Artificial Intelligence 4 Games

Monte Carlo Tree Search - Basics

2020

Jakub Kowalski, University of Wrocław

Monte Carlo Tree Search: Basics

Monte Carlo and Bandits

Literature

- ★ B. Abramson, B.: Expected-Outcome: A General Model of Static Evaluation. IEEE Trans. Pattern Anal. Mach. Intell., vol. 12, pp. 182 – 193, 1990.
- ★ Sheppard, B.: World-championship-caliber Scrabble. Artif. Intell., vol. 134, pp. 241–275, 2002.
- ★ Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time Analysis of the Multiarmed Bandit Problem. Mach. Learn., vol. 47, no. 2, pp. 235–256, 2002.

Monte Carlo Methods

A class of algorithms that rely on a usually very large number of repeated random-based sampling.

Key point is to avoid expensive (or even computationally impossible) calculations by approximating them with much easier to compute simplifications. As law of large numbers applies here, we can improve the quality of the approximation by increasing the number of performed runs.

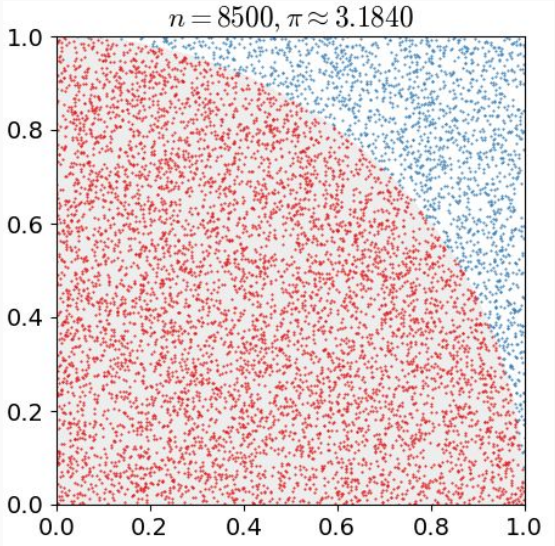
Monte Carlo (MC) methods are very popular in multiple domains, including:

- ➔ Physical sciences
- ➔ Engineering
- ➔ Computational biology
- ➔ Computer graphics
- ➔ ...
- ➔ AI for games

Monte Carlo Methods

Trivial example

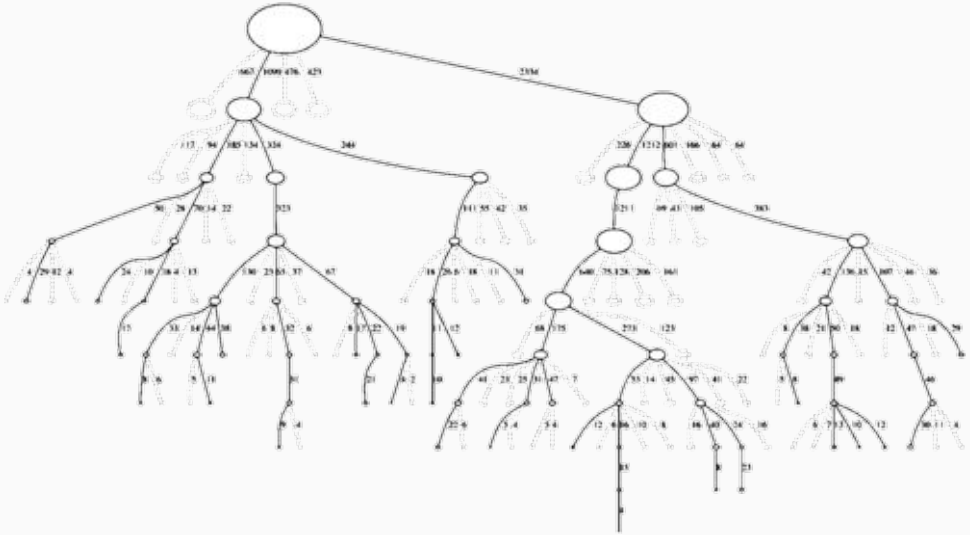
Calculating the value of π by counting how many points in 1×1 square fits within a quadrant.



https://en.wikipedia.org/wiki/Monte_Carlo_method

Weird example

Estimate the game outcome by calculating results of randomly simulating its plays.



https://www.cs.swarthmore.edu/~meeden/cs63/s19/reading/mcts.html

Monte Carlo Evaluation

Surprisingly, Monte Carlo method can be applied to evaluate the quality of a move.

