Monte Carlo Tree Search (MCTS) & AlphaZero

Lucas Janson and Sham Kakade

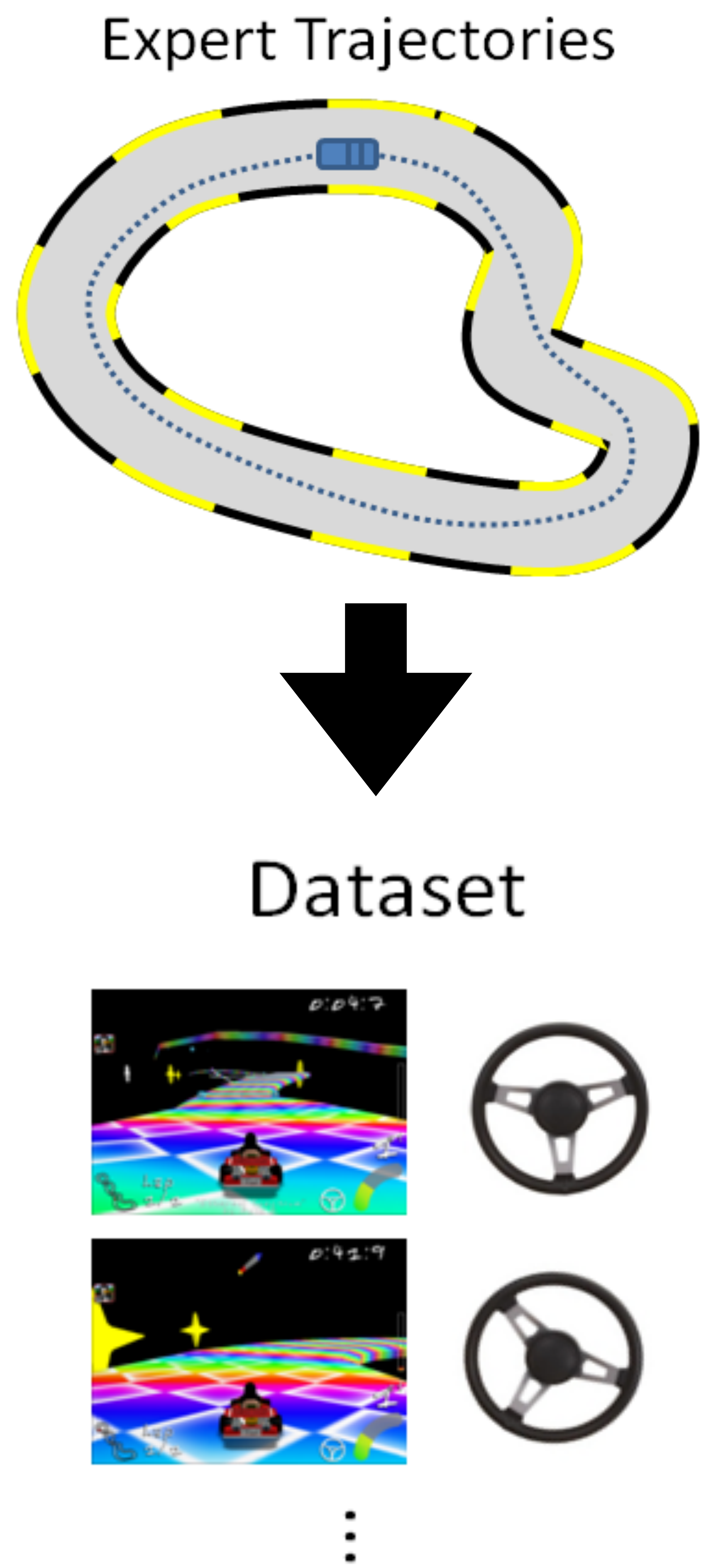
CS/Stat 184: Introduction to Reinforcement Learning
Fall 2023

Today

- Recap
- Game Playing: AlphaBeta Search/Rule Based Systems
- MCTS
- AlphaZero and Self-Play

Recap

Let's formalize the offline IL Setting and the Behavior Cloning algorithm



Finite horizon MDP $\mathscr{M}$

Ground truth reward $r(s, a) \in [0,1]$ is unknown;
Assume the expert has a good policy $\pi^\star$ (not necessarily opt)

We have a dataset of M trajectories: $\mathscr{D} = \{\tau_1, \ldots \tau_M\}$,
where $\tau_i = (s_h^i, a_h^i)_{h=0}^{H-1} \sim \rho_{\pi^\star}$

Goal: learn a policy from $\mathscr{D}$ that is as good as the expert $\pi^\star$

Theorem: IL is (almost) as easy as SL

$$\widehat{\pi} = \arg\min_{\pi \in \Pi} \sum_{i=1}^{M} \sum_{h=0}^{H-1} \ell\left(\pi, s_h^i, a_h^i\right)$$

Note a training and testing "mismatch"

$$\mathbb{1}\left(\widehat{\pi}(s_h^i) \neq a_h^i\right)$$

Theorem [BC Performance]:

suppose we assume supervised learning succeeds, with ϵ classification error:

$$\mathbb{E}_{\tau \sim \rho_{\pi^\star}} \left[\frac{1}{H} \sum_{h=0}^{H-1} \mathbf{1}\left[\widehat{\pi}(s_h) \neq \pi^\star(s_h)\right] \right] \leq \epsilon,$$

(where $\pi^\star$ is the experts policy, which need not be optimal)

then, under μ, we have:

$$|V^{\pi^\star} - V^{\widehat{\pi}}| \leq H^2 \epsilon$$

The quadratic amplification is annoying

Proof:

By the PDL

$$|V^{\pi^\star}(s) - V^{\hat{\pi}}(s)| = \left|\mathbb{E}_{\tau\sim\rho_{\pi^\star}}\left[\sum_{h=0}^{H-1} A_h^{\hat{\pi}}(s_h, a_h)\right]\right|$$

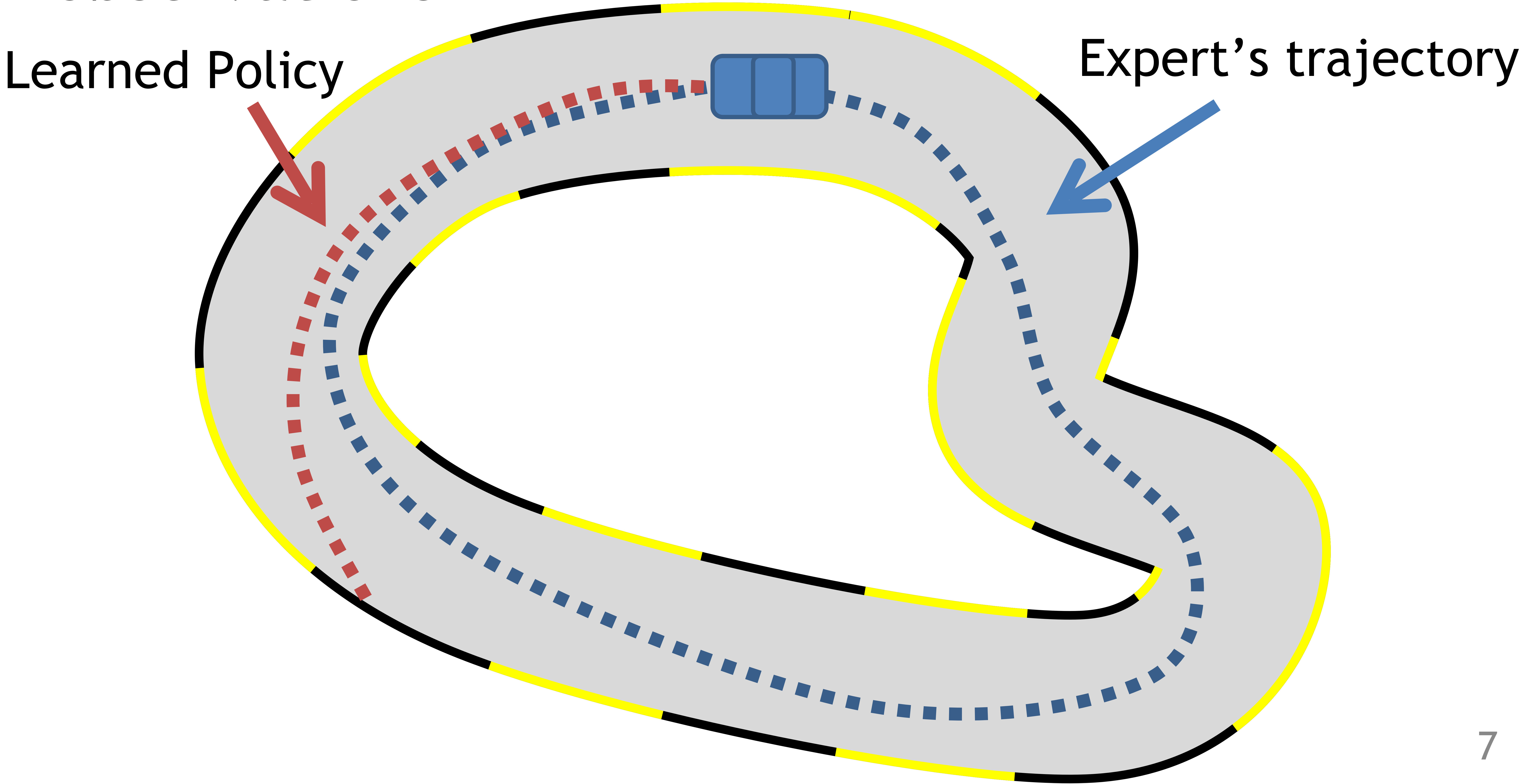
$$= \left|\mathbb{E}_{s_1,\ldots s_h\sim\rho_{\pi^\star}}\left[\sum_{h=0}^{H-1} A_h^{\hat{\pi}}(s_h, \pi^\star(s_h))\right]\right|$$

$$\leq H\left|\mathbb{E}_{\tau\sim\rho_{\pi^\star}}\left[\sum_{h=0}^{H-1} \mathbf{1}\left[\hat{\pi}(s_h) \neq \pi^\star(s_h)\right]\right]\right|$$

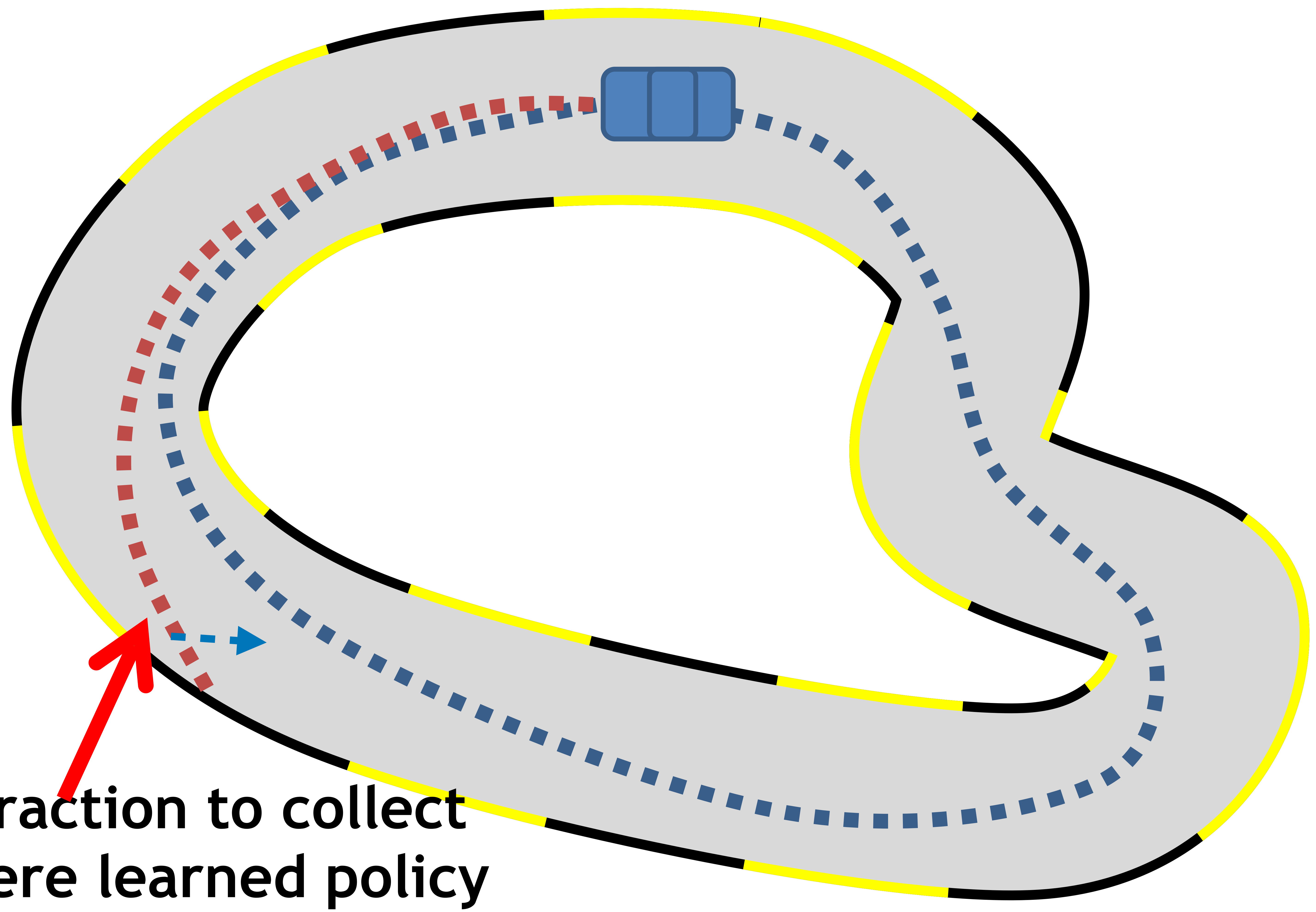
$$\leq H^2\epsilon$$

What could go wrong?

- Predictions affect future inputs/ observations



Intuitive solution: **Interaction**

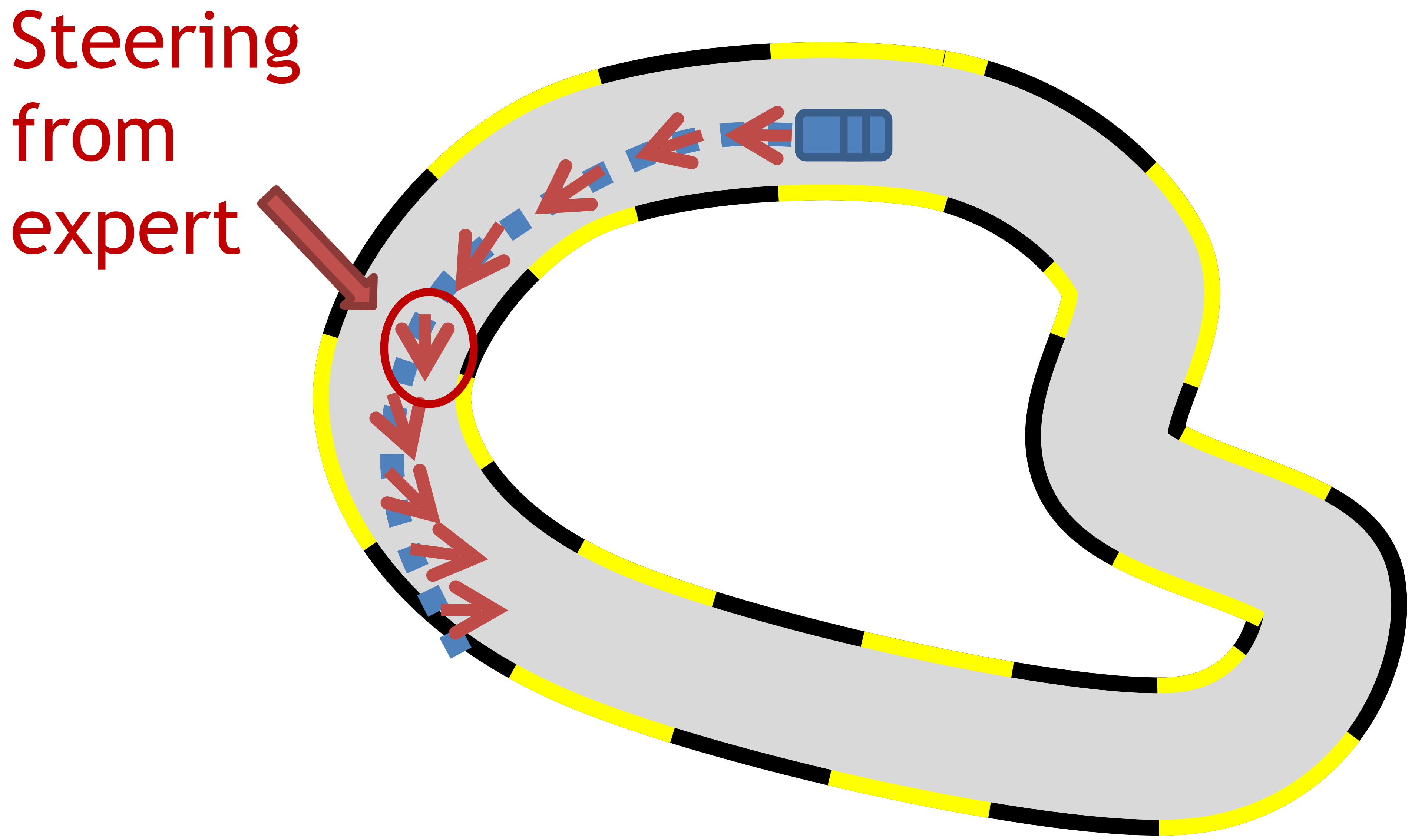


Use interaction to collect data where learned policy goes

DAgger: Dataset Aggregation [Ross11a]

