

What every game developer should know about

Mathematical Optimization (Part 3)

Intelligent Computational Media

Aalto University

Prof. Perttu Hämäläinen

perttu.hamalainen@aalto.fi

Contents: Alternatives to Deep RL in optimizing action sequences

- Forward search methods
- Neuroevolution

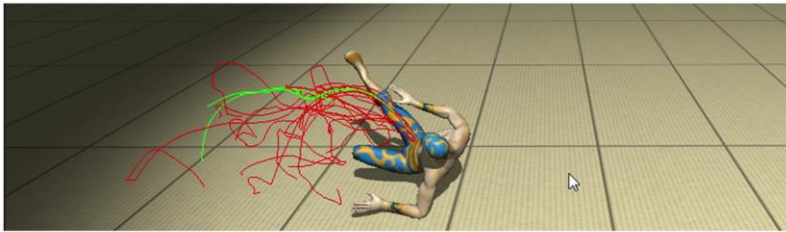


Let's start with a video.

Live demo

Controlling an agent through forward search

1. Simulate multiple action strategies up to a planning horizon or termination. Run simulations in parallel threads if possible
2. Step the "master simulation" forward using the best strategy
3. Repeat from the resulting state, possibly informing the forward simulations based on previous results. Planning horizon is shifted one time step forward (rolling horizon, receding horizon)



In this demo, there's no learning or training of neural networks. One starts the software and it just works.

This is because it's using forward search instead.

Deep RL vs. Forward Search

Deep RL:

- + Once trained, very computationally efficient
- Training can be very slow and may not converge

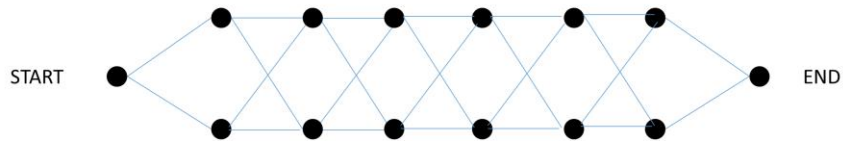
Search methods:

- + Get good results fast, without training neural networks
- Requires forward simulation (many times faster than real-time simulation)
- Requires capability to save and load simulation state

Combining search & neural networks can allow adjusting the tradeoffs.

Why is searching for action sequences hard?

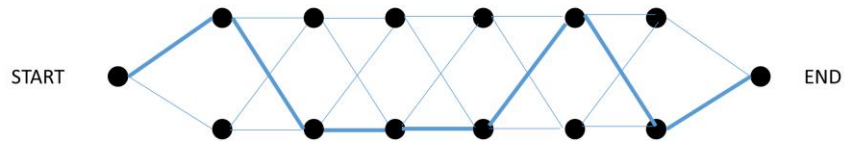
Curse of dimensionality



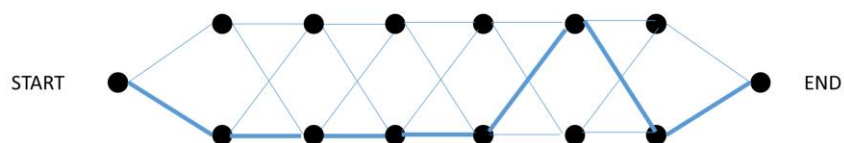
Consider finding the sequence of actions (moves) that take us from START to END along the shortest possible path.

Even in this simple graph, there are surprisingly many paths from start to end

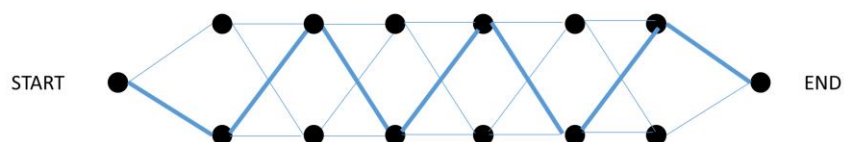
Curse of dimensionality



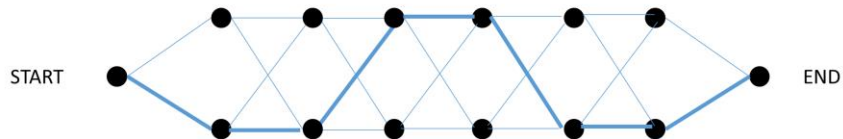
Curse of dimensionality



Curse of dimensionality



Curse of dimensionality



In fact, the number of possible paths grows exponentially with the number of vertices

Dynamic programming: Dijkstra's method



Clearly, brute-force checking all possible paths is not possible. Perhaps the most well-known algorithm, and one taught in CS 101 is Dijkstra's algorithm.