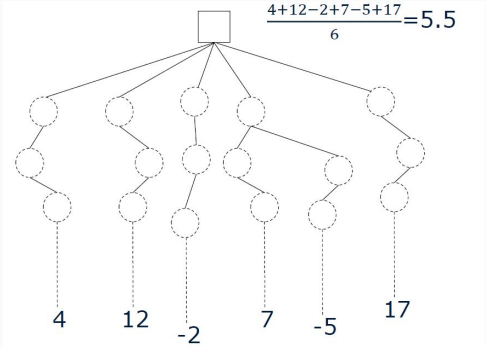
Simply, making random simulations of a game (a.k.a. playouts, rollouts) is useful approximate the game-theoretic value of a move / state.

Although containing obvious flaws because it does not allow any opponent modelling, such method was enough to achieve world champion level of play in Bridge and Scrabble.

We compute the Q-value of an action, by simply calculating the expected reward of that action:

$$Q(s,a) = W(s,a)/N(s,a)$$

Where ***N(s,a)*** is the number of times a game has been played from state ***s*** making an initial move ***a***, and ***W(s,a)*** is the sum of rewards gathered by the agent in those simulated playouts



Flat Monte Carlo (Flat MC)

Pseudocode

1. Start from the current state of the game ***s***.
2. For each legal action ***a*** in ***s*** set
   N(s,a) = 0; W(s,a) = 0
3. While there is time left:
   a. Randomize a legal action ***a***
   b. Perform a random simulation from ***s*** starting with ***a***.
   c. Let ***w*** be the obtained reward.
   d. Update state statistics:
      N(s,a) += 1; W(s,a) += w
4. Choose the action ***a*** leading to a state with the highest average ***Q(s,a)***.

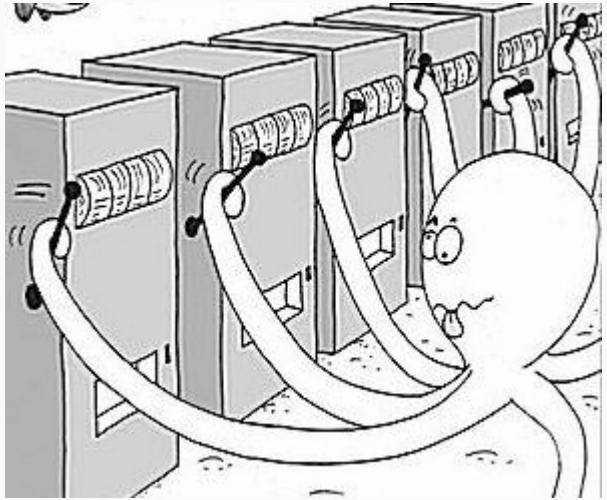
When it is useful?

★ Sanity check before implementing some more advanced algorithm
★ Flat MC is also often to estimate reasoner efficiency
★ We measure the engine speed by counting the number of simulations from the game tree root.

Multi-armed Bandit

Problem setting

- ➔ We have **K** slot machines, each associated with a reward-generating stream R_k.
- ➔ We have T rounds.
- ➔ In each round $t<T$ we need to choose a slot machine / arm $k \in \{1,\dots,K\}$.
- ➔ We receive the reward $R_k(t)$ (t-th element of the reward stream).
- ➔ Rewards of unchosen slot machines remain unknown.



Source: Microsoft Research

- ➔ We may assume that each R_k follows the probability distribution P_k, with mean μ_k.
- ➔ So the goal is to find the machine with the largest μ_k.

Regret

Our goal when approaching the multiarmed bandit problem is to minimize the *regret*.

Let ***μ**** be the maximum obtainable expected reward. There are three types of regret:

- *Instantaneous regret:* the difference between ***μ**** and the reward obtained by the last selected arm.
- *Simple regret*: after ***t*** iterations, the difference between ***μ**** and the arm believed to be the best at iteration ***t***.
- *Cumulative regret*: the sum of differences between ***μ**** and the reward obtained by the selected arm in each iteration.