1st iteration

Execute π_1 and Query Expert

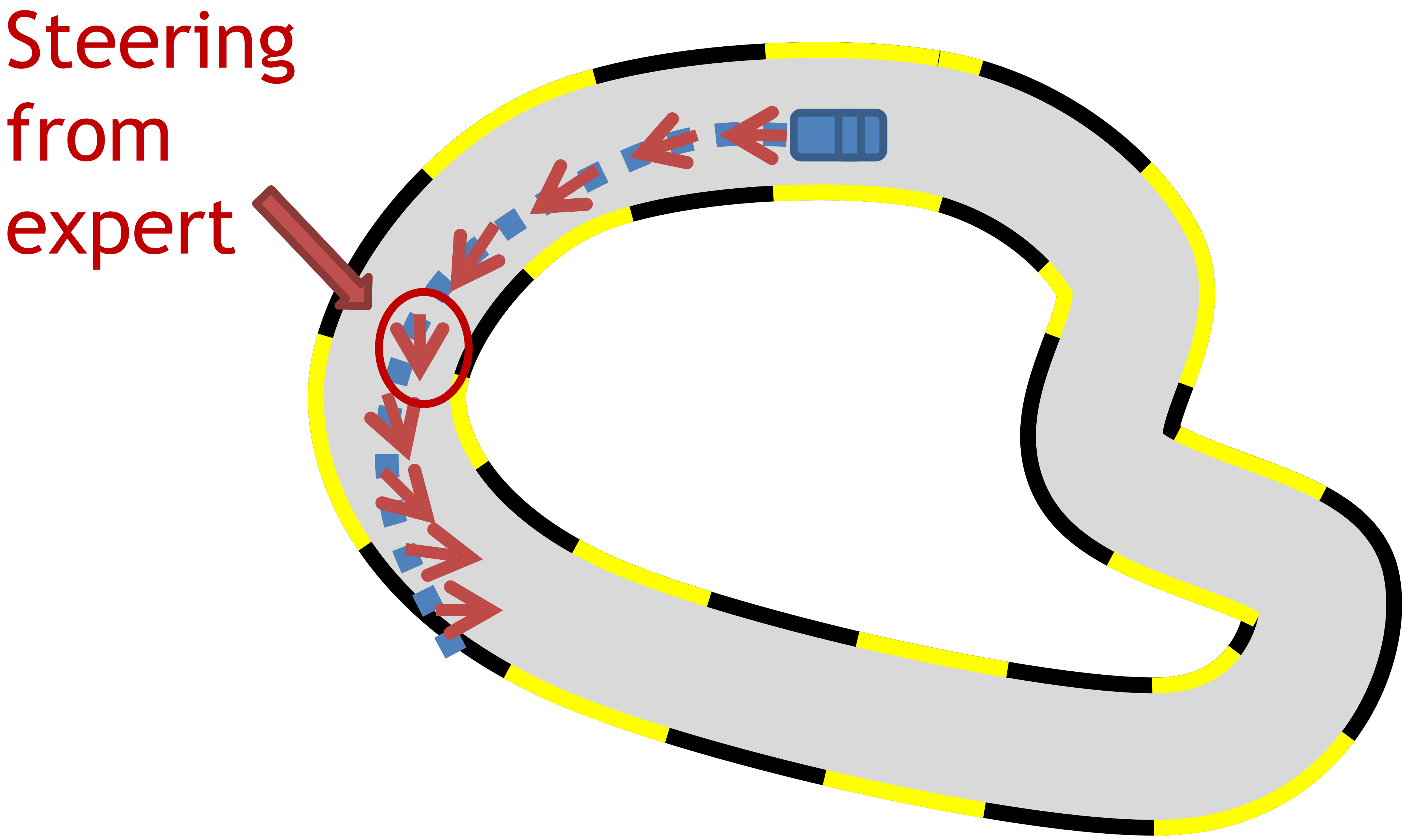


DAgger: Dataset Aggregation

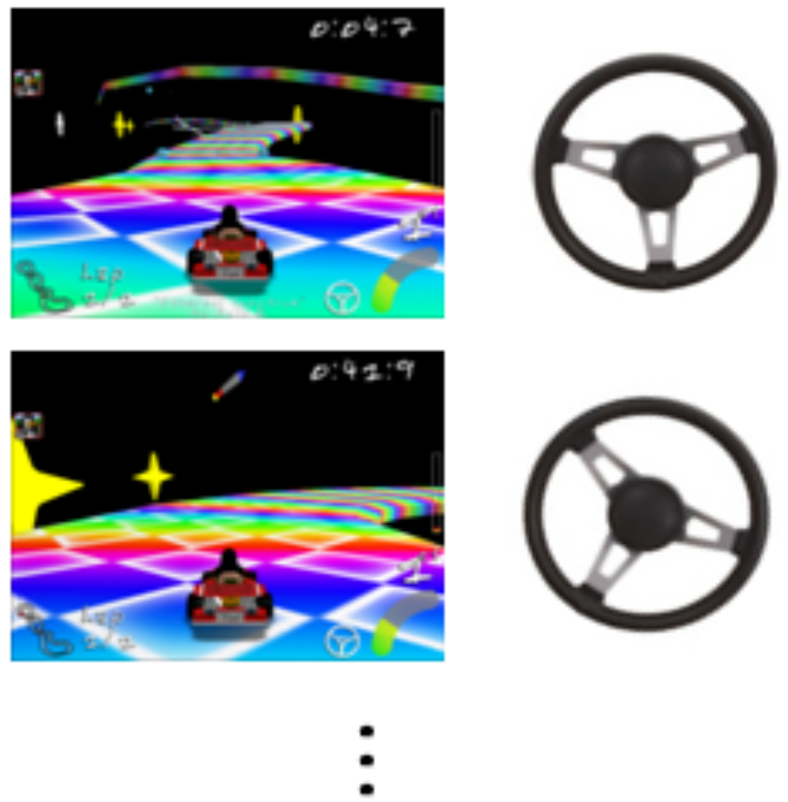
[Ross11a]

1st iteration

Execute π_1 and Query Expert



New Data

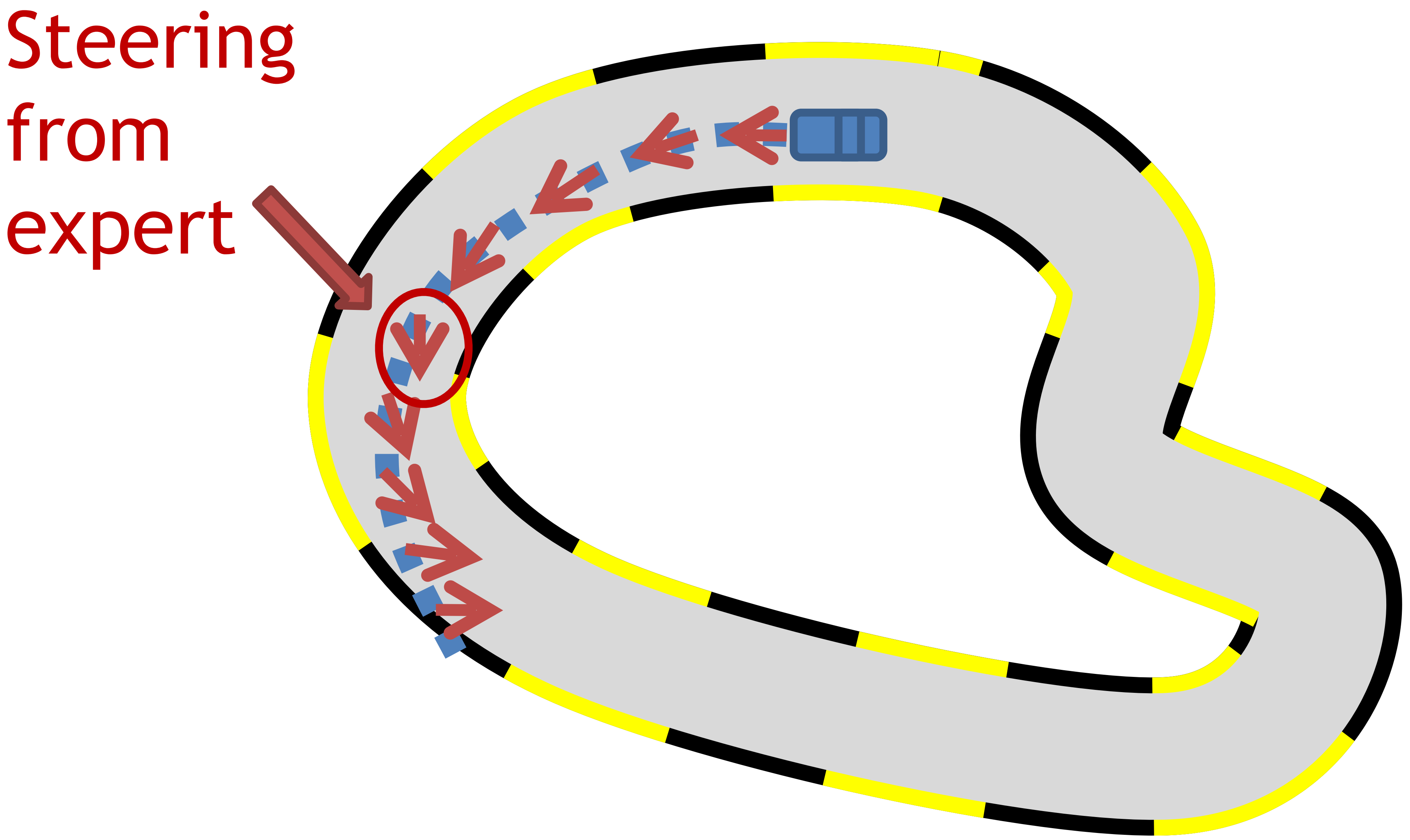


DAgger: Dataset Aggregation

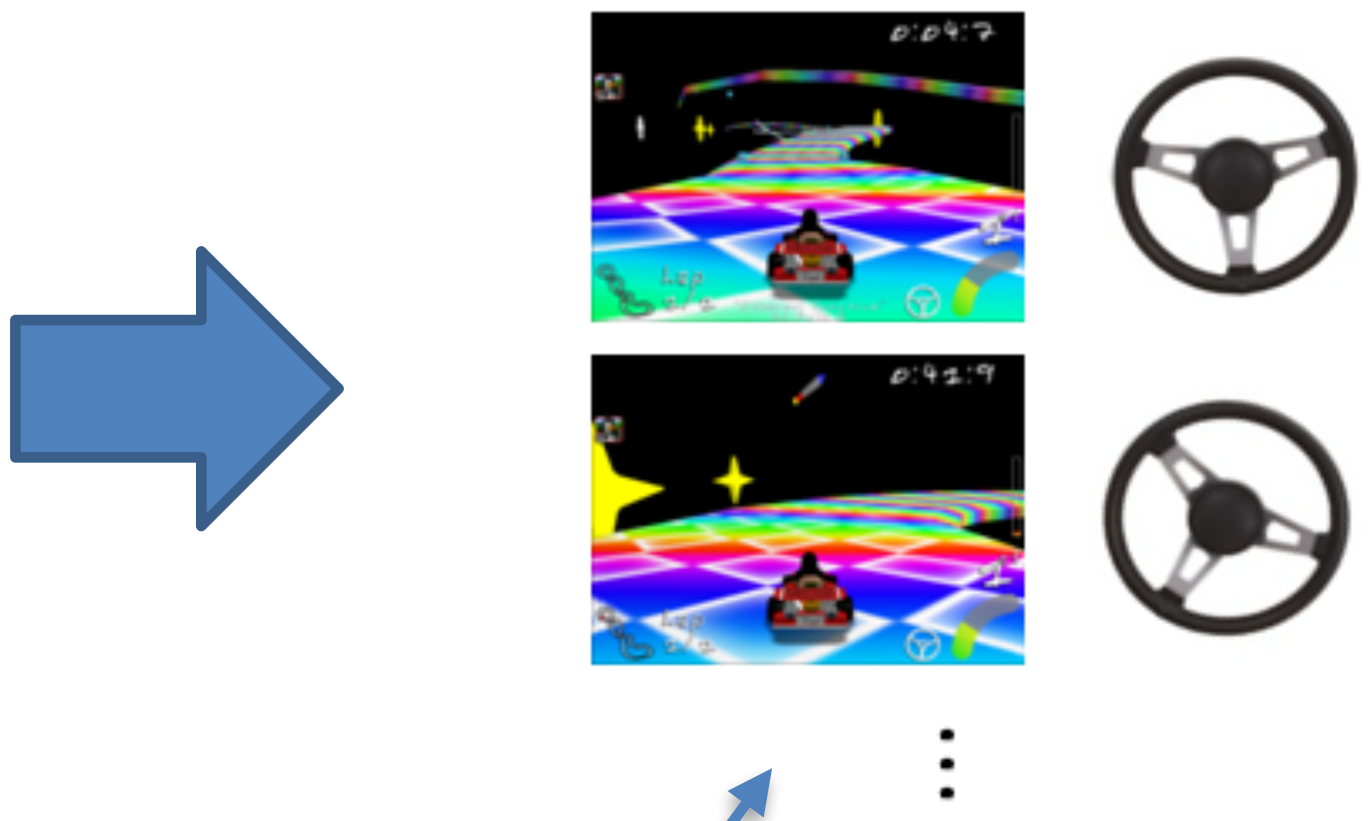
[Ross11a]

1st iteration

Execute π_1 and Query Expert



New Data



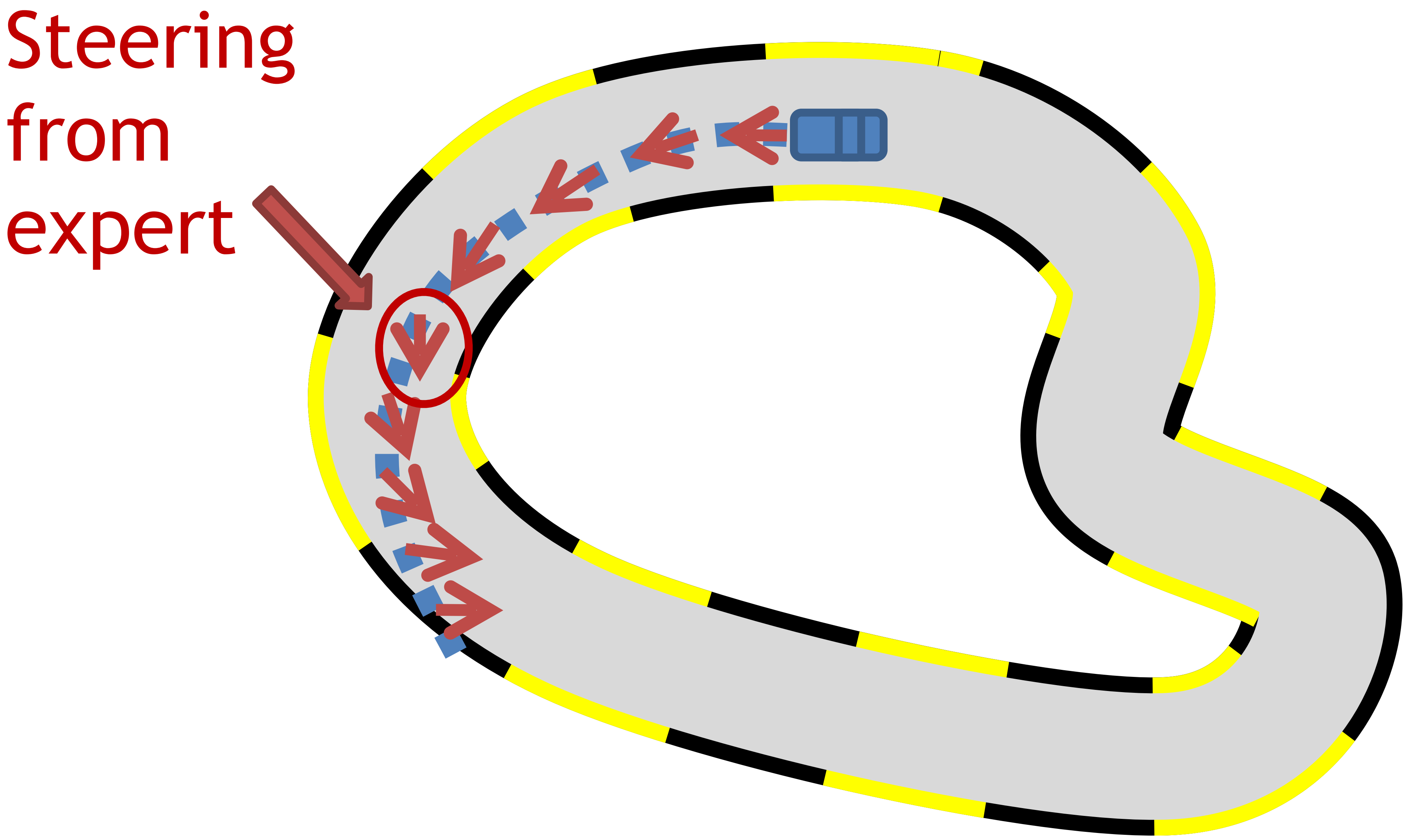
States from
the learned policy

DAgger: Dataset Aggregation

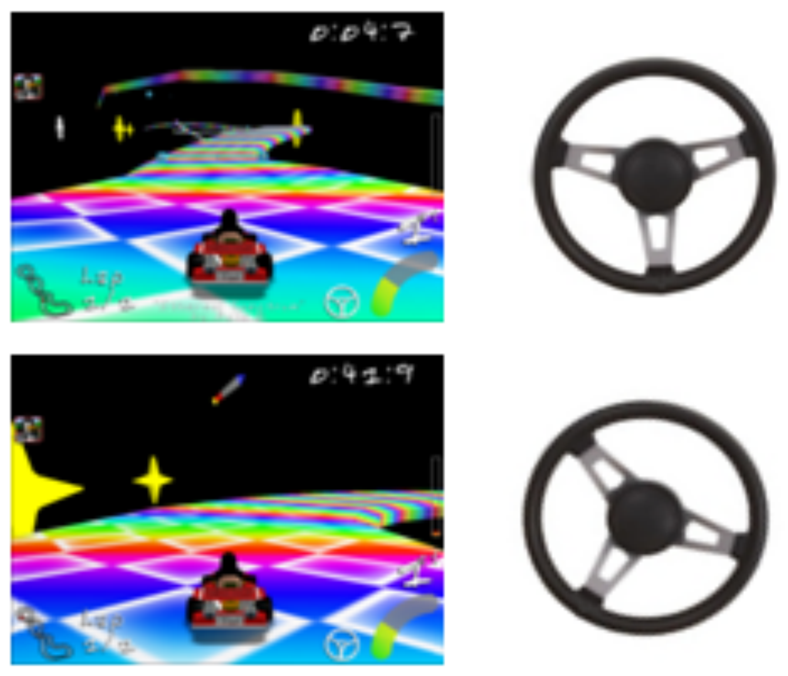
[Ross11a]