Green is the target, red is the starting point. At each iteration, the neighbors (blue circles) of the already visited nodes (colored) are considered, and the closest one to the starting point is visited. When visiting, the total distance from the starting point is stored in the node, here visualized as the color gradient. It is computed recursively as the sum of the visited node's distance from the closest node C in the already visited set, and the distance stored in C.

Although Dijkstra is slow, it's immensely faster than iterating over all possible paths. The number of paths grows exponentially with path length, whereas Dijkstra's worst case complexity is only $O(N^2)$, where N is the number of nodes. This is due to the use of the dynamic programming principle, i.e., breaking the problem down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. In Dijkstra's case, the solutions are the shortest path lengths from the nodes in the visited set – a new added node's shortest path is then updated simply by adding its distance to the stored value.

Animation by Subh83 - Own work, CC BY 3.0,
<https://commons.wikimedia.org/w/index.php?curid=14916903>

From Dijkstra to A*

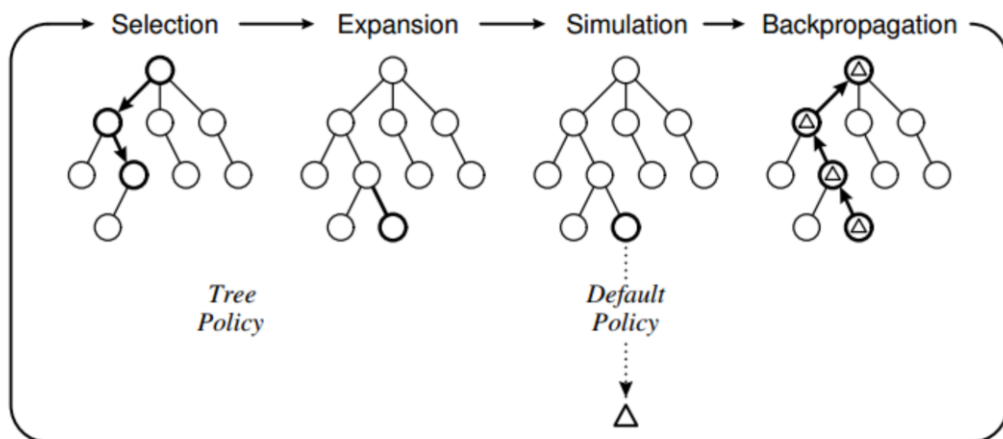


Instead of visiting the neighbor with shortest distance to the start node, one selects neighbor K with the shortest estimated whole path length between start and end. This is a sum of the realized distance from start to K and a heuristic distance from K to end, for example, Euclidian distance.

Why is it still hard?

- In 2d or 3d, this is trivial
- Unity's NavMesh implements pathfinding, no need to code yourself
- However, the search process becomes very slow with dozens or hundreds of possible actions
- A* heuristics hard to design for problems beyond pathfinding

Monte Carlo Tree Search to the rescue



MCTS is a common technique for adaptively exploring the tree of possible actions, without visiting every possible node. It also handles both single and multiplayer games.



<https://www.youtube.com/watch?v=HRIEUUC9TUA>

REPORT

A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play

David Silver^{1,2,*}, Thomas Hubert^{1,*}, Julian Schrittwieser^{1,*}, Ioannis Antonoglou¹, Matthew Lai¹, Arthur Guez¹, Marc Lanctot¹

* See all authors and affiliations

Science 07 Dec 2018:
Vol. 362, Issue 6419, pp. 1140-1144
DOI: 10.1126/science.aar6404

[Article](#)[Figures & Data](#)[Info & Metrics](#)[eLetters](#)[PDF](#)

One program to rule them all

Computers can beat humans at increasingly complex games, including chess and Go. However, these programs are typically constructed for a particular game, exploiting its properties, such as the symmetries of the board on which it is played. Silver *et al.* developed a program called AlphaZero, which taught itself to play Go, chess, and shogi (a Japanese version of chess) (see the Editorial, and the Perspective by Campbell). AlphaZero managed to beat state-of-the-art programs specializing in these three games. The ability of AlphaZero to adapt to various game rules is a notable step toward achieving a general game-playing system.

Science, this issue p. 1140; see also pp. 1087 and 1118

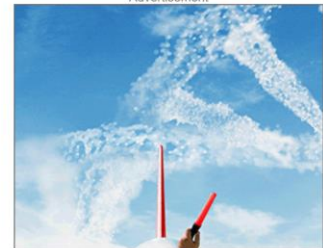
**Science**

Vol 362, Issue 6419
07 December 2018

[Table of Contents](#)
[Print Table of Contents](#)
[Advertising \(PDF\)](#)
[Classified \(PDF\)](#)
[Masthead \(PDF\)](#)

ARTICLE TOOLS[Email](#)[Print](#)[Alerts](#)[Citation tools](#)[Download Powerpoint](#)[Save to my folders](#)[Request Permissions](#)[Share](#)

Advertisement



A combination of MCTS and RL is also what powers AlphaZero, Google Deepmind's general game-playing AI that is the current master of Go and Chess, to name a few.

