Monte Carlo Tree Search (MCTS)

MCTS Overview

- Iteratively building partial search tree
- Iteration
  - Most urgent node
    - Tree policy
    - Exploration/exploitation
  - Simulation
    - Add child node
    - Default policy
  - Update weights

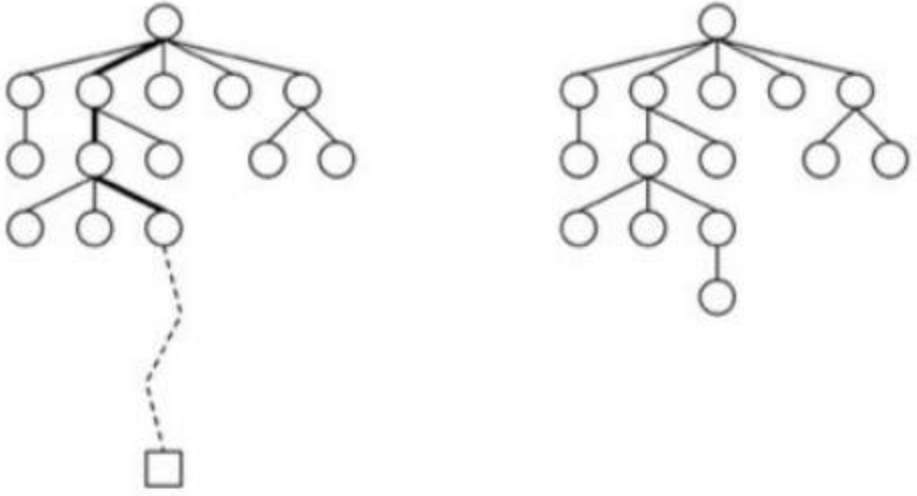


Fig. 1. The basic MCTS process [17].

Algorithm Overview

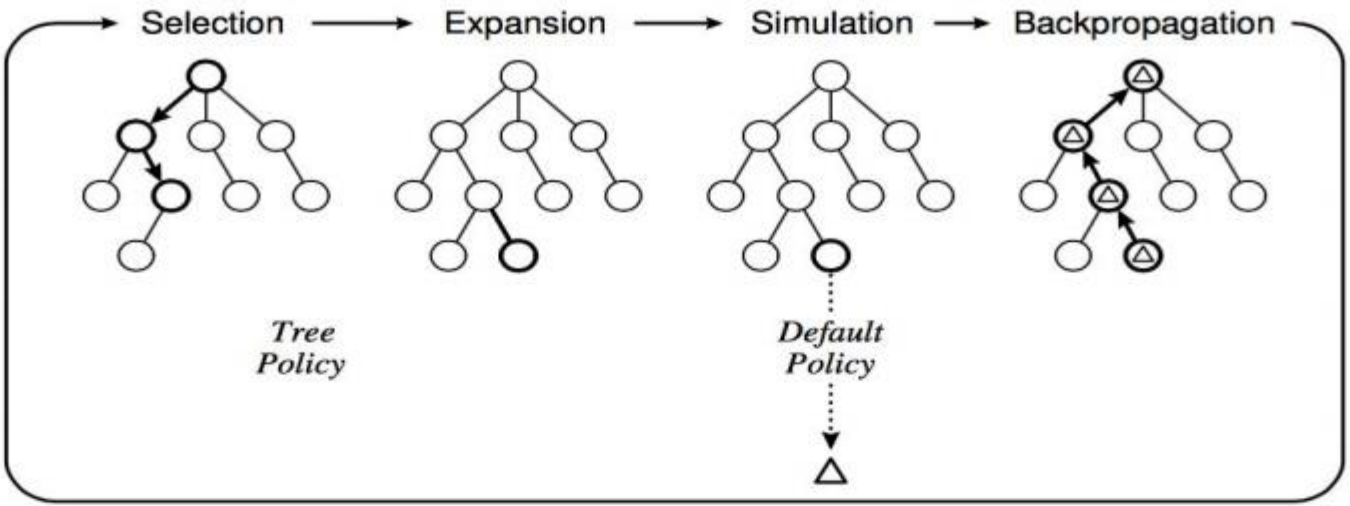


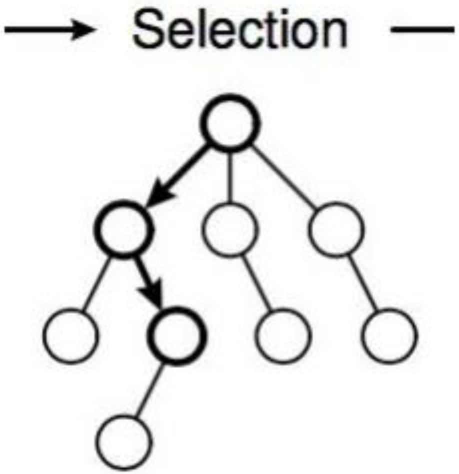
Fig. 2. One iteration of the general MCTS approach.

Policies

- Policies are crucial for how MCTS operates
- Tree policy
  - Used to determine how children are selected
- Default policy
  - Used to determine how simulations are run (ex. randomized)
  - Result of simulation used to update values

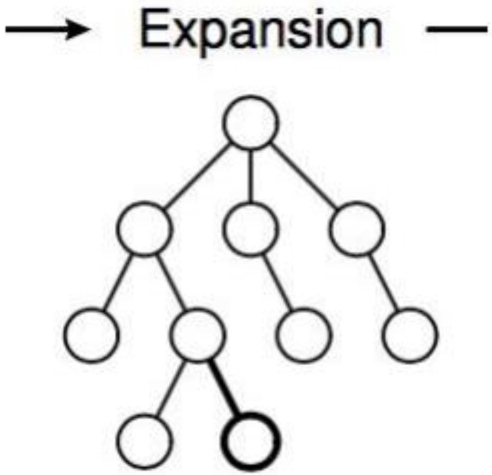
Selection

- Start at root node
- Based on Tree Policy select child
- Apply recursively - descend through tree
    - Stop when expandable node is reached
    - *Expandable* -
        - Node that is non-terminal and has unexplored children



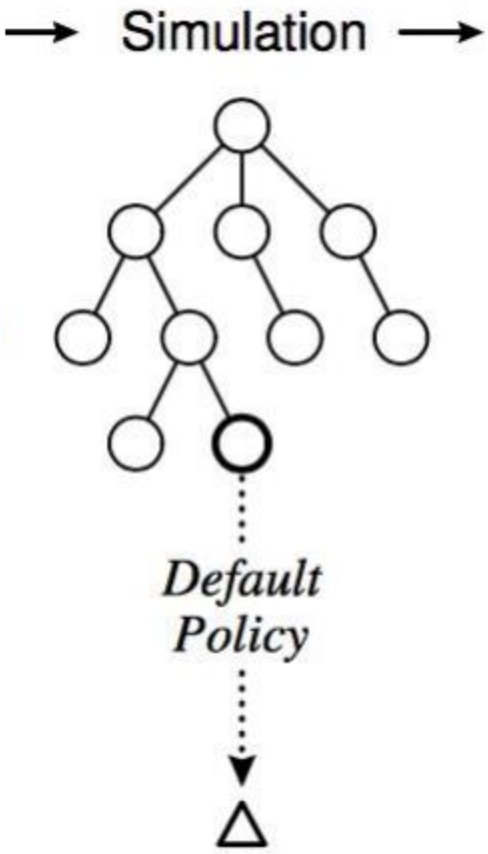
Expansion

- Add one or more child nodes to tree
  - Depends on what actions are available for the current position
  - Method in which this is done depends on Tree Policy



Simulation

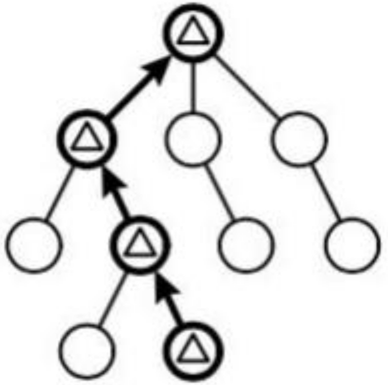
- Runs simulation of path that was selected
- Get position at end of simulation
- Default Policy determines how simulation is run
- Board outcome determines value