We are interested in minimizing the expected cumulative regret.

A bandit algorithm is optimal if its regret is

O(log T)

Exploration-Exploitation Dilemma

Exploration

Gather more information.

Random algorithm == full exploration.

Exploitation

Best choice with known information.

Greedy algorithm == full exploitation.

The regret of both algorithms is **linear**.

The main question of the bandit problem is how to properly balance exploration and exploitation.

Source: RSM Discovery

ε-greedy

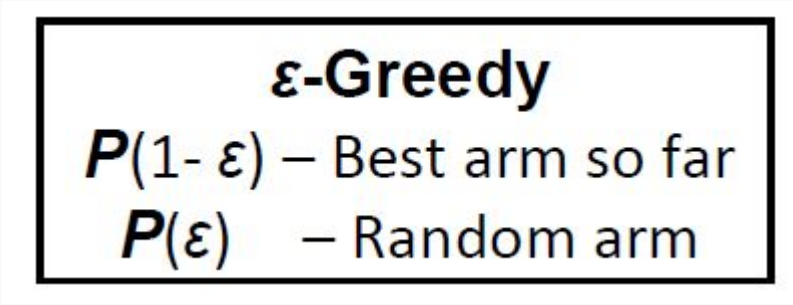
This is very simple but enough-to work action selection strategy.

From the current game state ***s***, we choose an action ***a*** for which we want to perform a simulation, thus improve its estimation ***Q(s,a)***.

- ➔ With the probability ε we uniformly choose a random legal action.
- ➔ Otherwise, we choose the action ***a*** with the highest expected payoff: ***W(s,a)/N(s,a)***

For ε>0 the search has non-zero exploration thus it (theoretically) converge to the optimum.

With proper ε, the regret bound is logarithmic.



UCB1

The most popular *upper confidence bound* (UCB) policy, which has an expected logarithmic growth of regret uniformly over ***n***. We have ***n*** being the overall number of plays so far, and $T_j(n)$ the number of times we have chosen the arm (action) ***j***, and an assumption that the rewards are from [0, 1].

The left term encourages the exploitation, while the right term is responsible for the exploration.

UCB1 (Auer et al (2002)).
Choose arm j so as to maximise:

$$\bar{X}_j + \sqrt{\frac{2 \log n}{T_j(n)}}$$

Mean so far

Upper bound on variance

Monte Carlo Tree Search

(MCTS)

Literature

- ★ Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, D., Colton, S. A survey of Monte Carlo tree search methods. TCIAIG, vol. 4, no. 1, pp. 1–43, 2012.
- ★ Kocsis, L., C. Szepesvári, C.: Bandit based Monte-Carlo Planning. Euro. Conf. Mach. Learn., pp. 282–293, 2006.

Monte Carlo Tree Search (MCTS)

So far, we were thinking only about the choice of the best action from a single game state in a “flat” manner. However, we may treat every game state as a multi-armed bandit problem, so they are “recursively” connected.

This is what MCTS is about. The algorithm progressively builds a partial game tree (called *MCTS tree*), based on the so-far results of its exploration, and with each iteration tries to improve its estimation about the values of each nodes / actions.

Our goal is to learn policy via estimating ***Q(s,a)***, i.e. a value of a game state ***s*** when performing action ***a***. Which is an expected reward when applying ***a*** from ***s*** - an average based on our previous trials.

In an MCTS Tree node ***v*** (associated with a game state ***s***) it is enough to remember:

- ➔ ***N(v)***, number of visits in node ***v***.
- ➔ ***W(v)***, the total simulation reward for games played through ***v***.

For a convenient tree traversal, in each node we usually remember a pointer to the parent and list of children with associated action labels.

MCTS pseudocode

```
function MCTSSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0))
```

How to choose final action?

