TÉCNICO LISBOA

Planning, Learning and Decision Making

Lecture 22 (and last!). Multi-armed bandits

Weighted majority

Weighted majority algorithm:

- Given a set of N "predictors" and $\eta < 1/2$
- Initialize predictor weights to $w_0(n) = 1, n = 1, \ldots, N$
- Make prediction based on the (weighted) majority vote
- Update weights of all wrong predictors as

$$w_{t+1}(n) = w_t(n)(1 - \eta)$$

EWA

Exponentially weighted averager:

- Given a set of N actions and $\eta > 0$
- Initialize weights to $w_0(a) = 1$, $a \in \mathcal{A}$
- Select an action according to the probabilities

$$p_t(a) = \frac{w_t(a)}{\sum_{a' \in \mathcal{A}} w_t(a')}$$

- Update weights of all actions as

$$w_{t+1}(a) = w_t(a) e^{-\eta c_t(a)}$$

Multi-armed bandit

Multi-armed bandit

Multi-armed bandit

- Sequential decision problem
- Game between agent and “nature”
- At each time step t:
  - Agent selects an action
  - Nature selects cost function
  - Agent gets the cost for its action

Multi-armed bandit

- How can we play this game?

Adversarial bandits

Multi-armed bandit

- Sequential decision problem
- Game between agent and “nature”
- At each time step t:
  - Agent selects an action
  - Nature selects cost function
  - Agent gets the cost for its action
- We make no assumptions on how the cost is selected (can be adversarial)

Setting is quite similar to EWA...

Recall EWA…

- Define a “confidence-level” for each action

Action 1	Action 2	Action 3	Action 4
			
Confidence: 1	Confidence: 1	Confidence: 1	Confidence: 1

Recall EWA…

- Select each action "proportionally" to its confidence:

$$p_t(a) = \frac{w_t(a)}{\sum_{a' \in \mathcal{A}} w_t(a')}$$

- When cost is revealed, we update each "confidence" according to the corresponding cost:

$$w_{t+1}(a) = w_t(a) e^{-\eta c_t(a)}$$

Cost

Recall EWA…

- When cost is revealed, we update each “confidence” according to the corresponding cost…

Cost: 0.1 **Cost: 0.3** **Cost: 1** **Cost: 0.7**

Confidence: 0.9 Confidence: 0.7 Confidence: 0.4 Confidence: 0.5

However…

However…

- We do not observe the cost for all actions, so we can only update weight for current action

$$w_{t+1}(a_t) = w_t(a_t)e^{-\eta c_t(a_t)}$$

- Actions experimented more often will have more updates, which will unbalance weights!

We must compensate

EXP3 - Step 1

- Ideally, as in EWA, we would like

$$w_t(a) = e^{-\eta \sum_{t=0}^{T-1} c_t(a)}$$

- However, we have

$$w_t(a) = e^{-\eta \sum_{t=0}^{T-1} c_t(a)\mathbb{I}(a_t=a)}$$

Only updated if $a_t = a$

EXP3

- By construction,

$$\mathbb{P}\left[a_t = a\right] = p_t(a)$$

- So, in expectation,

$$w_t(a) \approx e^{-\eta \sum_{t=0}^{T-1} c_t(a) p_t(a)}$$

- **Change:**
  - Use the update:

$$w_{t+1}(a_t) = w_t(a_t) e^{-\eta \frac{c_t(a_t)}{p_t(a_t)}}$$

Compensate

Summarizing…

- Given a set of N actions and $\eta > 0$
- Initialize weights to $w_0(a) = 1$, $a \in \mathcal{A}$
- Select an action a_t according to the probabilities

$$p_t(a) = \frac{w_t(a)}{\sum_{a' \in \mathcal{A}} w_t(a')}$$

- Update weight of action a_t as

$$w_{t+1}(a_t) = w_t(a_t) e^{-\eta \frac{c_t(a_t)}{p_t(a_t)}}$$

EXP3

- To measure the performance, we compare the total (expected) cost with that of the best action (in hindsight):

$$R_T = \mathbb{E}\left[\sum_{t=0}^{T-1} c_t(a_t)\right] - \min_{a \in \mathcal{A}} \sum_{t=0}^{T-1} c_t(a)$$

EXP3

- How much regret?
  - The analysis is similar to the WM and EWA, but involves trickier manipulations
  - The final bound on the regret is

$$R_T \leq \sqrt{2TN \log N}$$

EXP3

- This algorithm for adversarial multi-armed bandits is called **“EXPonentially weighted algorithm for EXPloration and EXPloitation”**, or EXP3
- It makes no assumptions on the process by which costs are selected (can be adversarial)
- Depends sublinearly on the number of actions
- Its regret is **sublinear in *T***

No-regret prediction

- We have that

$$R_T \leq \sqrt{2TN \log N}$$

Grows with

$\sqrt{T}$

(slower than T)

Stochastic bandits

Stochastic bandits

- We have available a set of actions
- The cost for each action comes from a **distribution** (fixed through time)

Action 1

Action 2

Action 3

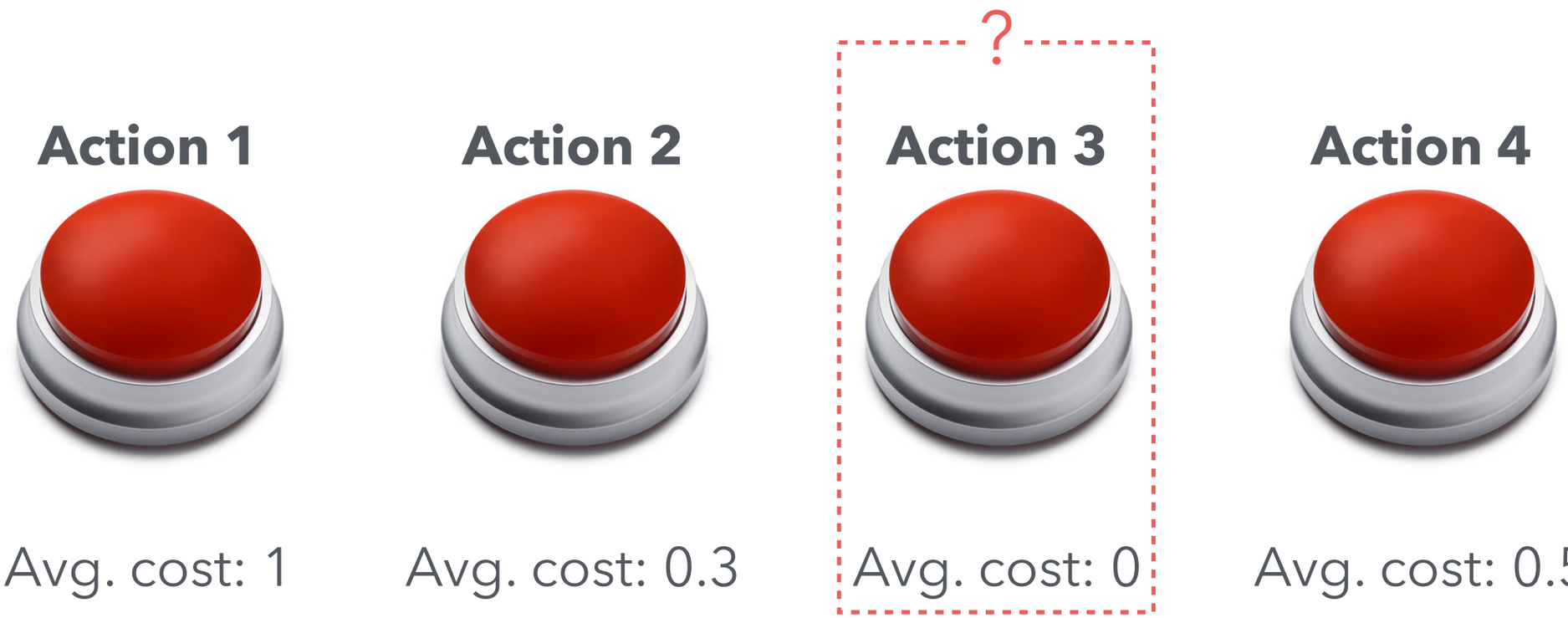
Action 4

Stochastic bandits

- What is the goal?
  - Select action with smallest average cost
- How can we compute the cost?
  - Get samples; average

Stochastic bandits

- After 20 steps:



Which action would you chose?

Why?

Situation 1:

- After 20 steps:

Action 1	Action 2	Action 3	Action 4
			
Avg. cost: 1	Avg. cost: 0.3	Avg. cost: 0	Avg. cost: 0.5
↓	↓	↓	↓
Played 9 times	Played 1 time	Played 9 times	Played 1 time

Situation 1:

- After 20 steps:

Action 1

Avg. cost: 1

Action 2

Avg. cost: 0.3

Action 3

Avg. cost: 0

Action 4

Avg. cost: 0.5

Situation 2:

- After 20 steps:

Action 1	Action 2	Action 3	Action 4
			
Avg. cost: 1	Avg. cost: 0.3	Avg. cost: 0	Avg. cost: 0.5
Played 1 time	Played 9 times	Played 1 time	Played 9 times

Situation 1:

- After 20 steps:

?

Action 1

Avg. cost: 1

Action 2

Avg. cost: 0.3

Action 3

Avg. cost: 0

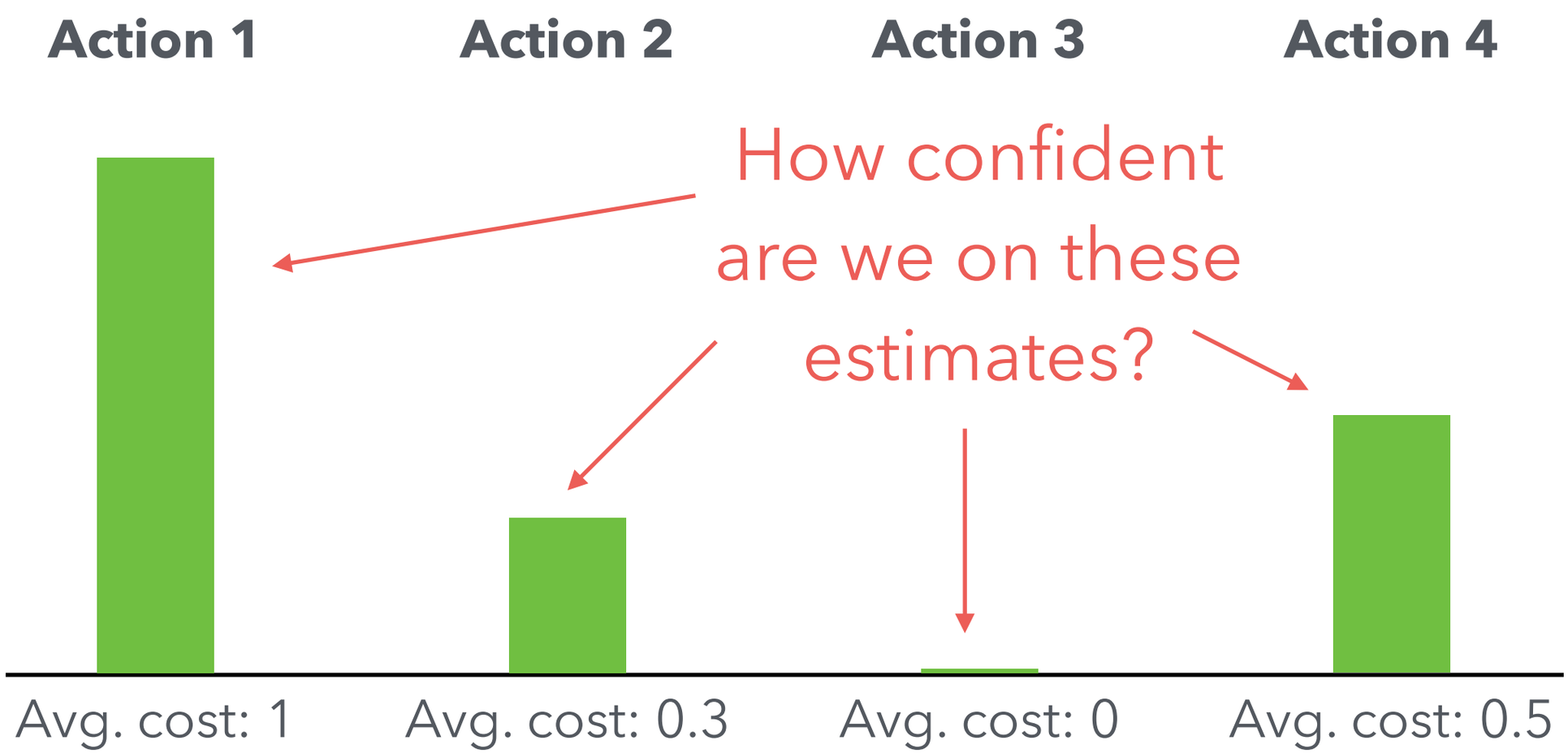
Action 4

Avg. cost: 0.5

Just the mean is not enough!

