Reinforcement Learning

Bandits Tools for Monte-Carlo Planning

Emilie Kaufmann

M2 MVA, 2025/2026

Planning in a MDP (or a game)

Planning (here) = deciding for the next action in a given state
$\neq$ learning a policy (i.e. how to act in all states)

Given a state s_1, several possible objective :

- find a good estimate of $V^\star(s_1)$
- find an action a such that $Q^\star(s_1, a)$ is close to $V^\star(s_1) = \max_b Q^\star(s_1, b)$
- find a good distribution over next actions

using as few calls to a generative model.

Example : decide the next move to play in a game
(call to the generative model = simulate a step in the game).

Outline

Monte Carlo Tree Search

MCTS is a family of methods that adaptively explore the tree of possible next states in a given state s_1, in order to find the best action in s_1.

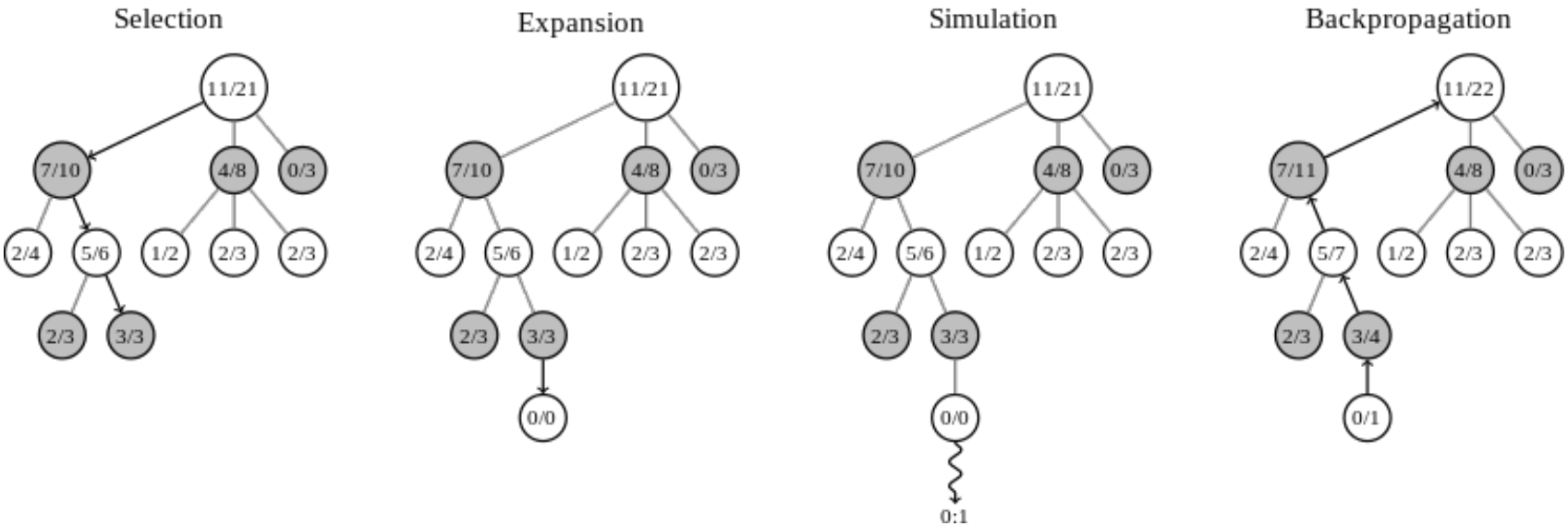


Figure – A generic MCTS algorithm for a game

MCTS requires a generative model (to sample trajectories from s_1)

Sparse Sampling

A MCTS algorithm for planning in a discounted MDP with sample complexity guarantees.

Function: **EstimateQ**(h, C, γ, G, s)
Input: depth h, width C, discount γ, generative model G, state s.
Output: A list $(\hat{Q}_h^*(s,a_1), \hat{Q}_h^*(s,a_2), \dots, \hat{Q}_h^*(s,a_k))$, of estimates of the $Q^*(s,a_i)$.