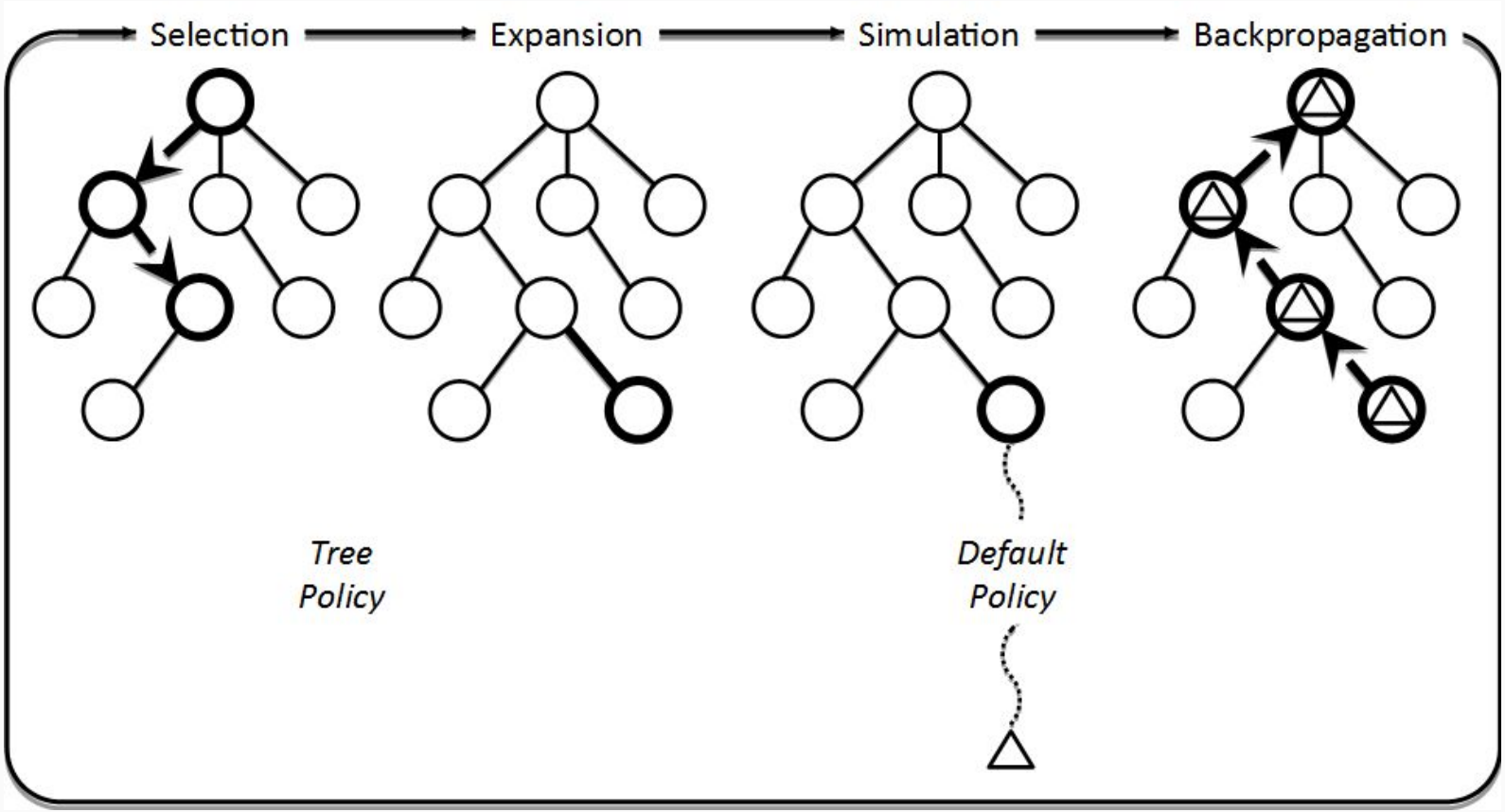
- ★ Largest number of visits
- ★ Highest expected reward
- ★ Use the same method as in the tree policy

The algorithm runs in a loop, iteratively building a search tree until some predefined computational budget (usually time / number of simulations) is reached.