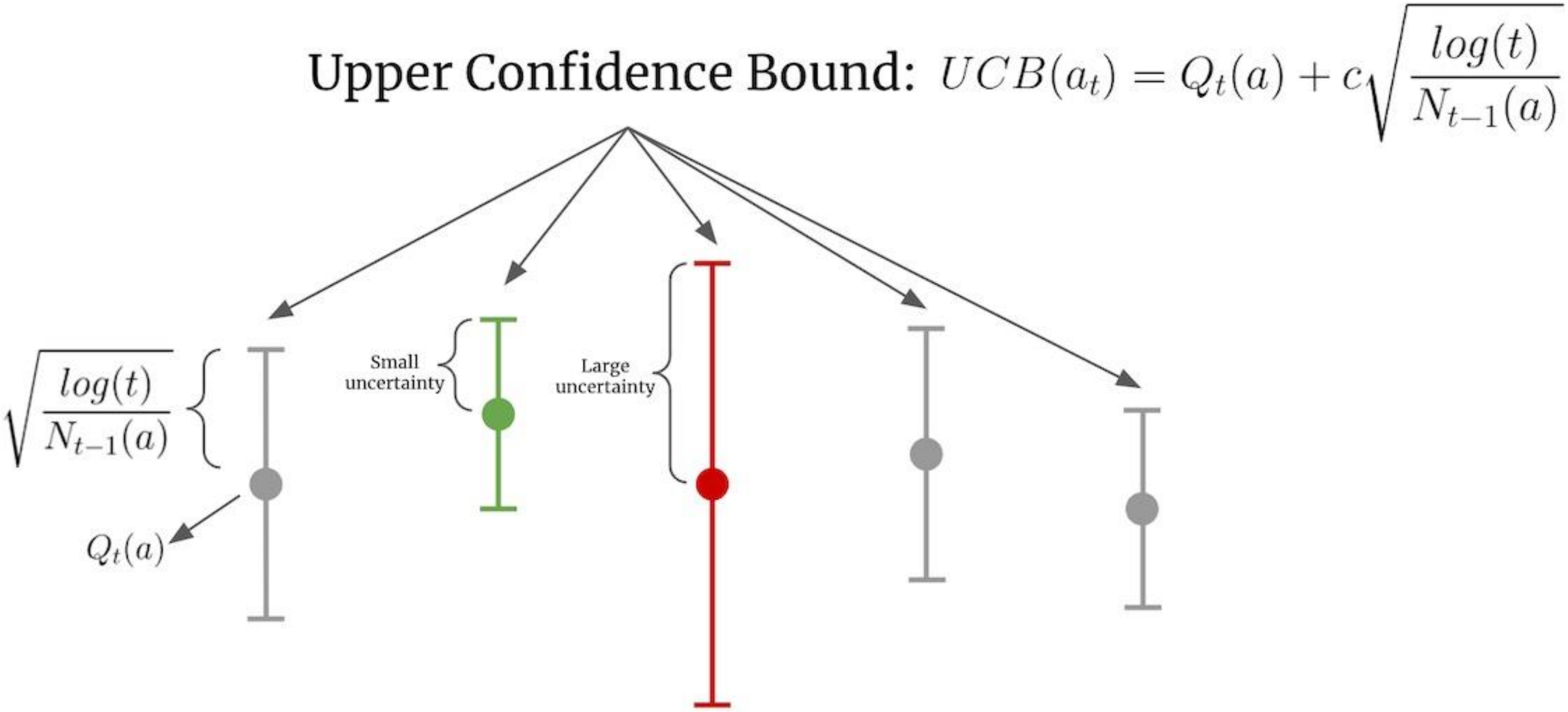
Backpropagation

- Moves backward through saved path
- Value of Node
    - representative of benefit of going down that path from parent
- Values are updated dependent on board outcome
    - Based on how the simulated game ends, values are updated

→ Backpropagation -



UCB in Bandits



Upper Confidence bounds applied to Trees (UCT) Algorithm

- Selecting child node: multi-armed bandit problem
  - UCB for child selection
- UCT

$$\underbrace{v_i}_{\text{value estimate}} + \underbrace{C}_{\text{tunable parameter}} \times \sqrt{\frac{\ln(\overbrace{N}^{\text{parent node visits}})}{\underbrace{n_i}_{\text{number of visits}}}}$$

- v: value estimate
- C: exploration parameter
- N: number of parent node visits
- n: number of visits

UCT Algorithm

$$\underbrace{v_i}_{\text{value estimate}} + \underbrace{C}_{\text{tunable parameter}} \times \sqrt{\frac{\ln(\overbrace{N}^{\text{parent node visits}})}{\underbrace{n_i}_{\text{number of visits}}}}$$

- n = 0 means infinite weight
    - Guarantees we explore each child at least once
- Each child has non-zero probability of selection
- Adjust C to change explore-exploit tradeoff

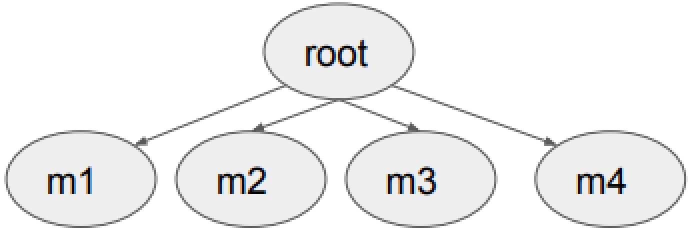
Theorem. MCTS with UCT action selection in the Selection phase finds an optimal policy [Kocsis and Szepesvári. ECML ‘06]

Example - The Game of Othello

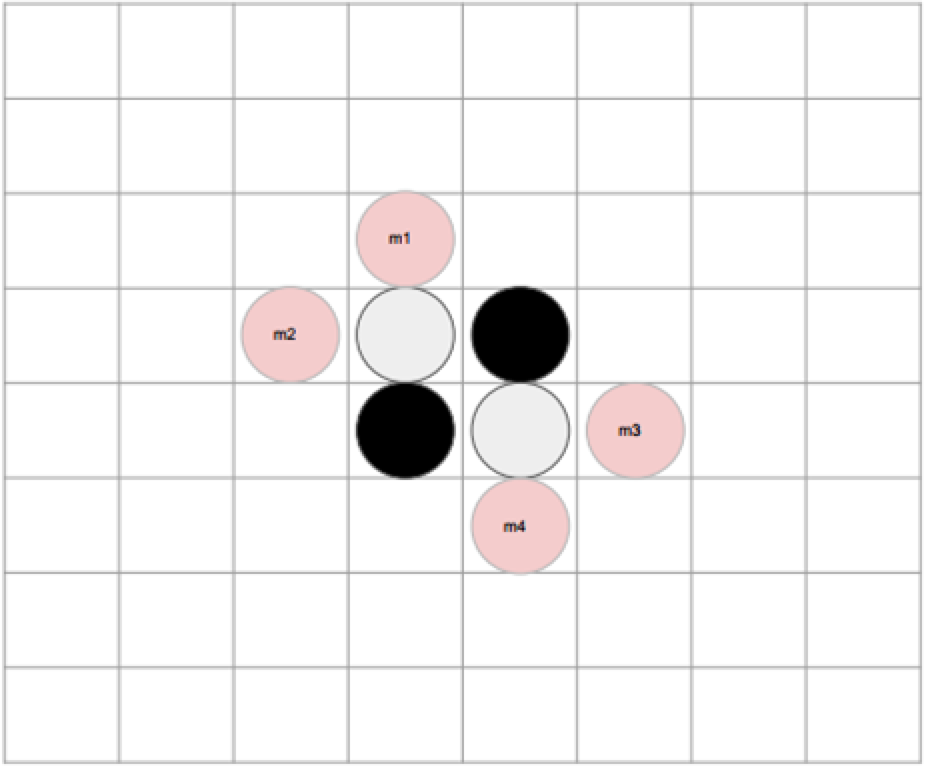
$$v_i + C \times \sqrt{\frac{\ln(N)}{n_i}}$$

value estimate
tunable parameter
parent node visits
number of visits

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)



- n_j - initially 0
  - all weights are initially infinity
- n - initially 0
- C_p - some constant > 0
  - For this example
  - C = (1 / 2√2)
- X_j - mean reward of selecting this position
  - [0, 1]
  - Initially N/A

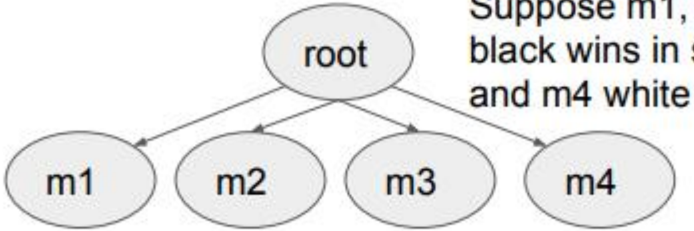


Example - The Game of Othello cont.