1st iteration

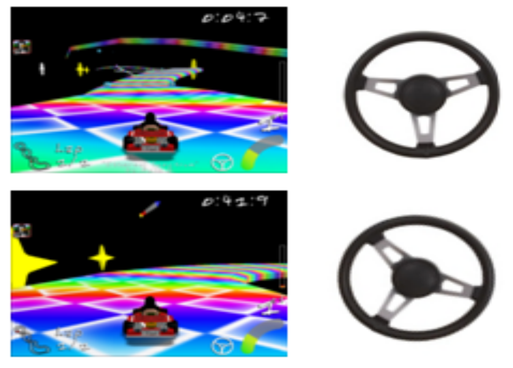
Execute π_1 and Query Expert



New Data

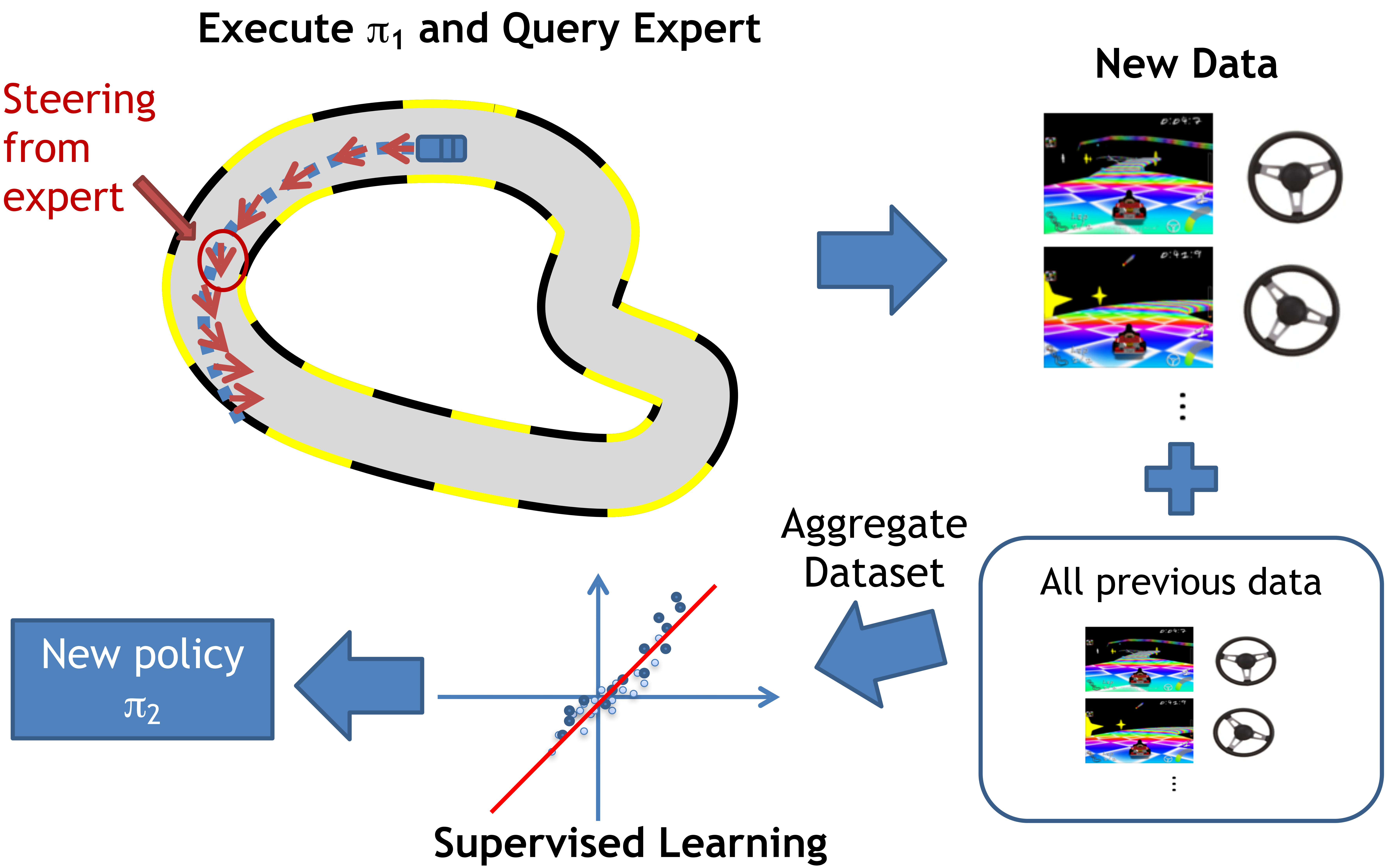


All previous data



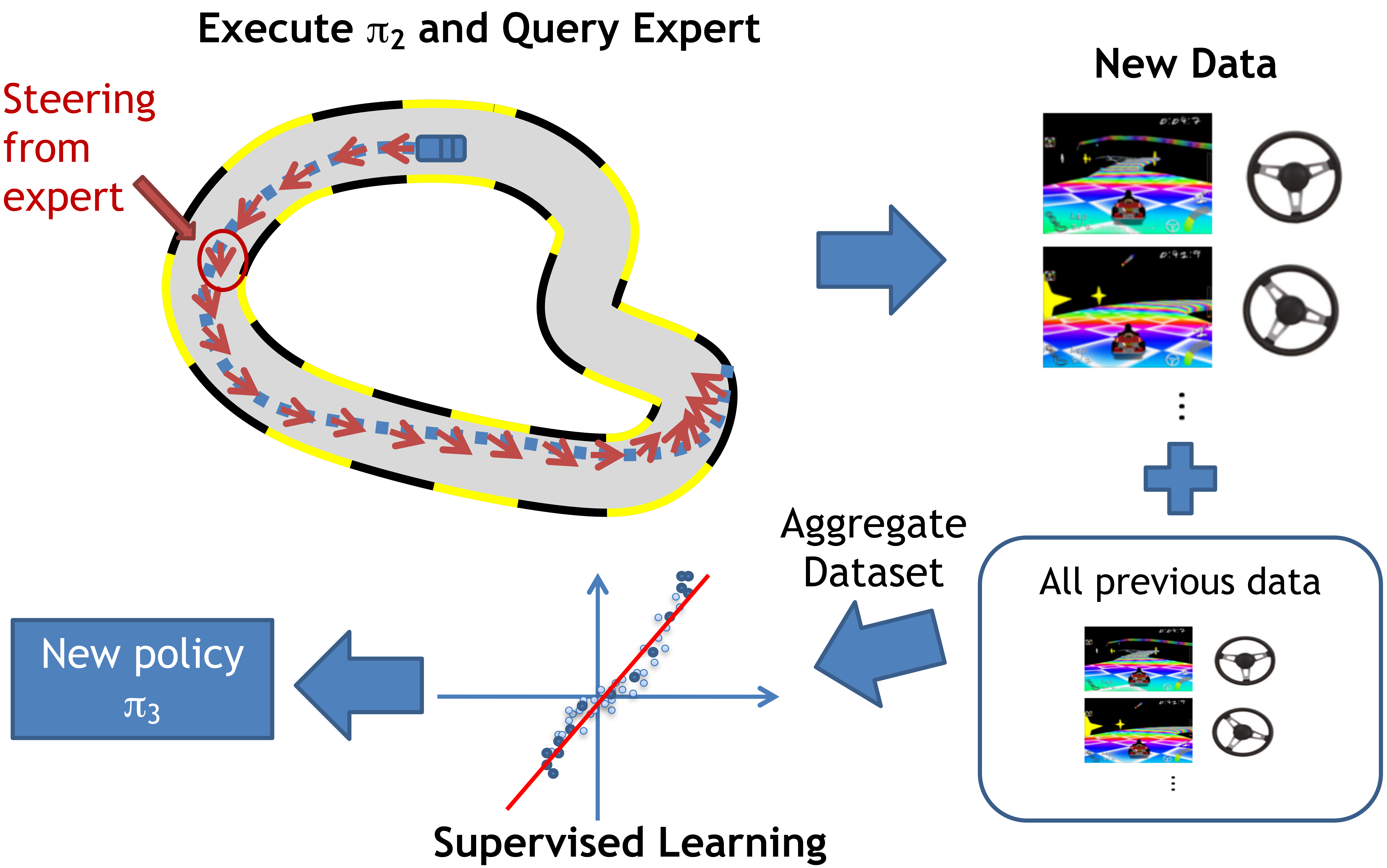
DAgger: Dataset Aggregation [Ross11a]

1st iteration



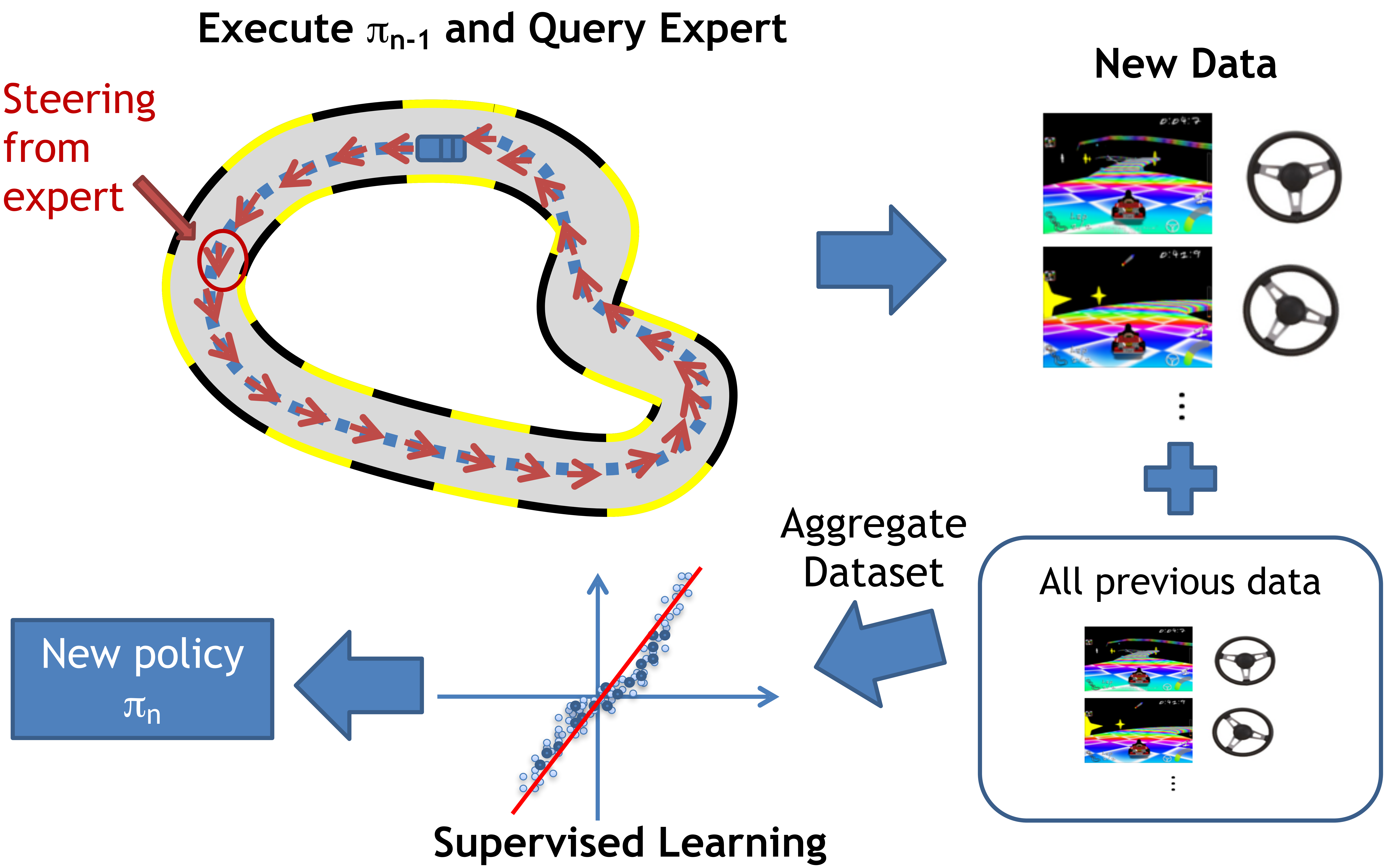
DAgger: Dataset Aggregation [Ross11a]

2nd iteration



DAgger: Dataset Aggregation [Ross11a]

n^{th} iteration



The DAgger algorithm

Initialize π^0, and dataset $\mathcal{D} = \emptyset$

For $t = 0 \to T-1$:

1. W/ π^t, generate dataset of trajectories $\mathcal{D}^t = \{\tau_1, \tau_2, \ldots\}$ where for all trajectories $s_h \sim \rho_{\pi^t}$, $a_h = \pi^\star(s_h)$
2. **Data aggregation:** $\mathcal{D} = \mathcal{D} \cup \mathcal{D}^t$
3. **Update policy via Supervised-Learning**: $\pi^{t+1} = \text{SL}\left(\mathcal{D}\right)$

In practice, the DAgger algorithm requires less human labeled data than BC.

