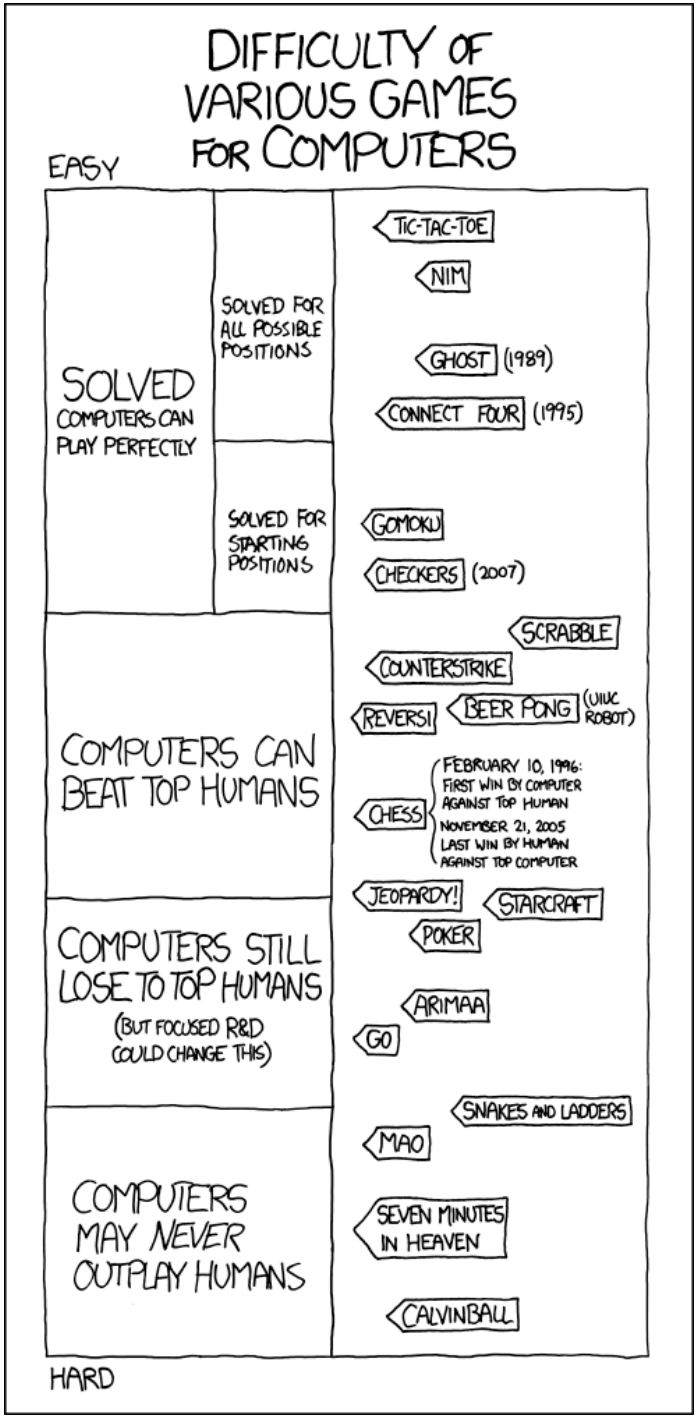
CSC384
Monte Carlo Game Tree Search
Summer 2019

Instructors: Alex Poole & Sonya Allin

Resources

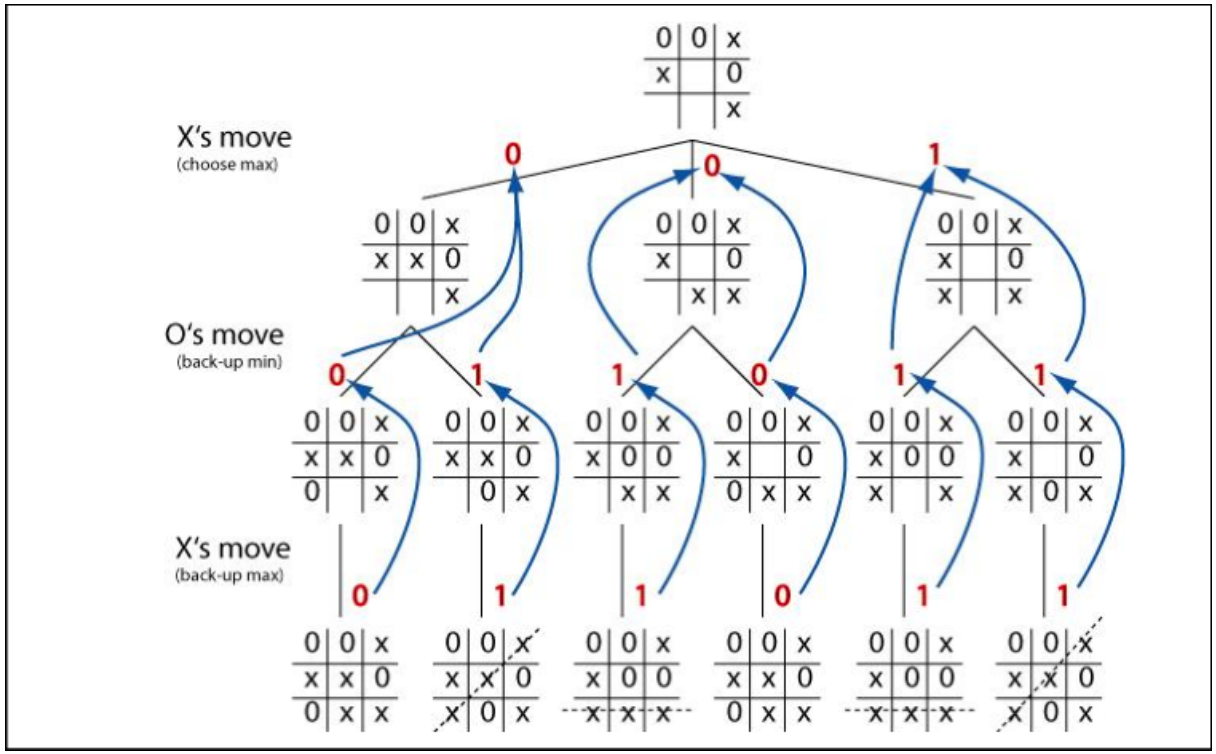
- Mastering the game of Go, David Silver et. al., Nature 2017
- AAAI-14 Games Tutorial, by Martin Muller
- International Seminar on New Issues in Artificial Intelligence, by Simon Lucas
- Introduction to Monte Carlo Tree Search, by Jeff Bradberry



Hierarchy of games (a bit outdated ...)

