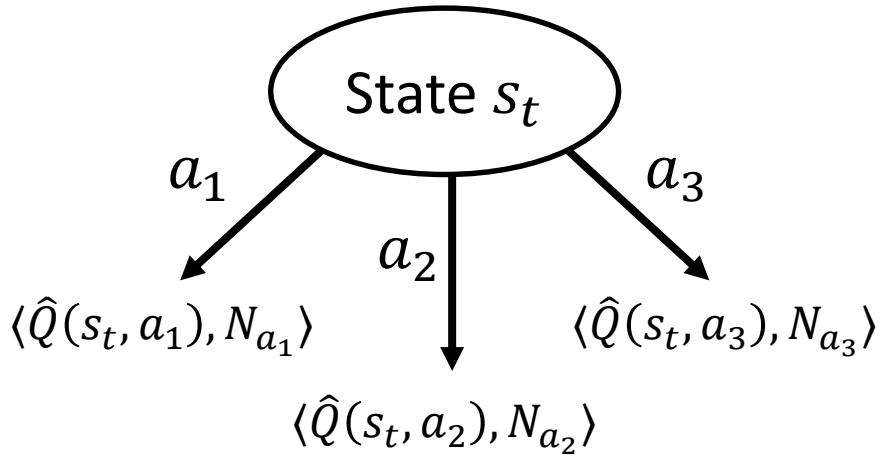
- **Goal:** Given a state $s_t \in \mathcal{S}$ and a policy $\pi_{rollout}$, we want to find the action $a_t \in \mathcal{A}$ which maximizes $Q^*(s_t, a_t) = \mathbb{E}[G_t | s_t, a_t]$

- **Approach**: Incrementally construct and traverse a search tree given a computation budget of K simulations and a horizon h
  - nodes represent states $s_t \in \mathcal{S}$ (and actions $a_t \in \mathcal{A}$)
  - search tree is used to „learn“ $\hat{Q} \approx Q^*$ via blackbox simulation



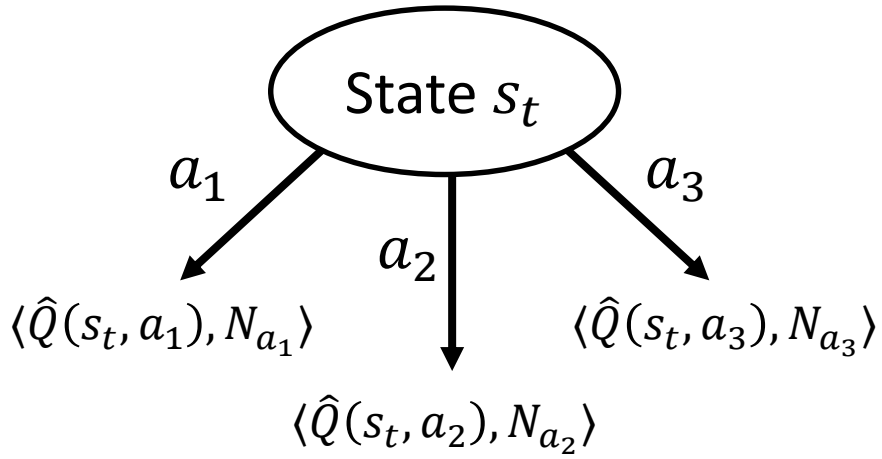
MCTS Phases

- MCTS consists of four phases:
  1. Selection
  2. Expansion
  3. Evaluation
  4. Backup



MCTS Phases - Selection

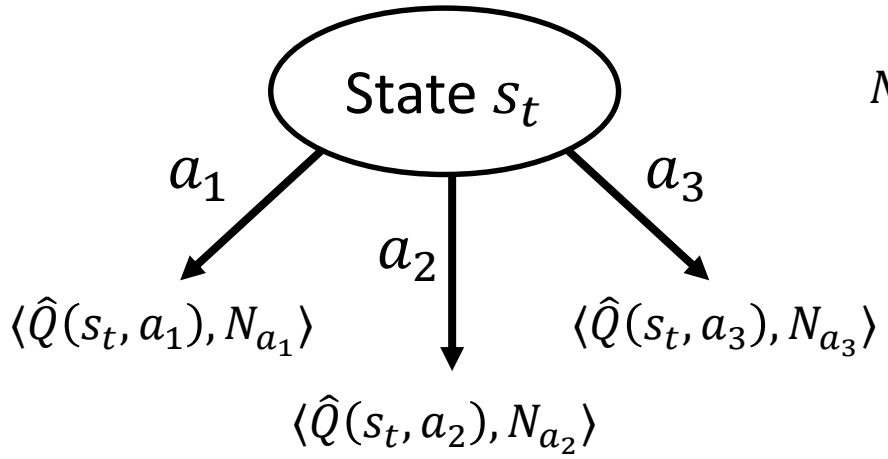
- MCTS phases:
  1. **Selection:**
     - given state $s_t \in \mathcal{S}$, select $a_t \in \mathcal{A}$ (until node is a leaf)
     - **How can we do this?** → **exploration/exploitation**
  2. Expansion
  3. Evaluation
  4. Backup