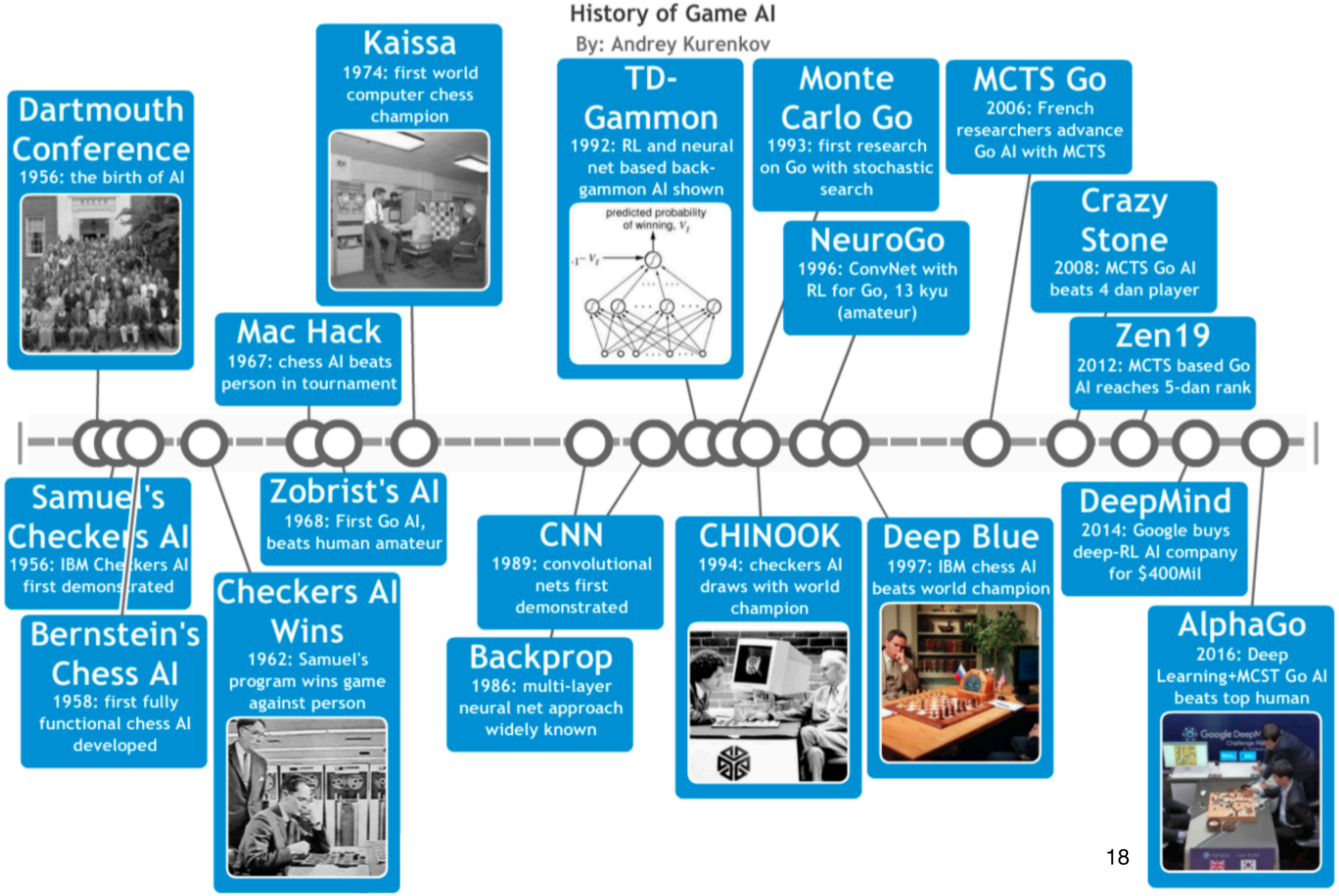
[Informal Theorem] Under more assumptions + assuming ϵ SL error is achievable, the DAgger algorithm has error: $|V^{\pi^\star} - V^{\widehat{\pi}}| \leq H\epsilon$

Today:

Today

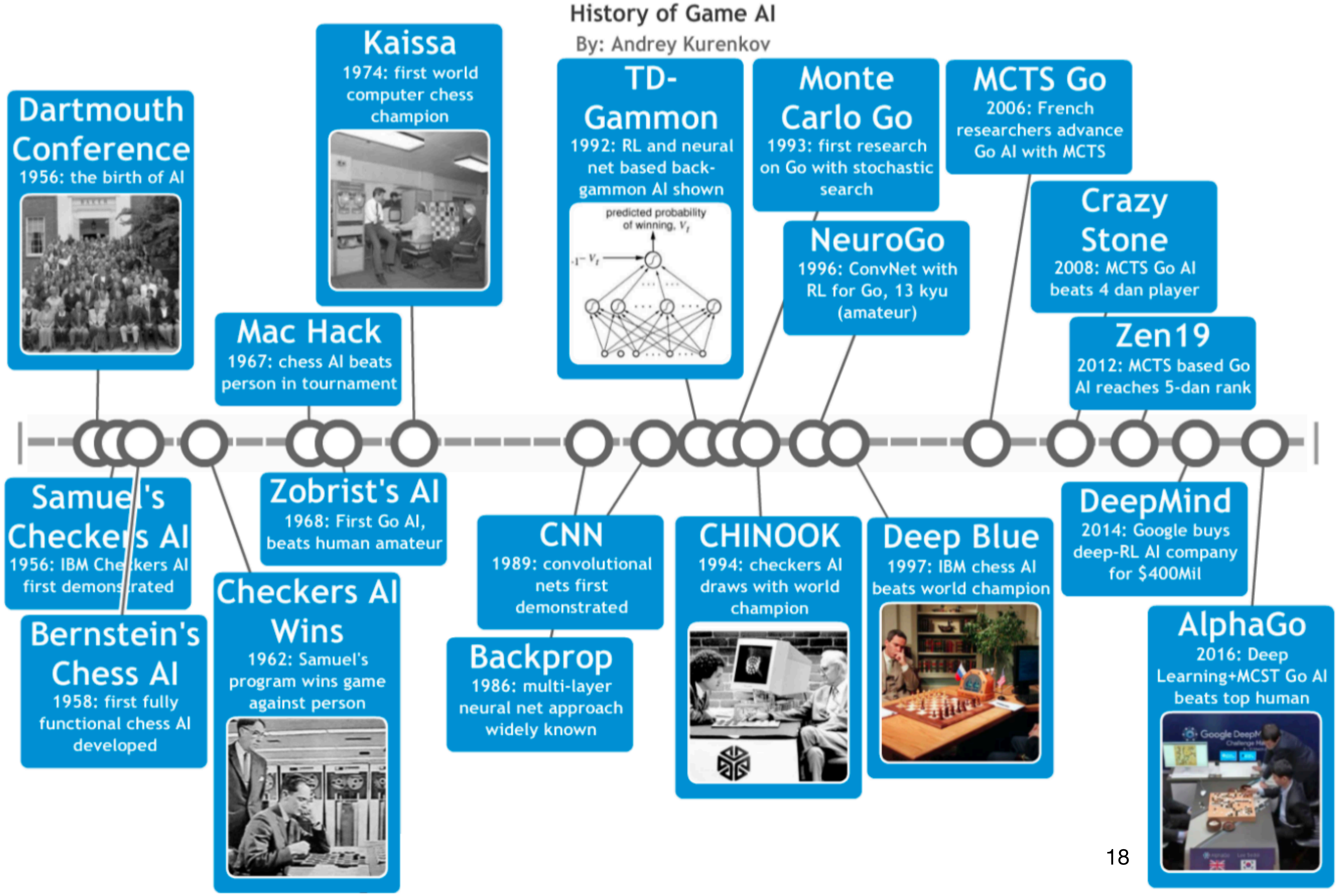
- Recap
- Game Playing: AlphaBeta Search/Rule Based Systems
- MCTS
- AlphaZero

Fascination with AI and Games…



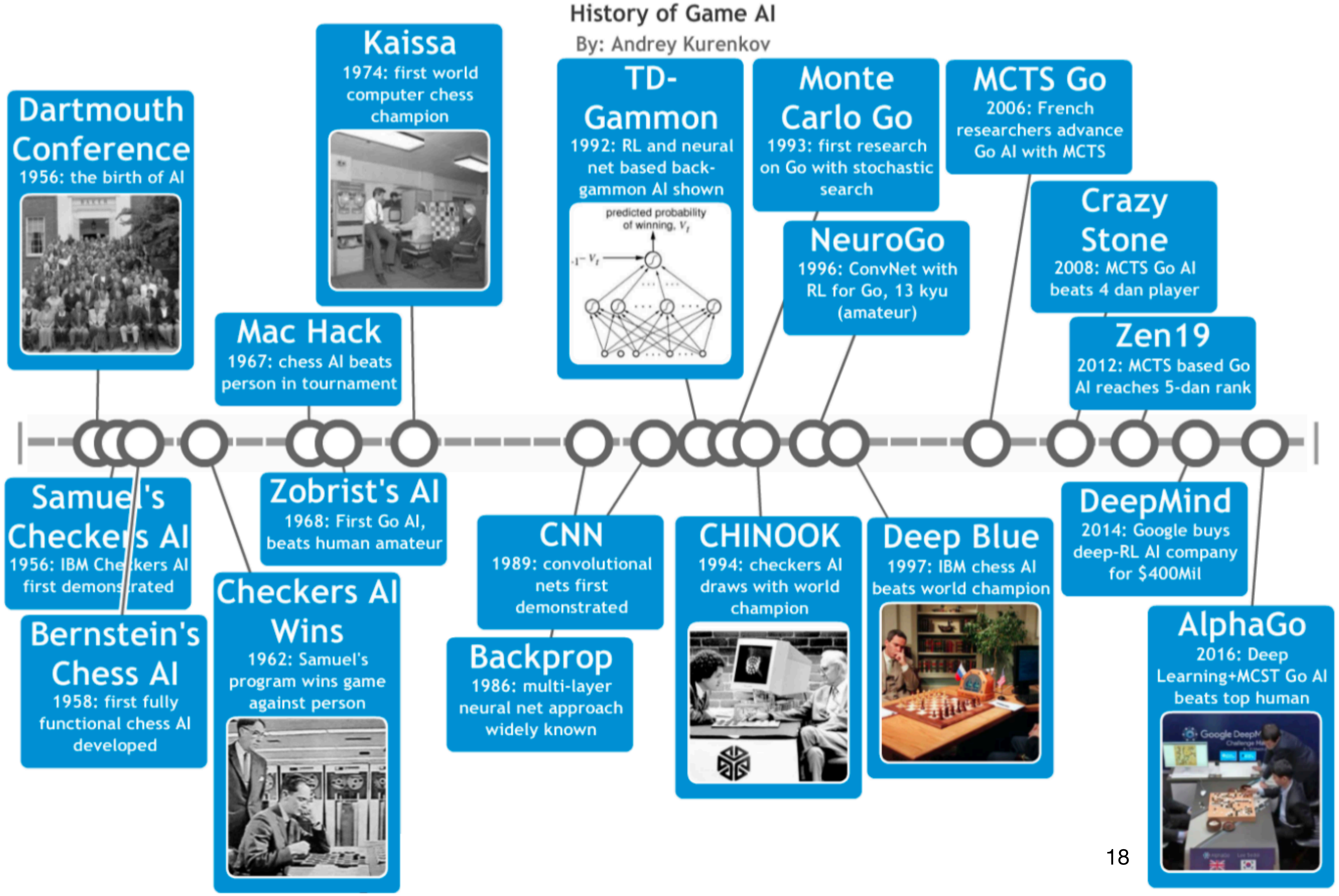
Fascination with AI and Games…

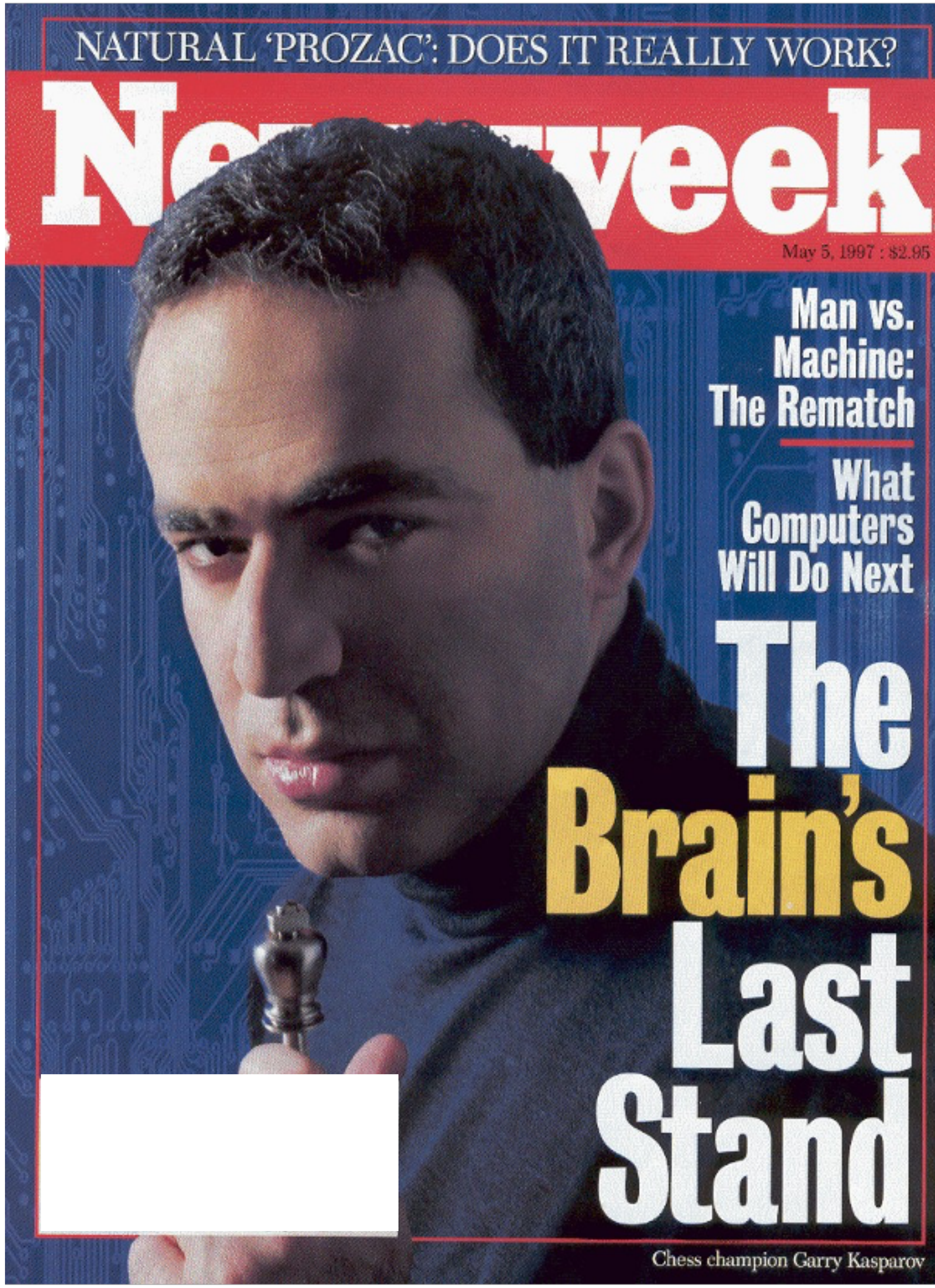
- DeepBlue v. Kasparov (1997)
  - winning in chess wasn’t a good indicator of “progress in AI”



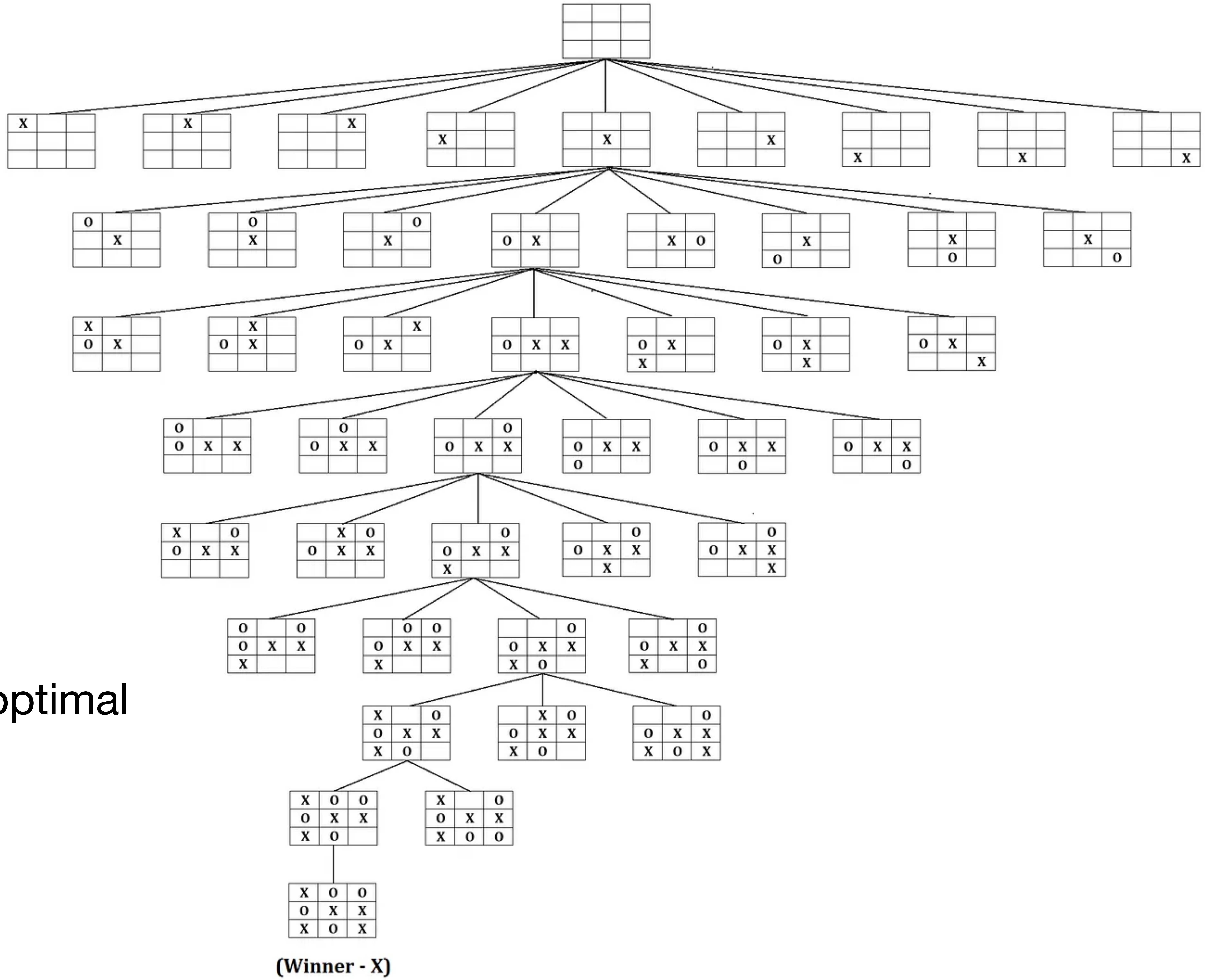
Fascination with AI and Games…

- DeepBlue v. Kasparov (1997)
  - winning in chess wasn't a good indicator of "progress in AI"