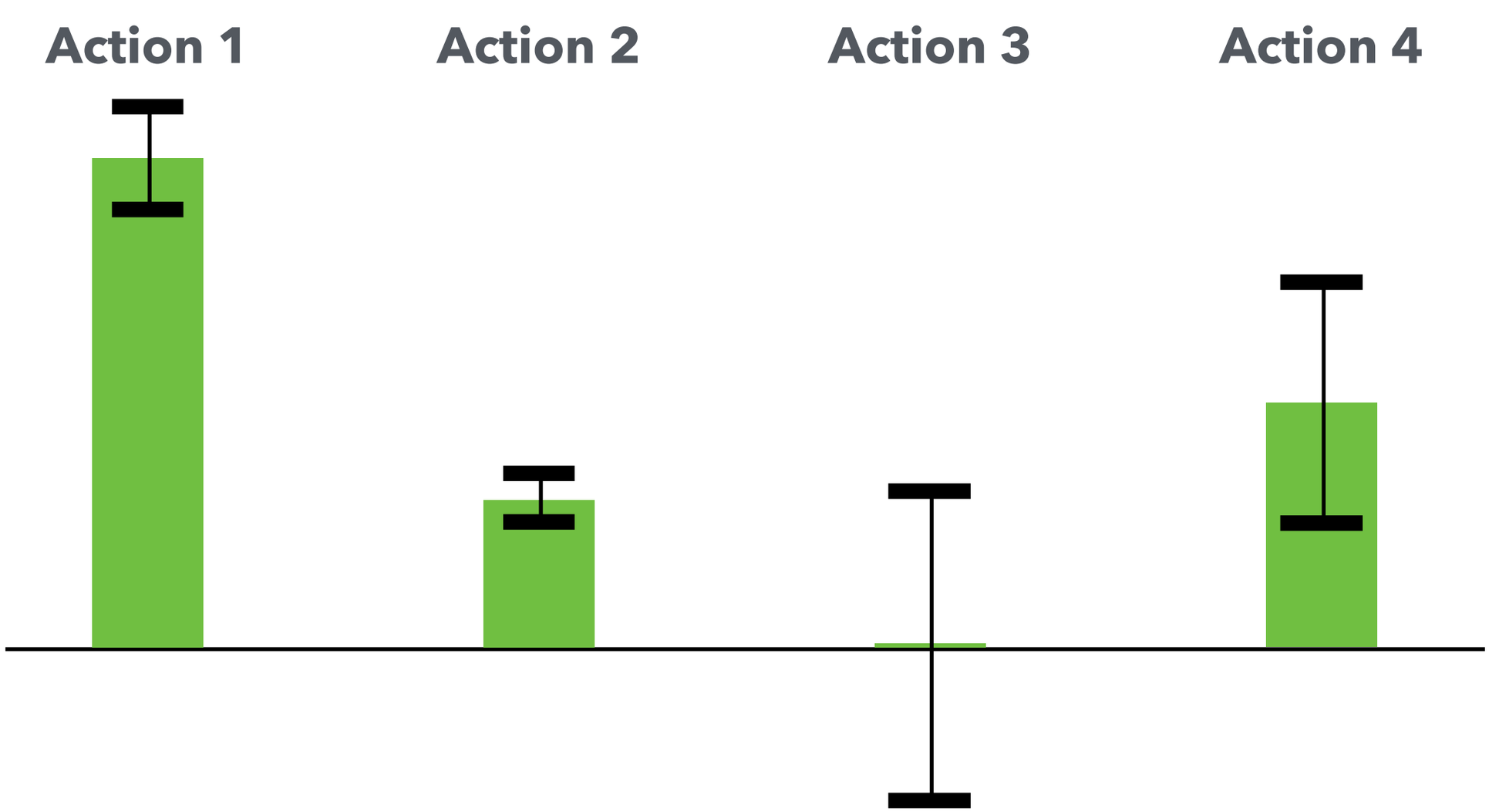
A different perspective

- After 20 steps:



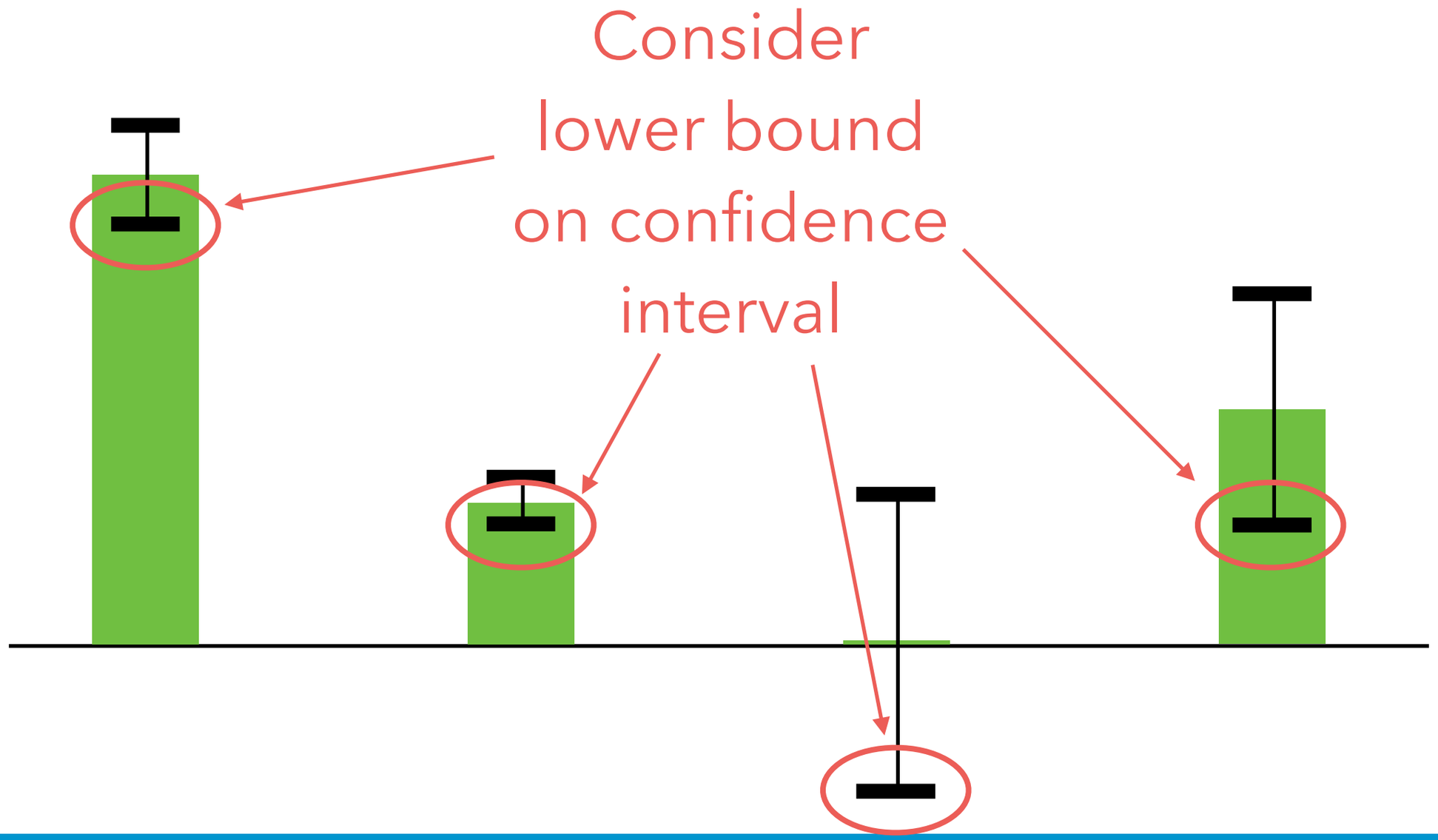
A different perspective

- After 20 steps (with confidence intervals):



A different perspective

- **Principle:** Optimism in the face of uncertainty



UCB algorithm

- Execute each action once
- From then on, at each step t, select action

$$a^* = \operatorname{argmin} \hat{c}(a) - \sqrt{\frac{2 \log t}{N_t(a)}}$$

Average cost

Confidence interval

Important notes

- In the literature, this algorithm was first proposed with rewards instead of costs
- With rewards, instead of minimizing the lower confidence bound, the algorithm maximizes the **upper confidence bound**, hence the name of the algorithm

Does this work?

- To measure the performance of UCB, we compare its total expected cost with the one optimal:

$$R_T = \mathbb{E}\left[\sum_{t=0}^{T-1} \mathrm{c}(a_t) - \min_{a \in \mathcal{A}} \sum_{t=0}^{T-1} \mathrm{c}(a)\right]$$

Random variables

Does this work?

- To measure the performance of UCB, we compare its total expected cost with the one optimal:

$$R_T = \mathbb{E}\left[\sum_{t=0}^{T-1} \mathrm{c}(a_t) - \min_{a \in \mathcal{A}} \sum_{t=0}^{T-1} \mathrm{c}(a)\right]$$

- Equivalently:

$$R_T = \sum_{a \in \mathcal{A}} \mathbb{E}\left[N_T(a)\right] c(a) - T c(a^*)$$

Expected n. of times that *a* is selected

Expected cost of optimal action

Does this work?

- To measure the performance of UCB, we compare its total expected cost with the one optimal:

$$R_T = \mathbb{E}\left[\sum_{t=0}^{T-1} \mathrm{c}(a_t) - \min_{a \in \mathcal{A}} \sum_{t=0}^{T-1} \mathrm{c}(a)\right]$$

- Equivalently:

$$R_T = \sum_{a \in \mathcal{A}} \mathbb{E}\left[N_T(a)\right] c(a) - T c(a^*)$$

$$= \sum_{a \in \mathcal{A}} \mathbb{E}\left[N_T(a)\right] \underbrace{(c(a) - c(a^*))}_{\Delta(a)}$$

Does this work?