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

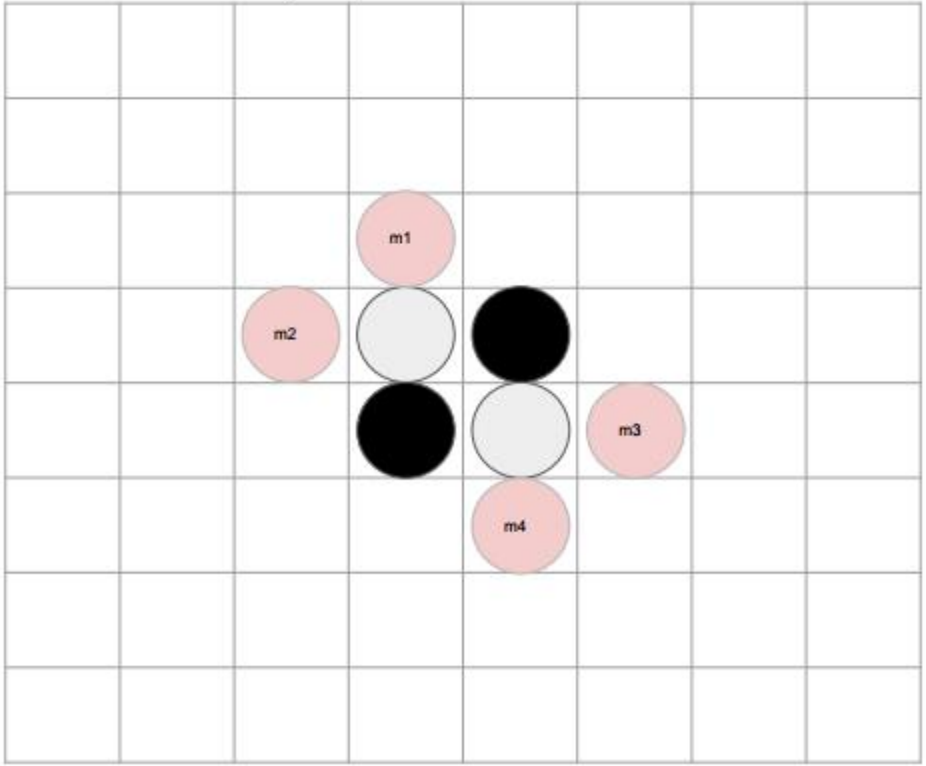
value estimate
tunable parameter
parent node visits
number of visits

After first 4 iterations:
Suppose m1, m2, m3 black wins in simulation and m4 white wins



	X_j	n	n_j
m1	1	4	1
m2	1	4	1
m3	1	4	1
m4	0	4	1

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)

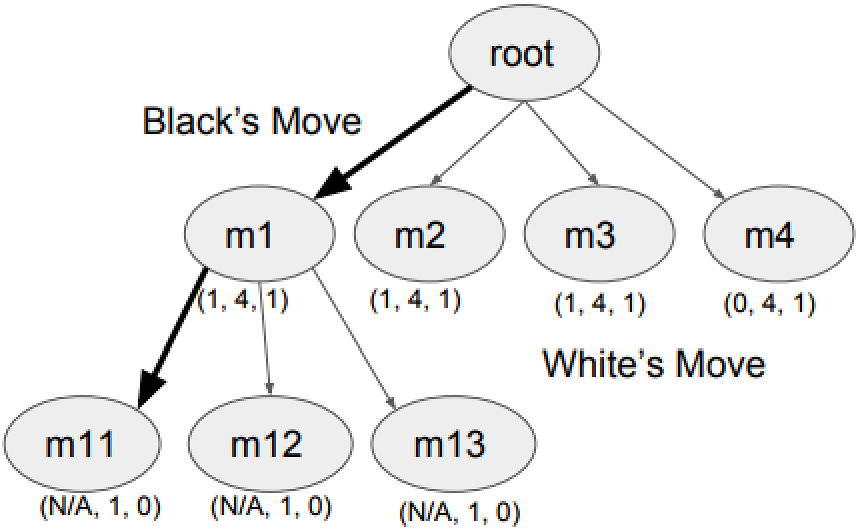


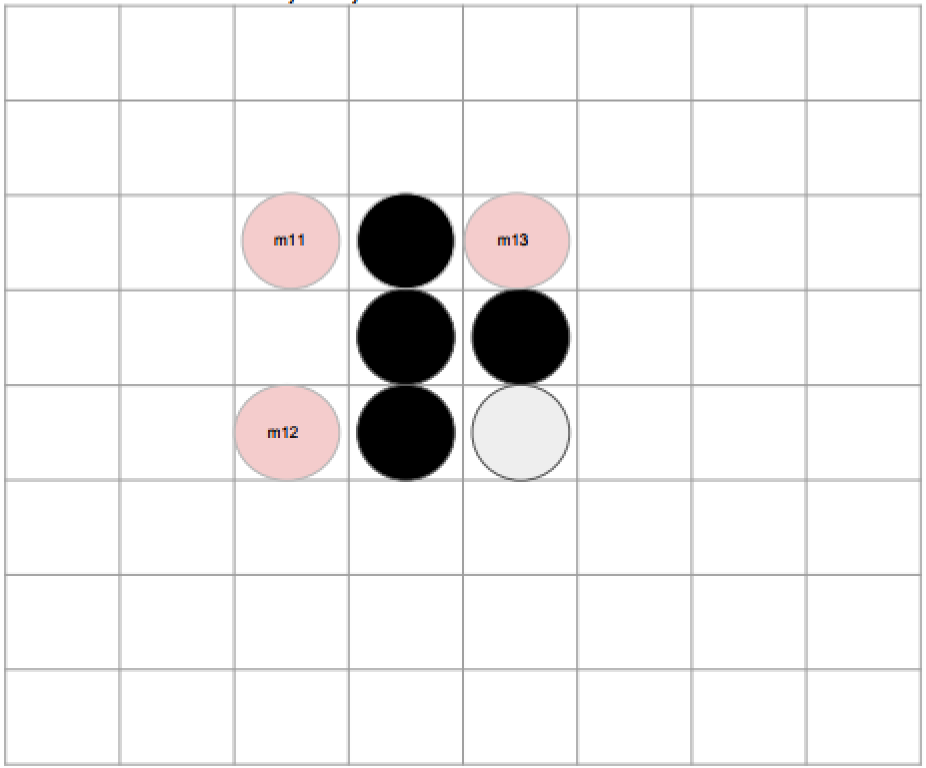
Example - The Game of Othello Iter #5

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

value estimate
tunable parameter
parent node visits
number of visits

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)





