Sequential Decision Making and Reinforcement Learning

(SDMRL)

MCTS

Sarah Keren

The Taub Faculty of Computer Science
Technion - Israel Institute of Technology

Monte-Carlo Tree Search

Monte Carlo Tree Search

Not l

Article Talk Read Edit View hi

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate
Contribute

Monte Carlo tree search

From Wikipedia, the free encyclopedia

In computer science, **Monte Carlo tree search** (**MCTS**) is a heuristic search algorithm for some kinds of decision processes, most notably those employed in software that plays board games. In that context MCTS is used to solve the game tree.

MCTS was combined with neural networks in 2016 for computer Go.[1] It has been used in other board games like chess and shogi,[2] games with incomplete information such as bridge[3] and poker,[4] as well as in turn-based-strategy video games (such as Total War: Rome II's implementation in the high level campaign AI[5]). MCTS has also been used in self-driving cars, for example in Tesla's Autopilot software.[6]

actice GeeksforGeeks

paration Topic-wise Practice C++ Java Python Competitive Programming Machine Learning HTML

Difficulty Level : Hard • Last Updated : 19 Jan, 2022

Monte Carlo Tree Search (MCTS) is a search technique in the field of Artificial Intelligence (AI). It is a probabilistic and heuristic driven search algorithm that combines the classic tree search implementations alongside machine learning principles of reinforcement learning.
In tree search, there's always the possibility that the current best action is actually not the most optimal action. In such cases, MCTS algorithm becomes useful as it continues to evaluate other alternatives periodically during the learning phase by executing them, instead of the current perceived optimal strategy. This is known as the " ***exploration-exploitation trade-off*** ". It exploits the actions and strategies that is found to be the best till now but also must continue to explore the local space of alternative decisions and find out if they could replace the current best.

towardsdatascience.com/monte-carlo-tree-search-158a917a8baa

SAGAR SHARMA
Aug 1, 2018 · 6 min read · Listen

Monte Carlo Tree Search

MCTS For Every Data Science Enthusiast

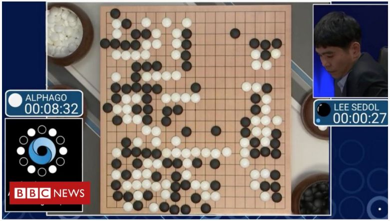
The Games like Tic-Tac-Toe, Rubik's Cube, Sudoku, Chess, Go and many others have common property that lead to exponential increase in the number of possible actions that can be played. These possible steps increase exponentially as the game goes forward. Ideally if you can predict every possible move and its result that may occur in the future. You can increase your chance of winning.

But since the moves increase exponentially — the computation power that is required to calculate the moves also goes through the roof.

Monte Carlo Tree Search is a method usually used in games to predict the path (moves) that should be taken by the policy to reach the final winning solution.

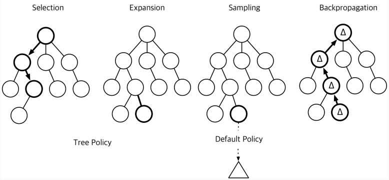
Monte Carlo Tree Search

- Combines exploration and exploitation to choose an action in a tree search setting
- Is relevant to classical planning, stochastic planning, and game playing
  - Revolutionized the world of computer Go
- Has many different variants
- Explosion in interest, applications far beyond games: Planning, motion planning, optimization, finance, energy management



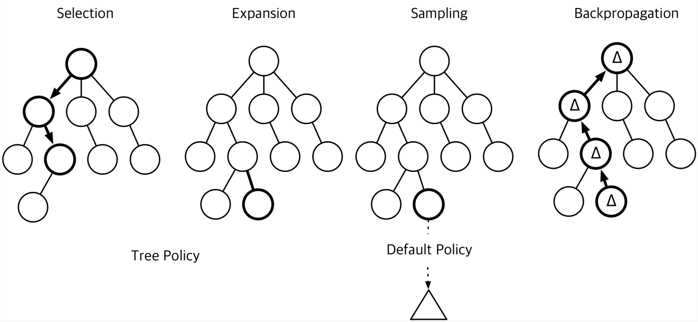
Key Idea: use **Monte Carlo simulation** to accumulate value estimates to guide towards highly rewarding trajectories in a **search tree**.