- Expanding $\mathbb{E}\left[N_T(a)\right]$, we get the bound

$$R_T \leq \sum_{a \neq a^*} \frac{8 \log T}{\Delta(a)} + 2 \sum_{a \in \mathcal{A}} \Delta(a)$$

UCB

- Only observes cost for selected actions (bandit problem)
- It assumes that costs follow an unknown (but fixed) distribution
- Its regret is **sublinear in *T***

No-regret prediction

- With some tricky manipulation, we can turn the bound into

$$R_T \leq \sqrt{CNT \log T}$$

Constant

Grows with $\sqrt{T \log T}$ (slower than T)

Overview:

- No assumptions on the cost process; observation of complete costs: EWA

$$R_T \leq \sqrt{\frac{T}{2} \log N}$$

- No assumptions on the cost process; observation of single-action cost (bandit setting): EXP3

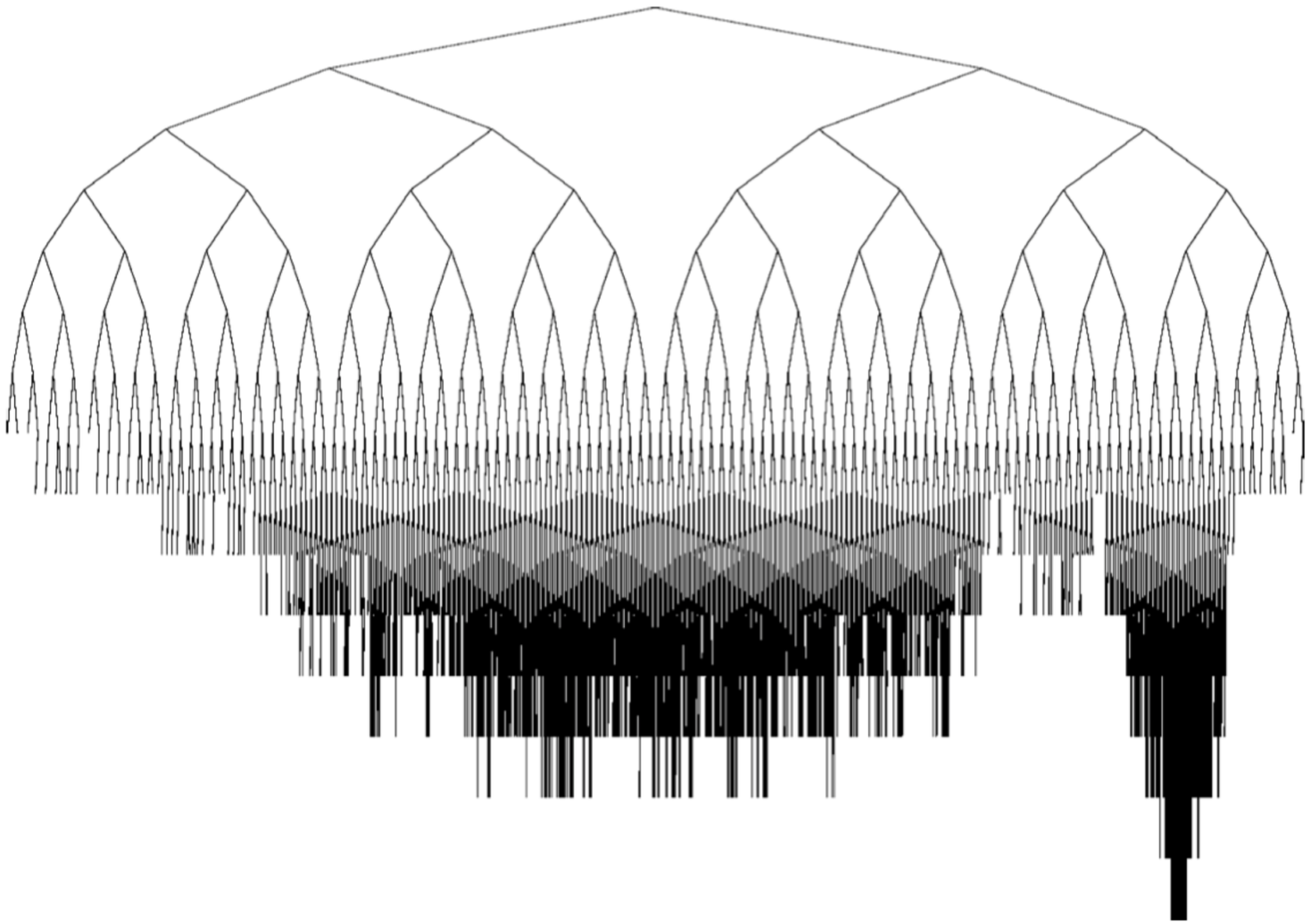
$$R_T \leq \sqrt{2TN \log N}$$

- Costs sampled from distribution; observation of single-action cost (bandit setting): UCB

$$R_T \leq \sqrt{CNT \log T}$$

TÉCNICO LISBOA

Applications



Monte Carlo Tree Search

Monte Carlo Tree Search

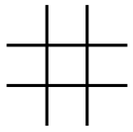
- Application of UCB
- Very efficient planning method for large domains
- **Idea:**
  - Build a search tree incrementally
  - Use sampling to compute node values

MCTS (cont.)

- Four stages:
  - Select a node to expand (SELECTION)
  - Expand it (EXPANSION)
  - Simulate the process starting from that node (SIMULATION)
  - Back-propagate the resulting value (BACK-PROPAGATION)

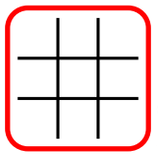
Example (Tic-Tac-Toe)

1. Select node to expand



$Q = 0$
$N = 0$

Example (Tic-Tac-Toe)



$Q = 0$
$N = 0$

1. Select node to expand

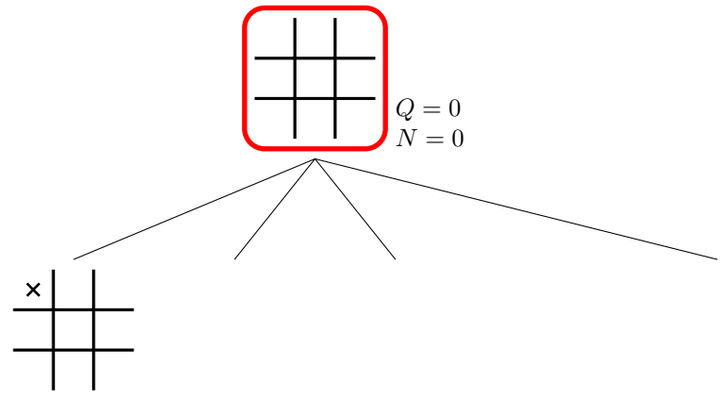
2. Expand it

Example (Tic-Tac-Toe)

1. Select node to expand

2. Expand it

3. Simulate games from the leaves



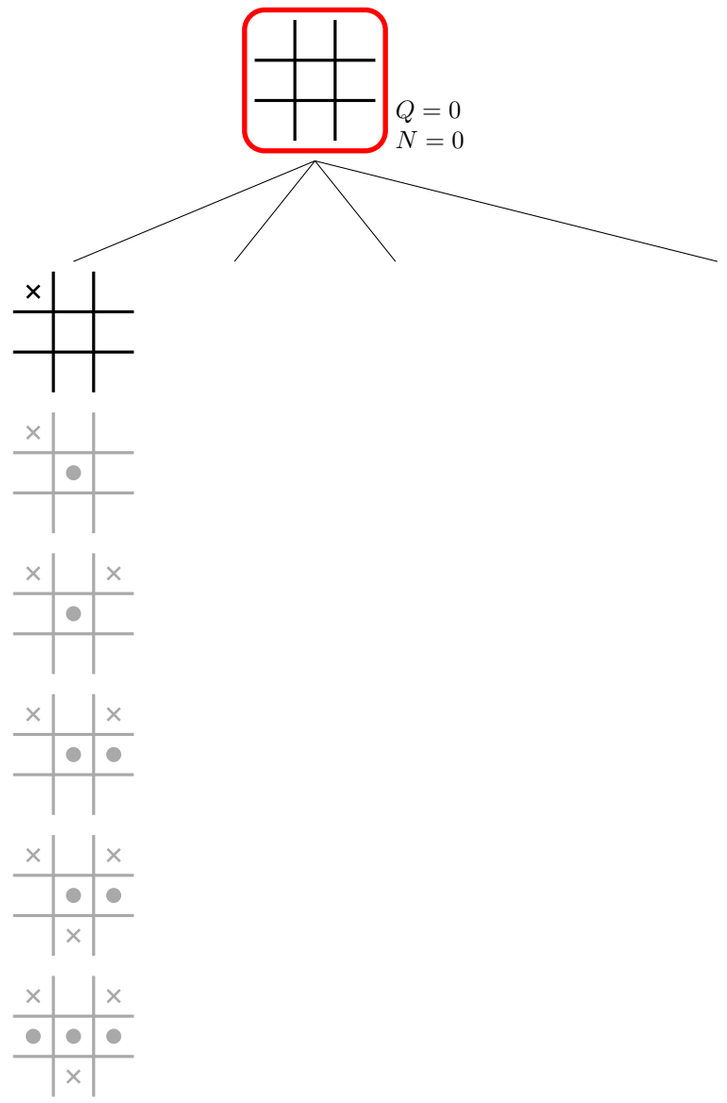
Example (Tic-Tac-Toe)

1. Select node to expand

2. Expand it

3. Simulate games from the leaves

4. Back-propagate values



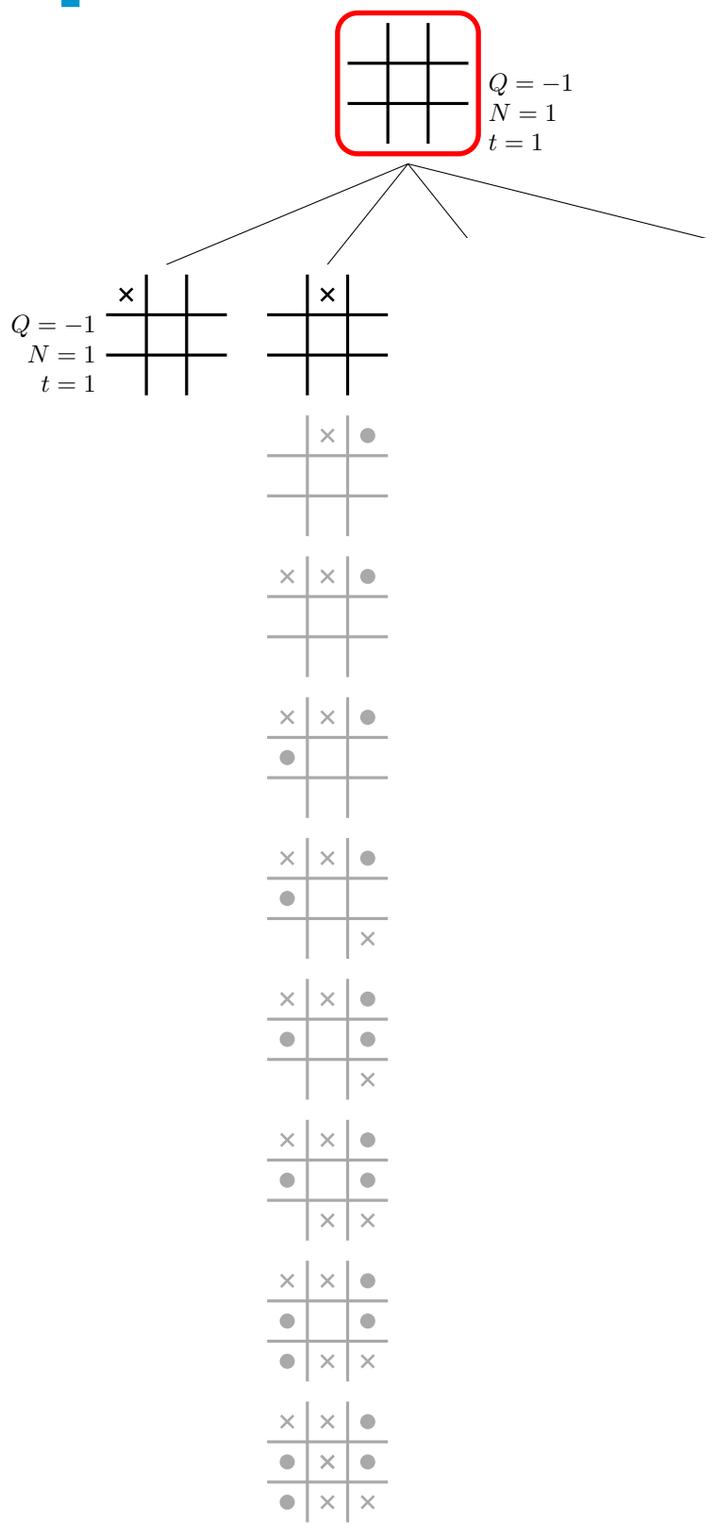
Example (Tic-Tac-Toe)

1. Select node to expand

2. Expand it

3. Simulate games from the leaves

4. Back-propagate values

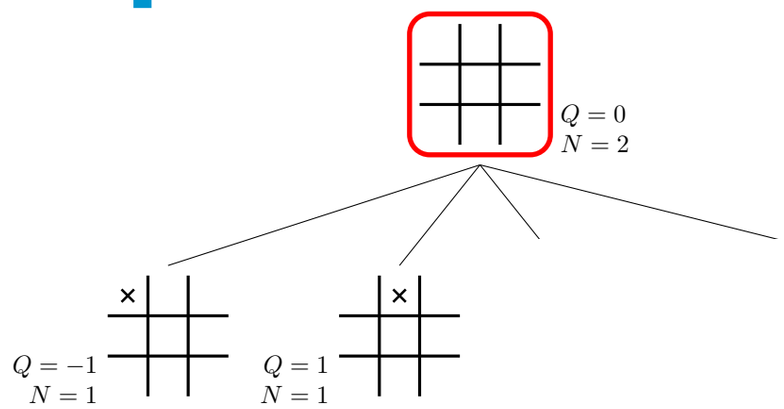


Example (Tic-Tac-Toe)