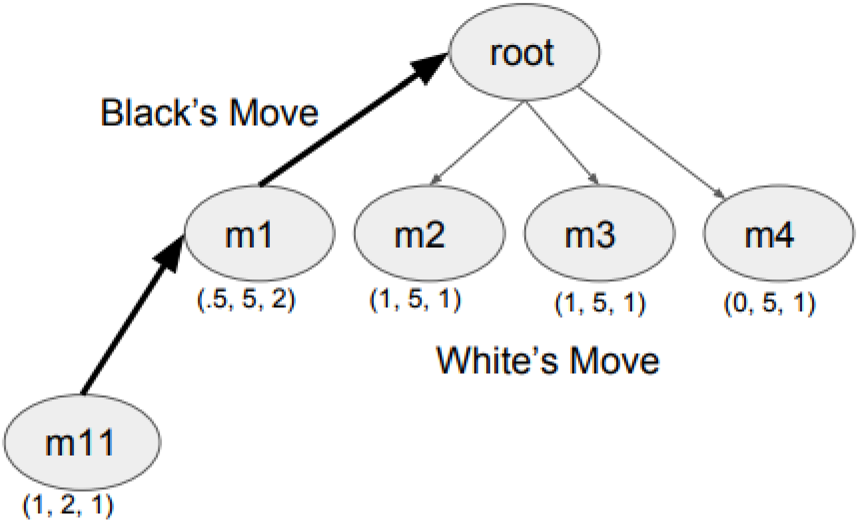
- First selection picks m1
- Second selection picks m11

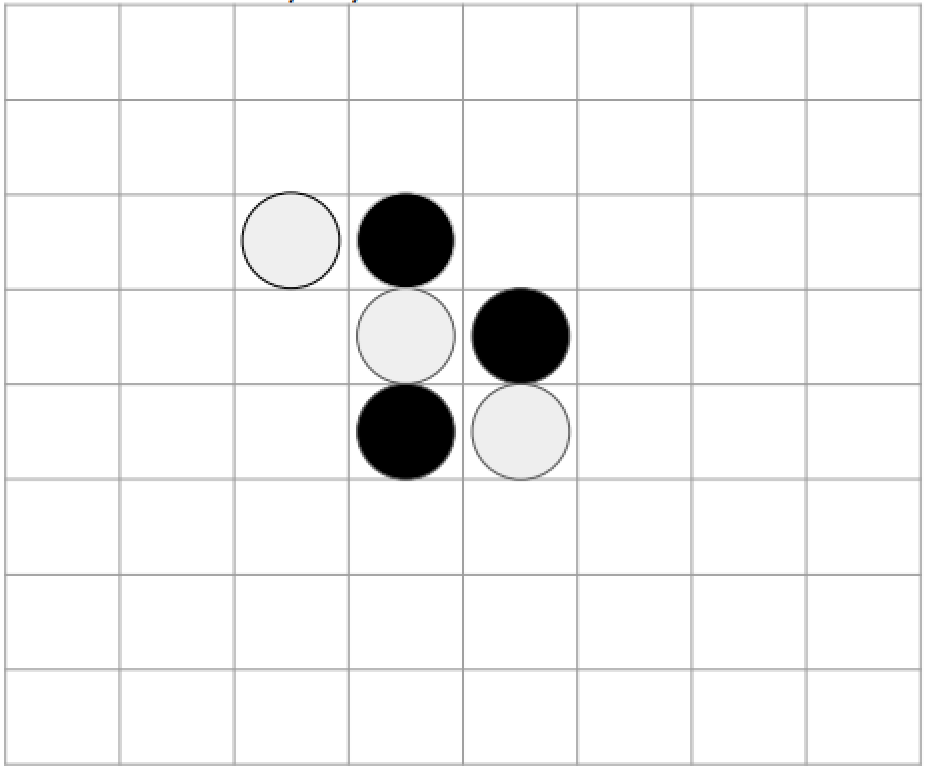
Example - The Game of Othello Iter #5

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

value estimate
tunable parameter
parent node visits
number of visits

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)

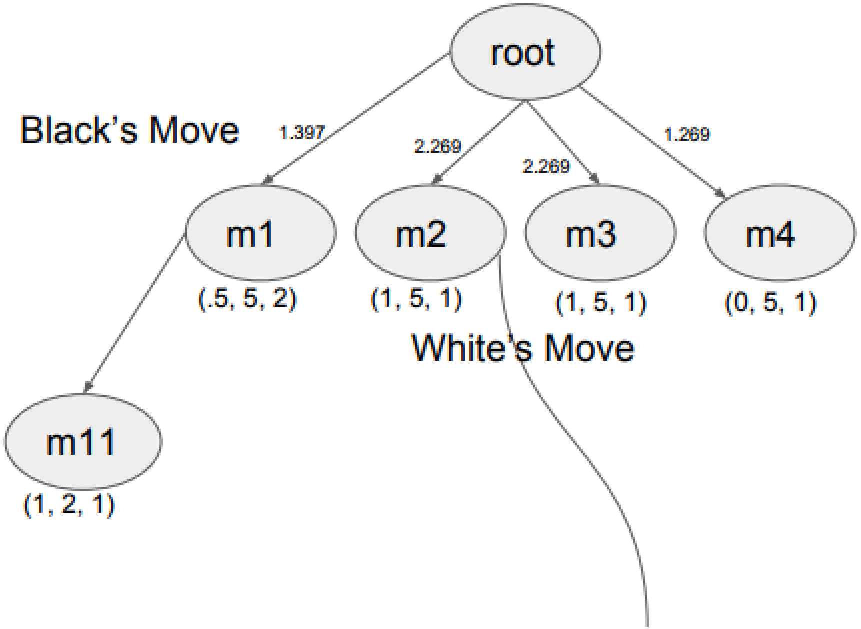




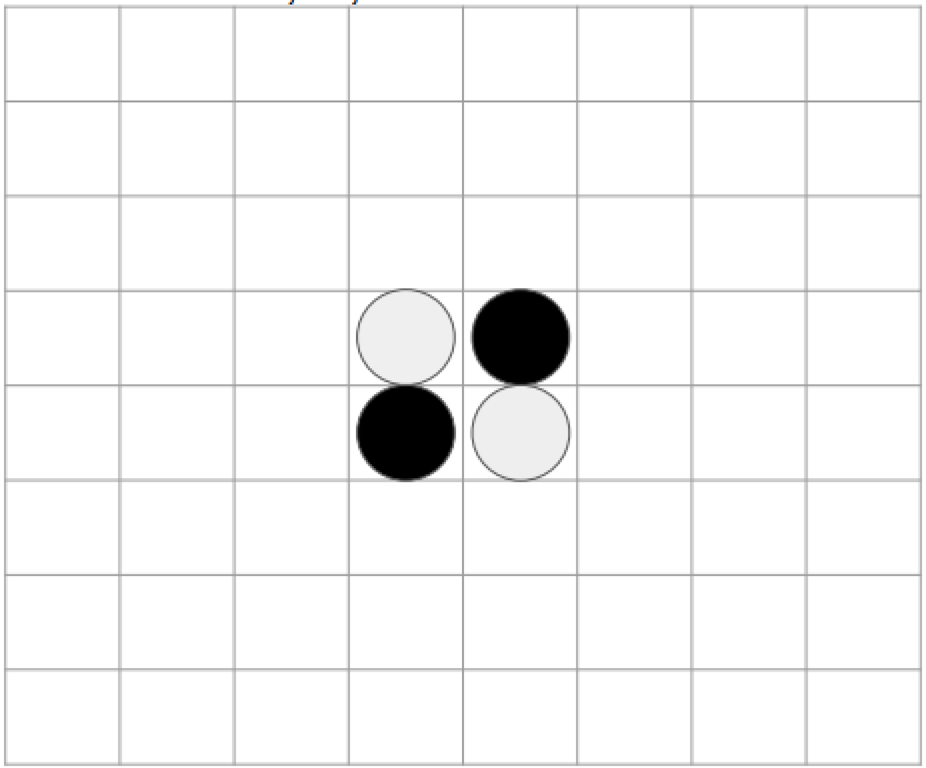
- Run a simulation
- White Wins
- Backtrack, and update mean scores accordingly.