An MCTS iteration consists of four phases:

- ➔ *Selection*: descending through the MCTS tree following the chosen path of actions
- ➔ *Expansion*: adding new node to the tree
- ➔ *Simulation*: running a simulated game from the newly added node
- ➔ *Backpropagation*: updating statistics of each node in MCTS tree along the way based on the result of the simulation

Single MCTS iteration



Selection Phase

```
function TreePolicy(v)
    while v is nonterminal do
        if v not fully expanded then
            return Expand(v)
        else
            v ← BestChild(v, Cp)
    return v

function BestChild(v, c)
    return  arg max_{v' ∈ children of v}  Q(v')/N(v') + c sqrt(2 ln N(v) / N(v'))
```

Until we find the node that needs expansion we descends the tree determining which action (state/node) to choose next.

This is called a **selection strategy**.

We often name the algorithm **UCT** (Upper Confidence bounds applied to Trees) if the selection strategy is UCB-based. Notice the ***c*** constant for balancing exploration-exploitation (larger ***c*** leads to more explorative behaviour).

Also note that (usually) the node to expand does not have to be the leaf of the tree. We expand the first encountered node containing some untried action.

Expansion Phase

function EXPAND(v)
choose $a \in$ untried actions from $A(s(v))$
add a new child v' to v
with $s(v') = f(s(v), a)$
and $a(v') = a$
return v'

Expansion simply adds new node to the tree: with initializing nodes values, proper pointer to the parent, etc.

In practice, to simplify and speed up this random selection of untried actions, I tend to initialize a node with a shuffled array containing legal actions. Then, it is enough to remember a pointer of the last expanded action. If it is equal to the array's length, the node is fully expanded. Otherwise, we use the action it is pointing at and increment it.