1. Select node to expand

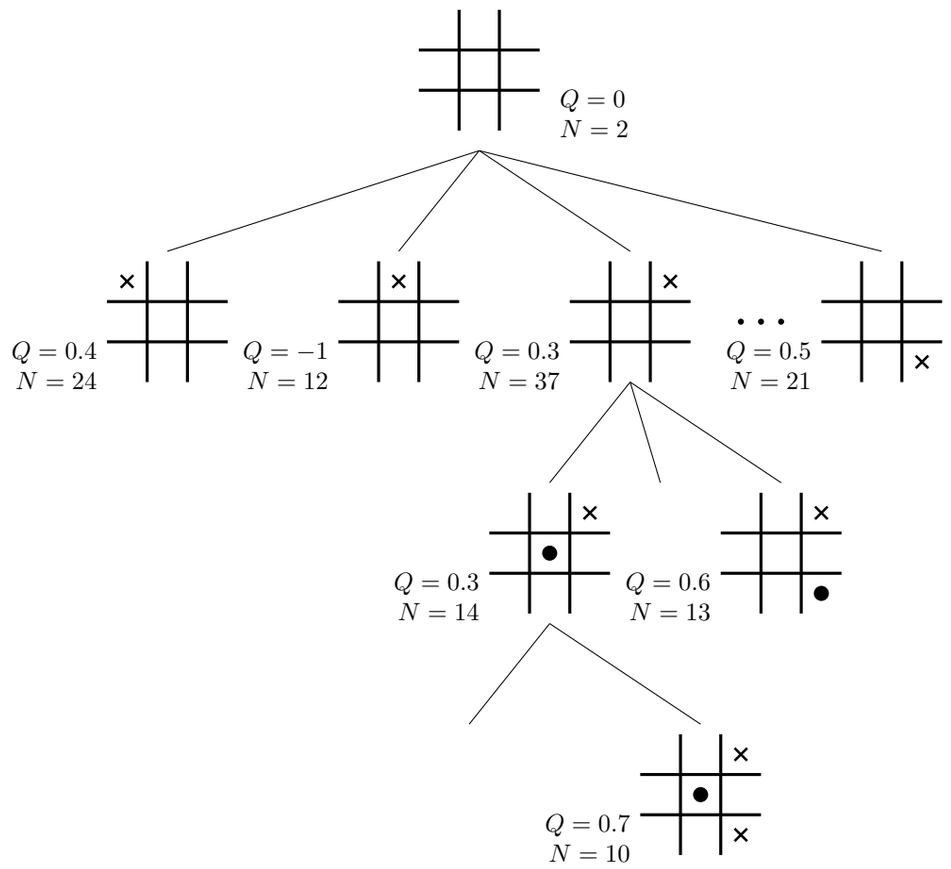
2. Expand it

3. Simulate games from the leaves



4. Back-propagate values

After several iterations



Selection

- How is the node to be expanded selected?
  - Several approaches exist
  - Most standard nowadays is UCB
  - The resulting MCTS algorithm is called UCT (UCB for Trees)
  - At each node, selects the node *i* minimizing

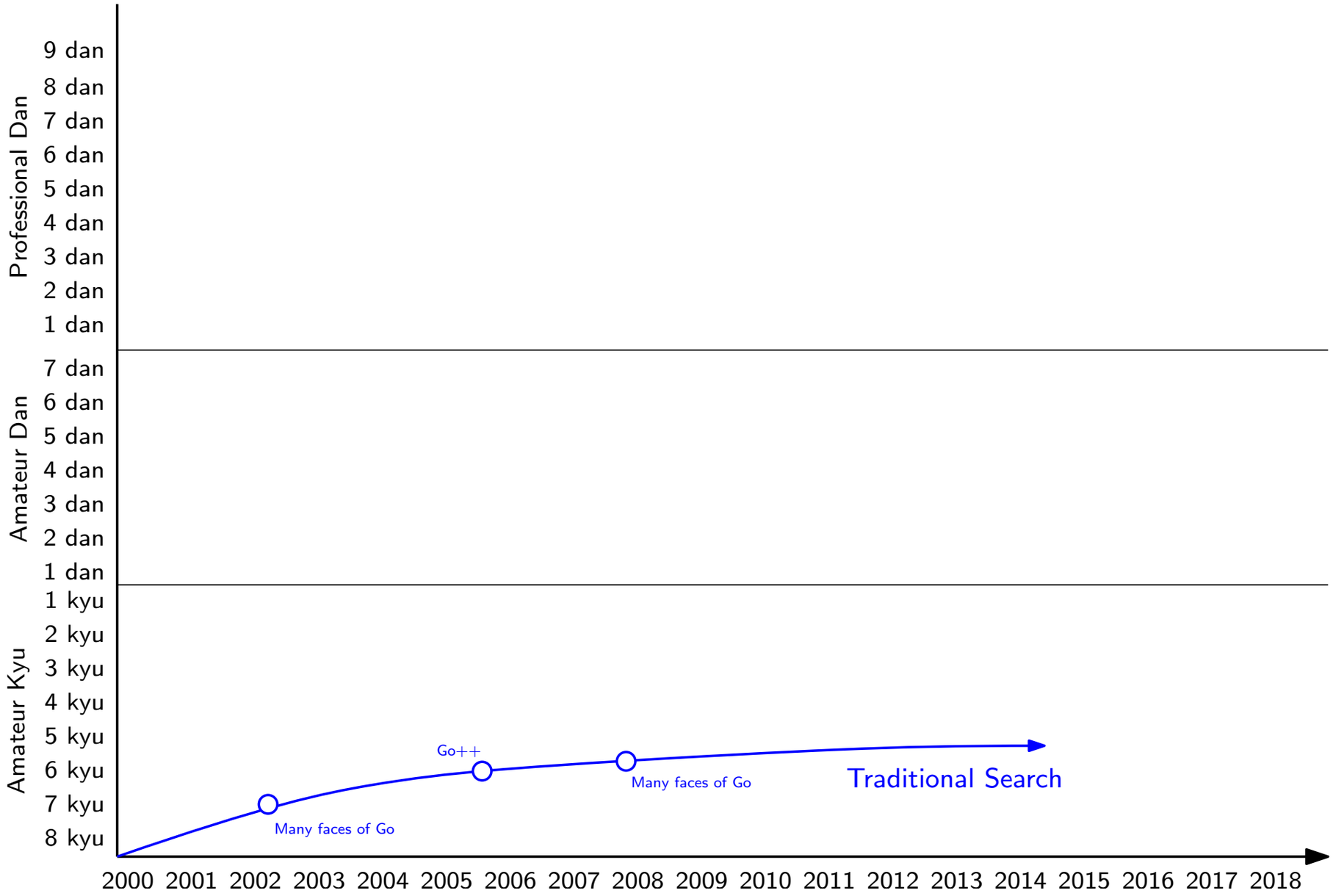
$$Q(i) - c\sqrt{\frac{\log(t)}{N(i)}}$$

UCB expression!