Example - The Game of Othello Iter #6

$$\underbrace{v_i}_{\text{value estimate}} + \underbrace{C}_{\text{tunable parameter}} \times \sqrt{\frac{\ln(\overbrace{N}^{\text{parent node visits}})}{\underbrace{n_i}_{\text{number of visits}}}}$$



(X_j, n, n_j) - (*Mean Value, Parent Visits, Child Visits*)



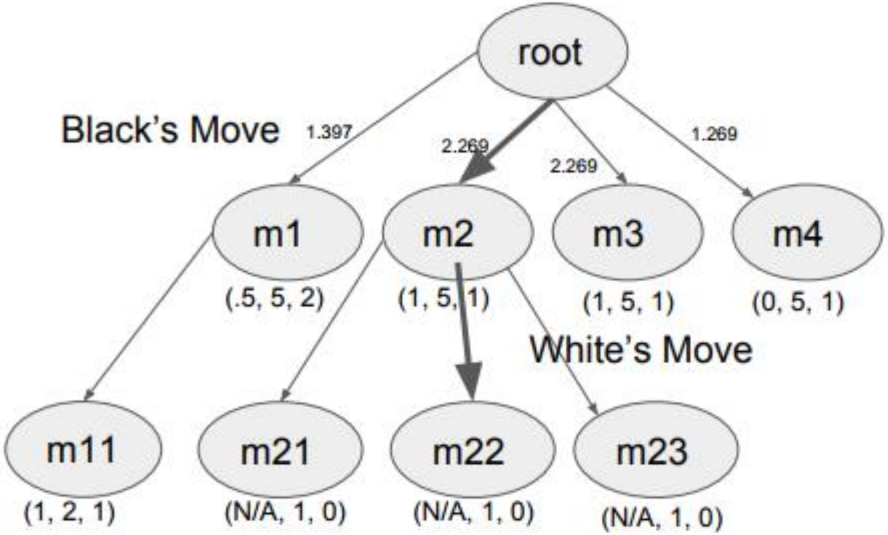
- Suppose we first select m2

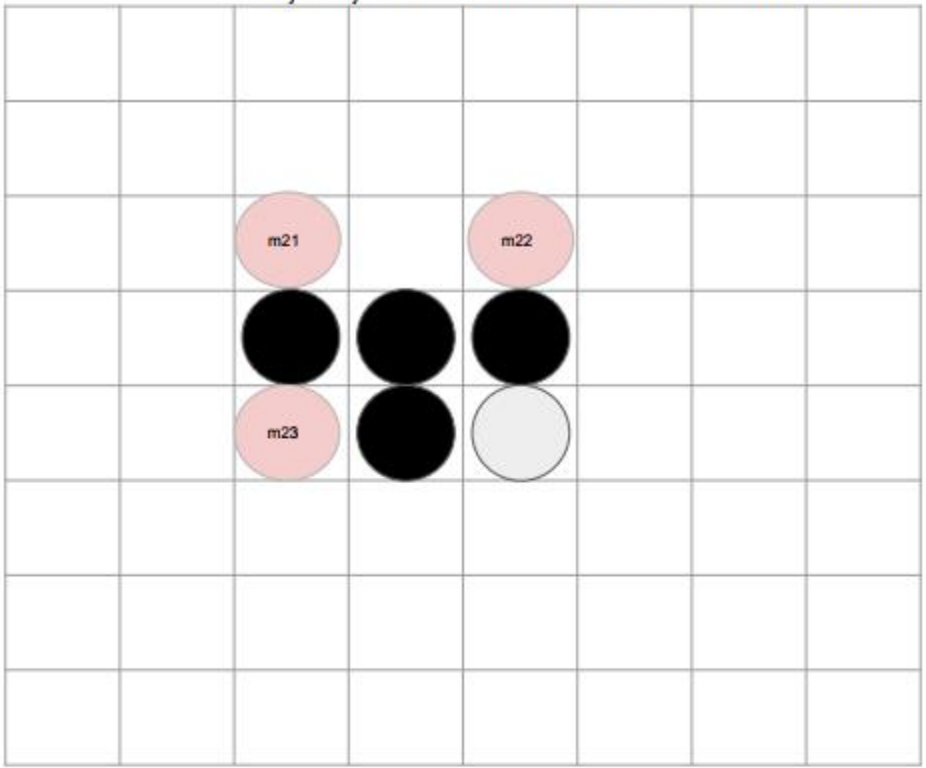
Example - The Game of Othello Iter #6

$$\underbrace{v_i}_{\text{value estimate}} + \underbrace{C}_{\text{tunable parameter}} \times \sqrt{\frac{\ln(N)}{n_i}}$$

N — parent node visits; n_i — number of visits

(X_j, n, n_j) - (*Mean Value, Parent Visits, Child Visits*)





- Suppose we pick m22

Example - The Game of Othello Iter #6

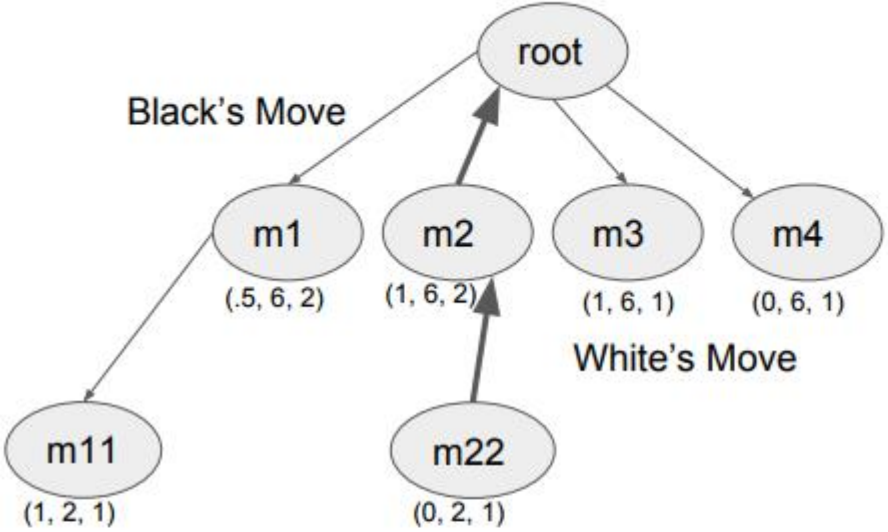
$$v_i + C \times \sqrt{\frac{\ln(N)}{n_i}}$$