1. If $h = 0$, return $(0, \dots, 0)$.
2. For each $a \in A$, use G to generate C samples from the next-state distribution $P_{sa}(\cdot)$. Let S_a be a set containing these C next-states.
3. For each $a \in A$ and let our estimate of $Q^*(s,a)$ be

$$\hat{Q}_h^*(s,a) = R(s,a) + \gamma \frac{1}{C} \sum_{s' \in S_a} \textbf{EstimateV}(h-1, C, \gamma, G, s'). \quad (5)$$

4. Return $(\hat{Q}_h^*(s,a_1), \hat{Q}_h^*(s,a_2), \dots, \hat{Q}_h^*(s,a_k))$.

Function: **EstimateV**(h, C, γ, G, s)
Input: depth h, width C, discount γ, generative model G, state s.
Output: A number $\hat{V}_h^*(s)$ that is an estimate of $V_h^*(s)$.

1. Let $(\hat{Q}_h^*(s,a_1), \hat{Q}_h^*(s,a_2), \dots, \hat{Q}_h^*(s,a_k)) :=$ **EstimateQ**(h, C, γ, G, s).
2. Return $\max_{a \in \{a_1, \dots, a_k\}} \{\hat{Q}_h^*(s,a)\}$.

Function: **Algorithm** $\mathcal{A}(\epsilon, \gamma, R_{max}, G, s_0)$
Input: tolerance ϵ, discount γ, max reward R_{max}, generative model G, state s_0.
Output: An action a.

1. Let the required horizon H and width C parameters be calculated as given as functions of ϵ, γ and R_{max} in Theorem1.
2. Let $(\hat{Q}_H^*(s,a_1), \hat{Q}_H^*(s,a_2), \dots, \hat{Q}_H^*(s,a_k)) :=$ **EstimateQ**(H, C, γ, G, s_0).
3. Return $\arg\max_{a \in \{a_1, \dots, a_k\}} \{\hat{Q}_H^*(s,a)\}$.

[Kearns et al., 2002]

Parameters tuning :

$$H = \left\lceil \frac{\ln(V_{\max}/\lambda)}{\ln(1/\gamma)} \right\rceil$$

$$C = 3 \frac{V_{\max}^2}{\lambda^2} H \ln\left(\frac{k H V_{\max}^2}{\lambda^2} \right)$$

$$\lambda = \frac{\epsilon(1-\gamma)^2}{4}, \quad V_{\max} = \frac{R_{\max}}{1-\gamma}$$

k : branching factor

Sample complexity :

$$\left(\frac{k}{\epsilon(1-\gamma)} \right)^{\left(\frac{1}{1-\gamma \ln\left(\frac{1}{\epsilon(1-\gamma)} \right)} \right)}$$

calls to the generative model to get $|V^{\mathcal{A}}(s) - V^{\star}(s)| \leq \epsilon$