A Survey of Monte Carlo Tree Search Methods

Cameron Browne, *Member, IEEE*, Edward Powley, *Member, IEEE*, Daniel Whitehouse, *Member, IEEE*, Simon Lucas, *Senior Member, IEEE*, Peter I. Cowling, *Member, IEEE*, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton

Abstract—Monte Carlo Tree Search (MCTS) is a recently proposed search method that combines the precision of tree search with the generality of random sampling. It has received considerable interest due to its spectacular success in the difficult problem of computer Go, but has also proved beneficial in a range of other domains. This paper is a survey of the literature to date, intended to provide a snapshot of the state of the art after the first five years of MCTS research. We outline the core algorithm's derivation, impart some structure on the many variations and enhancements that have been proposed, and summarise the results from the key game and non-game domains to which MCTS methods have been applied. A number of open research questions indicate that the field is ripe for future work.

Index Terms—Monte Carlo Tree Search (MCTS), Upper Confidence Bounds (UCB), Upper Confidence Bounds for Trees (UCT), Bandit-based methods, Artificial Intelligence (AI), Game search, Computer Go.

1 INTRODUCTION

MONTE Carlo Tree Search (MCTS) is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results. It has already had a profound impact on Artificial Intelligence (AI) approaches for domains that can be represented as trees of sequential decisions, particularly games and planning problems.

In the five years since MCTS was first described, it has become the focus of much AI research. Spurred on by some prolific achievements in the challenging task of computer Go, researchers are now in the pro-

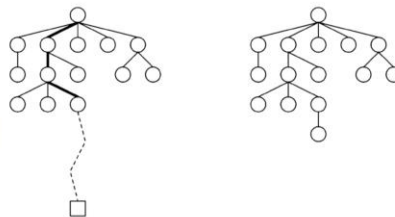


Fig. 1. The basic MCTS process [17].

<http://mcts.ai/pubs/mcts-survey-master.pdf>

A useful article explaining MCTS and its variants

Monte Carlo Tree Search

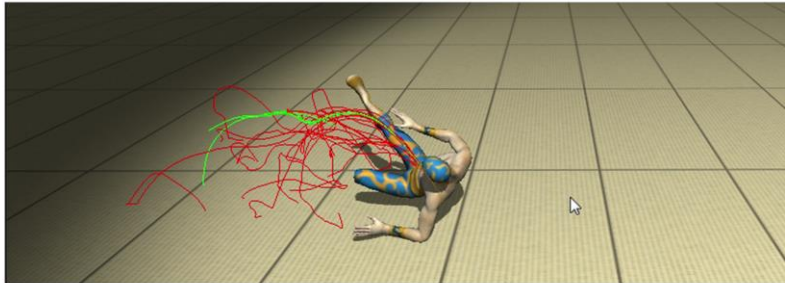
The best visualizations I could find:

<https://int8.io/monte-carlo-tree-search-beginners-guide/>

What about continuous actions?

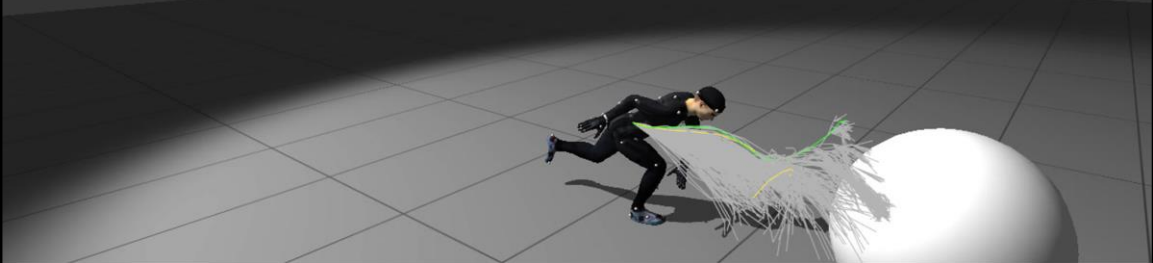
Our SIGGRAPH 2014 work: no tree search

- Simply run a number of rollouts from the current state
- Search space reduced by encoding action sequences as pose splines
- Take action following the best rollout
- The distribution of rollout actions is adapted



Control Particle Belief Propagation (C-PBP)