value estimate

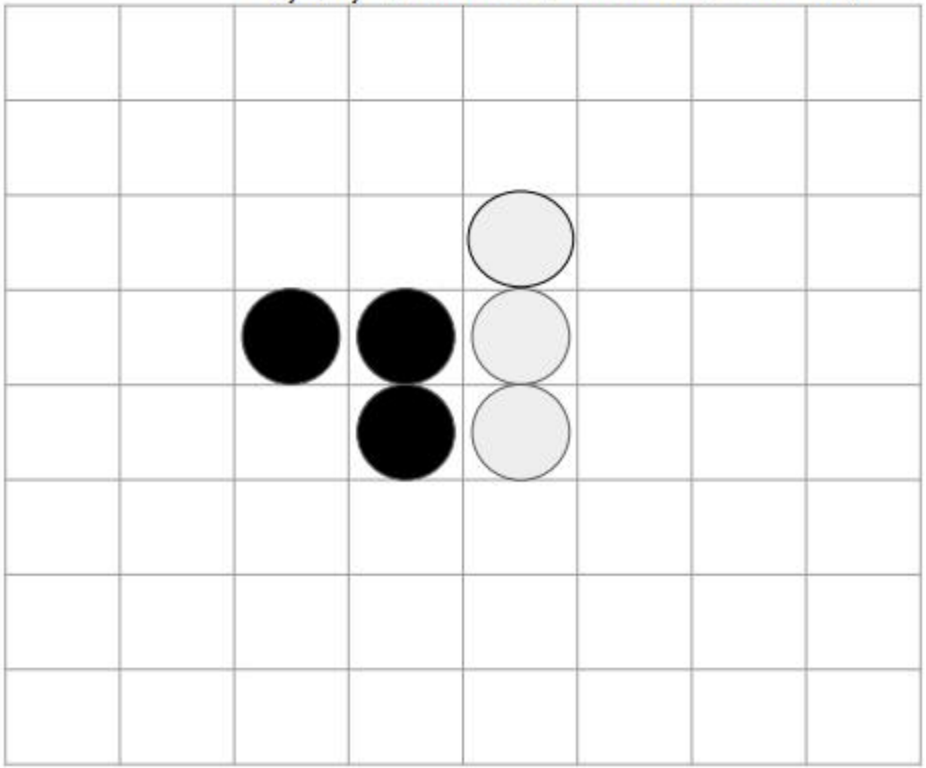
tunable parameter

parent node visits

number of visits



(X_j, n, n_j) - *(Mean Value, Parent Visits, Child Visits)*

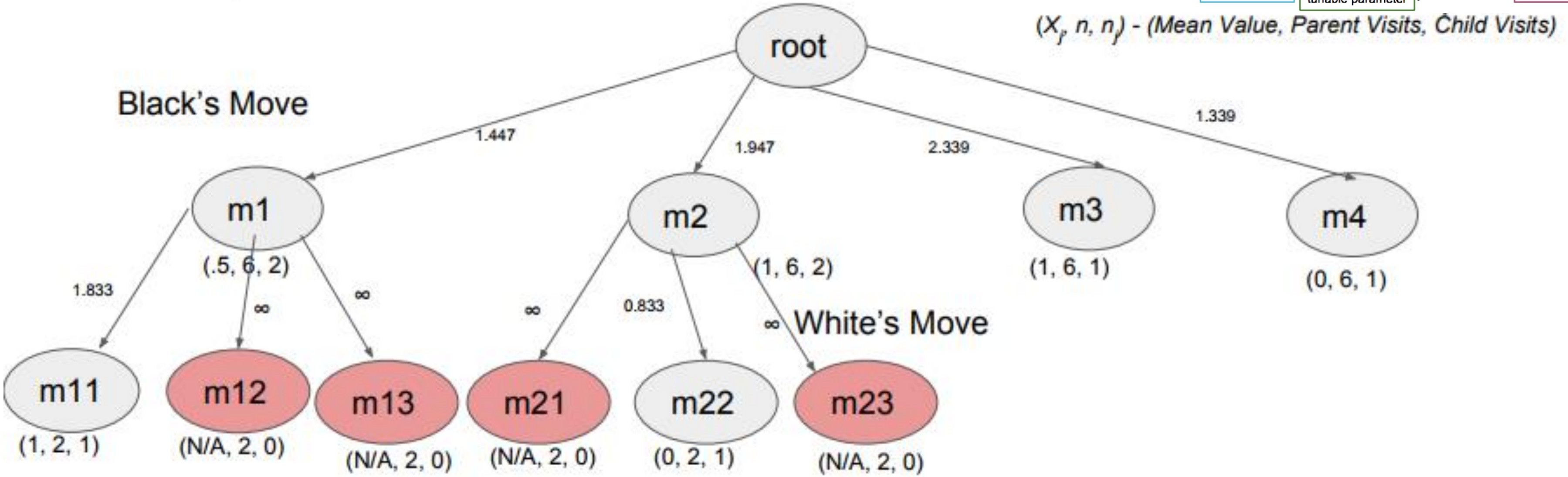


- Run simulated game from this position.
- Suppose black wins the simulated game.
- Backtrack and update values

Example - The Game of Othello Iter #6

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

parent node visits
value estimate
tunable parameter
number of visits



- This is how our tree looks after 6 iterations.
- Red Nodes not actually in tree
- Now given a tree, actual moves can be made using max, robust, max-robust, or other child selection policies.
- Only care about subtree after moves have been made

AlphaGo

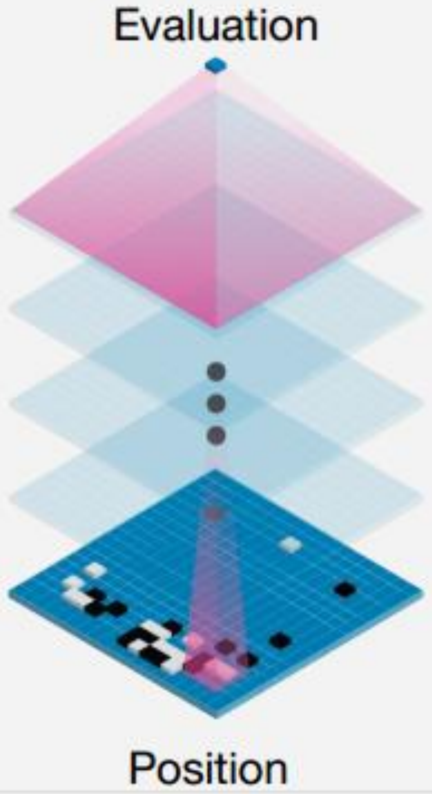

LEE SEDOL

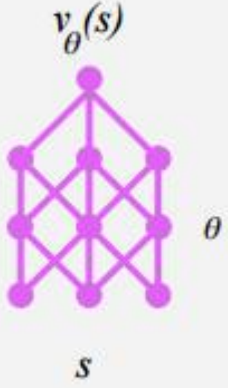
AlphaGo

Uses a value network and policy network to augment MCTS

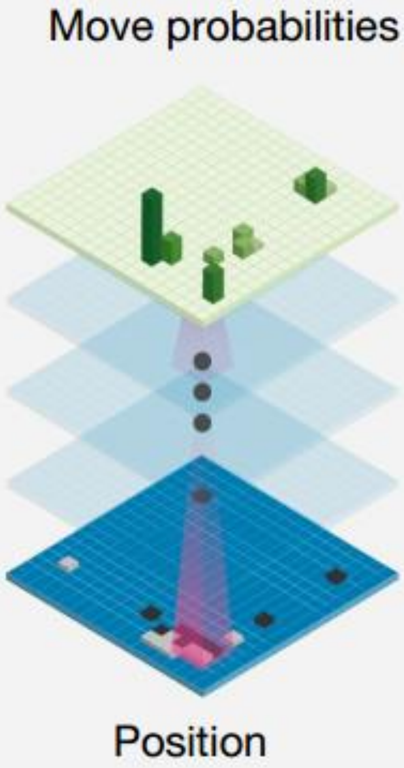
1. Policy network first trained on professional Go games and then trained further using reinforcement learning
2. Value network trained using self-play using the policy network
3. Then MCTS is run leveraging the two networks