MCTS Phases - Selection

- MCTS phases:
  1. **Selection:**
     - given state $s_t \in \mathcal{S}$, select $a_t \in \mathcal{A}$ (until node is a leaf)
     - **how can we do this?** → **multi-armed bandits (e.g., UCB1)**
  2. Expansion
  3. Evaluation
  4. Backup

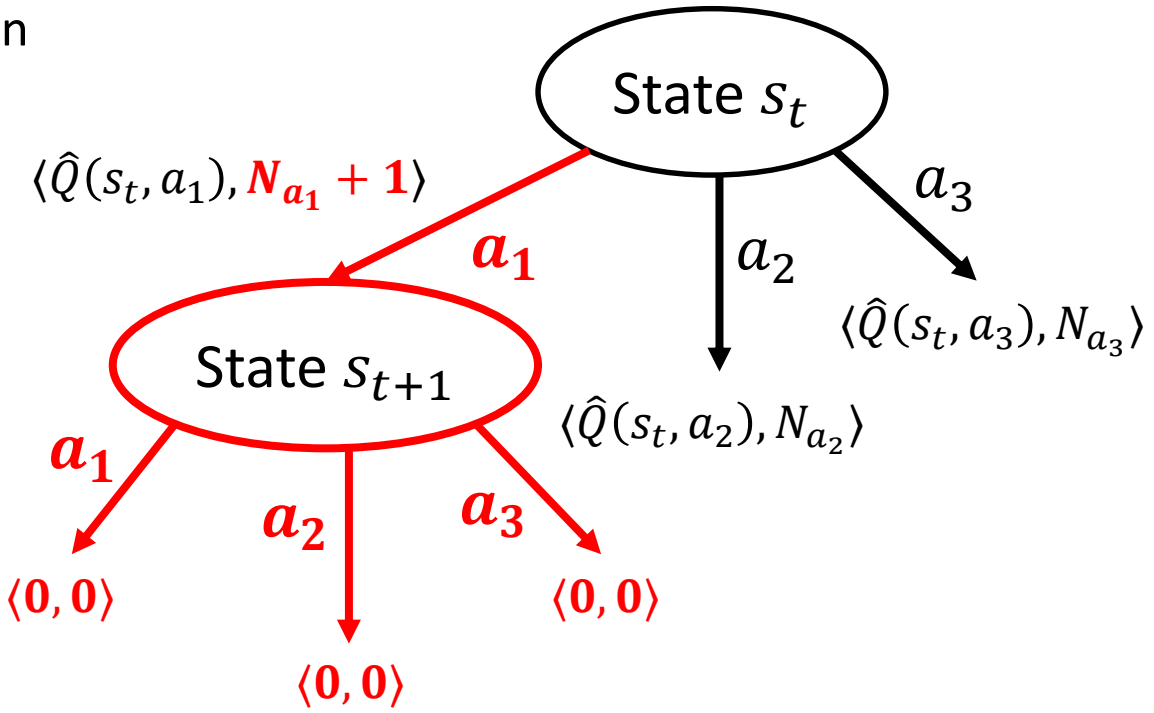
$$\boldsymbol{\pi_{tree}(s_t) = argmax_{a_t \in \mathcal{A}}\{\hat{Q}(s_t, a_t) + c\sqrt{\frac{2logN}{N_{a_t}}}\}}$$