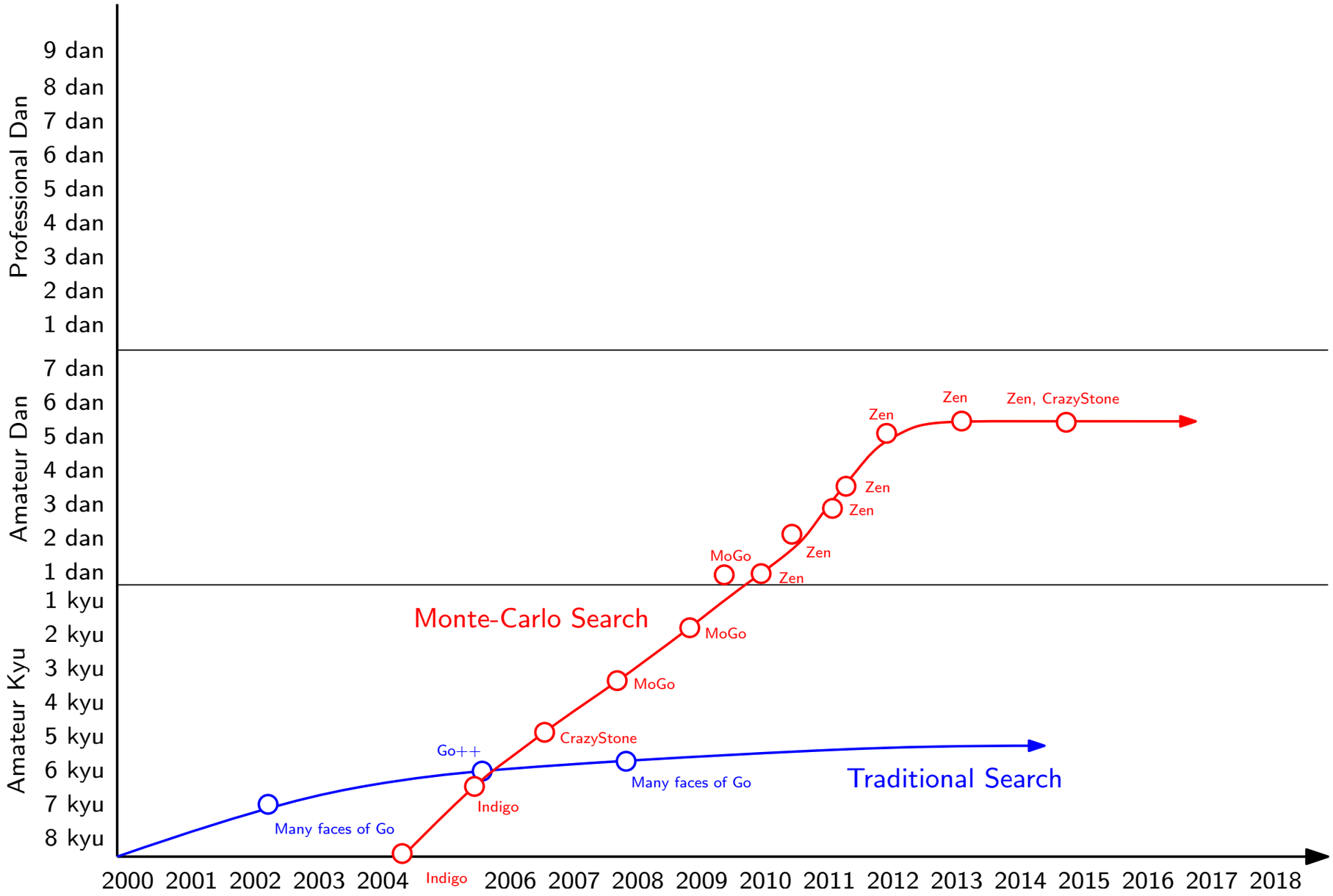
MCTS

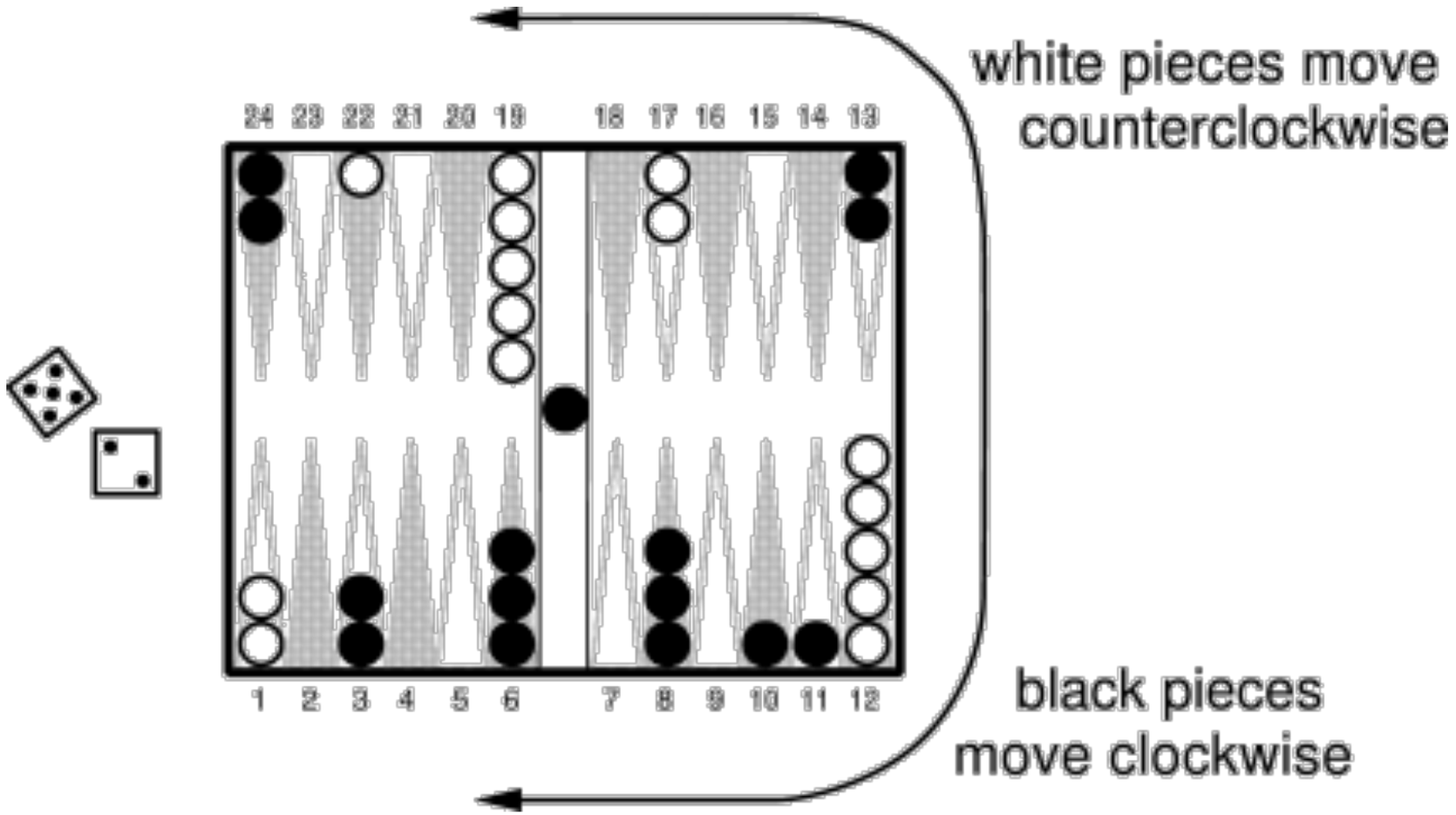
- The use of the “UCB heuristic” allows the tree expansion to follow the most promising route
- It is able to effectively “navigate” the game tree to and focus on areas that are more likely to reach victory

Evolution of Computer Go



Evolution of Computer Go



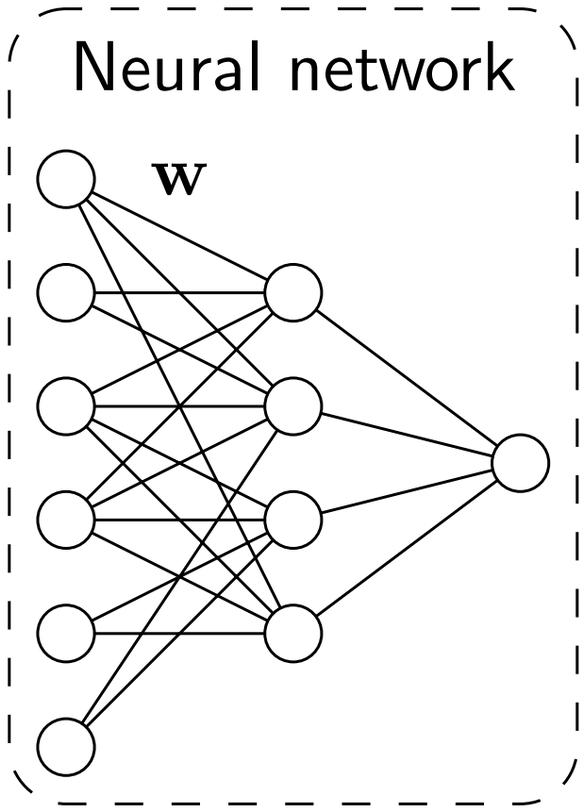

white pieces move
counterclockwise
24 23 22 21 20 19 18 17 16 15 14 13
1 2 3 4 5 6 7 8 9 10 11 12
black pieces
move clockwise

TD Gammon

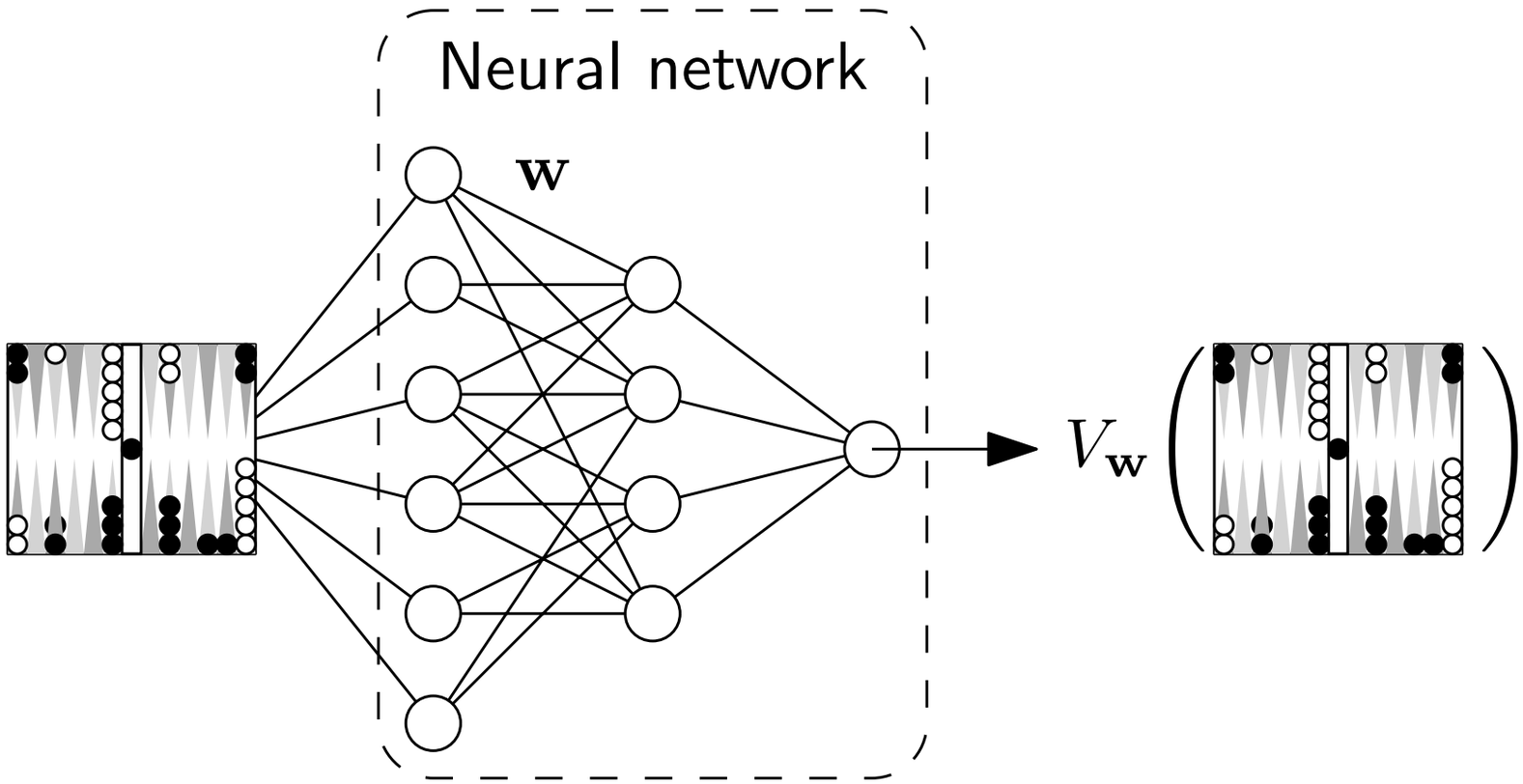
TD-Gammon

- Combines neural networks with TD-learning
- First computer program to achieve top human-level play in backgammon
- Neural network with a total of 80 hidden units
- Learned by playing games against itself

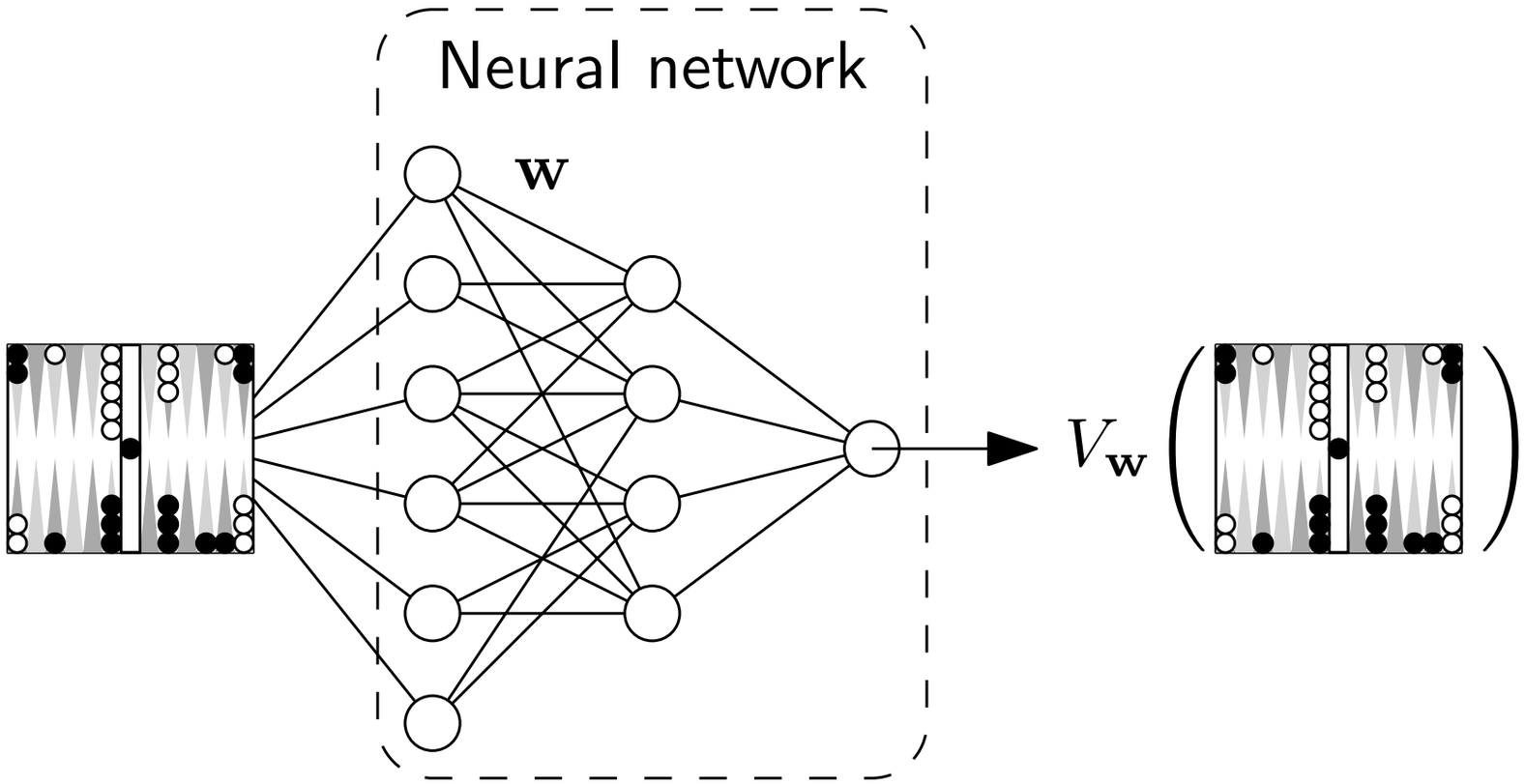
Illustration



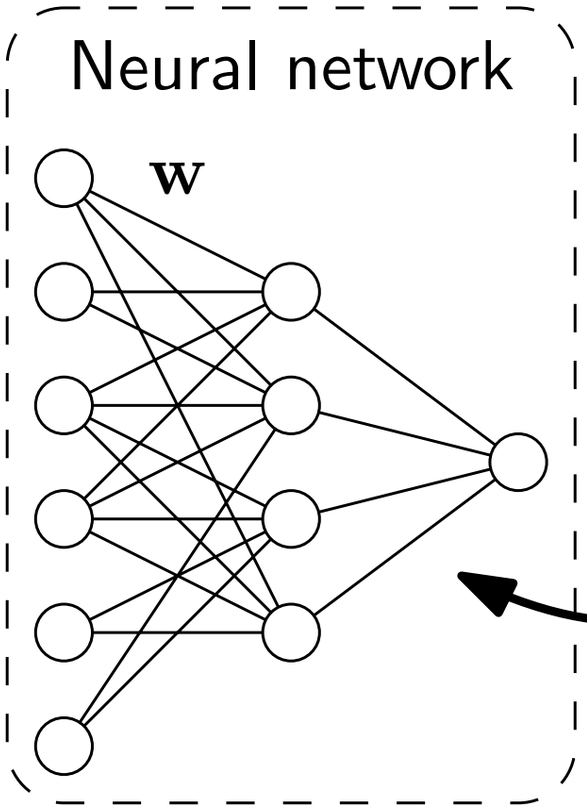
Illustration



Illustration



Illustration



$$\text{Error} = V_{\mathbf{w}}\left(\right) - V_{\mathbf{w}}\left(\right)$$

Training

- Neural network update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \mathbf{z}_t(X)(V_{\mathbf{w}}(X_{t+1}) - V_{\mathbf{w}}(X_t))$$ TD update

where $\mathbf{z}_t$ is the eligibility trace, given as

$$\mathbf{z}_{t+1}(X) = \mathbf{z}_t(X)\lambda + \nabla_{\mathbf{w}} V_{\mathbf{w}}(X)$$