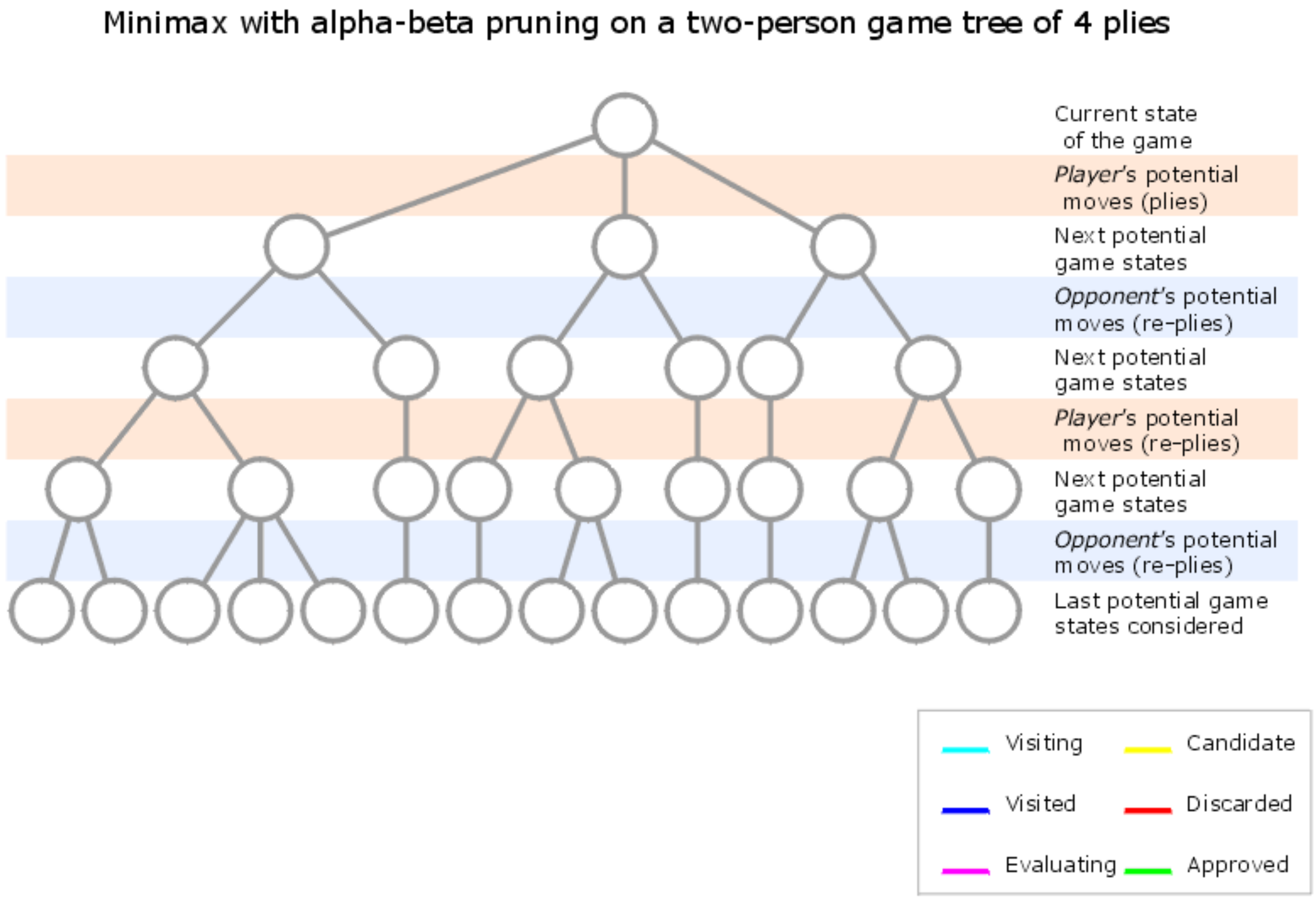
Game Trees for Two Player Games



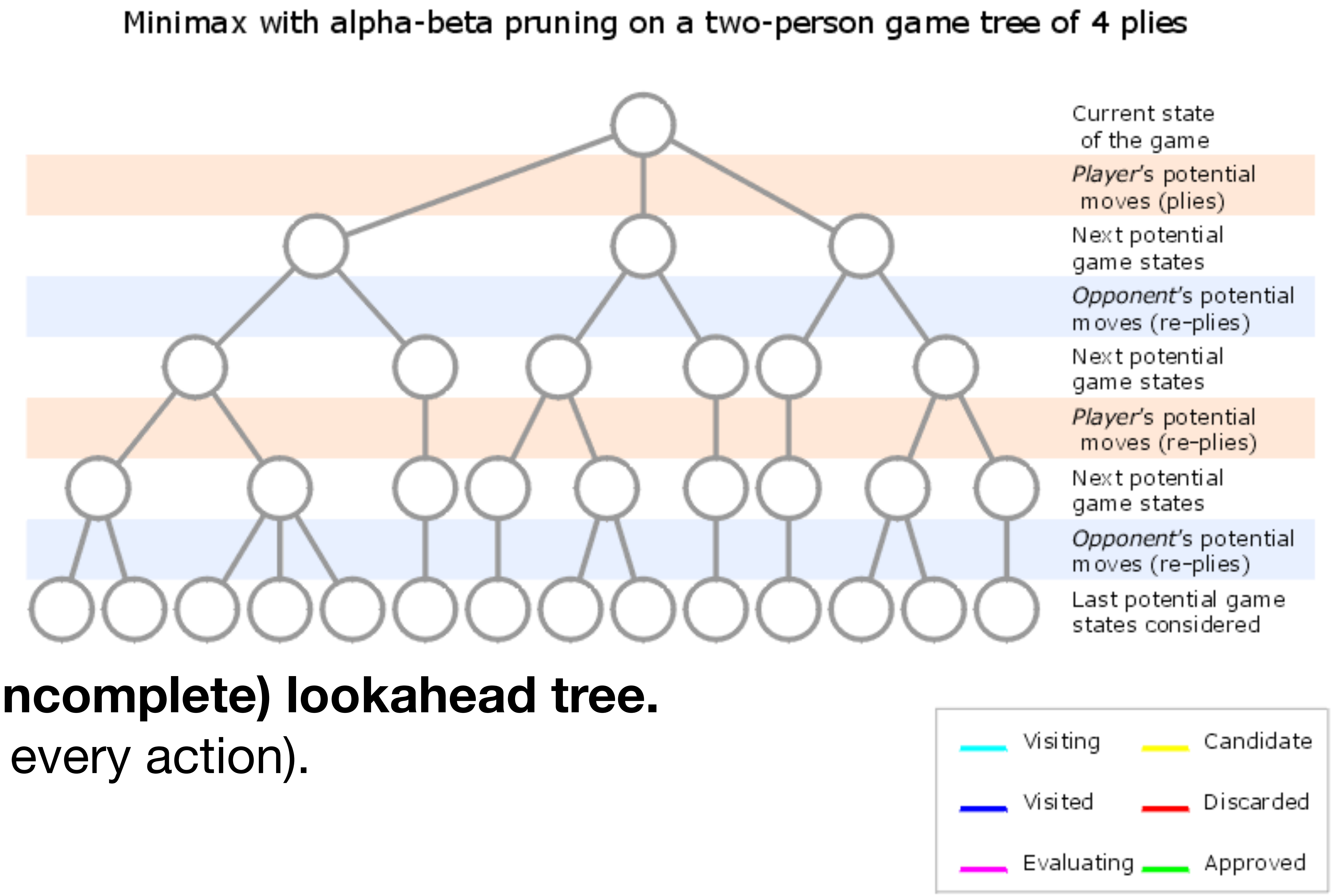
- In principle, one could work out the optimal strategy for any zero-sum game with lookahead.

Figure not fully expanded.

AlphaBeta Search

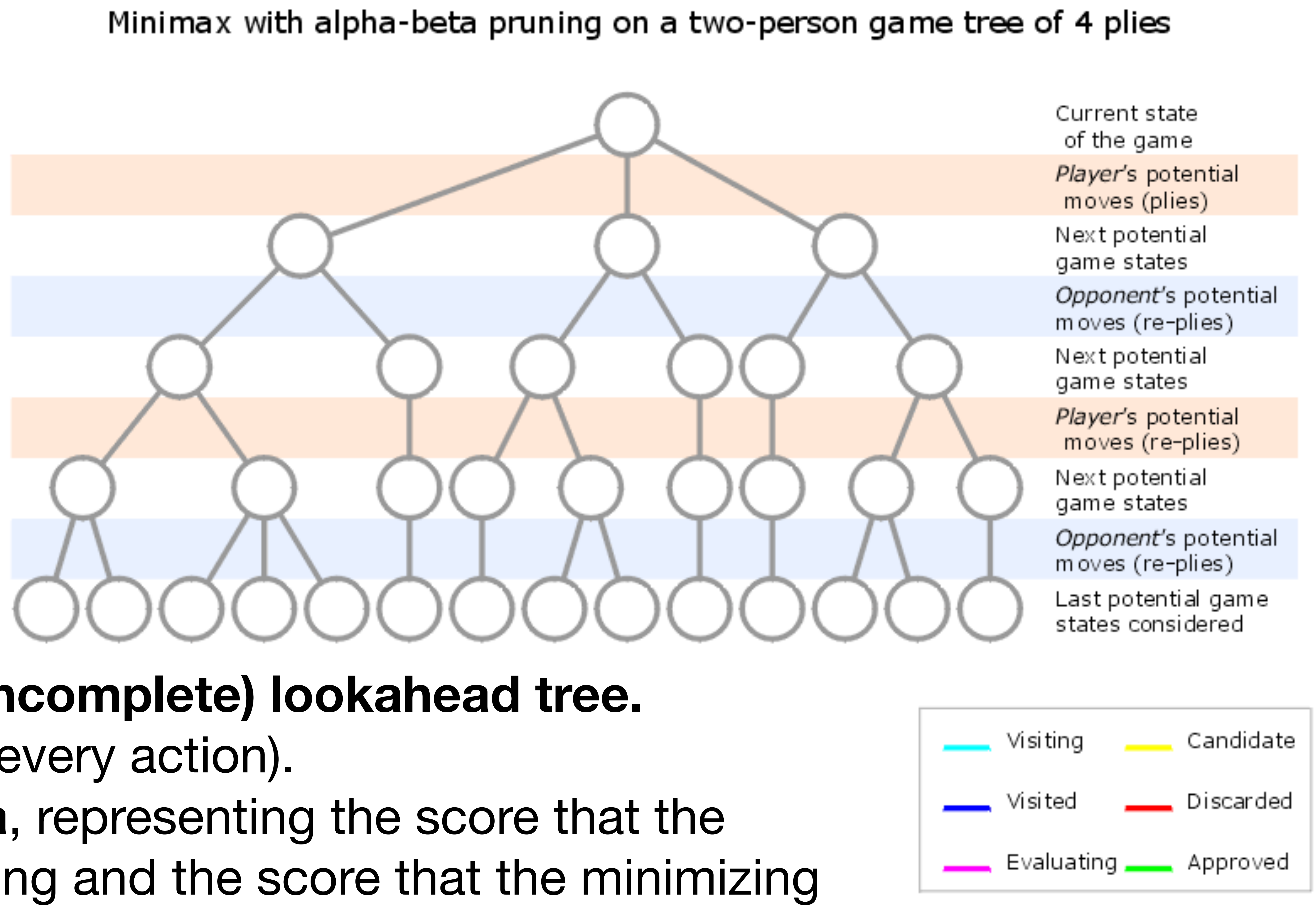


AlphaBeta Search



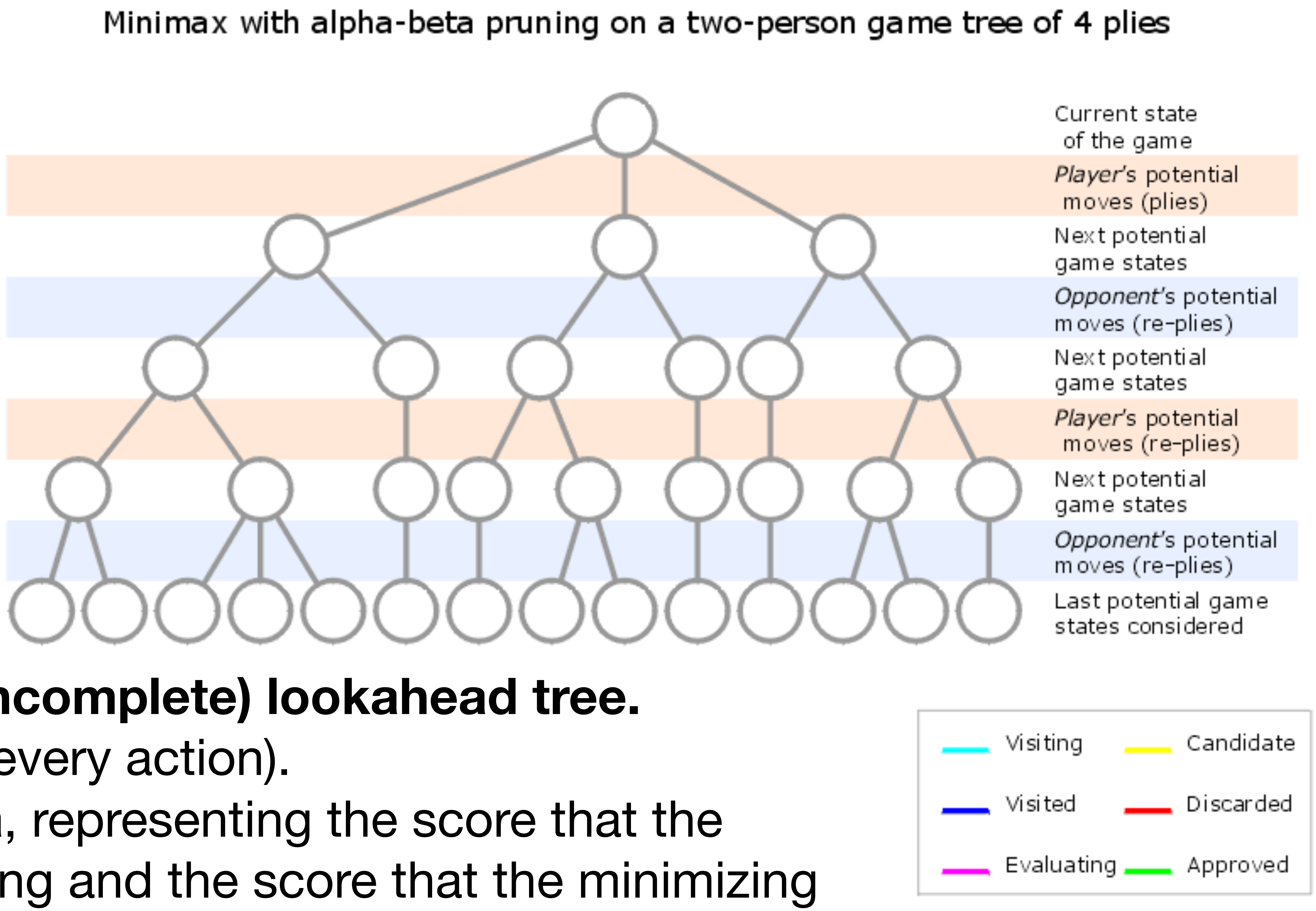
- **To take a single move, we build an (incomplete) lookahead tree.** (a lookahead tree is built before taking every action).

AlphaBeta Search



- **To take a single move, we build an (incomplete) lookahead tree.** (a lookahead tree is built before taking every action).
  - maintain two values, **alpha** and **beta**, representing the score that the maximizing player is assured of getting and the score that the minimizing player is assured of getting.
  - assume opponents will always try to do "best responses"

AlphaBeta Search



- **To take a single move, we build an (incomplete) lookahead tree.** (a lookahead tree is built before taking every action).
  - maintain two values, **alpha** and **beta**, representing the score that the maximizing player is assured of getting and the score that the minimizing player is assured of getting.
  - assume opponents will always try to do “best responses”
  - **Need a heuristic for which branches to search.**
  - **Try to prune away as may branches as we can.**

Stockfish 15.1

Strong open source chess engine

Download Stockfish

Latest from the blog

2022-12-04: Stockfish 15.1

2022-11-18: ChessBase GmbH and the Stockfish team reach an agreement and end their legal dispute

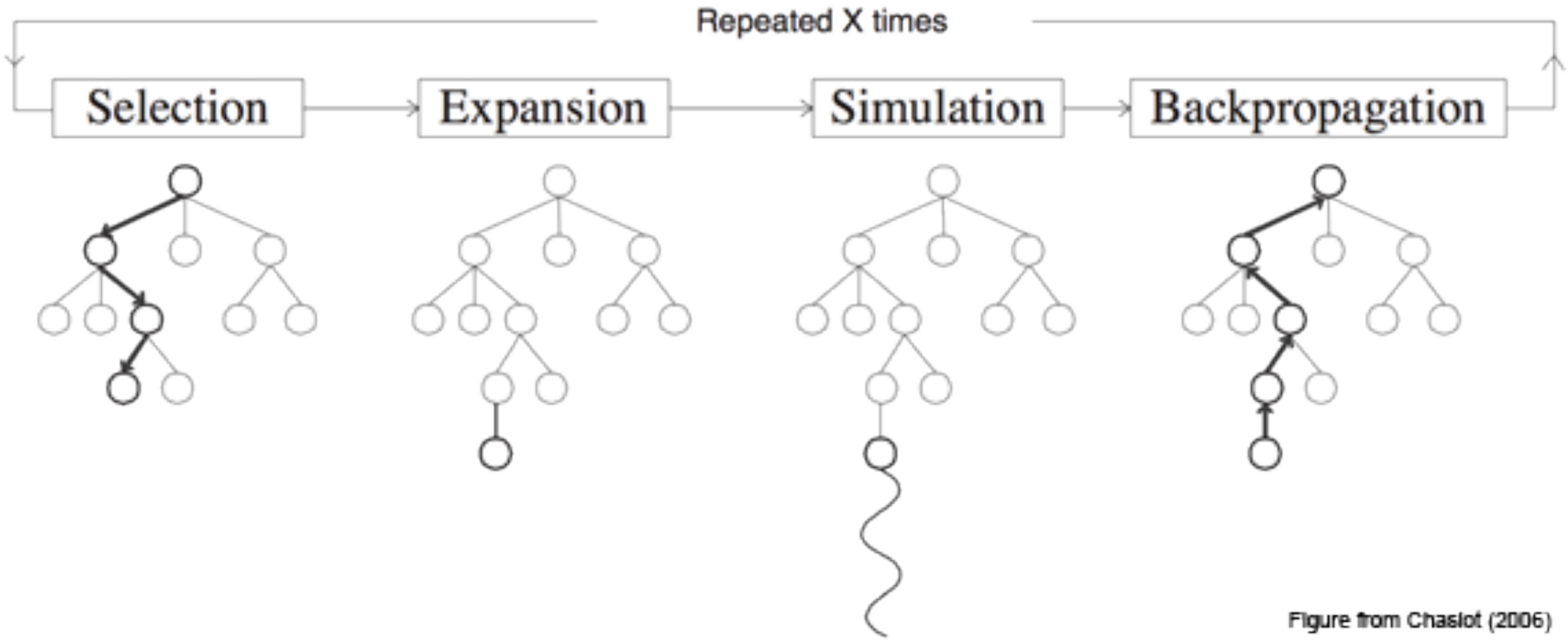
2022-06-22: Public court hearing soon!