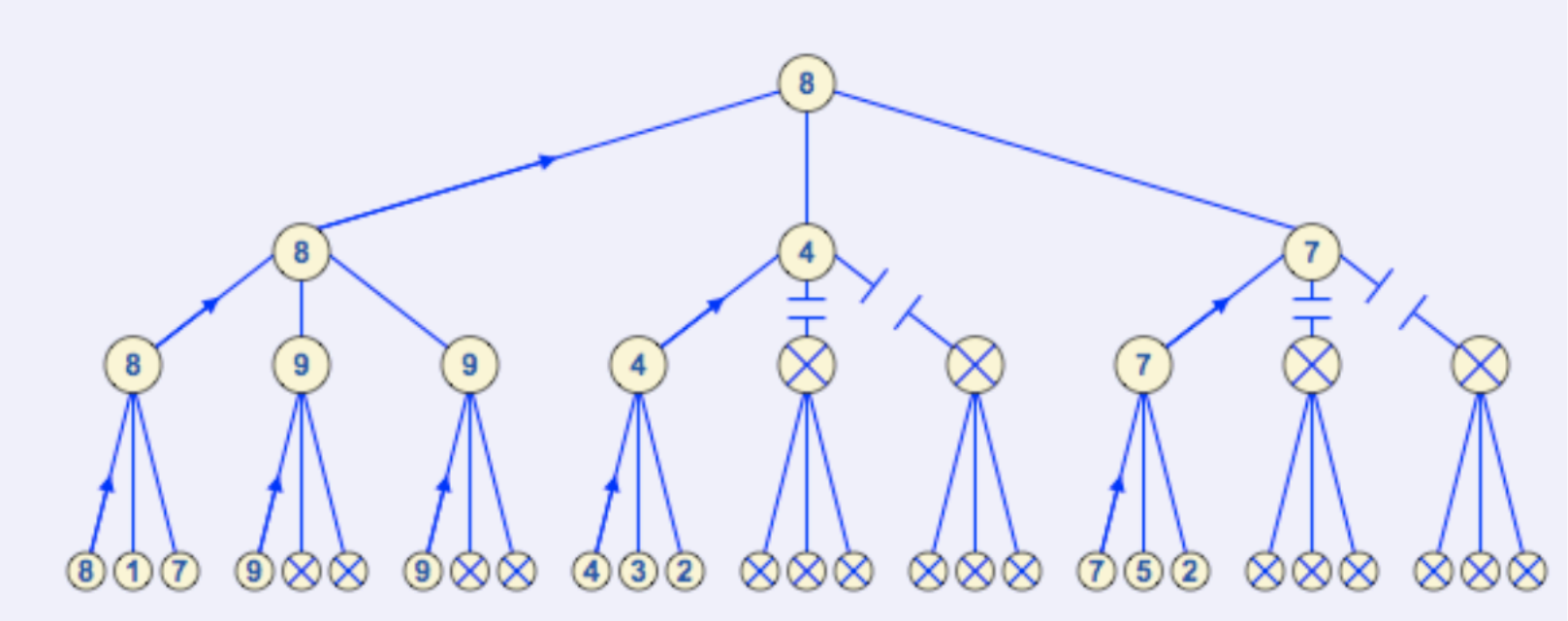
https://xkcd.com/1002/

Minimax and Alpha-Beta



- Works well for "small" games
- Optimal strategy requires traversal of entire tree

Minimax and Alpha-Beta

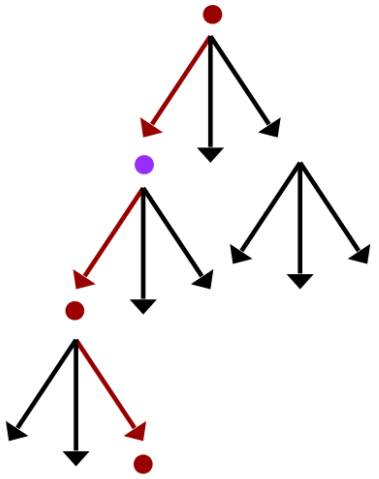


- Alpha-Beta improves performance of minimax (works for Chess, Arimaa)
  - *In best case, what is the performance of Alpha-Beta relative to Minimax?*
- Optimal policies still requires information about entire tree

Larger Games

But what about when the game tree is extremely large? Maybe we could:

- Restrict our focus to parts of the tree that look promising (like A*)
- Try evaluation functions (heuristics)
- Try sampling



Naive Idea

We could use simulations directly as an evaluation function for alpha-beta. But …

- A single simulation is very noisy, only 0/1 signal
- Running many simulations for one evaluation can be very slow
  - typical speed of chess programs: **1 million** eval/second
  - Go: 1 million moves/second, 400 moves/simulation, 100 simulations/eval = **25** eval/second

Where to search in our tree?

Instead, we can use the results of simulations to guide the growth of our game tree. This means balancing:

Exploitation: We can choose to make the best decision given current information.

Exploration: We can choose to gather more information about the tree.

A good long-term strategy may involve short-term sacrifices.

We want enough information to make good decisions.

An explore vs. exploit example

Dilemma reflected in the multi-armed bandit problem:

- We've a choice of several "one armed bandit" arms
- Each arm pull is independent of other pulls
- Each arm has a fixed, unknown average payoff

Which arm has the best average payoff?

An explore vs. exploit example

Want to **explore** all arms
BUT, if we explore too much, may sacrifice reward

Want to **exploit** promising arms more often
BUT, if we exploit too much, can get stuck with sub-optimal values

Need to balance between **exploration** and **exploitation**

An explore vs. exploit example

A **policy** is a strategy for choosing arm to play at time t, given arm selections and outcomes at times 1, …, t - 1

Examples:

- Uniform policy (play a least-played arm, break ties randomly)
- Greedy (play an arm with highest empirical payoff)
- ε-greedy (play arm with highest payoff with probability 1-ε, else play randomly)
- Decaying ε-greedy $\varepsilon_t = \varepsilon_t * (\alpha)/(t + \alpha)$

What is a good policy?

One that minimizes **regret** = losses from playing a sub-optimal arm.