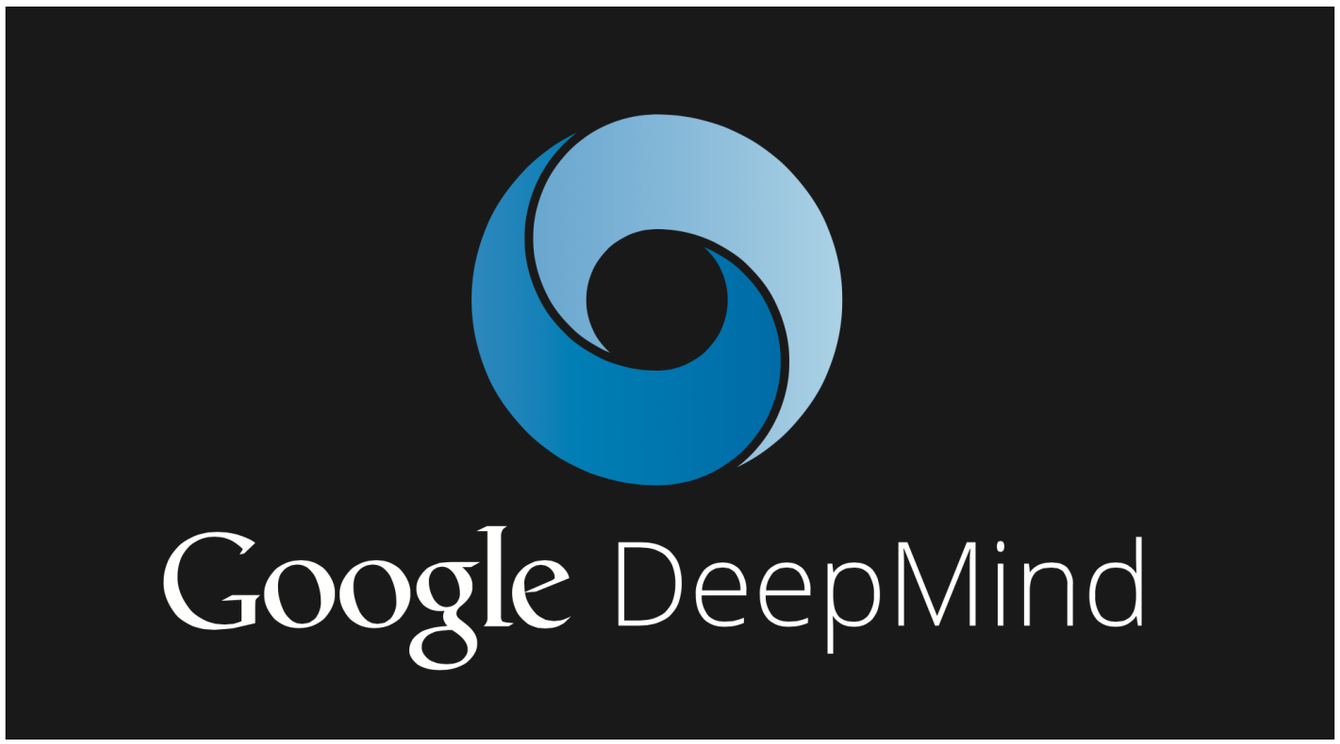
Other facts

- TD-Gammon was developed at IBM by Gerald Tesauro, in 1992
- The program played several world top players, attaining similar performance
- The program introduced several important changes to the way the game is played – for example, TD-Gammon used an opening previously thought to be poor but which was eventually adopted by top players

AlphaGo

Some facts…

- *AlphaGo* is a Go playing program developed by UK-based company Deep Mind (owned by Google)
- Deep Mind was also behind the system that learned to play Atari games at human level



How AlphaGo works…

- Basic idea: **MCTS**
  - Simulate good opponents (for simulation)
  - Compute robust optimal board values (to perform selection, etc.)

Don’t know optimal policy!

How AlphaGo works…

- Basic idea: **MCTS**
  - Simulate good opponents
  - Compute robust optimal board values
- Compute a great policy
- Evaluate great policy

How AlphaGo works…

- Basic idea: **MCTS**
  - Simulate good opponents → Policy network
  - Compute robust optimal board values
- Compute a great policy → Policy network 2
- Evaluate great policy → Value network

Modules of AlphaGo

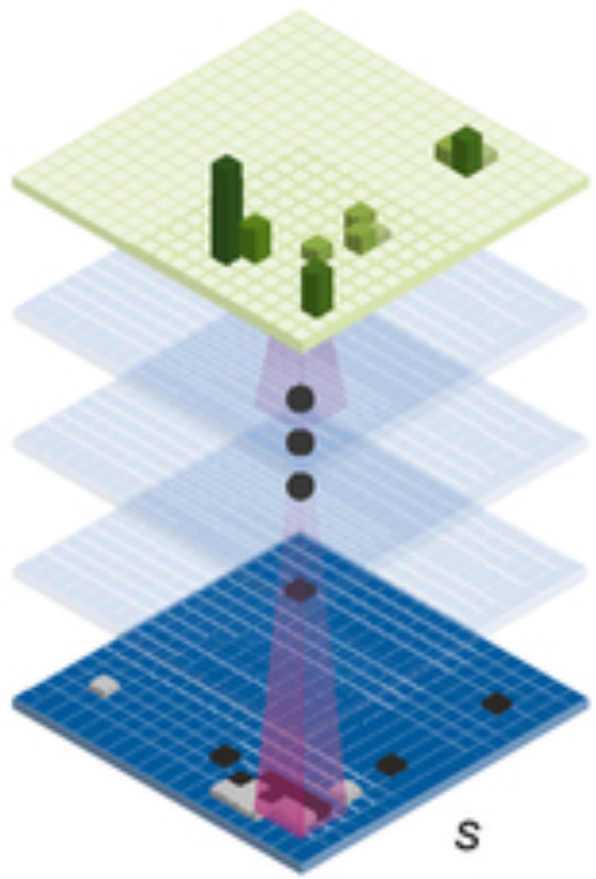
- Policy network trained to predict human plays
- Policy network trained to select “winning” plays
- Value network to evaluate board positions

SL policy network

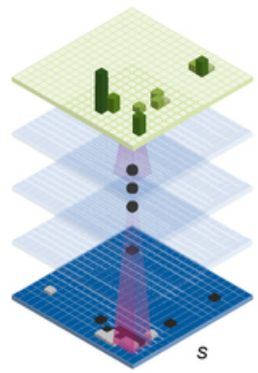
- Optimized to predict human moves
- Trained from 30×10^6 boards from experts
- Attained a prediction accuracy of 57% (almost 15% above existing predictors)

Policy network
(Supervised learning)

$p_\sigma(a|s)$



Smaller (but faster) network

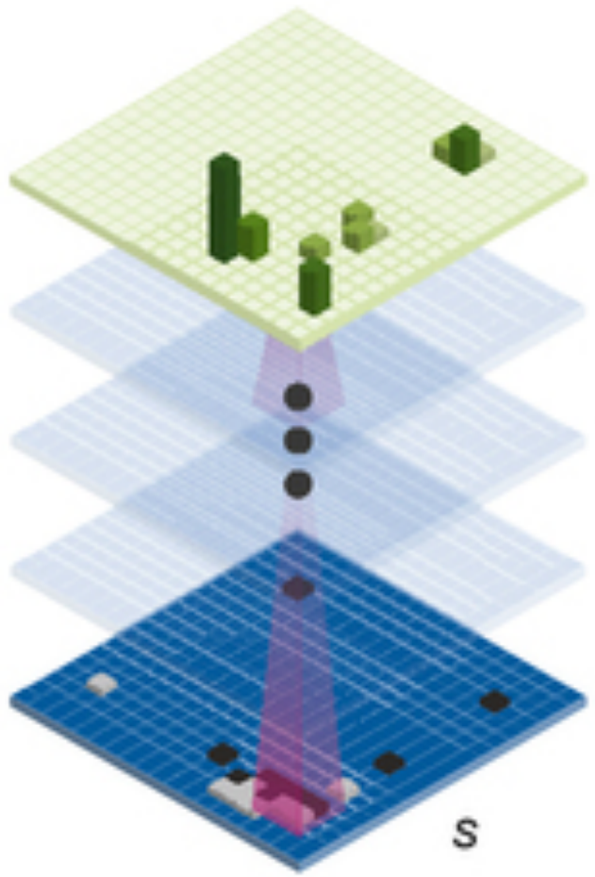


RL policy network

- Optimized to “win”
- Trained in self play (against previous versions of itself)
- Network weights updated using policy gradient

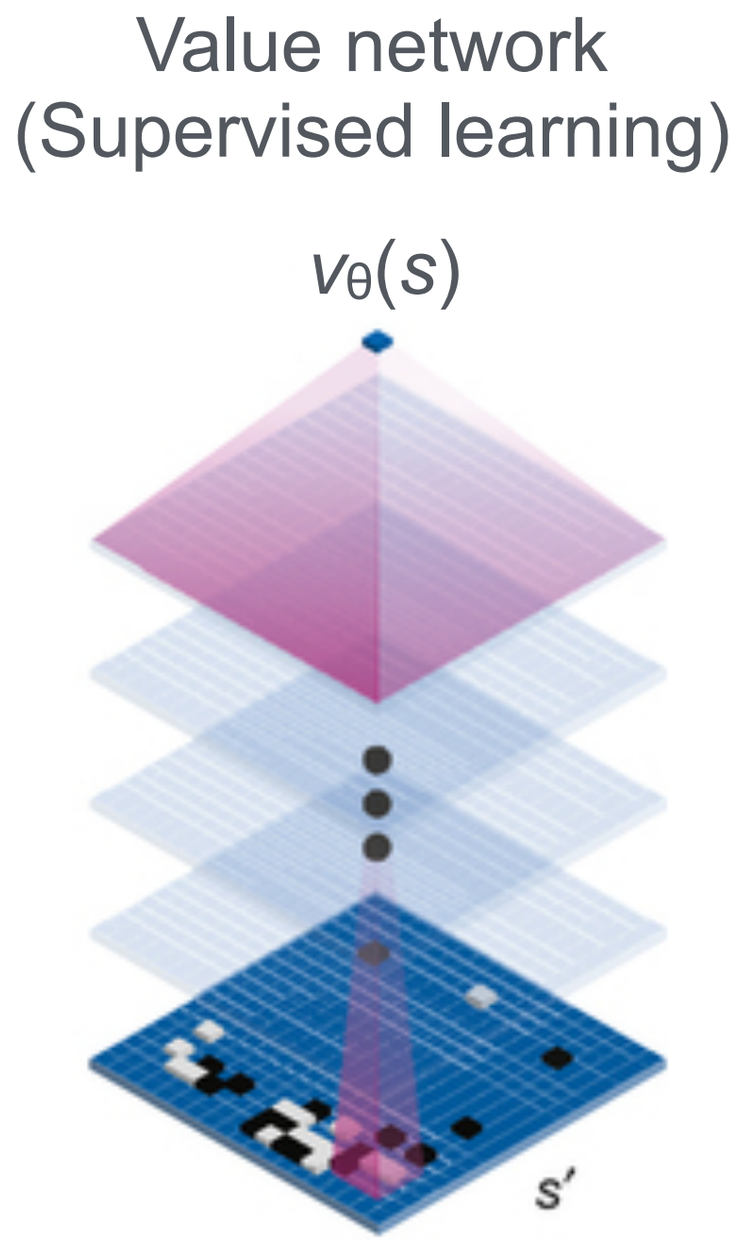
Policy network
(Reinforcement learning)