$$N = \sum_{a_t \in \mathcal{A}} N_{a_t}$$

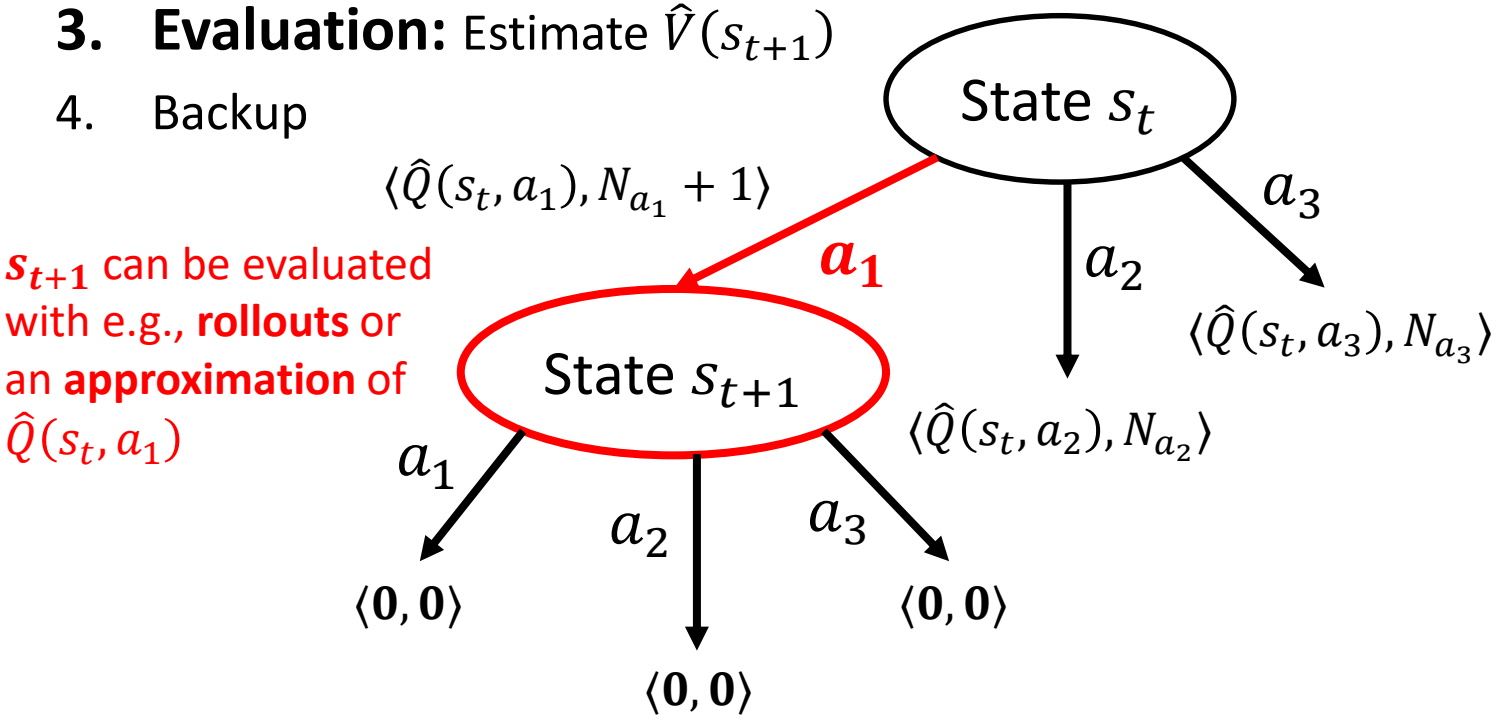
MCTS Phases - Expansion

- MCTS phases:
  1. Selection
  2. **Expansion**
  3. Evaluation
  4. Backup



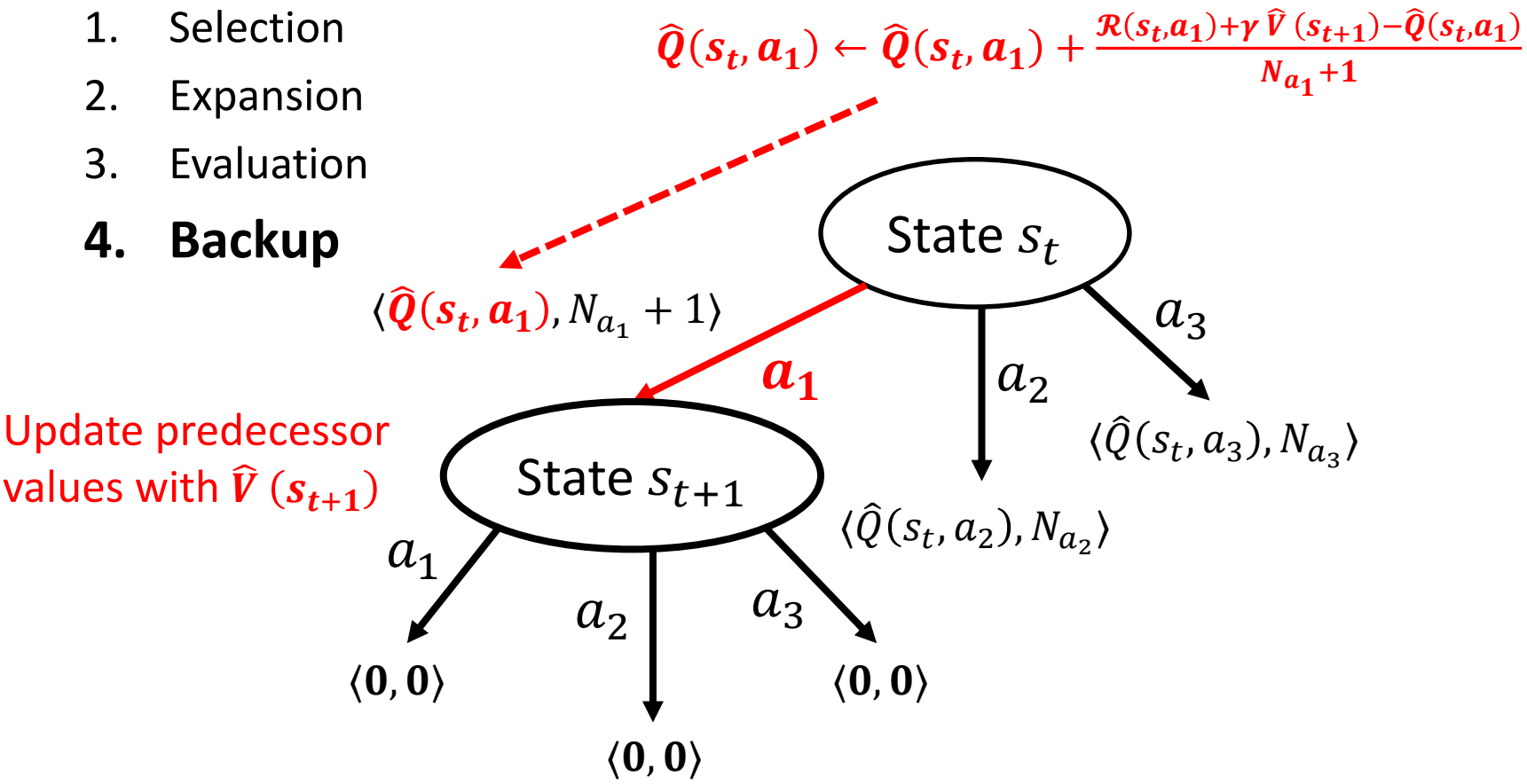
MCTS Phases - Evaluation

- MCTS phases:
  1. Selection
  2. Expansion
  3. **Evaluation:** Estimate $\hat{V}(s_{t+1})$
  4. Backup



MCTS Phases - Backup

- MCTS phases:
  1. Selection
  2. Expansion
  3. Evaluation
  4. **Backup**



MCTS and Policy Iteration

- MCTS phases:
  1. **Selection** → **Policy Improvement** (e.g., maximize UCB1)
  2. Expansion
  3. **Evaluation**
  4. **Backup** → **Policy Evaluation** (estimate $Q^*(s_t, a_t)$)

- Given infinite simulations K, MCTS converges to the optimal best first tree for state $s_t \in \mathcal{S}$, which corresponds to the **optimal policy** $\pi^*(s_t)$

Summary

- Dynamic Programming vs. Monte Carlo Planning

Algorithm	Evaluation	Improvement	Model Type	Optimality?
Policy Iteration	V^{π} and/or Q^{π} of current π	Maximize V^{π} and/or Q^{π}	Explicit	Yes
Value Iteration	V^{n} via Bellman Update	Maximize current V^{n}	Explicit	Yes
MC Rollouts	$Q^{\pi_{rollout}}$ of $\pi_{rollout}$	Maximize $Q^{\pi_{rollout}}$ (at the end only)	Generative	No (depends on $\pi_{rollout}$)
MCTS	$Q^{\pi_{tree}}$ of current π_{tree}	Maximize $Q^{\pi_{tree}}$	Generative	Yes (with inifinite time)

Summary

- What we know so far:
  - Multi-armed bandits
  - Markov Decision Processes (MDPs)
  - Policies and Value Functions
  - Optimally solve MDPs with Dynamic Programming
  - Approximately solve MDPs with Monte Carlo Search

- What we don‘t know (yet):
  - How to find solutions without a model?

Thank you!