Three Bandits

A

P(A wins) = 60%

B

P(B wins) = 55%

C

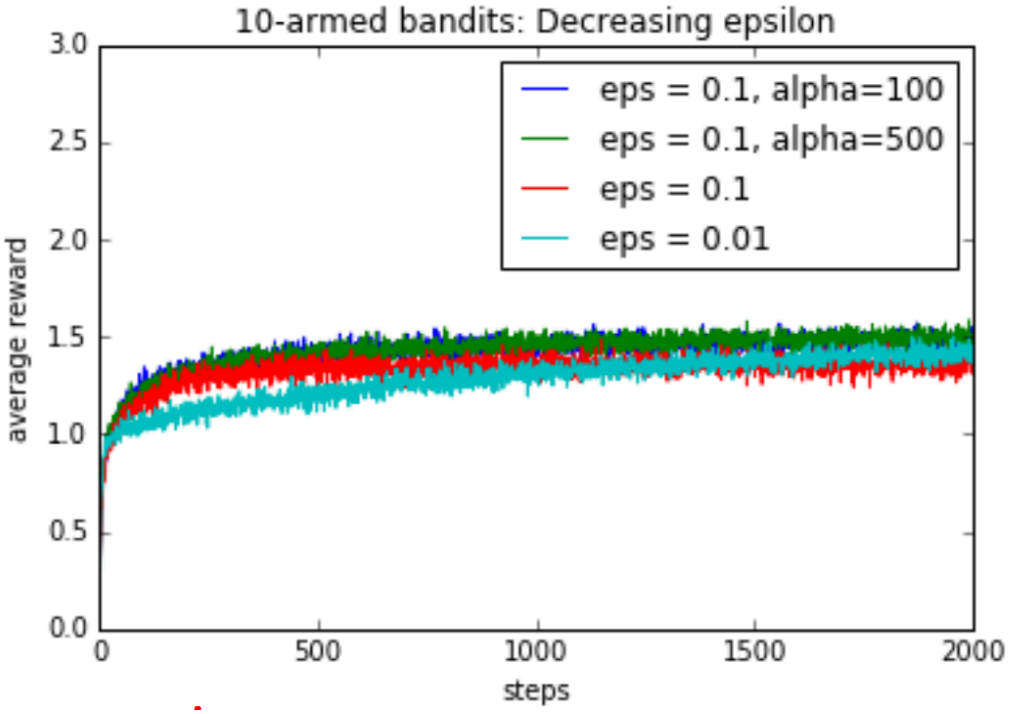
P(C wins) = 40%

Each pull of an arm is either

- A win (payoff 1) or
- A loss (payoff 0)

Machine A is the best arm … **but we don't know that a priori**

An explore vs. exploit example



10 bandits, 2000 pulls.

Each bandit returns a reward drawn from a Gaussian that is randomly centered between 0 and 1.

Graph illustrates average reward using ε-greedy and decaying ε-greedy strategies.

(Figure from Arora and Hazan's AI class at Princeton)

ε-greedy methods

ε-greedy methods work well but ….

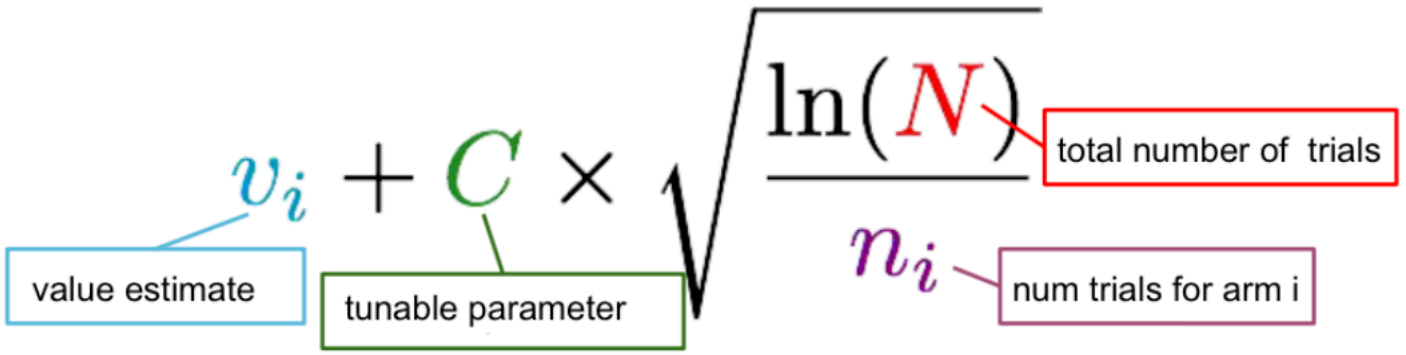
-- still explore "randomly" and may neglect promising arms
-- may continue to "explore" arms that have already been pulled many times.

Upper Confidence Bound

1. Initialize an estimated value for each arm to something high (optimistic, encourages exploration)
2. Update values as we gauge performance of each arm
3. Reduce incentive to explore as information is gathered (i.e. as confidence in estimates increases).

Leads to the UCB1 formula (Auer et al 2002):

First, try each arm once.
Then, at each time step, choose arm i that maximizes:

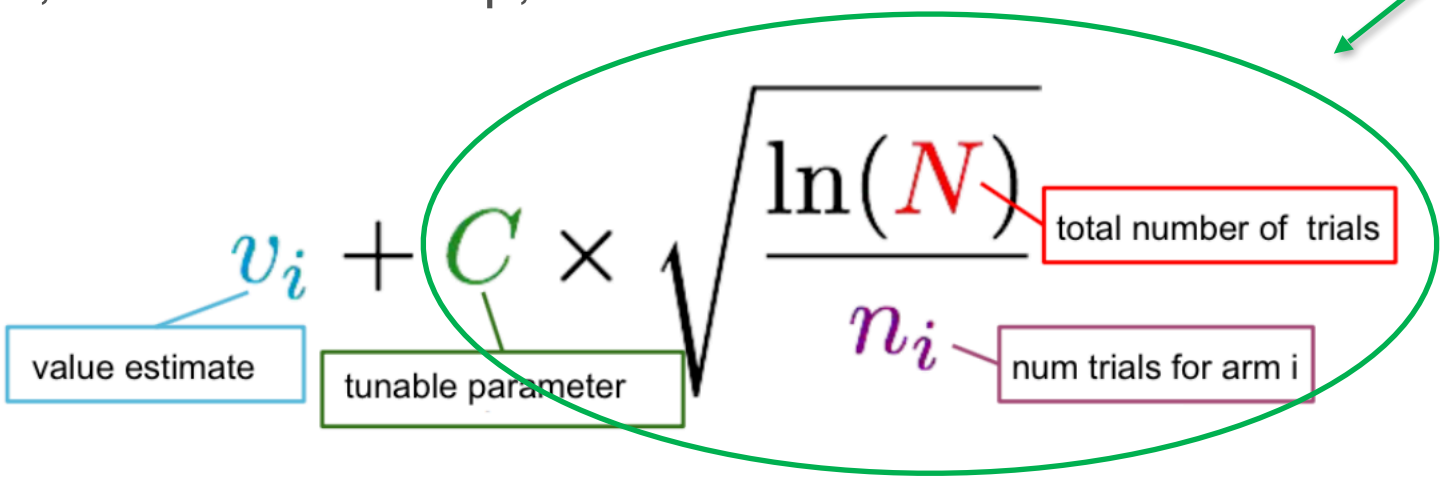


Upper Confidence Bound

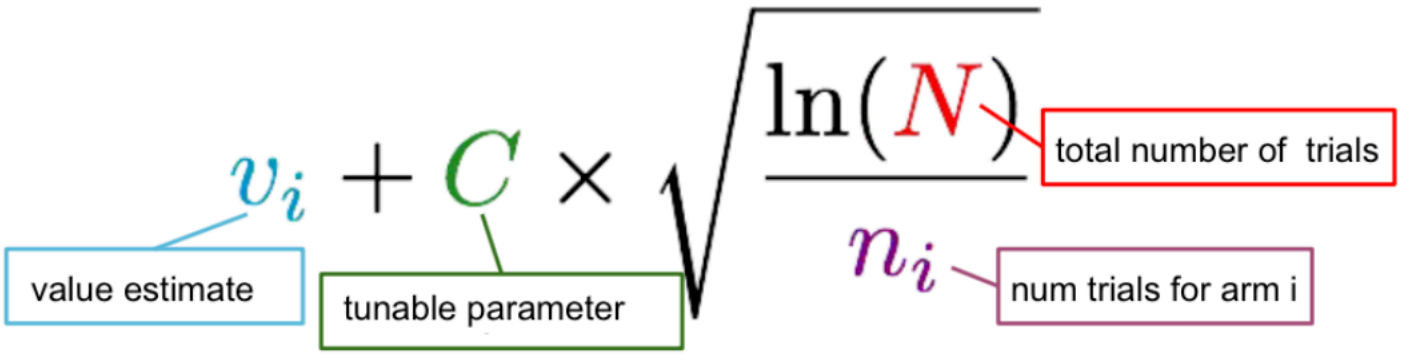
1. Initialize an estimated value for each arm to something high (optimistic, encourages exploration)
2. Update values as we gauge performance of each arm
3. Reduce incentive to explore as information is gathered (i.e. as confidence in estimates increases).

Leads to the UCB1 formula (Auer et al 2002):

First, try each arm once.
Then, at each time step, choose arm i that maximizes:



Upper Confidence Bound

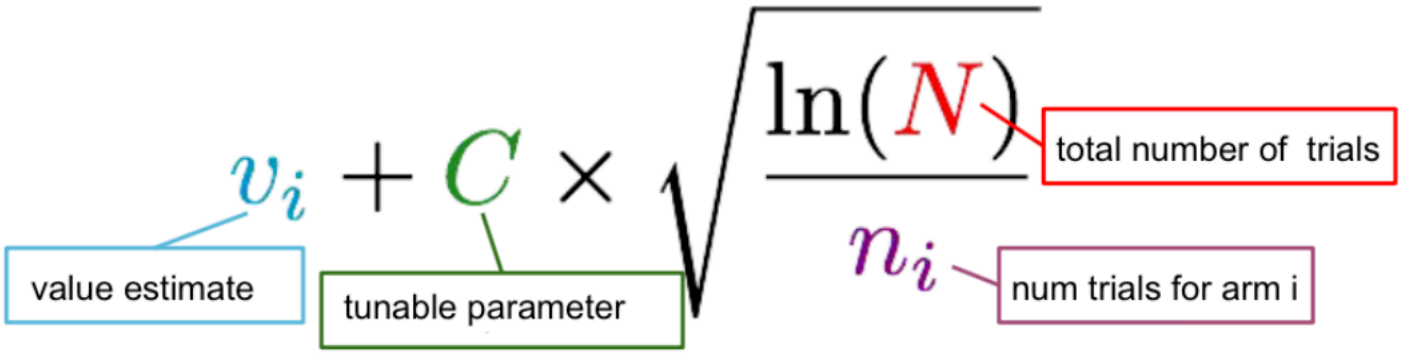


Confidence interval around v_i is large when relative number of trials (n_i) is small; shrinks in proportion to the square root of n_i

- When there is uncertainty about v_i, "exploration" term is large

We will explore if n_i is much less than N

Upper Confidence Bound



UCB1 has some nice theoretical properties:

- Allows for selection of optimal arm (or policy) as N nears infinity
- Eventually achieves logarithmic total regret
- Many other algorithms (e.g. greedy) achieve regret that is a linear function of N