- Each simulation consists of two phases
  - **Tree policy:** pick actions to maximise values
  - **Default / roll-out policy:** pick actions randomly to simulate a trajectory.
- Repeat (each simulation)
  - **Evaluate** states $Q(S, A)$ by Monte-Carlo evaluation
  - **Improve** tree policy e.g. by ϵ-greedy.
- Converges on the optimal search tree $Q(S, A) \rightarrow q(S, A)$



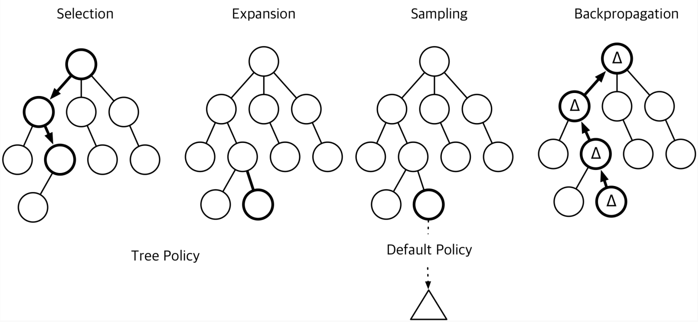
Monte Carlo Tree Search

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

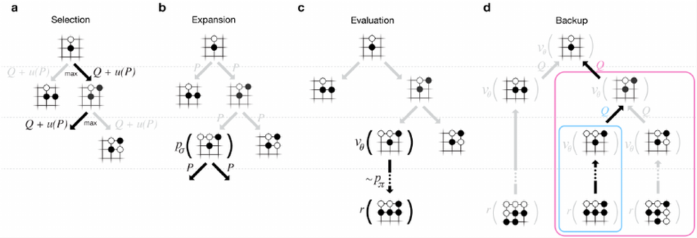


Monte Carlo Tree Search

- Selection
- Expansion
- Sampling / Simulation
- Back-propogation



Example: MCTS for GO



MCTS is not just for games

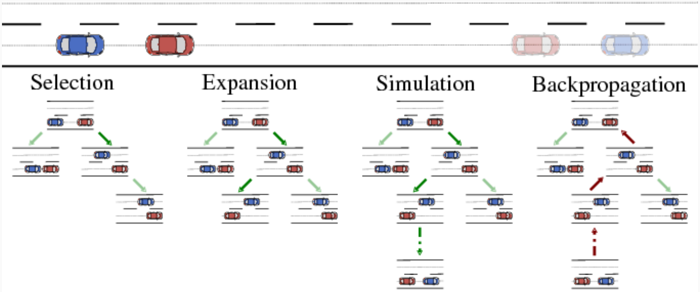


Fig. 1: Phases of Monte Carlo Tree Search for an overtaking maneuver: the

https://www.semanticscholar.org/paper/Decentralized-Cooperative-Planning-for-Automated-Kurzer-Zhou/585b73322365ba2d0afa7449691c81cb98777599

Selection Policy

- Use results of simulations to guide growth of the game tree
  - Exploitation: focus on promising moves
  - Exploration: focus on moves where uncertainty about evaluation is high
- Seems like two contradictory goals
  - Theory of bandits can help

Selection Policy: Multi-Armed Bandit Problem

- We can choose among several arms
- Each arm pull is independent of other pulls
- Each arm has fixed, unknown average payoff
- Which arm has the best average payoff?

Selection Policy: UCB1 [Auer et al 02]

- First, try each arm once
- Each arm pull is independent of other pulls
- Then, at each time step:
- Choose arm i that maximizes the UCB1 formula for the upper confidence

$$v_i + C \times \sqrt{\frac{ln(N)}{n_i}}$$

where

- v_i - current estimation of the value of bandit i
- C - tunable parmater to balance exploration / exploitation
- N total number of trials
- n_i - no. of trials for bandit

How is this relevant to search trees ?

Selection Policy: UCT (UCB applied to trees)

- UCB makes single decision
- What about sequences of decisions (e.g. planning games?)