Sparse Sampling

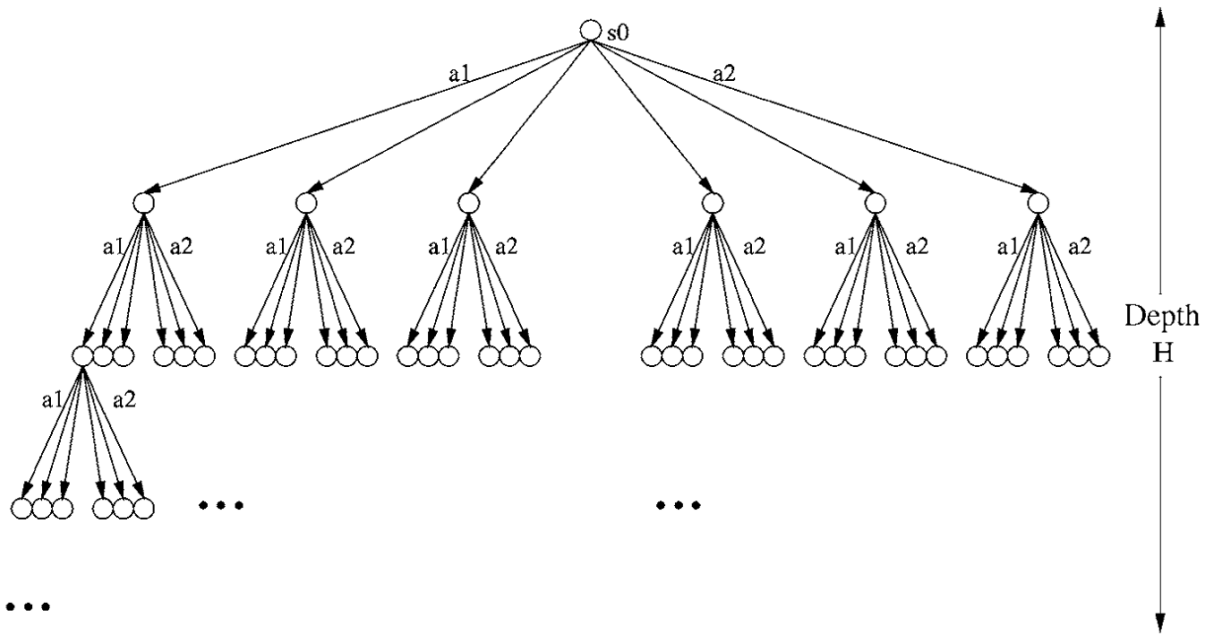


Figure 2. Sparse look-ahead tree of states constructed by the algorithm (shown with $C = 3$, actions a_1, a_2).

➜ one can go beyond recursive algorithms with bandit approaches

Outline

The UCT algorithm

Bandit-Based Monte-Carlo planning : to select a path in the tree, run a bandit algorithm each time a children (next action) needs to be selected

UCT = UCB for Trees [Kocsis and Szepesvári, 2006]

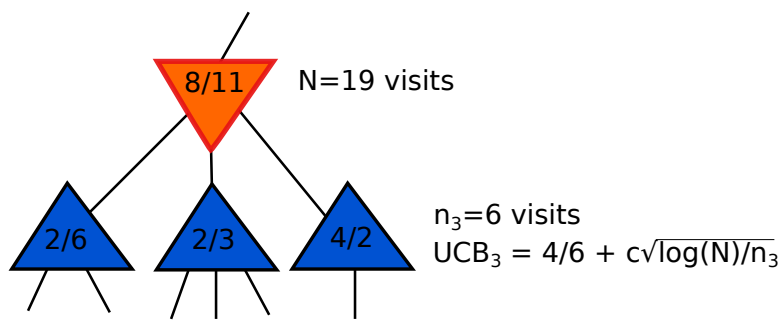
UCT in a Game Tree

In a MAX node s (= root player move), select an action

$$\underset{a \in \mathcal{C}(s)}{\operatorname{argmax}} \ \frac{S(s,a)}{N(s,a)} + c\sqrt{\frac{\log\left(\sum_b N(s,b)\right)}{N(s,a)}}$$

$N(s,a)$: number of visits of (s,a)

$S(s,a)$: number of visits of (s,a) ending with the root player winning



The UCT algorithm

Bandit-Based Monte-Carlo planning : to select a path in the tree, run a bandit algorithm each time a children (next action) needs to be selected

UCT = UCB for Trees [Kocsis and Szepesvári, 2006]

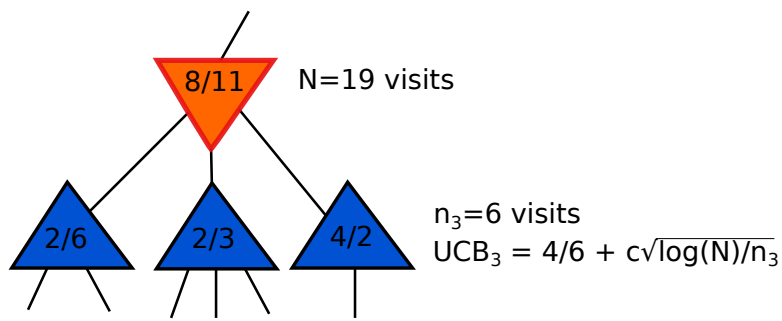
UCT in a Game Tree

In a MIN node s (= adversary move), select an action

$$\underset{a \in \mathcal{C}(s)}{\operatorname{argmin}} \ \frac{S(s,a)}{N(s,a)} - c\sqrt{\frac{\log\left(\sum_b N(s,b)\right)}{N(s,a)}}$$