$p_\rho(a|s)$



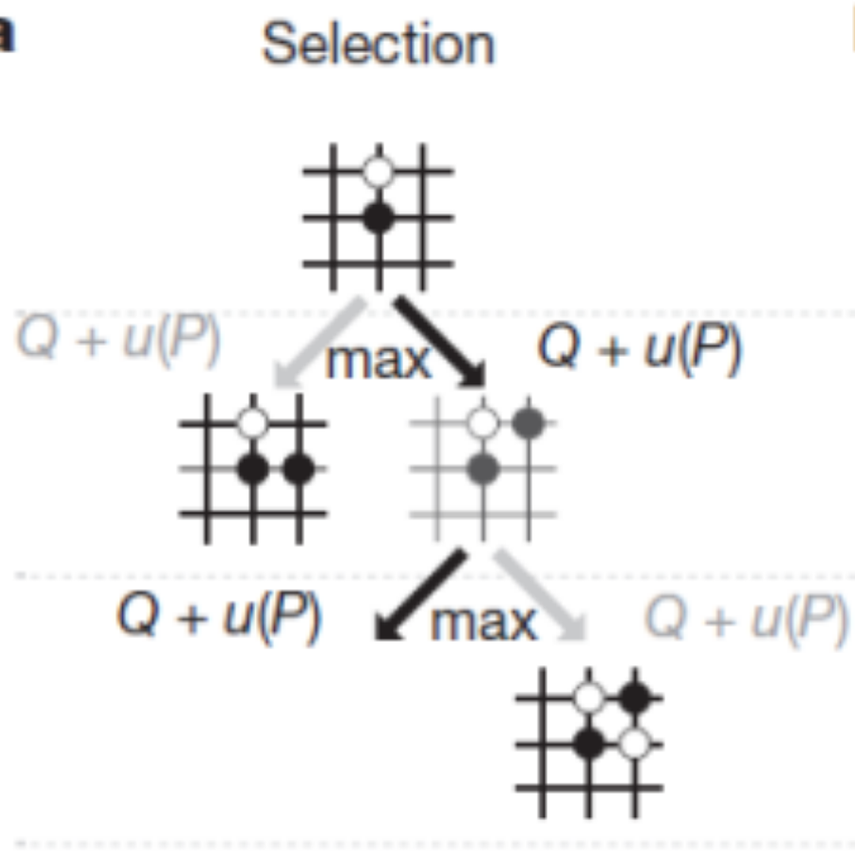
Value network

- Used to evaluate board positions
- Trained from (sampled) board positions obtained during self-play games using the policy obtained from the RL network



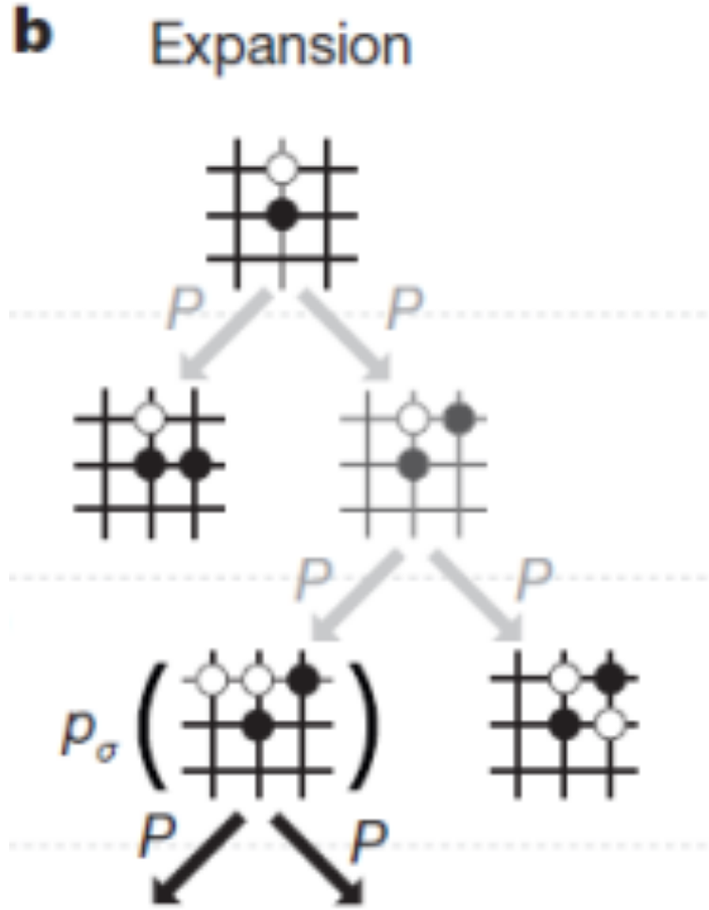
MCTS in AlphaGo

- **Selection:**
  - Tree is traversed from root to leaf by selecting UCB-like heuristic
  - The "bonus" $u(P)$ depends on the prior probability P, provided by the SL policy network



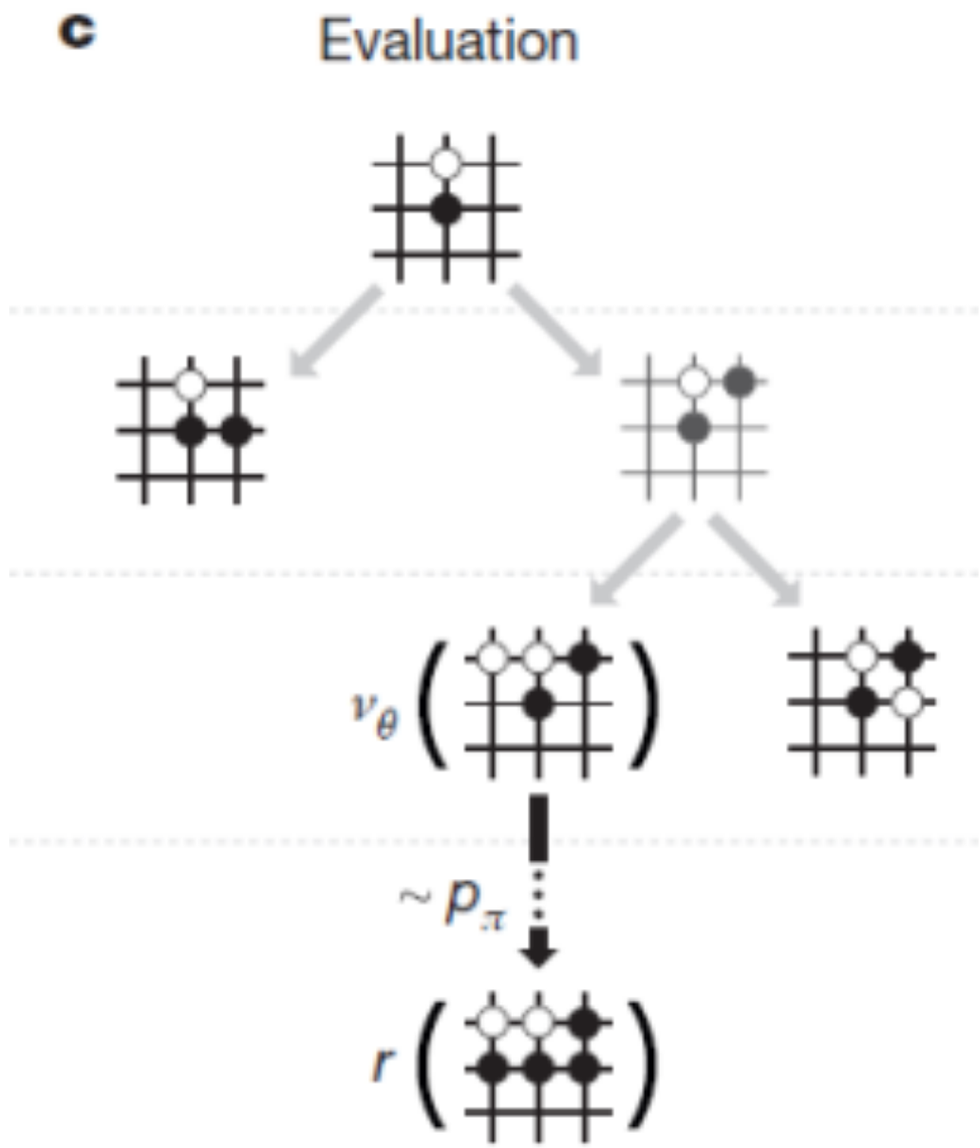
MCTS in AlphaGo

- **Expansion:**
  - New leaf nodes are created and assigned prior probabilities *P* from SL network



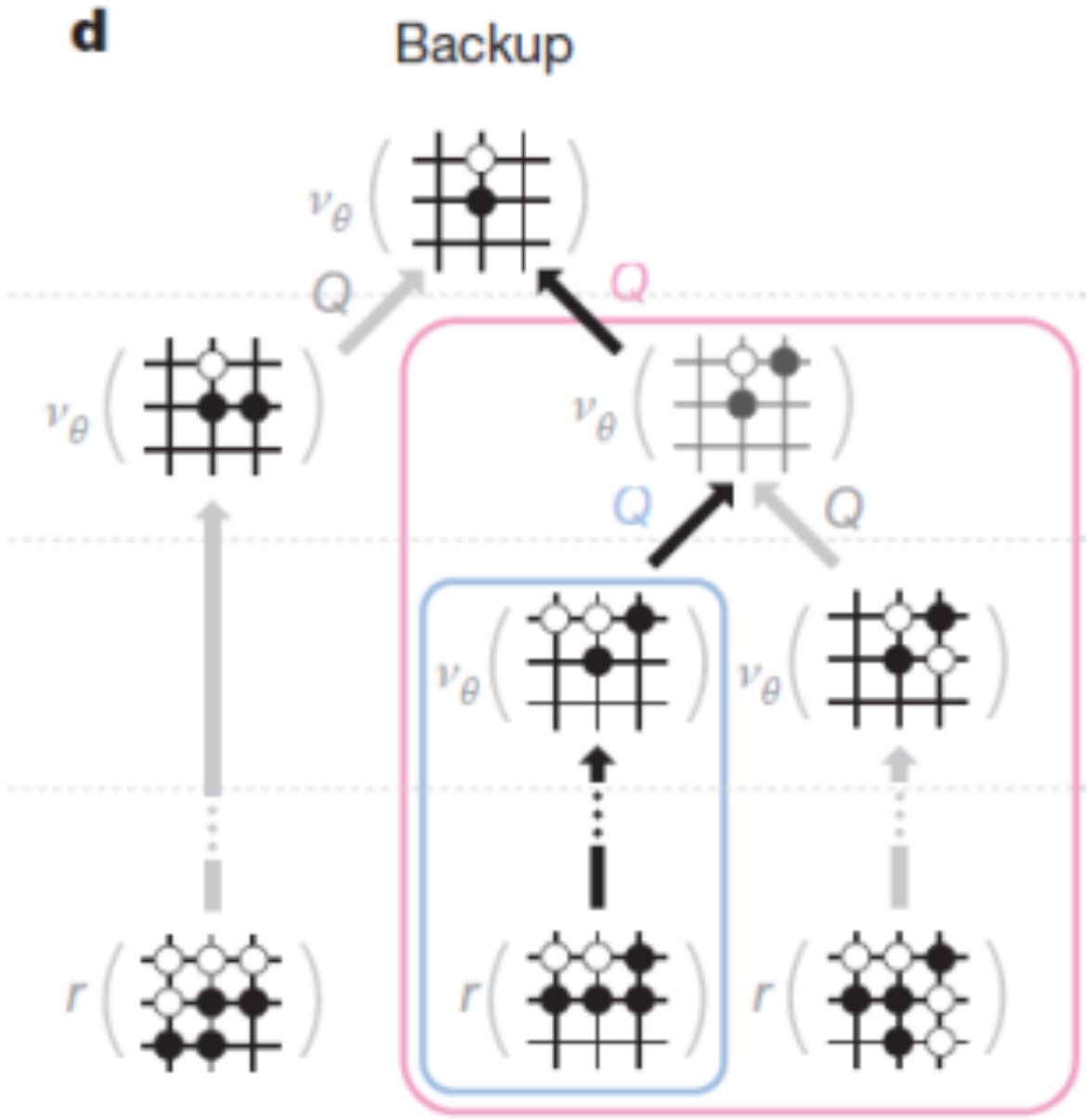
MCTS in AlphaGo

- **Evaluation:**
  - Games are “simulated” until the end using the small network