Selection Policy: UCT (UCB applied to trees)

- UCB makes single decision
- What about sequences of decisions (e.g. planning games?)
- Answer: use a look-ahead tree
  - Bandit arm ≈ move in a game
  - Payoff ≈ quality of move
  - Regret ≈ difference to best move
- Apply UCB-like formula for node selection
  - Choose "optimistically" where to expand next

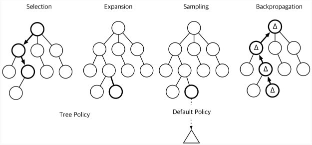
$$UCB(s) = \frac{U(s)}{N(s)} + C \times \sqrt{\frac{ln(N(s.parent))}{N(s)}}$$

where

- $U(s)$ - total utility of all rollouts that went through s
- N(s) - number of rollouts though s.

Simulation / Roll-out Policy

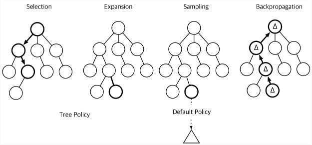
- Default roll-out policy is to make uniform random moves
- Goal is to find strong correlations between initial position and result of a simulation
- Domain independent techniques for games : Ideas ?



From Sarit Kraus: *https://u.cs.biu.ac.il/~krauss/advai2018/MCTS.pdf*

Simulation / Roll-out Policy

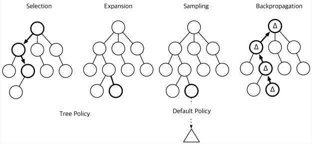
- Default roll-out policy is to make uniform random moves
- Goal is to find strong correlations between initial position and result of a simulation
- Domain independent techniques for games : Ideas ?
    - If there is an immediate win, take it
    - Avoid immediate losses
    - Avoid moves that give opponent immediate win
    - Last Good Reply
    - Using prior knowledge



From Sarit Kraus: *https://u.cs.biu.ac.il/~krauss/advai2018/MCTS.pdf*

Simulation / Roll-out Policy

- Last Good Reply (Drake 2009), Last Good Reply with Forgetting (Baier et al 2010)
- Machine-learned pattern values (Silver 2009)
- Simulation balancing (Silver and Tesauro 2009)
- Using prior knowledge



From Sarit Kraus: *`https://u.cs.biu.ac.il/~krauss/advai2018/MCTS.pdf`*

MCTS efficiency

MCTS efficiency

- The time to perform a rollout is linear in the depth of the tree
- This gives plenty of time to consider multiple rollouts
- For example
  - if:
    - Branching factor b=32
    - Average game length (tree depth) is d=100
    - We can compute 10^9 moves
  - Minimax can search 6 ply deep
  - AlphaBeta can search up to 12 ply deep
  - MCTS can do 10^7 rollouts
- Works great for games with
  - Large branching factor (then minimax can't search deep enough)
  - Poor evaluation function

MCTS for MDPs

A **Markov Decision Process**(MDP) is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where

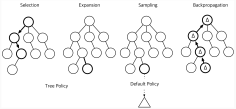
- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{P}$ is a state transition probability matrix
  $\mathcal{P}^a_{s,s'} = \mathcal{P}[S_{t+1} = s' | S_t = s, A_t = a]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$, and
- **optional:** γ is a discount factor

How to perform ?

- Selection
- Expansion
- Sampling / Simulation
- Back-propogation

MCTS: Summery

- UCB, UCT are very important algorithms in both theory and practice with well-founded convergence guarantees under relatively weak conditions
- Applicable to a variety of games and other applications, as it is domain independent
- Basis for extremely successful programs for games and many other applications
- Very general algorithm for decision making
  - Works with very little domain-specific knowledge
  - (But) needs a simulator of the domain
  - Can take advantage of knowledge when present
  - Anytime algorithm - can stop the algorithm and provide answer immediately, though improves answer with more time



Recap and what next

- Spectrum of approaches to planning with
  - deterministic and stochastic actions
  - full observability
  - next: What about partial observability ?

Do we still need to know about planning?

by Subbarao Kambhampati:
"Human, grant me the serenity to accept the things I cannot learn, the data to learn the things I can, and the wisdom to know the difference."[1]

[1] My addition: and the ability to know what to do about it