It’s a “rule-based” system.

Today

- Recap
- Game Playing: AlphaBeta Search/Rule Based Systems
- MCTS
- AlphaZero and Self-Play

MCTS:
Monte Carlo Tree Search



- AlphaBeta pessimistic approach may not lead to effective heuristics.
- **MCTS: to decide on an action, we build a lookahead tree.** (and repeat)
  Input: game state/node "R"; Output: single action to take at R
  - For two player games
  - When building the lookahead tree, we use a heuristic to estimate the "value" of taking action "a" at any node "s"
    (no minmax values estimated).
- **Applicable to "small" games.**

ActionSelectionSubroutine

ActionSelectionSubroutine

Input: game state (“root node” R), # playouts N

ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N

For rollouts $t = 1 : N$

ActionSelectionSubroutine

rollouts ⟷ trajectories

Input: game state ("root node" R), # playouts N

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {win, lose}

ActionSelectionSubroutine

Input: game state (“root node” R), # playouts N

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {win, lose}
   a. For player $X \in \{A, B\}$, at current state s, define $s' = NextState(s, a)$ and define:

$$\text{UCB score}_t(s, a) = \frac{\text{\#wins for player X at } s'}{\text{\#visits to } s'} + C \times \sqrt{\frac{\log(\text{total visits to } s)}{\text{\#visits to } s'}}$$

ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {win, lose}
   a. For player $X \in \{A, B\}$, at current state s, define $s' = NextState(s, a)$ and define:

$$\text{UCB score}_t(s, a) = \frac{\text{\#wins for player X at } s'}{\text{\#visits to } s'} + C \times \sqrt{\frac{\log(\text{total visits to } s)}{\text{\#visits to } s'}}$$

   b. Choose and "take" action:

$$\hat{a} = \arg\max_a \text{UCB score}(s, a)$$

ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {win, lose}
   a. For player $X \in \{A, B\}$, at current state s, define $s' = NextState(s, a)$ and define:
   $$\text{UCB score}_t(s, a) = \frac{\text{\#wins for player X at } s'}{\text{\#visits to } s'} + C \times \sqrt{\frac{\log(\text{total visits to } s)}{\text{\#visits to } s'}}$$
   b. Choose and "take" action:
   $$\hat{a} = \arg\max_a \text{UCB score}(s, a)$$
2. Update stats: For all visited states s in this "roll-out",

ActionSelectionSubroutine

Input: game state (“root node” R), # playouts N

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode ∉ {win, lose}
   a. For player $X \in \{A, B\}$, at current state s, define $s' = NextState(s, a)$ and define:

   $$\text{UCB score}_t(s, a) = \frac{\text{\#wins for player X at } s'}{\text{\#visits to } s'} + C \times \sqrt{\frac{\log(\text{total visits to } s)}{\text{\#visits to } s'}}$$

   b. Choose and “take” action:

   $$\hat{a} = \arg\max_a \text{UCB score}(s, a)$$

2. Update stats: For all visited states s in this “roll-out”,
   c. update visit counts:

   $$[\text{\#visits to } s'] = [\text{\#visits to } s'] + 1$$

ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode ∉ {win, lose}
   a. For player $X \in \{A, B\}$, at current state s, define $s' = NextState(s, a)$ and define:
      $$\text{UCB score}_t(s, a) = \frac{\text{\#wins for player X at } s'}{\text{\#visits to } s'} + C \times \sqrt{\frac{\log(\text{total visits to } s)}{\text{\#visits to } s'}}$$
   b. Choose and "take" action:
      $$\hat{a} = \arg\max_a \text{UCB score}(s, a)$$
2. Update stats: For all visited states s in this "roll-out",
   c. update visit counts:
      $$[\text{\#visits to } s'] = [\text{\#visits to } s'] + 1$$
   d. for winner X and if s was visited by X:
      $$[\text{\#wins for X at } s'] = [\text{\#wins for X at } s'] + 1$$

ActionSelectionSubroutine

Input: game state (“root node” R), # playouts N

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode ∉ {win, lose}
   a. For player $X \in \{A, B\}$, at current state s, define $s' = NextState(s, a)$ and define:
   $$\text{UCB score}_t(s, a) = \frac{\text{\#wins for player X at } s'}{\text{\#visits to } s'} + C \times \sqrt{\frac{\log(\text{total visits to } s)}{\text{\#visits to } s'}}$$
   b. Choose and “take” action:
   $$\hat{a} = \arg\max_a \text{UCB score}(s, a)$$
2. Update stats: For all visited states s in this “roll-out”,
   c. update visit counts:
   $$[\text{\#visits to } s'] = [\text{\#visits to } s'] + 1$$
   d. for winner X and if s was visited by X:
   $$[\text{\#wins for X at } s'] = [\text{\#wins for X at } s'] + 1$$

(data structure: only need to keep track of stats at visited states)

ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {win, lose}
   a. For player $X \in \{A, B\}$, at current state s, define $s' = NextState(s, a)$ and define:
   $$\text{UCB score}_t(s, a) = \frac{\text{\#wins for player X at } s'}{\text{\#visits to } s'} + C \times \sqrt{\frac{\log(\text{total visits to } s)}{\text{\#visits to } s'}}$$
   b. Choose and "take" action:
   $$\hat{a} = \arg\max_a \text{UCB score}(s, a)$$
2. Update stats: For all visited states s in this "roll-out",
   c. update visit counts:
   $$[\text{\#visits to } s'] = [\text{\#visits to } s'] + 1$$
   d. for winner X and if s was visited by X:
   $$[\text{\#wins for X at } s'] = [\text{\#wins for X at } s'] + 1$$

(data structure: only need to keep track of stats at visited states)

Output: return the action $\hat{a} = \arg\max_a \text{UCB score}_N(\text{Root Node } R, a)$

Example Diagram:

"rollout"

Example Diagram:

- **Obtaining the t-th rollout** (steps called **Selection/Expansion/Simulation):** Start from "root R" and select successive child nodes until a the game ends.

Example Diagram:

- **Obtaining the t-th rollout** (steps called **Selection/Expansion/Simulation):**
  Start from "root R" and select successive child nodes until a the game ends.
  - At state s (for player X), choose action a leading to $s' = NextState(s, a)$ which maximizes:

Example Diagram:

- **Obtaining the t-th rollout** (steps called **Selection/Expansion/Simulation):**
  Start from "root R" and select successive child nodes until a the game ends.
  - At state s (for player X), choose action a leading to $s' = NextState(s, a)$ which maximizes:

$$\text{UCB score}_t(s,a) = \frac{\text{\#wins for player X at } s'}{\text{\#visits to } s'} + C \times \sqrt{\frac{\log(\text{total visits to } s)}{\text{\#visits to } s'}}$$

Example Diagram:

Example Diagram:

$s_1, s_2, s_3, \ldots s_{50}$ | $r=0$ (we lost).

- **The update step for the t-th rollout (“backpropagation”):**
  Use the result of the rollout to update information in the nodes on the visited path:
  [#visits to s'] = [#visits to s'] + 1
  [#wins for X at s'] = [#wins for X at s'] + 1

Example Diagram:

Example Diagram:

- **Repeat all steps N times**, (so we do N roll-outs)
- **select the "best" action at the root node R** (the game state):

$$\hat{a} = \arg\max_{a} \text{UCB score}_N(\text{Root Node } R, a)$$

MCTS also“works” with a simulator for (single-agent) RL

MCTS also“works” with a simulator for (single-agent) RL

- The basic idea of “roll-outs/lookahead” is common, if we have a simulator:
  e.g. MPC (Model Predictive Control)/iLQR

MCTS also“works” with a simulator for (single-agent) RL

- The basic idea of “roll-outs/lookahead” is common, if we have a simulator:
  e.g. MPC (Model Predictive Control)/iLQR
- MCTS also applicable to RL, but:

MCTS also“works” with a simulator for (single-agent) RL

- The basic idea of “roll-outs/lookahead” is common, if we have a simulator:
  e.g. MPC (Model Predictive Control)/iLQR
- MCTS also applicable to RL, but:
  - need the number of states in the lookahead tree to be “small”
    (e.g. doesn’t work if we tend not to visit the same state again)

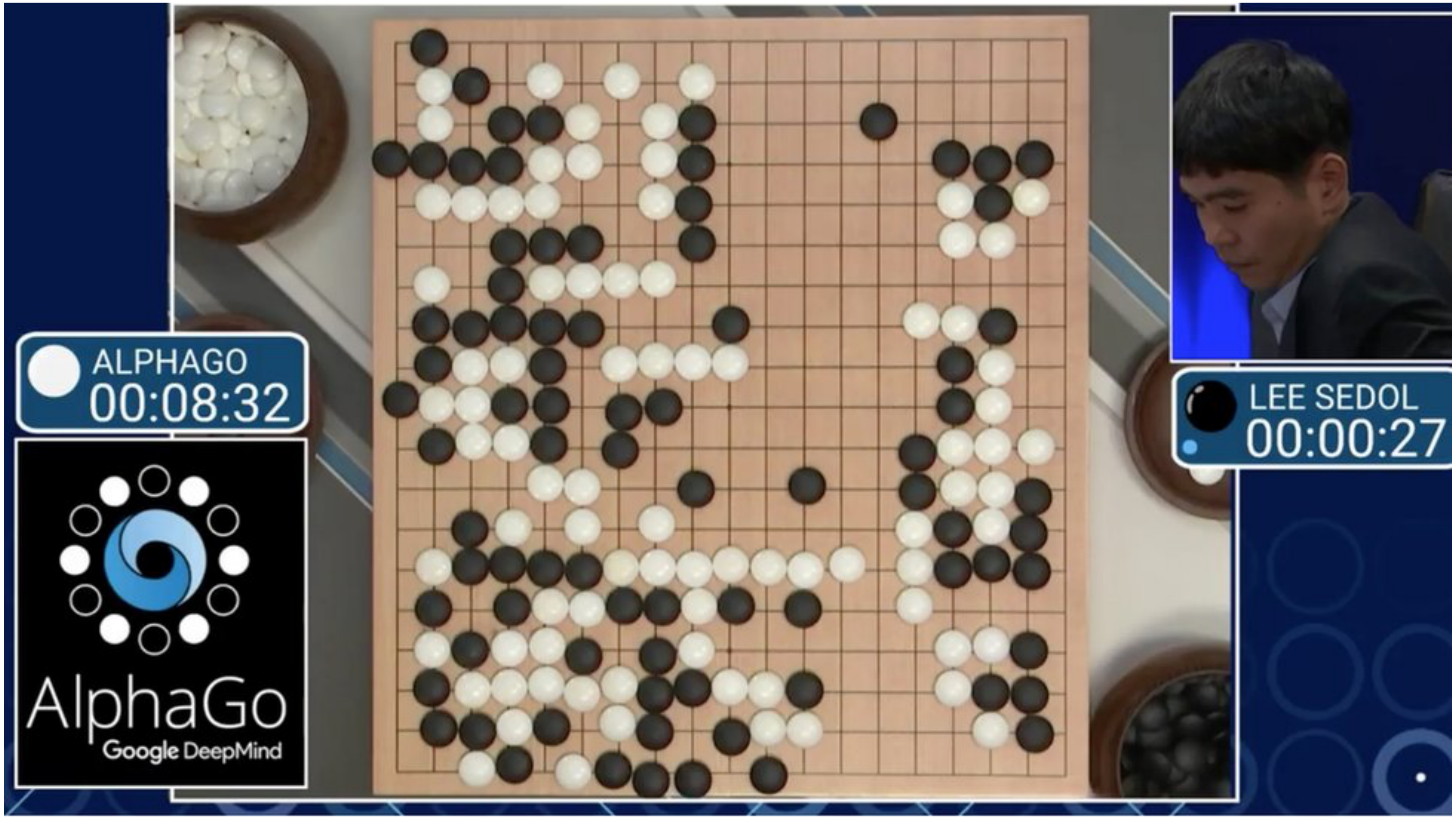
Today

- Recap
- Game Playing: AlphaBeta Search/Rule Based Systems
- MCTS
- ✓ AlphaZero and Self-Play

AlphaGo

AlphaGo versus Lee Sedol
4–1

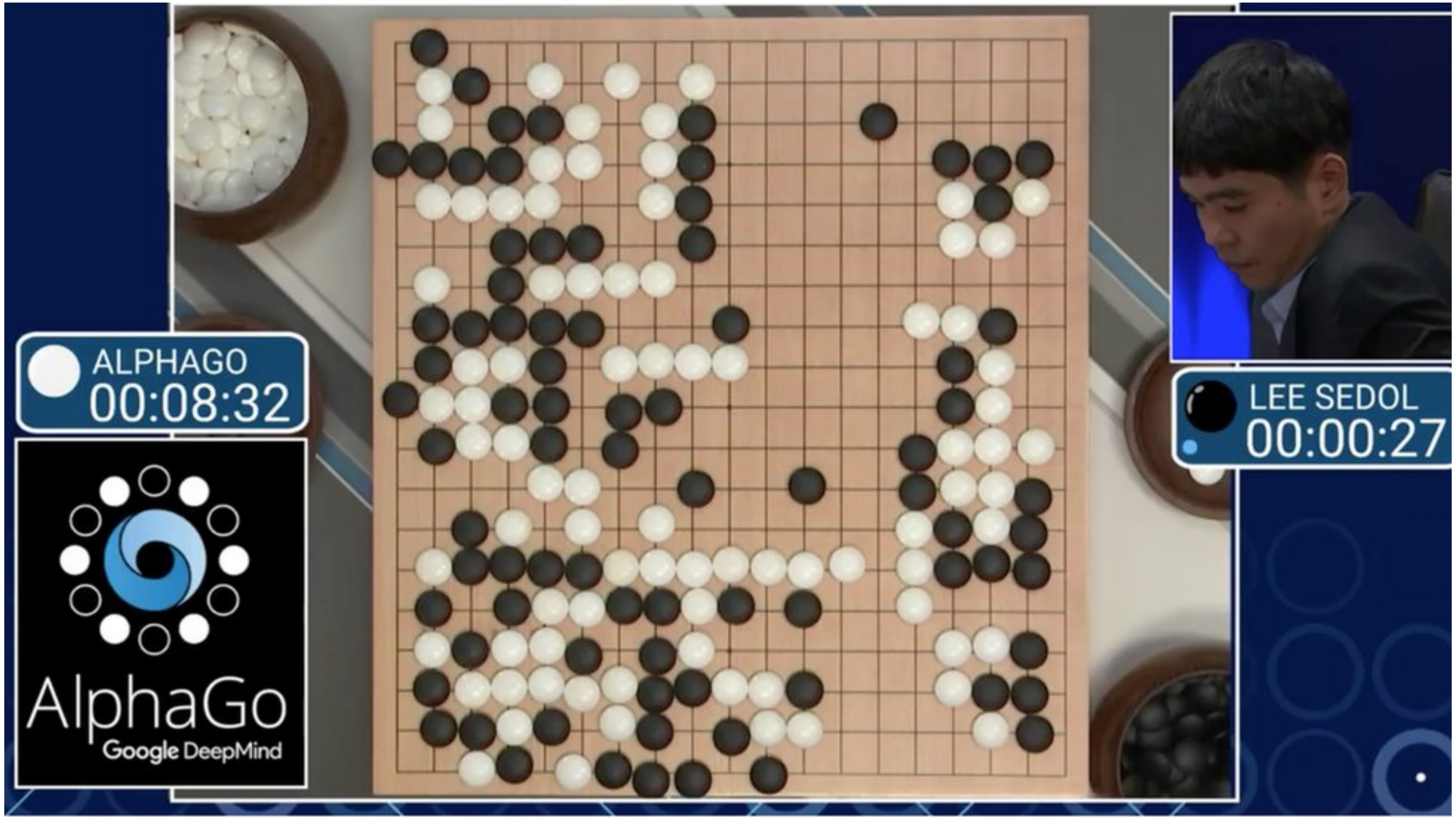
Seoul, South Korea, 9–15 March 2016	
Game one	AlphaGo W+R
Game two	AlphaGo B+R
Game three	AlphaGo W+R
Game four	Lee Sedol W+R
Game five	AlphaGo W+R



AlphaGo

AlphaGo versus Lee Sedol
4–1

Seoul, South Korea, 9–15 March 2016	
Game one	AlphaGo W+R
Game two	AlphaGo B+R
Game three	AlphaGo W+R
Game four	Lee Sedol W+R
Game five	AlphaGo W+R



- Lots of moving parts:
  - **Imitation Learning:** first, the algo estimates the values from historical games.
  - It then uses an **MCTS-stye lookahead** with **learned value functions**.
- **AlphaZero** (2017) is a simpler more successful approach.