$N(s,a)$: number of visits of (s,a)

$S(s,a)$: number of visits of (s,a) ending with the root player winning



The UCT algorithm

Bandit-Based Monte-Carlo planning : to select a path in the tree, run a bandit algorithm each time a children (next action) needs to be selected

UCT = UCB for Trees [Kocsis and Szepesvári, 2006]

UCT in a Game Tree

In a MAX node s (= root player move), select an action

$$\operatorname*{argmax}_{a \in \mathcal{C}(s)} \frac{S(s,a)}{N(s,a)} + c\sqrt{\frac{\log\left(\sum_b N(s,b)\right)}{N(s,a)}}$$

$N(s,a)$: number of visits of (s,a)
$S(s,a)$: number of visits of (s,a) ending with the root player winning

When a leaf (or some maximal depth) is reached :

- a playout is performed (play the game until the end with a simple heuristic, or produce a random evaluation of the leaf position)
- the outcome of the playout (typically 1/0) is stored in all the nodes visited in the previous trajectory

The UCT algorithm

- first good AIs for Go where based on variants on UCT
- it remains a heuristic (no sample complexity guarantees, parameter c fined-tuned for each application)
- many variants have been proposed

[Browne et al., 2012]

Outline

Alpha Zero

AlphaZero learns a good policy by using a MCTS algorithm guided by a neural network

$\neq$ pure play-out based MCTS

Input

A neural network predicting a policy $\boldsymbol{p} \in \Delta(\mathcal{A})$ and a value $v \in \mathbb{R}$ from the current state s : $(\boldsymbol{p}, v) = f_\theta(s)$.

The MCTS algorithm maintains for each visited state/action the counts and cumulated values + a vector of prior action probabilities :

$$\{N(s,a), S(s,a), P(s,a)\}$$

Alpha Zero

AlphaZero learns a good policy by using a MCTS algorithm guided by a neural network

$\neq$ pure play-out based MCTS

Input

A neural network predicting a policy $\boldsymbol{p} \in \Delta(\mathcal{A})$ and a value $v \in \mathbb{R}$ from the current state s : $(\boldsymbol{p}, v) = f_\theta(s)$.

The MCTS algorithm maintains for each visited state/action the counts and cumulated values + a vector of prior action probabilities :

$$\{N(s,a), S(s,a), P(s,a)\}$$

Selection step : in some state s, choose the next action to be

$$\underset{a \in \mathcal{C}(s)}{\operatorname{argmax}} \left[\frac{S(s,a)}{N(s,a)} + c \times P(s,a) \frac{\sqrt{N(s)}}{1+N(s,a)} \right]$$

for some (fine-tuned) constant c.

Alpha Zero

AlphaZero learns a good policy by using a MCTS algorithm guided by a neural network

$\neq$ pure play-out based MCTS

Input

A neural network predicting a policy $\boldsymbol{p} \in \Delta(\mathcal{A})$ and a value $v \in \mathbb{R}$ from the current state s : $(\boldsymbol{p}, v) = f_\theta(s)$.

The MCTS algorithm maintains for each visited state/action the counts and cumulated values + a vector of prior action probabilities :

$$\{N(s,a), S(s,a), P(s,a)\}$$

Expansion step : once a leaf s_L is reached, compute $(\boldsymbol{p}, v) = f_\theta(s_L)$.

- Set v to be the value of the leaf
- For all possible next actions b :
  - ➜ initialize the count $N(s_L, b) = 0$
  - ➜ initialize the prior probability $P(s_L, b) = \boldsymbol{p}_b$ (possibly add some noise)

Alpha Zero

AlphaZero learns a good policy by using a MCTS algorithm guided by a neural network

$\neq$ pure play-out based MCTS

Input

A neural network predicting a policy $\boldsymbol{p} \in \Delta(\mathcal{A})$ and a value $v \in \mathbb{R}$ from the current state s : $(\boldsymbol{p}, v) = f_\theta(s)$.