MCTS in AlphaGo

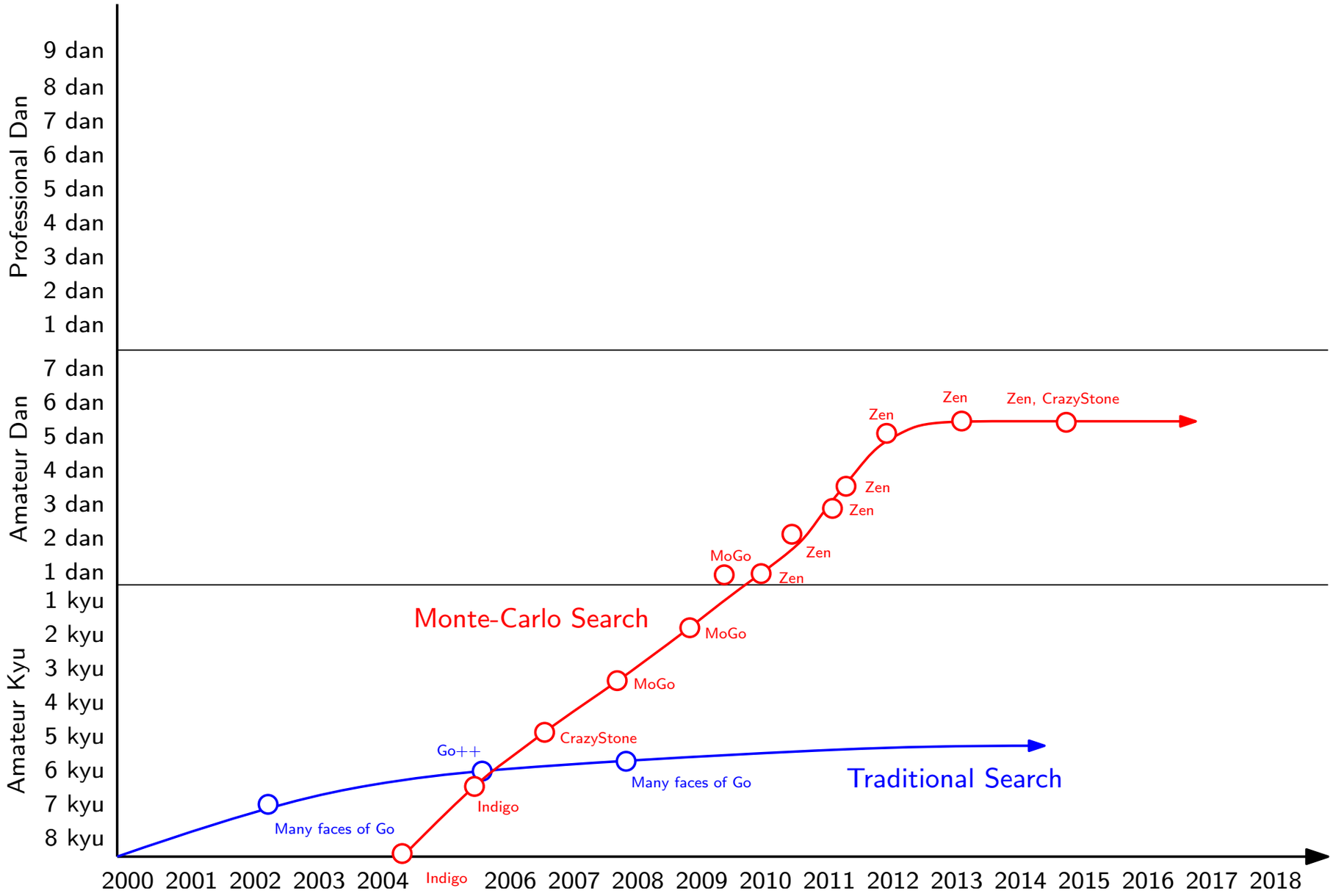
- **Evaluation:**
  - Value obtained from simulation is combined with value from value network
  - Q values are computed from by averaging the resulting evaluations



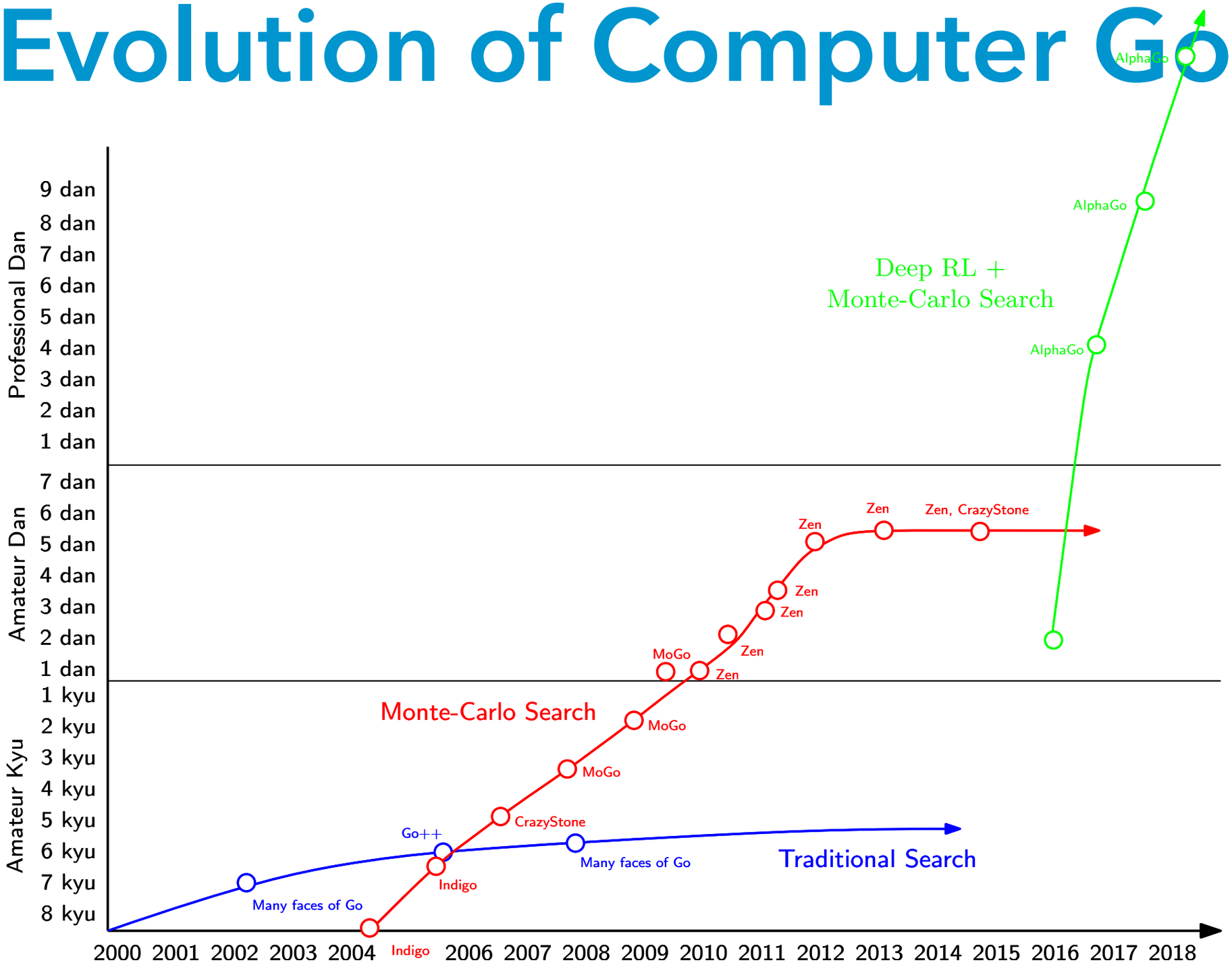
Meet AlphaGo

Distributed *AlphaGo*: 1202 CPUs and 176 GPUs

Evolution of Computer Go



Evolution of Computer Go



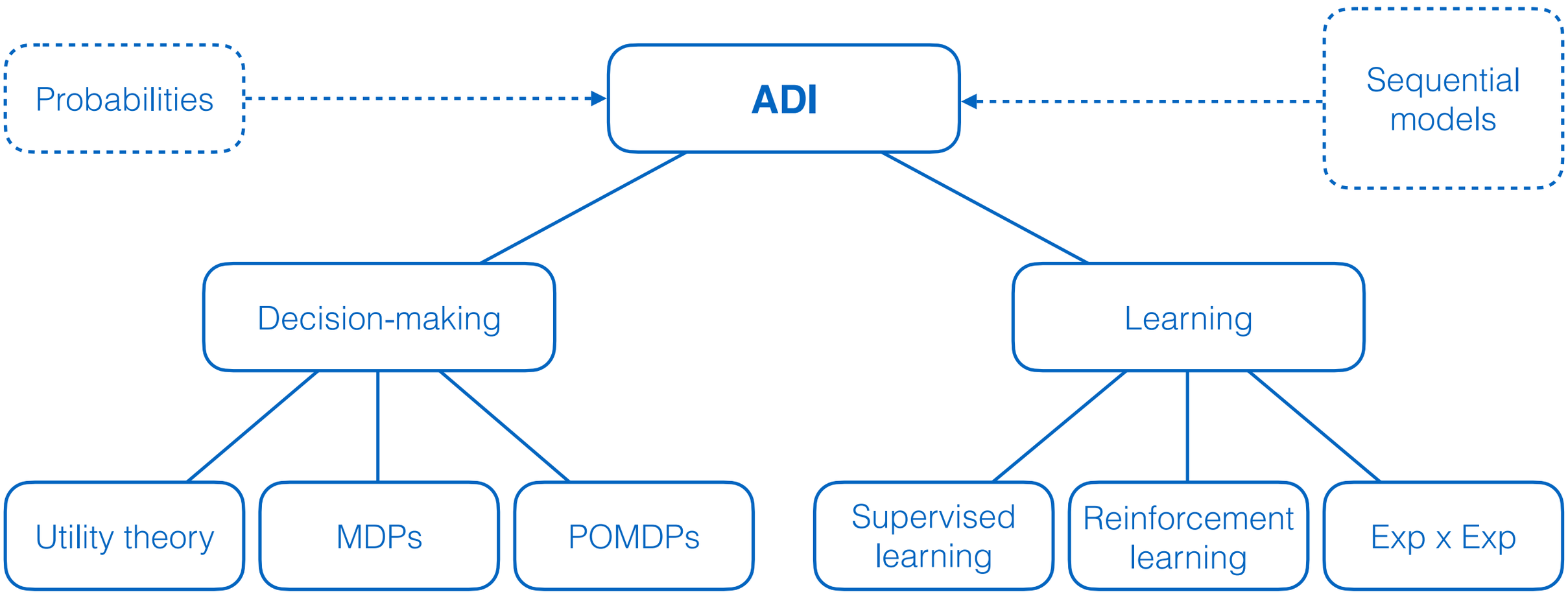
TÉCNICO LISBOA

To conclude...

What we set out to do…

- Understand the main issues involved in **decision making**, in face of **uncertainty**
- Familiarize with some of the main tools for **planning** and **learning** in such settings

Learning and Decision Making



"There is no end to education. It is not that you read a book, pass an examination, and finish with education. The whole of life, from the moment you are born to the moment you die, is a process of learning."

– Jiddu Krishnamurti


"That's all Folks!"