Simulation Phase

```
function DefaultPolicy(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

Each time we add a new node to the tree, we use a default policy to simulate the gameplay until it reaches a terminal state.

What we are interested in are the players' rewards for this state. (Thus we do not need an evaluation function. But...)

"Default" default policy is to choose every move uniformly at random.

For some well-known game we may improve this part by using smarter policy that reflects the real gameplay better.

Backpropagation Phase

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

Starting with the node newly added to the MCTS tree, we update statistics of every node along the way to the root.

Some application remarks

Reusing tree

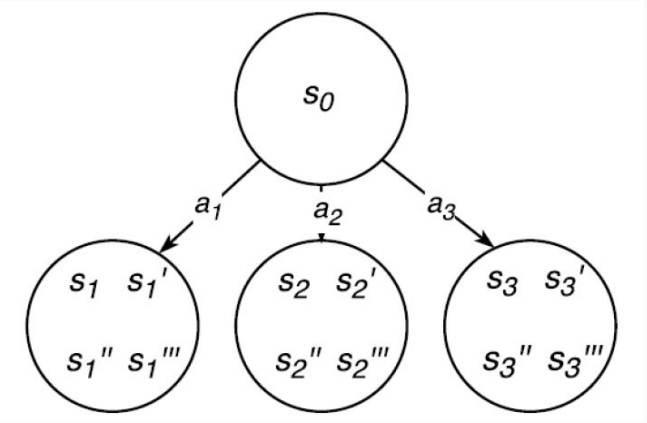
After a turn, we simply make a proper child node a new root, unchain it from the rest of the tree to free the space, and continue search downwards.

Sometimes it may be beneficial to decay the values of the previous-turn tree by some $\gamma \in [0,1]$ to increase the impact of newly gathered knowledge.

When a significant game change occurs (e.g. a new level, or some global game reverse) it is better to rebuild the tree from the scratch.

Open Loop approach

We can deal with nondeterminism “for free”. If the game state is not stored within a tree (which is a usual approach), but we only apply action each time, a node can represent a belief state.



Source: Mark Winands

And that’s MCTS

A simple layout that leads to a very versatile and customizable algorithm that causes **revolution in playing Computer Go**, but it was also successfully applied to, e.g.:

- ★ General Game Playing,
- ★ General Video Game Playing,
- ★ Two-player turn-based games (Hex, Amazons, Lines of Action, ...)
- ★ Puzzles (SameGame, Sudoku, …)
- ★ Real life problems (hospital planning, smart grids, space exploration (that’s not a real life problem))
- ★ Multiplayer board games (Chinese Checkers, Blokus, Focus)
- ★ Stochastic / imperfect information games (MtG, Chinese Dark Chess, Hearthstone)
- ★ Real-time games (Ms Pac-Man)
- ★ Video games (StarCraft, Total War: Rome, Fable Legends)

Summary

Summary: MCTS Pros and Cons

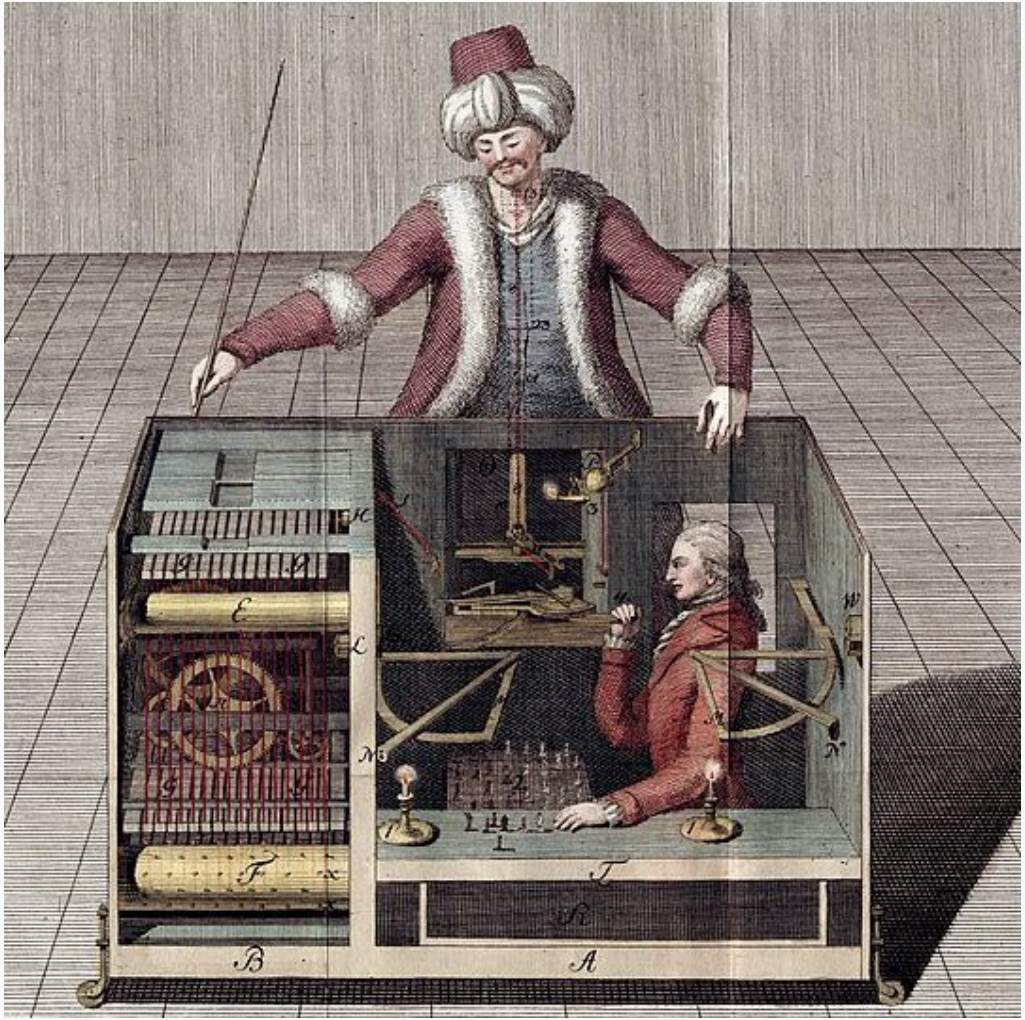
Pros

- ★ Anytime
- ★ Knowledge-free
- ★ Asymmetric
- ★ Converges to Minimax
- ★ Works well for imperfect information, nondeterminism, multiple players, simultaneous moves
- ★ Many well-established enhancements

Cons

- ❖ Requires forward model (game reasoner), which may be bothersome and error-prone
- ❖ Requires significant efficiency to start working
- ❖ Still, sometimes needs heuristic evaluation
- ❖ Rarely works off-the-shelf (usually requires tuning or even larger adaptation)

Thanks!



Bonus reference quiz