The MCTS algorithm maintains for each visited state/action the counts and cumulated values + a vector of prior action probabilities :

$$\{N(s,a), S(s,a), P(s,a)\}$$

Back-up step : for all ancestor s_t, a_t in the trajectory that end in leaf s_L,

$$\begin{aligned} N(s_t,a_t) &\leftarrow N(s_t,a_t) + 1 \\ S(s_t,a_t) &\leftarrow S(s_t,a_t) + v \end{aligned}$$

Alpha Zero

AlphaZero learns a good policy by using a MCTS algorithm guided by a neural network

$\neq$ pure play-out based MCTS

Input

A neural network predicting a policy $\boldsymbol{p} \in \Delta(\mathcal{A})$ and a value $v \in \mathbb{R}$ from the current state s : $(\boldsymbol{p}, v) = f_\theta(s)$.

The MCTS algorithm maintains for each visited state/action the counts and cumulated values + a vector of prior action probabilities :

$$\{N(s,a), S(s,a), P(s,a)\}$$

Output of the planning algorithm ? select an action a at random according to

$$\pi(a) = \frac{N(s_0,a)^{1/\tau}}{\sum_b N(s_0,b)^{1/\tau}}$$

for some (fine-tuned) temperature τ.

Training the neural network

- In AlphaGo, f_θ was trained on a database of games played by human
- In AlphaZero, the network is trained using only self-play

[Silver et al., 2016, Silver et al., 2017]

Let θ be the current parameter of the network $(\boldsymbol{p}, v) = f_\theta(s_L)$.

1. generate N games where each player uses MCTS(θ) to select the next action a_t (and output a probability over actions π_t)

$$\mathcal{D} = \bigcup_{i=1}^{\text{Nb games}} \left\{ (s_t, \pi_t, \pm r_{T_i}) \right\}_{t=1}^{T_i}$$

   T_i : length of game i, $r_{T_i} \in \{-1, 0, 1\}$ outcome of game i for one player

2. Based on a sub-sample of $\mathcal{D}$, train the neural network using stochastic gradient descent on the loss function

$$L(s, \boldsymbol{\pi}, z; \boldsymbol{p}, v) = (z - v)^2 - \boldsymbol{\pi}^\top \log(\boldsymbol{p}) + c\|\theta\|^2$$

A nice actor-critic architecture

AlphaZero alternates between

- **The actor** : MCTS(θ)
  generates trajectories guided by the network f_θ but still exploring

➜ act as a policy improvement
($N = 25000$ games played, in which the choice of each move uses MCTS with 1600 simulations)

- **The critic** : neural network f_θ
  updates θ based on trajectories followed by the critic

➜ evaluate the actor's policy

Monte-Carlo Tree-Search

- Practise : Alpha Zero (and other variants) led to big successes for solving games (and other more sophisticated tasks later)
- ➜ requires a lot of enginnering to make it work, e.g. in choosing the form of the MCTS algorithm used

- Theory :
  - ➜ very little theory, especially with Neural Network as the function approximator [Shah et al., 2022]
  - ➜ some attempts to use Best Arm Identification Tools for MCTS in simplified games [Kaufmann and Koolen, 2017]


Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012).

A survey of monte carlo tree search methods.

IEEE Transactions on Computational Intelligence and AI in games,, 4(1) :1–49.


Kaufmann, E. and Koolen, W. M. (2017).

Monte-Carlo tree search by best arm identification.

In *Advances in Neural Information Processing Systems (NeurIPS)*.


Kearns, M. J., Mansour, Y., and Ng, A. Y. (2002).

A sparse sampling algorithm for near-optimal planning in large markov decision processes.

Machine Learning, 49(2-3) :193–208.


Kocsis, L. and Szepesvári, C. (2006).

Bandit based monte-carlo planning.

In *Proceedings of the 17th European Conference on Machine Learning*.


Shah, D., Xie, Q., and Xu, Z. (2022).

Nonasymptotic analysis of monte carlo tree search.

Operation Research, 70(6) :3234–3260.


Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016).

Mastering the game of go with deep neural networks and tree search.

Nature, 529 :484–489.


Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017).

Mastering the game of go without human knowledge.

Nature, 550 :354–.