Hämäläinen et al., SIGGRAPH 2015



$$\mathcal{P}(\mathbf{z}) \propto \left(\prod_k \psi_k(\mathbf{z}_k) \right) \left(\prod_{k=1}^K \Psi_{\text{fwd}}(\mathbf{z}_{k-1}, \mathbf{z}_k) \right) \left(\prod_{k=0}^{K-1} \Psi_{\text{bwd}}(\mathbf{z}_{k+1}, \mathbf{z}_k) \right)$$

In this paper, we for the first time implemented a form of sampling-based or Monte Carlo Tree Search. However, the mathematical framework and reasoning was based on Belief Propagation. In hindsight, this was overly cumbersome, although the results were certainly better

Hämäläinen, Perttu, Joose Rajamäki, and C. Karen Liu. "Online control of simulated humanoids using particle belief propagation." *ACM Transactions on Graphics (TOG)* 34.4 (2015): 81.

FDI-MCTS (Rajamäki & Hämmäläinen, 2017)

- Simple MCTS variant for fixed-horizon continuous control. Simplification of C-PBP, no cumbersome math.
- Combines deep neural networks and tree search
- Benefit: Very fast convergence, humanoid walking in less than a minute on a single computer
- Drawback: Requires forward simulation, the resulting policy not always as robust as, e.g., with PPO



This video shows a humanoid agent learning to walk from scratch. Stable locomotion emerges in less than a minute of CPU time.

Recent continuous control MCTS

- Moerland, Thomas M., et al. "A0C: Alpha zero in continuous action space." *arXiv preprint arXiv:1805.09613* (2018).

<https://arxiv.org/abs/1805.09613>

My group's current topics

- Implement and benchmark continuous control MCTS variants for 3D simulated character control
- Find the best ways to combine continuous control MCTS with RL

Predictive Physics Simulation in Game Mechanics

ACM CHI PLAY 2017

Perttu Hämäläinen (Aalto University)

Xiaoxiao Ma (Aalto University)

Jari Takatalo (Aalto University)

Julian Togelius (NYU Tandon School of Engineering)

Finally, I'd like to point out that forward search can lead to interesting game mechanics, if designed for the player instead of AI.

Contents: Alternatives to RL

- Forward search methods
- **Neuroevolution**

Neuroevolution

- Apply an evolutionary optimization method like CMA-ES directly to policy network parameters θ
- Some algorithms like NEAT optimize both network parameters and network architecture
- Optimization objective $f(\theta)$ = sum of rewards over episode

One iteration:

1. Sample parameters $\theta_1, \dots, \theta_N$ for N neural networks
2. Run 1 or more episode with each network, compute $f(\theta)$
3. Update the sampling distribution

Very similar to RL, but more simple, as one doesn't need to train any networks, compute advantages etc.! Instead, one just samples random network parameters.

Evolutionary optimization = optimization methods that mutate and recombine "populations" of individuals, inspired by biological evolution. CMA-ES is inspired by this tradition, although later on, it has been analyzed and understood from different perspectives, e.g., sampling-based gradient descent with adaptive step size.

Geijtenbeek 2013 (CMA-ES neuroevolution)

Flexible Muscle-Based Locomotion for Bipedal Creatures

SIGGRAPH ASIA 2013

**Thomas Geijtenbeek
Michiel van de Panne
Frank van der Stappen**

This paper from way before deep reinforcement learning still has the best looking humanoid locomotion results.

It combines CMA-ES neuroevolution with some problem-specific engineering like using a state machine that switches between different policy networks for different parts of the gait.

CMA-ES works well in this case, as the dedicated networks can be small, with only hundreds of parameters for CMA-ES to optimize.

LM-MA-ES can easily optimize networks with a few thousand parameters. Beyond that, PPO or PPO-CMA provide faster convergence in my experience.

Recent papers on neuroevolution

Such, Felipe Petroski, et al. "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning." *arXiv preprint arXiv:1712.06567* (2017)

Salimans, Tim, et al. "Evolution strategies as a scalable alternative to reinforcement learning." *arXiv preprint arXiv:1703.03864* (2017).

These papers propose methods for making neuroevolution efficient with large neural networks.

Optimizing action sequences: Summary

Deep RL:

- + Produces a policy network that can compute actions very efficiently
- Training can be very slow

Neuroevolution:

- + Produces a policy network that can compute actions very efficiently
- + Simple to implement: only policy network, no training
- With large policy networks, can be even slower than Deep RL and/or requires extra tricks that make implementation more complex

Forward search methods like MCTS:

- + Find good actions fast, without training or optimizing neural networks
- Requires forward simulation (many times faster than real-time simulation)
- Requires capability to save and load game or simulation state