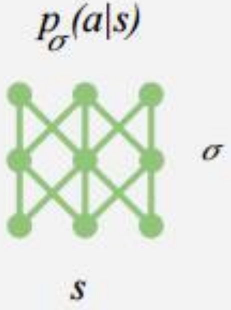
Value network





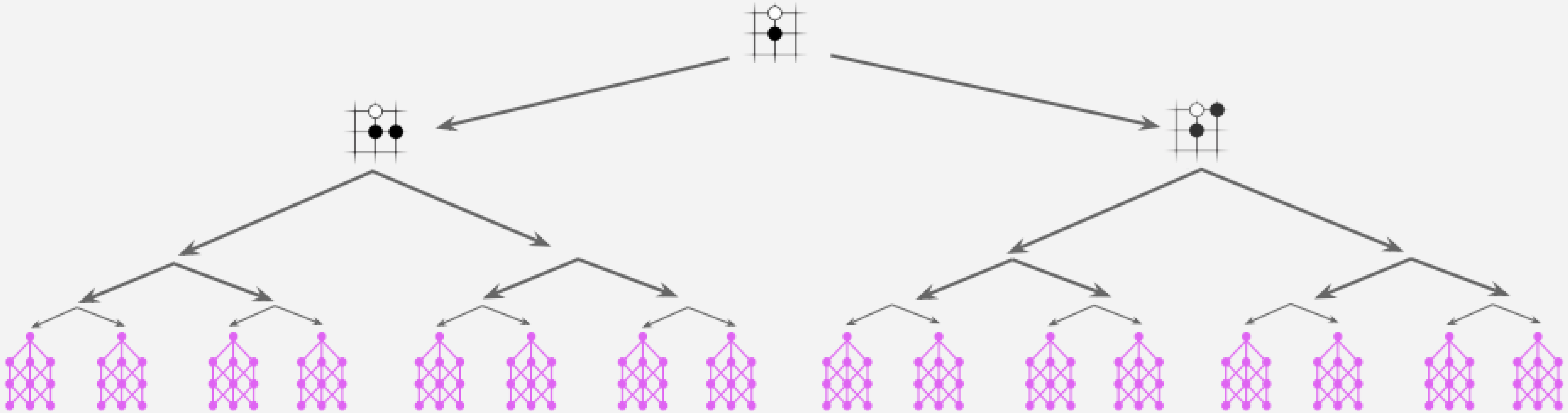
Policy network





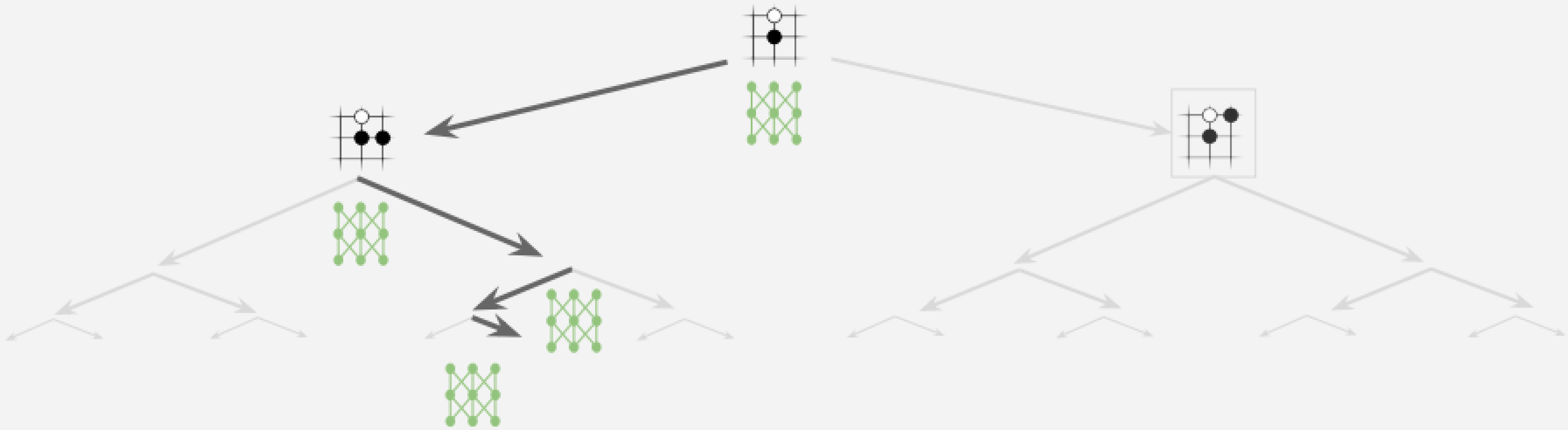
High-level idea 1:

Reducing depth with value network



High-level idea 2:

Reducing breadth with policy network



AlphaGo’s MCTS

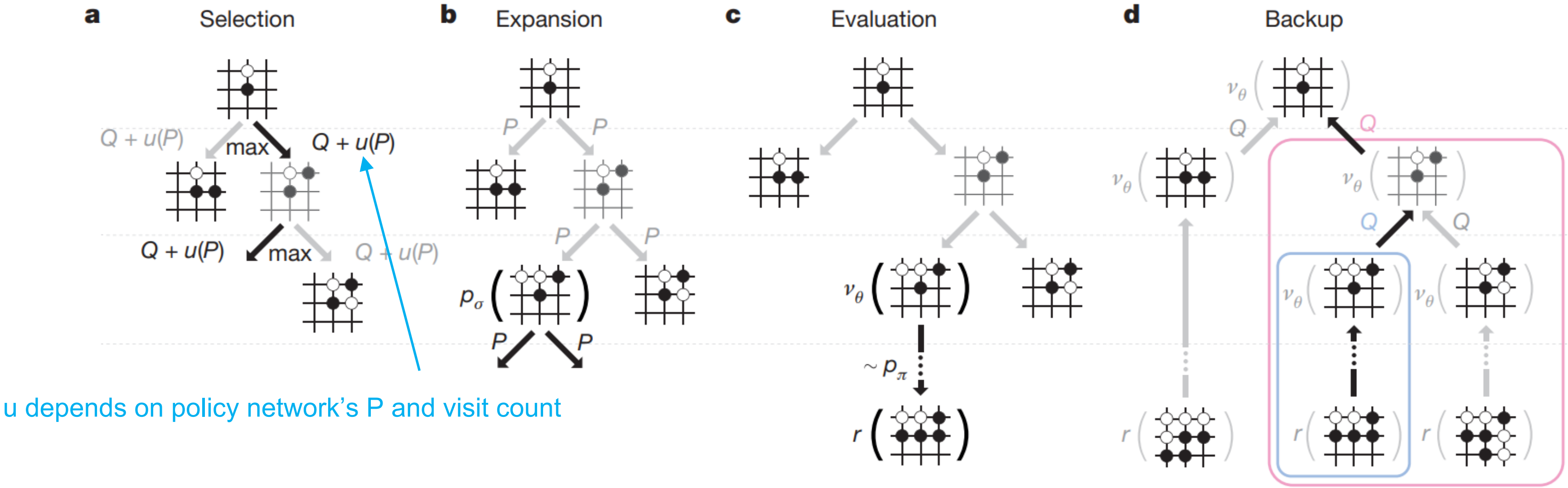


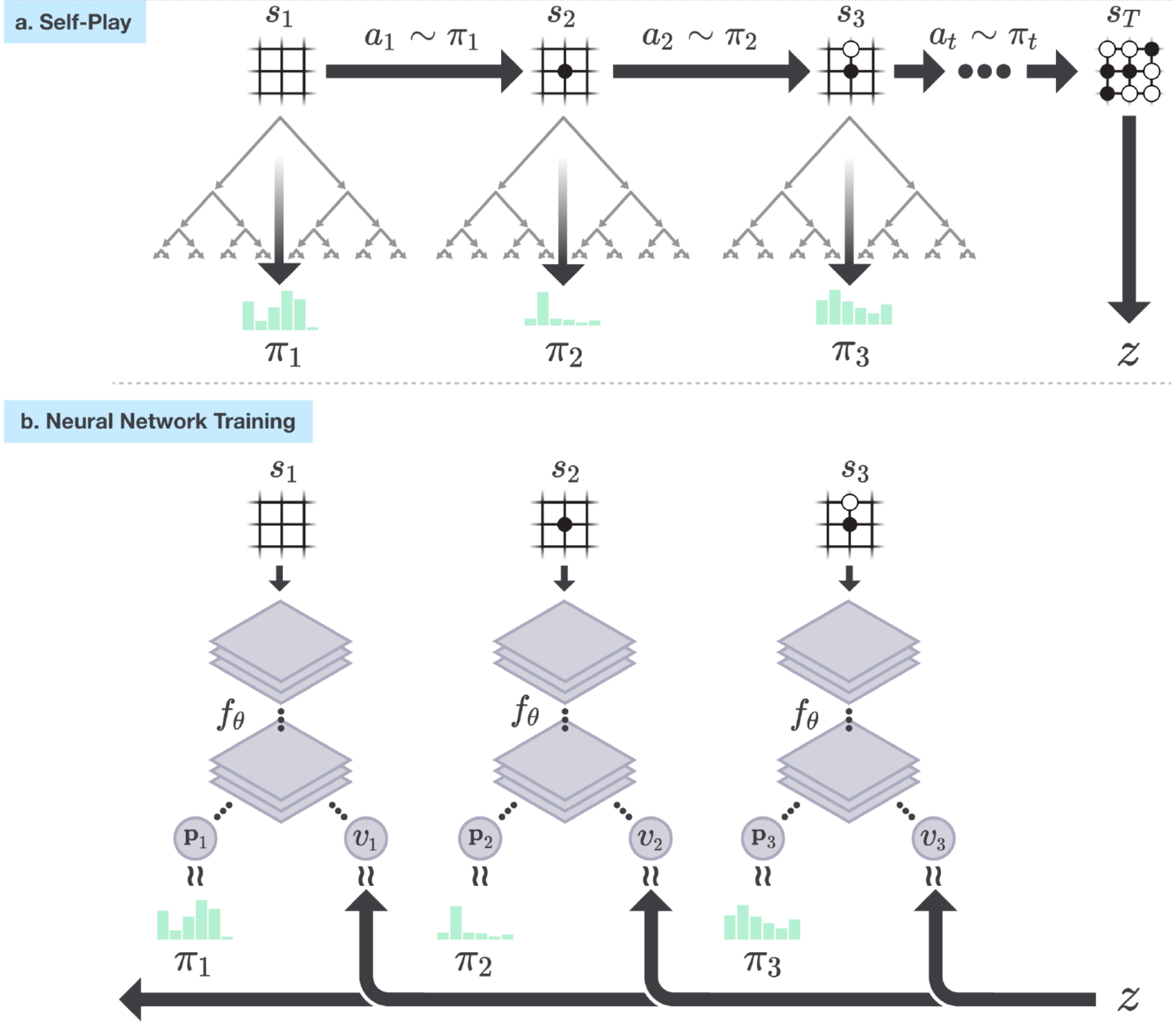
Figure 3 | Monte Carlo tree search in AlphaGo. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

Once search is complete, the algorithm selects the most visited move from the root.