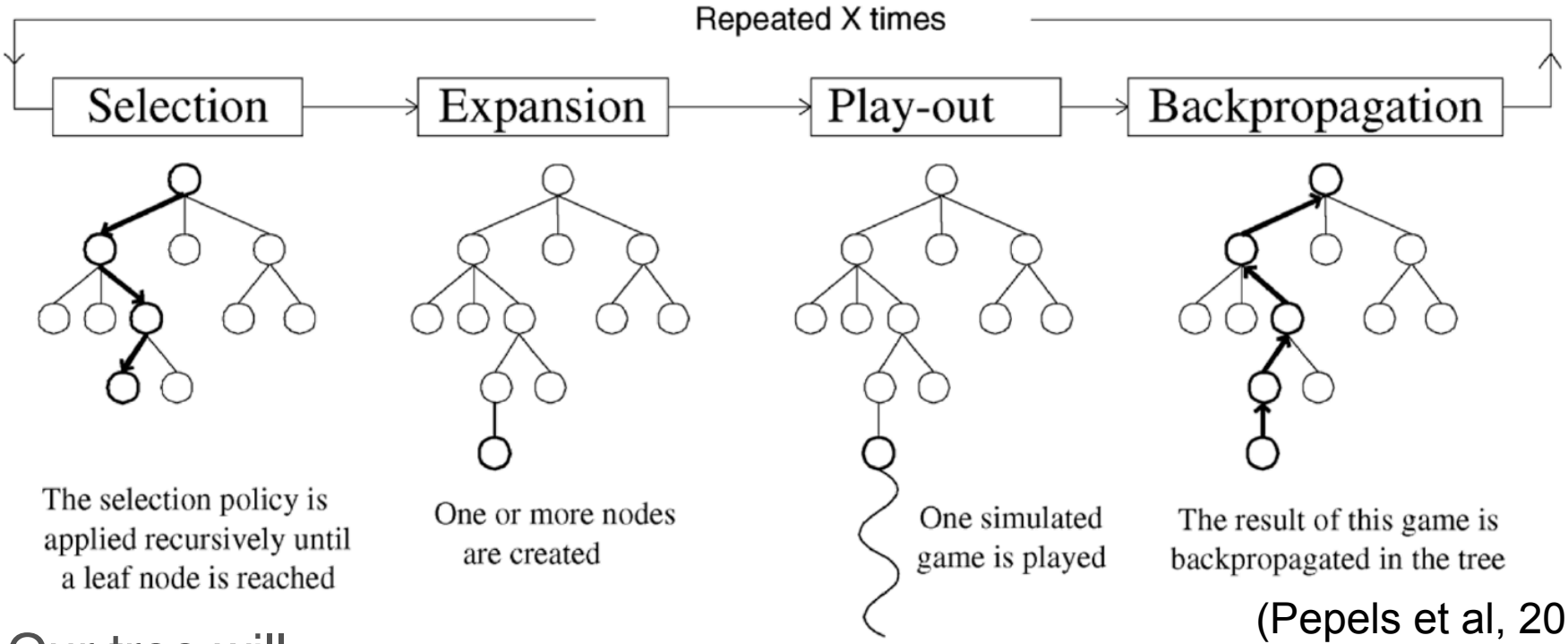
From UCB to a Game Tree

UCB makes a single decision,
…. but a game requires a sequence of decisions!

Make look-ahead (or game) tree that:

Equates **pulling the arm of a bandit** to playing a move.
Equates **payoff** with a move's utility, or value.
Equates **regret** with the losses we incur by making a sub-optimal move.

From UCB to a Game Tree



(Pepels et al, 2014)

Our tree will:

Apply a UCB-like formula to value interior nodes of tree

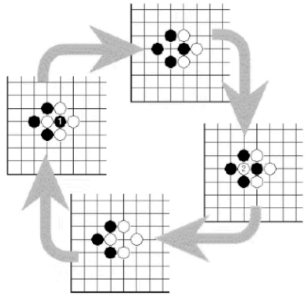
- We can choose “optimistically” where to expand next

Use rollouts (simulations) to gather information, update values.

Back to Go

The rules:
https://www.youtube.com/watch?v=5PTXdR8hLIQ

1. Play takes place on a 19x19 grid; players are black and white.
2. At each turn, players place a coloured “stone” on grid.
3. Players may skip turns, but must sacrifice a stone if they do.
3. Stones surrounded by an opponent are “captured”.
4. No self-sacrifices, no repetition of previous boards allowed.
6. Game ends when players “pass” in succession
7. Goal is to gather the most “territory”



Why Go is Hard

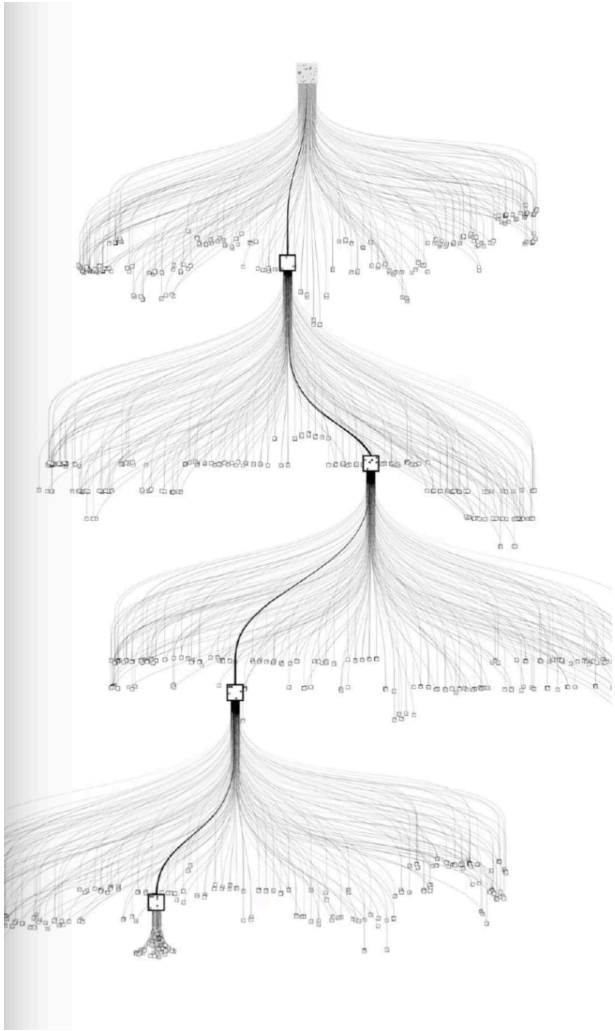
Go offers ~200 possible moves at each step
(vs. 35 for chess)
Each game of Go lasts an average of 100 moves
(vs. 40 for chess)
What does this mean for the size of the game tree?

In addition, engineering a good heuristic is hard!

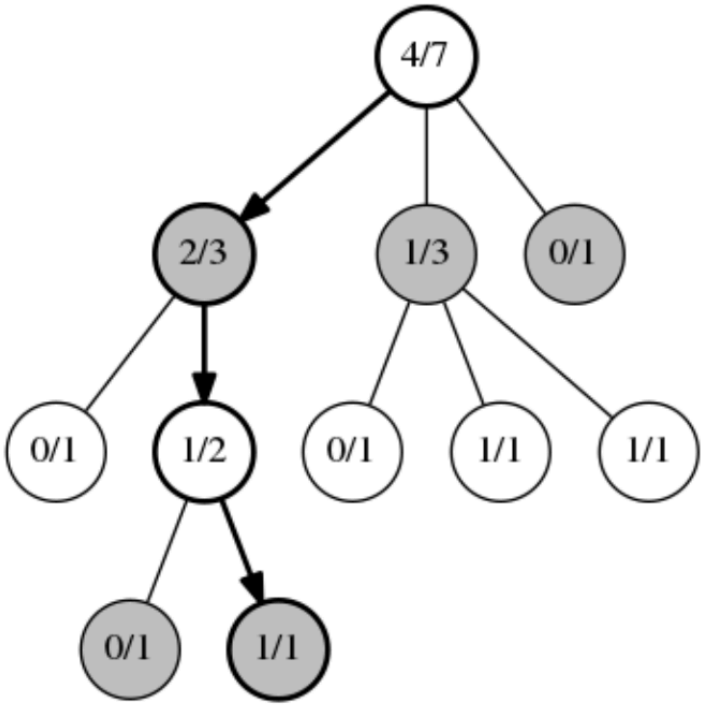
Very similar looking positions can have completely different outcomes:
-- person with the most pieces may not be "winning"
-- estimating "territories" is hard