AlphaZero

AlphaZero

- AlphaZero: MCTS + DeepLearning

AlphaZero

- AlphaZero: MCTS + DeepLearning
  - There is a value network and policy network:
    - a **value network** estimating for the state of the board $v_\theta(s)$
    - A **policy network** $p_\theta(a \mid s)$ that is a probability vector over all possible actions. (think $p_\theta(a \mid s)$ of as an estimate of which actions the “subroutine” selects)

AlphaZero

- AlphaZero: MCTS + DeepLearning
  - There is a value network and policy network:
    - a **value network** estimating for the state of the board $v_\theta(s)$
    - A **policy network** $p_\theta(a \mid s)$ that is a probability vector over all possible actions.
      (think $p_\theta(a \mid s)$ of as an estimate of which actions the "subroutine" selects)
  - There is a **termination condition** for each rollout,
    e.g. each rollout is no longer than K steps

AlphaZero: ActionSelectionSubroutine

AlphaZero: ActionSelectionSubroutine

Input: game state (“root node” R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

AlphaZero: ActionSelectionSubroutine

Input: game state (“root node” R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

For rollouts $t = 1 : N$

AlphaZero: ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {termination condition}

AlphaZero: ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {termination condition}
   a. At current state s, define $s' = NextState(s, a)$ and define:

$$\text{UCB score}_t(s, a) = \text{AvValue}(s') + C \cdot p_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{total visits to s})}{\text{\#visits to } s'}}$$

AlphaZero: ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {termination condition}
   a. At current state s, define $s' = NextState(s, a)$ and define:

   $$\text{UCB score}_t(s, a) = \text{AvValue}(s') + C \cdot p_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{total visits to s})}{\text{\#visits to } s'}}$$

   b. Choose and "take" action:

   $$\hat{a} = \arg\max_a \text{UCB score}_t(s, a)$$

AlphaZero: ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {termination condition}
   a. At current state s, define $s' = NextState(s, a)$ and define:
   $$\text{UCB score}_t(s, a) = \text{AvValue}(s') + C \cdot p_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{total visits to s})}{\text{\#visits to } s'}}$$
   b. Choose and "take" action:
   $$\hat{a} = \arg\max_a \text{UCB score}_t(s, a)$$
2. Update stats: For all visited states s in this "roll-out",

AlphaZero: ActionSelectionSubroutine

Input: game state (“root node” R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {termination condition}
    a. At current state s, define $s' = NextState(s, a)$ and define:

    $$\text{UCB score}_t(s, a) = \text{AvValue}(s') + C \cdot p_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{total visits to s})}{\text{\#visits to } s'}}$$

    b. Choose and “take” action:

    $$\hat{a} = \arg\max_a \text{UCB score}_t(s, a)$$

2. Update stats: For all visited states s in this “roll-out”,
    c. Let C be the terminal node in this rollout.

AlphaZero: ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {termination condition}
   a. At current state s, define $s' = NextState(s, a)$ and define:

      $$\text{UCB score}_t(s, a) = \text{AvValue}(s') + C \cdot p_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{total visits to s})}{\text{\#visits to } s'}}$$

   b. Choose and "take" action:

      $$\hat{a} = \arg\max_a \text{UCB score}_t(s, a)$$

2. Update stats: For all visited states s in this "roll-out",
   c. Let C be the terminal node in this rollout.
   d. Update counts: $N(s) \leftarrow N(s) + 1$

AlphaZero: ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {termination condition}
   a. At current state s, define $s' = NextState(s, a)$ and define:
   $$\text{UCB score}_t(s, a) = \text{AvValue}(s') + C \cdot p_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{total visits to s})}{\text{\#visits to } s'}}$$
   b. Choose and "take" action:
   $$\hat{a} = \arg\max_a \text{UCB score}_t(s, a)$$
2. Update stats: For all visited states s in this "roll-out",
   c. Let C be the terminal node in this rollout.
   d. Update counts: $N(s) \leftarrow N(s) + 1$
   e. If state s was for player A: $AvValue(s) \leftarrow \frac{N(s)}{N(s)+1} AvValue(s) + \frac{1}{N(s)+1} v_\theta(C)$

AlphaZero: ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {termination condition}
   a. At current state s, define $s' = NextState(s, a)$ and define:

   $$\text{UCB score}_t(s, a) = \text{AvValue}(s') + C \cdot p_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{total visits to s})}{\text{\#visits to } s'}}$$

   b. Choose and "take" action:

   $$\hat{a} = \arg\max_a \text{UCB score}_t(s, a)$$

2. Update stats: For all visited states s in this "roll-out",
   c. Let C be the terminal node in this rollout.
   d. Update counts: $N(s) \leftarrow N(s) + 1$
   e. If state s was for player A: $AvValue(s) \leftarrow \dfrac{N(s)}{N(s)+1} AvValue(s) + \dfrac{1}{N(s)+1} v_\theta(C)$
   f. If state s was for player B: same update but with $-v_\theta(C)$

AlphaZero: ActionSelectionSubroutine

Input: game state ("root node" R), # playouts N, **value network** $v_\theta(s)$, **policy network** $p_\theta(a \mid s)$

For rollouts $t = 1 : N$

1. Obtain the t-th roll-out: While CurrentNode $\notin$ {termination condition}
   a. At current state s, define $s' = NextState(s, a)$ and define:

      $$\text{UCB score}_t(s, a) = \text{AvValue}(s') + C \cdot p_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{total visits to s})}{\text{\#visits to } s'}}$$

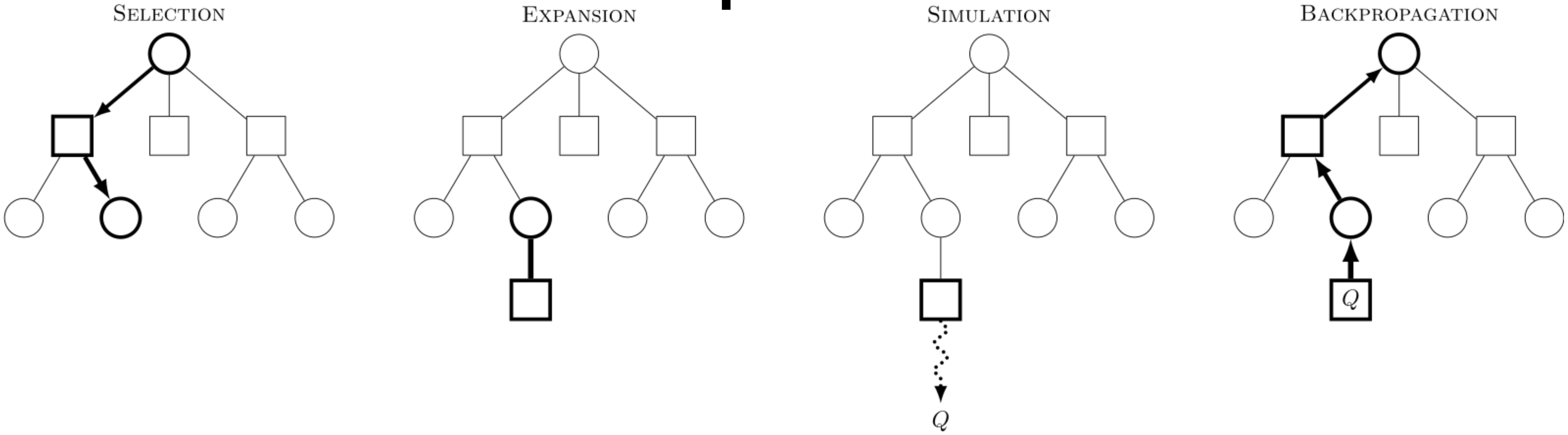
   b. Choose and "take" action:

      $$\hat{a} = \arg\max_a \text{UCB score}_t(s, a)$$

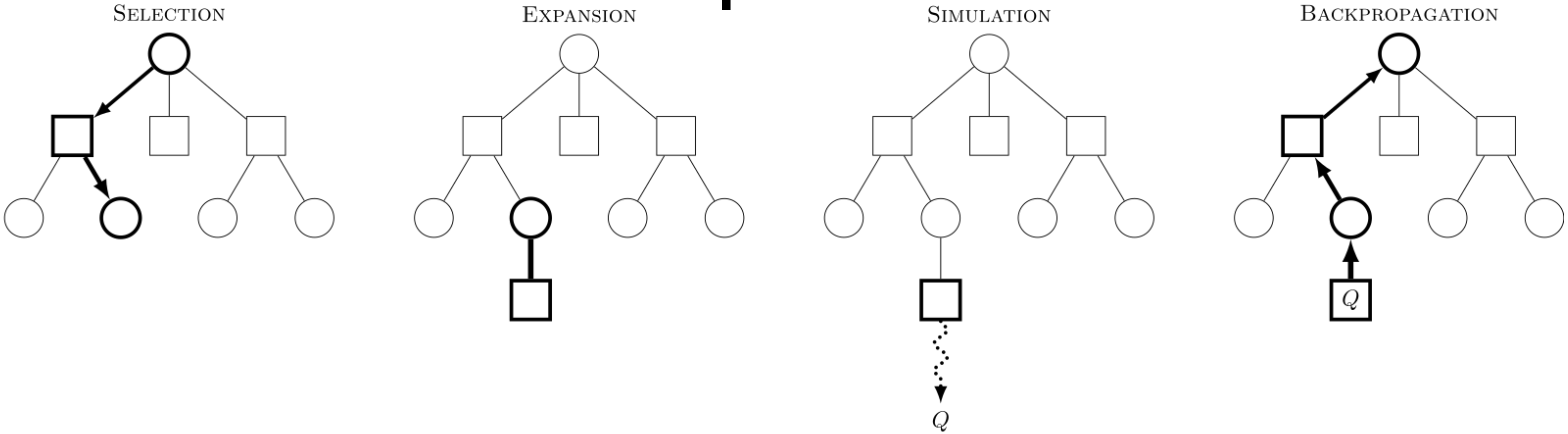
2. Update stats: For all visited states s in this "roll-out",
   c. Let C be the terminal node in this rollout.
   d. Update counts: $N(s) \leftarrow N(s) + 1$
   e. If state s was for player A: $AvValue(s) \leftarrow \frac{N(s)}{N(s)+1} AvValue(s) + \frac{1}{N(s)+1} v_\theta(C)$
   f. If state s was for player B: same update but with $-v_\theta(C)$

Output: return the action $\hat{a} = \arg\max_a \text{UCB score}_N(\text{Root Node } R, a)$

AlphaZero

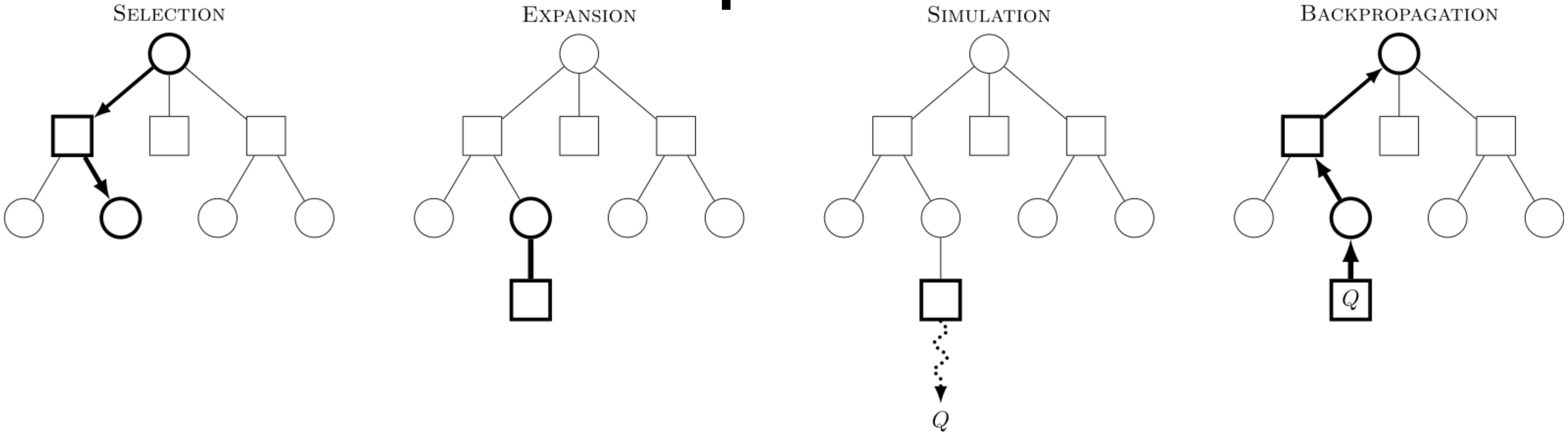


AlphaZero



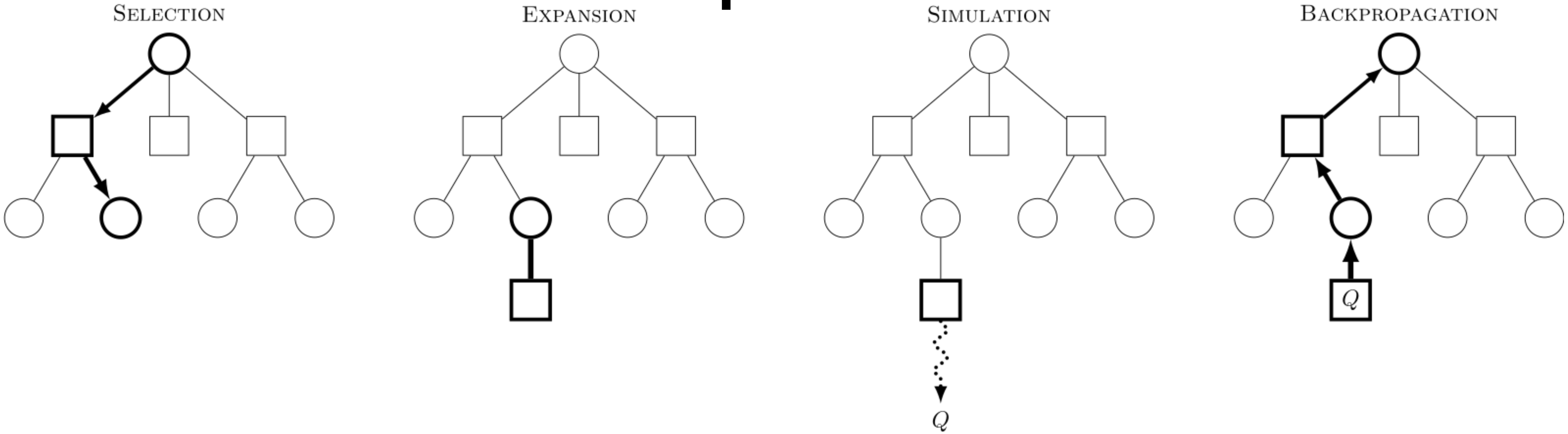
- **Obtaining the t-th rollout** (steps called **Selection/Expansion/Simulation):**

AlphaZero



- **Obtaining the t-th rollout** (steps called **Selection/Expansion/Simulation):**
  - Start from “root R” (current game) and do a rollout of no more than K steps.
  - At state s, choose action a leading to $s' = NextState(s, a)$ which maximizes:

AlphaZero

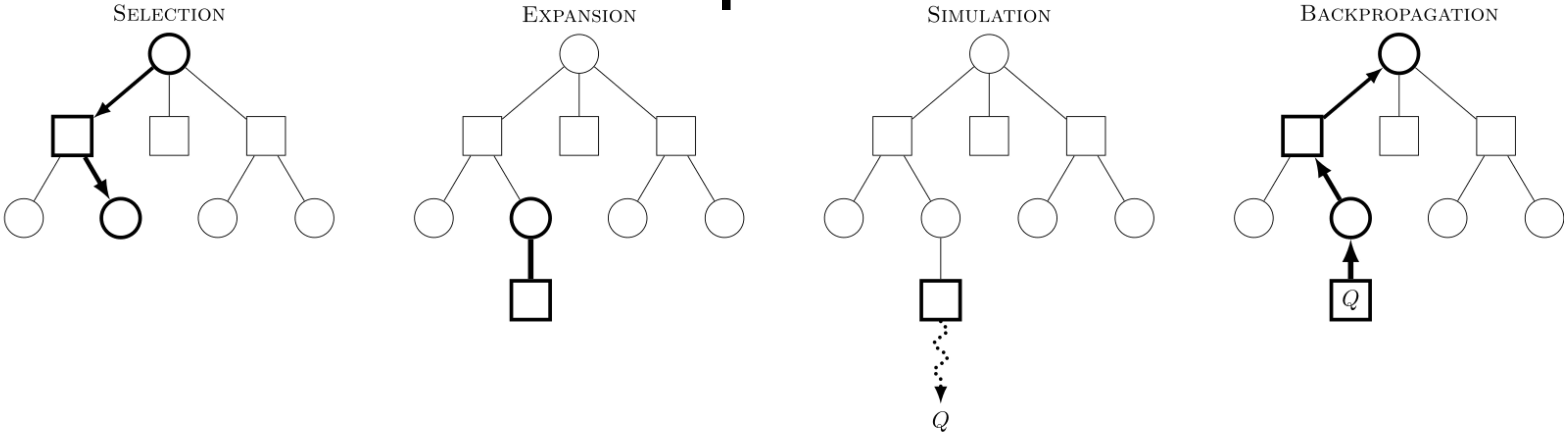


- **Obtaining the t-th rollout** (steps called **Selection/Expansion/Simulation):**
  - Start from “root R” (current game) and do a rollout of no more than K steps.
  - At state s, choose action a leading to $s' = NextState(s, a)$ which maximizes:

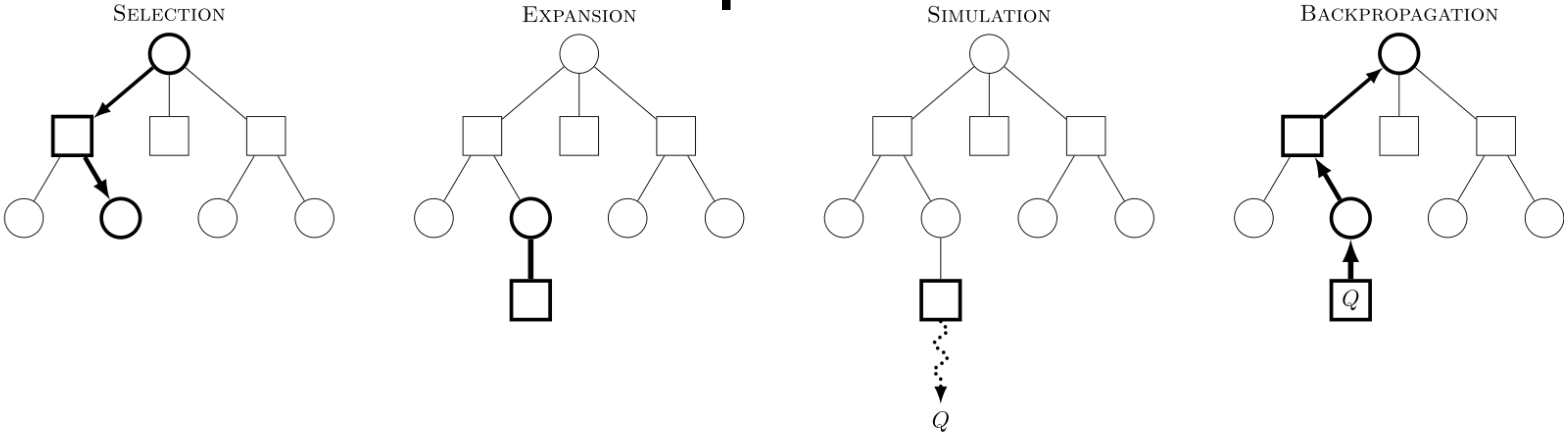
$$\text{UCB score}(a) = \text{AvValue}(s') + C \cdot p_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{total visits to s})}{\text{\#visits to s'}}}$$

  - We’ll specify AverageValue(s') soon.
    - in MCTS, this average was $\frac{\text{\#wins at s'}}{\text{\#visits to s'}}$

AlphaZero

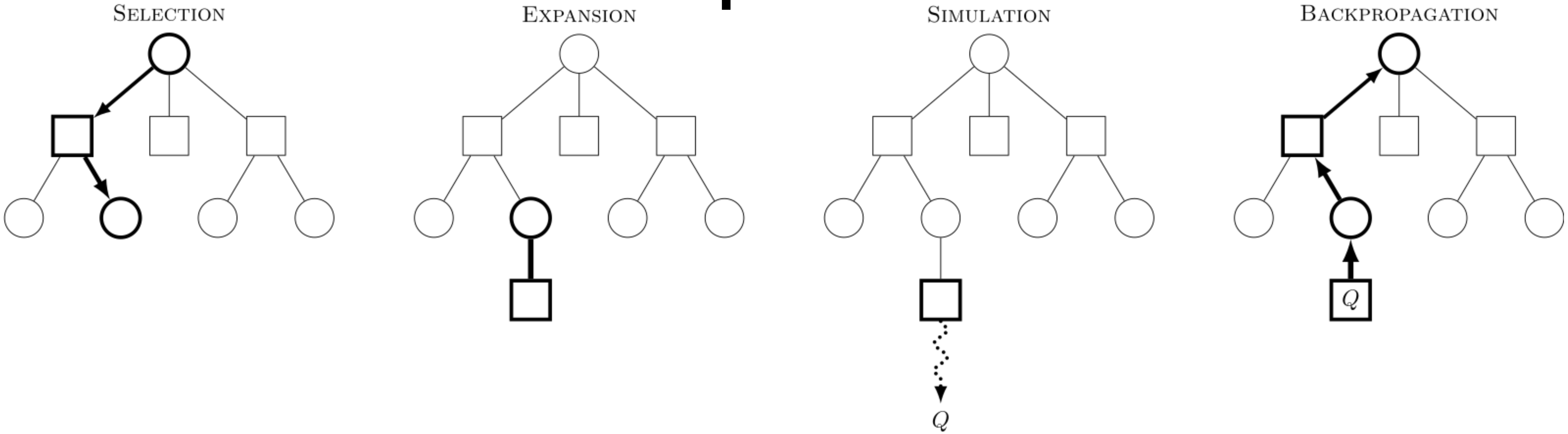


AlphaZero



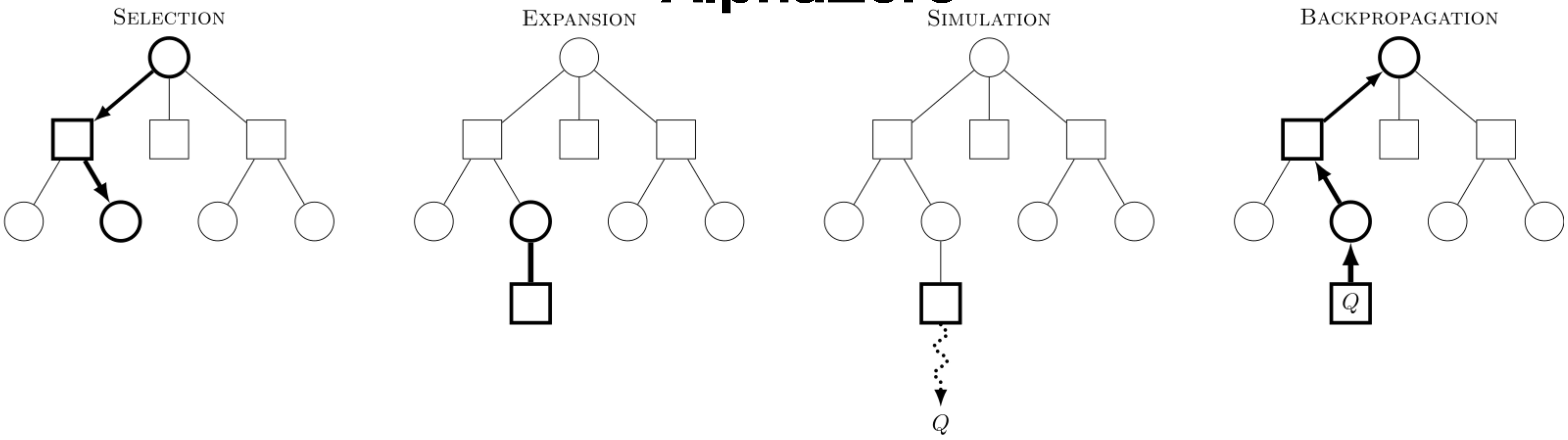
- **The update step for the t-th rollout (“backpropagation”):**

AlphaZero



- **The update step for the t-th rollout ("backpropagation"):**
  - Suppose the Simulation ends at node C after K steps.

AlphaZero



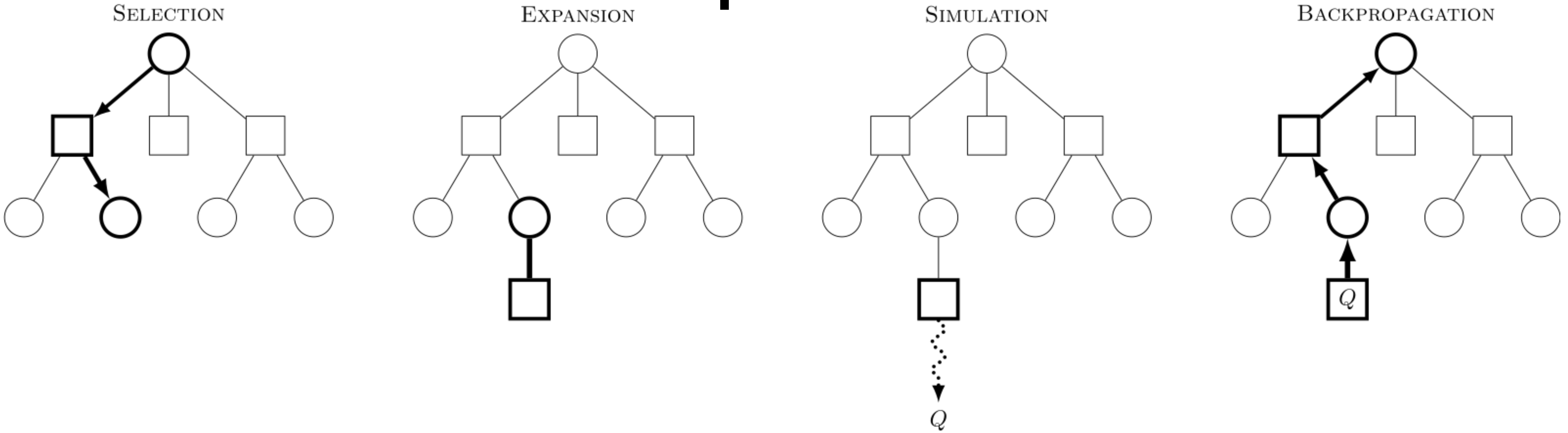
- **The update step for the t-th rollout ("backpropagation"):**
  - Suppose the Simulation ends at node C after K steps.
  - Update AvValue(s) on all s in the path from the root R to C (for player A):

$$AvValue(s) \leftarrow \frac{N(s)}{N(s)+1} AvValue(s) + \frac{1)}{N(s)+1} v_\theta(C)$$

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

(use negative values for player B)

AlphaZero



- **The update step for the t-th rollout ("backpropagation"):**
  - Suppose the Simulation ends at node C after K steps.
  - Update AvValue(s) on all s in the path from the root R to C (for player A):

$$AvValue(s) \leftarrow \frac{N(s)}{N(s)+1} AvValue(s) + \frac{1)}{N(s)+1} v_\theta(C)$$

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

(use negative values for player B)

- **Repeat all steps N times**, then **select "best" action at the root node R** (the game state).

AlphaZero: Learning

AlphaZero: Learning

- Input: dataset of M self-play games

AlphaZero: Learning

- Input: dataset of M self-play games
  - The point in the dataset is of the (s_t, a_t, R_t), which says action a_t was taken in state s_t and the game resulted in outcome R_t (e.g. win=1,loose=-1, draw=0)

AlphaZero: Learning

- Input: dataset of M self-play games
  - The point in the dataset is of the (s_t, a_t, R_t), which says action a_t was taken in state s_t and the game resulted in outcome R_t (e.g. win=1,loose=-1, draw=0)
- Supervised Learning: try learn θ so to predict the actions and rewards

$$Loss(\theta) = \sum_t (v_\theta(s_t) - R_t)^2 - \log p_\theta(a_t \mid s_t)$$

AlphaZero: Learning

- Input: dataset of M self-play games
  - The point in the dataset is of the (s_t, a_t, R_t), which says action a_t was taken in state s_t and the game resulted in outcome R_t (e.g. win=1,loose=-1, draw=0)
- Supervised Learning: try learn θ so to predict the actions and rewards

$$Loss(\theta) = \sum_t (v_\theta(s_t) - R_t)^2 - \log p_\theta(a_t \mid s_t)$$

AlphaZero was trained solely via self-play, using 5,000 first-generation TPUs to generate the games and 64 second-generation TPUs to train the neural networks. In parallel, the in-training AlphaZero was periodically matched against its benchmark (Stockfish, elmo, or AlphaGo Zero) in

Comparing Monte Carlo tree search searches, AlphaZero searches just 80,000 positions per second in chess and 40,000 in shogi, compared to 70 million for Stockfish and 35 million for elmo. AlphaZero compensates for the lower number of evaluations by using its deep neural network to

Chess [edit]

In AlphaZero's chess match against Stockfish 8 (2016 TCEC world champion), each program was given one minute per move. Stockfish was allocated 64 threads and a hash size of 1 GB,[1] a setting that Stockfish's Tord Romstad later criticized as suboptimal.[7][note 1] AlphaZero was trained on chess for a total of nine hours before the match. During the match, AlphaZero ran on a single machine with four application-specific TPUs. In 100 games from the normal starting position, AlphaZero won 25 games as White, won 3 as Black, and drew the remaining 72.[8] In a series of twelve, 100-game matches (of unspecified time or resource constraints) against Stockfish starting from the 12 most popular human openings, AlphaZero won 290, drew 886 and lost 24.[1]

Shogi [edit]

AlphaZero was trained on shogi for a total of two hours before the tournament. In 100 shogi games against elmo (World Computer Shogi Championship 27 summer 2017 tournament version with YaneuraOu 4.73 search), AlphaZero won 90 times, lost 8 times and drew twice.[8] As in the chess games, each program got one minute per move, and elmo was given 64 threads and a hash size of 1 GB.[1]

Go [edit]

After 34 hours of self-learning of Go and against AlphaGo Zero, AlphaZero won 60 games and lost 40.[1][8]

Comparing Monte Carlo tree search searches, AlphaZero searches just 80,000 positions per second in chess and 40,000 in shogi, compared to 70 million for Stockfish and 35 million for elmo. AlphaZero compensates for the lower number of evaluations by using its deep neural network to

Chess [edit]

In AlphaZero's chess match against Stockfish 8 (2016 TCEC world champion), each program was given one minute per move. Stockfish was allocated 64 threads and a hash size of 1 GB,[1] a setting that Stockfish's Tord Romstad later criticized as suboptimal.[7][note 1] AlphaZero was trained on chess for a total of nine hours before the match. During the match, AlphaZero ran on a single machine with four application-specific TPUs. In 100 games from the normal starting position, AlphaZero won 25 games as White, won 3 as Black, and drew the remaining 72.[8] In a series of twelve, 100-game matches (of unspecified time or resource constraints) against Stockfish starting from the 12 most popular human openings, AlphaZero won 290, drew 886 and lost 24.[1]

Shogi [edit]

AlphaZero was trained on shogi for a total of two hours before the tournament. In 100 shogi games against elmo (World Computer Shogi Championship 27 summer 2017 tournament version with YaneuraOu 4.73 search), AlphaZero won 90 times, lost 8 times and drew twice.[8] As in the chess games, each program got one minute per move, and elmo was given 64 threads and a hash size of 1 GB.[1]

Go [edit]

After 34 hours of self-learning of Go and against AlphaGo Zero, AlphaZero won 60 games and lost 40.[1][8]

Cup

Year	Time Controls	Result	Ref
2018	30+10	1st	[63]
2019	30+5	2nd[note 1]	[64]
2019	30+5	2nd	[65]
2019	30+5	1st	[66]
2020	30+5	1st	[67]
2020	30+5	3rd	[68]
2020	30+5	1st	[69]
2021	30+5	1st	[70]
2021	30+5	1st	[71]
2022	30+3	1st	[72]
2023	30+3	2nd	[73]

Leela Chess Zero

Original author(s) Gian-Carlo Pascutto, Gary Linscott

Leela Chess Zero (abbreviated as **LCZero**, **lc0**) is a free, open-source, and deep neural network–based chess engine and volunteer computing project. Development has been spearheaded by programmer Gary Linscott, who is also a developer for the Stockfish chess engine. Leela Chess Zero was adapted from the Leela Zero Go engine,[1] which in turn was based on Google's AlphaGo Zero project.[2] One of the purposes of Leela Chess Zero was to verify the methods in the AlphaZero paper as applied to the game of chess.

MuZero

- **MuZero**
  - Basically AlphaZero but we don’t know game rules.
  - We learn the transition function as we play.

Summary:

1. Search is powerful: MCTS
2. Search + learning is better: AlphaZero

self-play

Attendance:
[bit.ly/3RcTC9T]

Feedback:
[bit.ly/3RHtlxy]