AlphaGo Zero

AlphaGo Zero

- No human data besides rules of the game
- The value and policy network are trained in self-play **in the context of MCTS** instead of human data or without search
  - MCTS as a policy improvement operator!
- Trained on 4 TPUs for 70 days
  - Compared to tens of thousands of TPUs for Gemini



a. Self-Play
s_1
$a_1 \sim \pi_1$
s_2
$a_2 \sim \pi_2$
s_3
$a_t \sim \pi_t$
s_T
π_1
π_2
π_3
z
b. Neural Network Training
s_1
s_2
s_3
f_θ
f_θ
f_θ
$\mathbf{p}_1$
v_1
$\mathbf{p}_2$
v_2
$\mathbf{p}_3$
v_3
π_1
π_2
π_3
z

Neural Network Loss Function

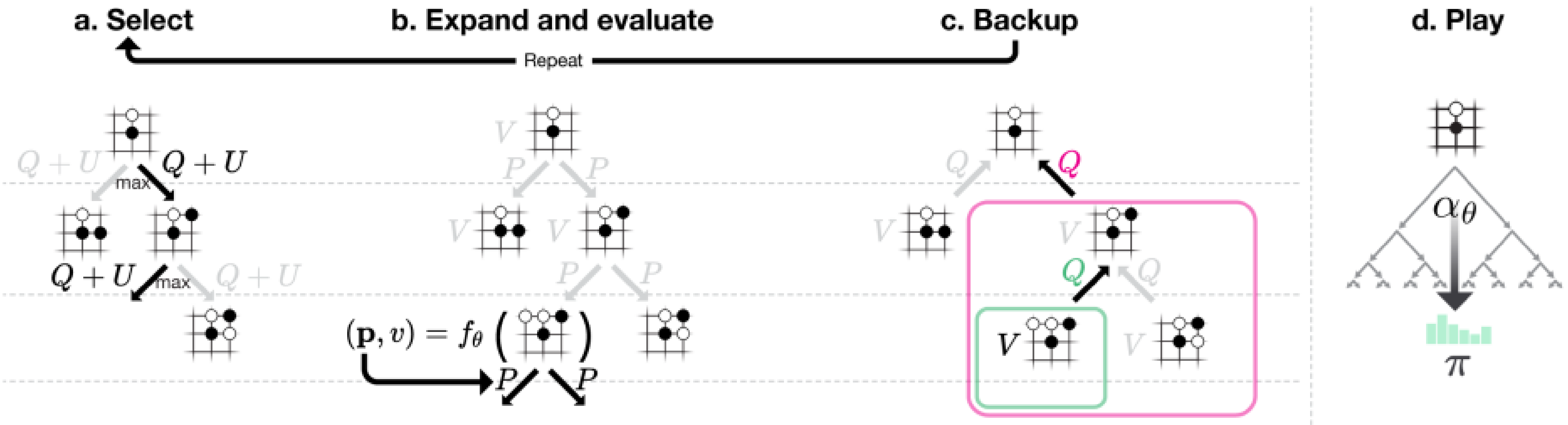
$$(\boldsymbol{p}, v) = f_\theta(s) \quad \text{and} \quad l = (z - v)^2 - \boldsymbol{\pi}^{\mathrm{T}} \log \boldsymbol{p} + c\|\theta\|^2$$

Value error

Maximise similarity of the neural network move probabilities p to the search probabilities π

Regularizer

Search Algorithm



Once the search is complete, search probabilities π are returned proportional to $N^{1/\mu}$, where N is the visit count of each move from the root state and μ is a parameter controlling temperature

Search Algorithm

- Each node s in the search tree contains edges (s, a) for all legal actions
- Each edge stores a set of statistics, {N(s, a), W(s, a), Q(s, a), P(s, a)}
    - N: number of visits to that edge
    - W: Total value
    - Q: Average value
    - P: Policy output

$$a_t = \operatorname{argmax}(Q(s_t, a) + U(s_t, a))$$

$$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

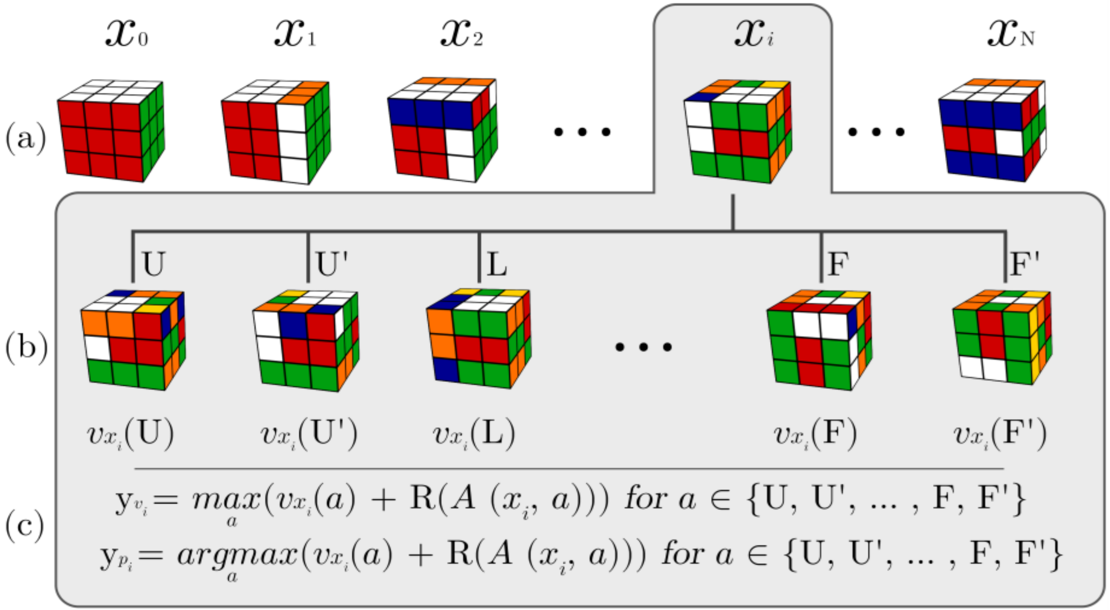
Expand and Evaluate

- When we reach a leaf node, we run the state through the neural network to get a value estimate and policy estimate
- Each edge (N, W, Q) is initialized to 0
- Backup value

Backup

- We update N, W, Q with the value that the neural network proposes
- $N(s, a) = N(s, a) + 1$
- $W(s,a) = W(s,a) + v$
- $Q(s, a) = W(s, a) / N(s, a)$

These Techniques are Useful Also in Single-Agent Settings



E.g.1: Rubik's cube
McAleer et al. "Solving the Rubik's cube with approximate policy iteration." *ICLR*. 2018.
Agostinelli et al. "Solving the Rubik's cube with deep reinforcement learning and search." *Nature Machine Intelligence.* 2019

E.g.2: Edge test selection in kidney exchange
McElfresh, Curry, Sandholm, Dickerson, "Improving Policy-Constrained Kidney Exchange via Pre-Screening", NeurIPS-20]