Monte Carlo Tree Search (MCTS)

- Build a “statistics” tree (detailing values of nodes using UCB like formula)
- Uses statistics to focus on “interesting” node during search
- Determined node values by simulation
- Effectively limit focus on selected branches, searching to greater depths along “interesting” paths



Step 1: Selection

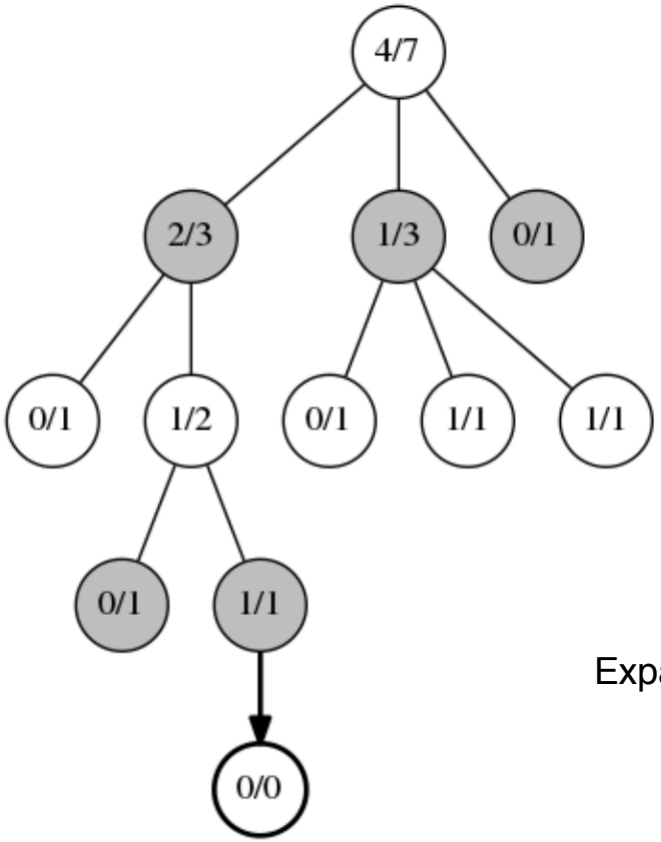


Apply selection policy recursively until a leaf node is reached

Note that values at each node reflect the perspective of the current player

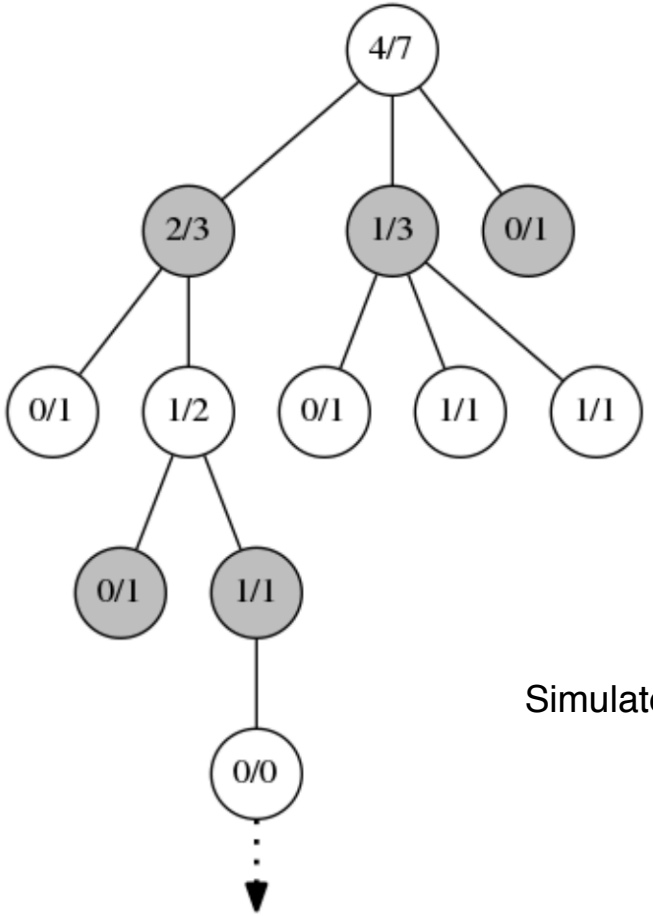
Also note that values here do not include “exploration” terms, just averages

Step 2: Expansion



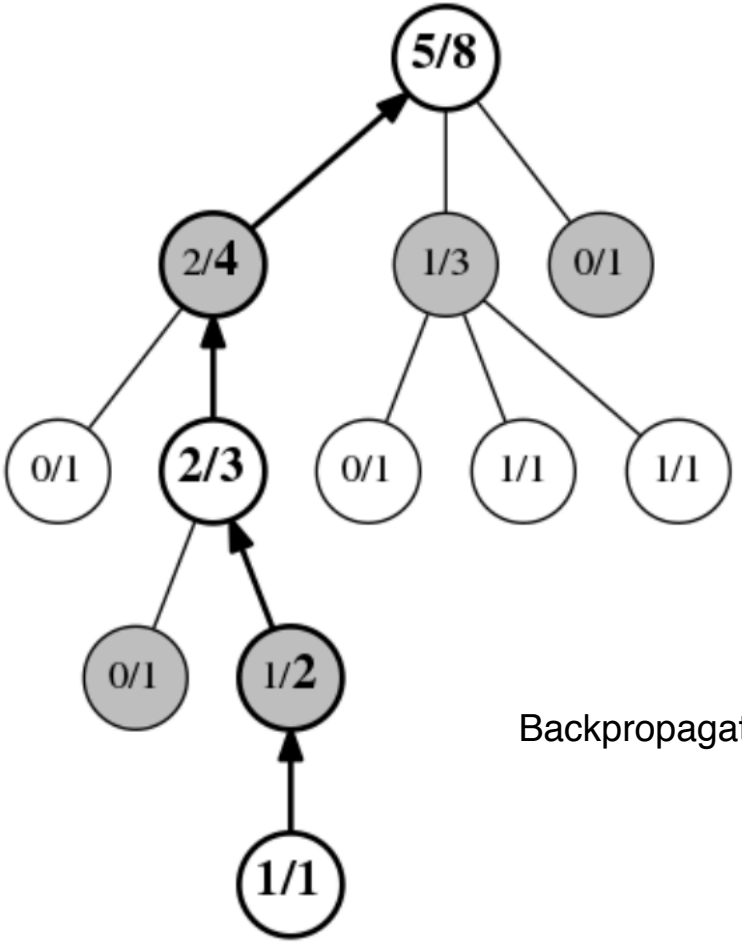
Expand, or explore, leaf nodes.

Step 3: Simulation



Simulate game play from the unexplored node.

Step 4: Backpropagation



Backpropagate results (update statistics)

Which Move to Take?

While Time Allows, MCTS will:

- Select and expand nodes based on summary statistics
- Simulate to grow the tree

Once time is up, which move to play?

- Highest average value
- Highest UCB value
- Most simulated value

Which Move to Take?

While Time Allows, MCTS will:

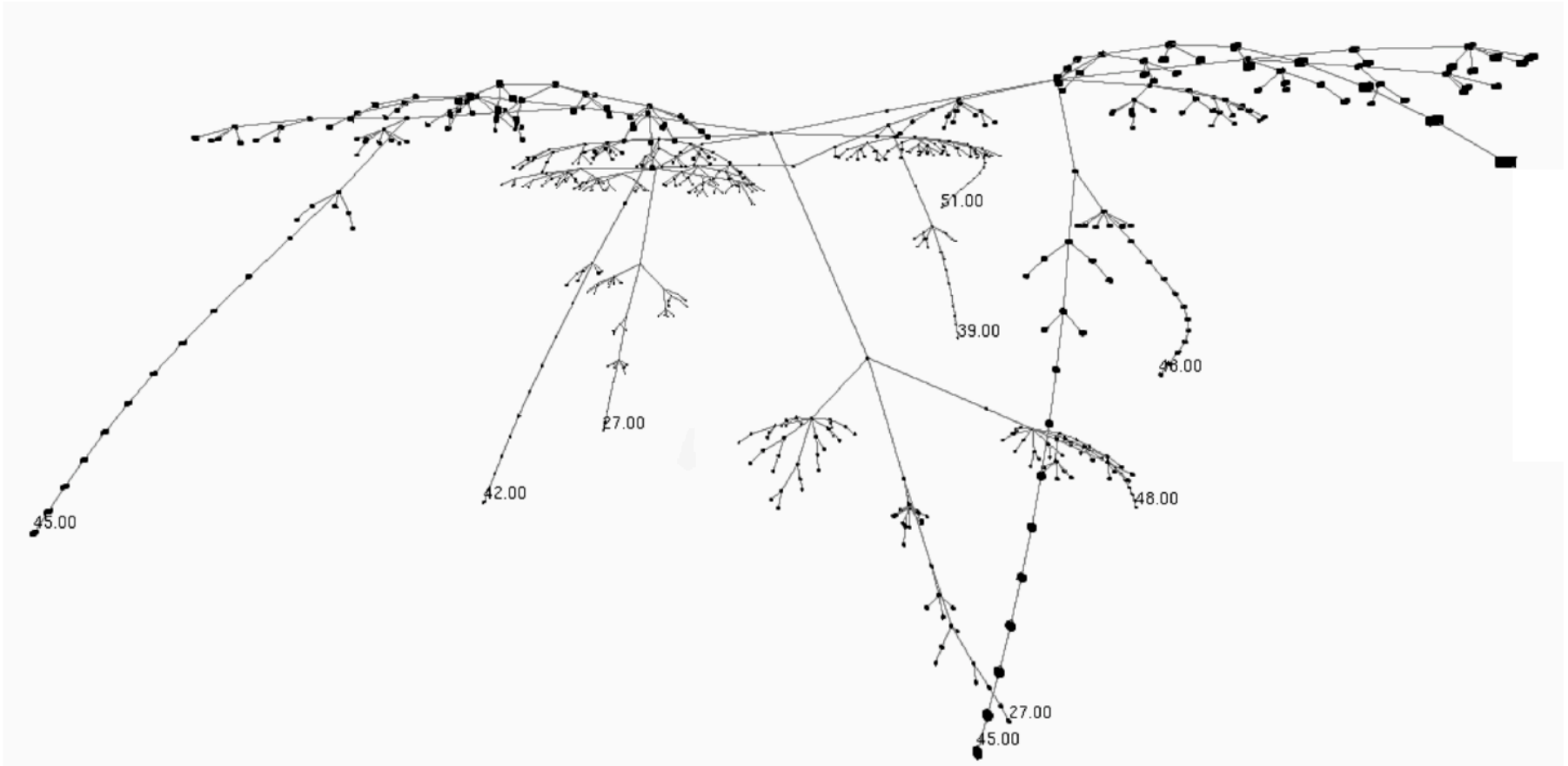
- Select and expand nodes based on summary statistics
- Simulate to grow the tree

Once time is up, which move to play?

- Highest average value
- Highest UCB value
- **Most simulated value**

MCTS looks at the most interesting moves more often

Sample MCTS Tree



(CadiaPlayer, Bjornsson and Finsson, IEE T-CIAIG 2009)

Summary of advantages and disadvantages of MCTS

No need for explicit evaluation function (or heuristic)

Applies to a variety of games, relatively domain independent

Focuses on promising parts of the game tree, which is particularly valuable when games have a large branching factor

Can yield a decision at any time, but decisions improve with time.

But …

A promising branch may still lead to a loss; losses may not be discovered at random

Improving MCTS

By default, UCB would see us make uniform random moves in order to explore.
At each layer, however, the branching factor of Go is still ~200.
That's still a lot of moves!

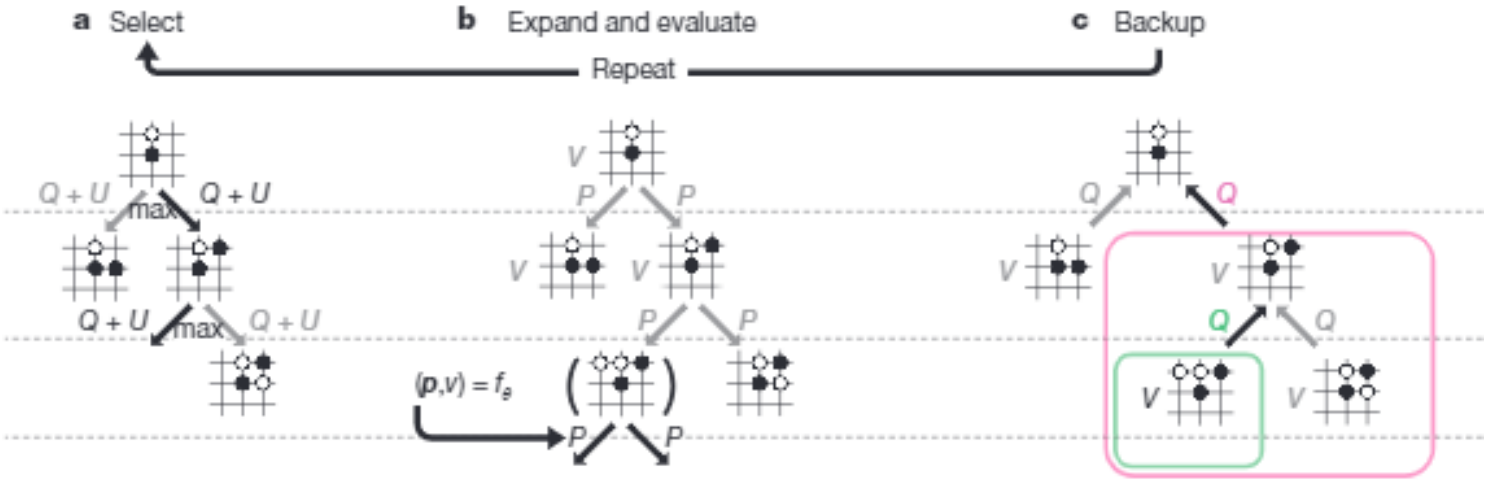
We can decrease the moves we explore using **prior knowledge.** One example:

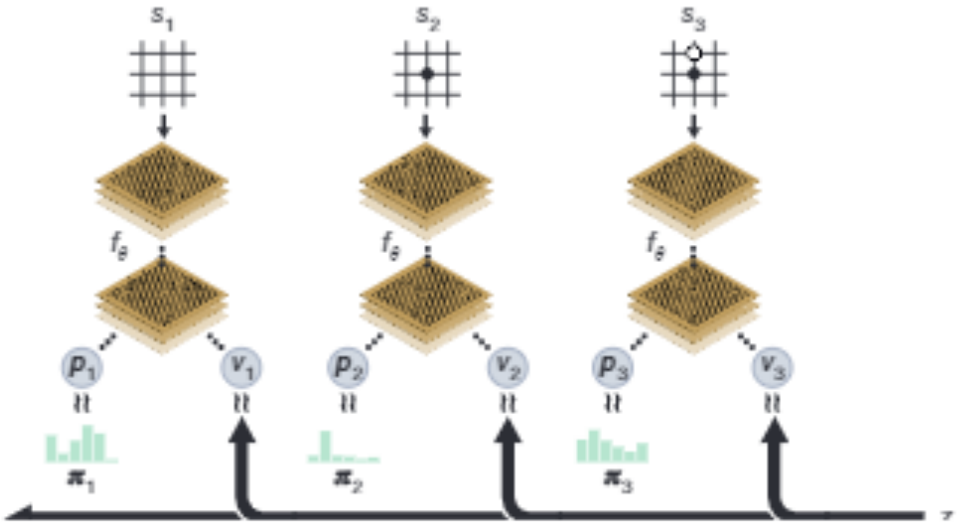
Last Good Reply (Drake 2009):

After winning a simulation, store (opponent move, our answer) move pairs

- Execute the same "reply" in future simulations
- Delete stored move pair if the move fails (Baier et al 2010)

Improving MCTS





AlphaGo`s exploration was guided by trained neural networks. These took in a board position, and returned a value and probability distribution over subsequent moves. Information was then backpropagated to the root.

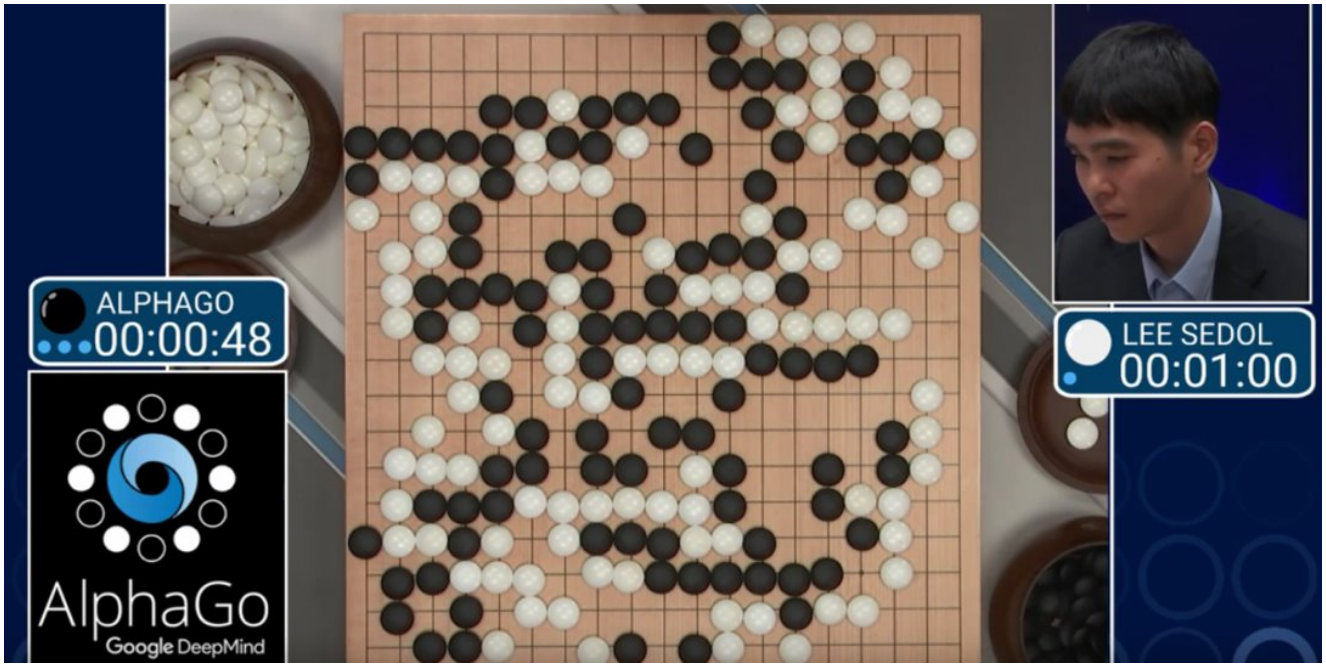
(Silver et. al, 2017)

Other MCTS improvements

Quantize state space – mix of MCTS and Dynamic Programming
-- Search graph rather than a tree

Parallelize search
-- Extra simulations should never hurt

Adaptive search
-- Dynamically alter simulation strategies based on play



Summary

- UCB, UCT are very important algorithms in both theory and practice with well-founded convergence guarantees under relatively weak conditions
- General applicability to a wide variety of problems
- Basis for extremely successful programs for games and many other applications
- Strategy has informed many current applications in planning, motion planning, optimization, finance, energy management