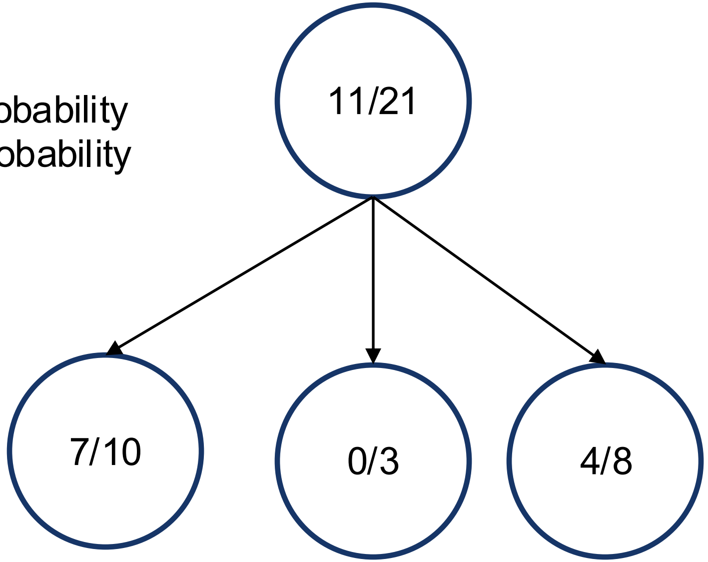
Monte Carlo Tree Search

Which root node should we select to run simulations on?

Wins/Total (for player 1)



Idea #1: Epsilon Greedy

Select random node with epsilon probability
Select best node with (1-epsilon) probability

Problem:

If we are very confident a node is bad, then we want 0 probability of expanding it.

Epsilon-greedy treats every action other than the best action equally.

Idea #2: Thompson Sampling

Sample action with probability proportional to value of each action (wins/total)

$$p(a_t) \propto e^{w/N}$$

Advantage: Prioritizes sampling actions based on quality (i.e., higher probability of simulating second best action than worst action)

Disadvantage: Does not take into account uncertainty

Multi-Arm Bandits

Multi-Arm Bandits

Single-armed bandit

Multi-Arm Bandits

Single-armed bandit

Multi-Arm Bandits

Single-armed bandit

Multi-Arm Bandits

Single-armed bandit

Multi-Arm Bandits

Single-armed bandit

Multi-Arm Bandits

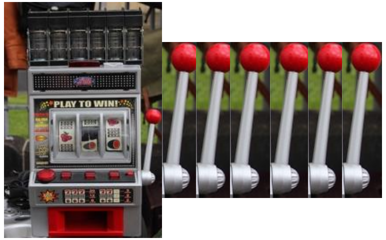
Single-armed bandit

Multi-Arm Bandits

Single-armed bandit

Multi-Arm Bandits

Single-armed bandit



Multi-Arm Bandits

Single-armed bandit

Multi-Arm Bandits

When an arm is pulled, the rewards are random.

Each arm returns a reward with (different) unknown mean and variance

Single-armed bandit

Multi-Arm Bandits

When an arm is pulled, the rewards are random.

Each arm returns a reward with (different) unknown mean and variance

Bandit Problems are essentially MDPs with a single state

Single-armed bandit

How can get as much reward as possible over N pulls?

Multi-Arm Bandits

When an arm is pulled, the rewards are random.

Each arm returns a reward with (different) unknown mean and variance

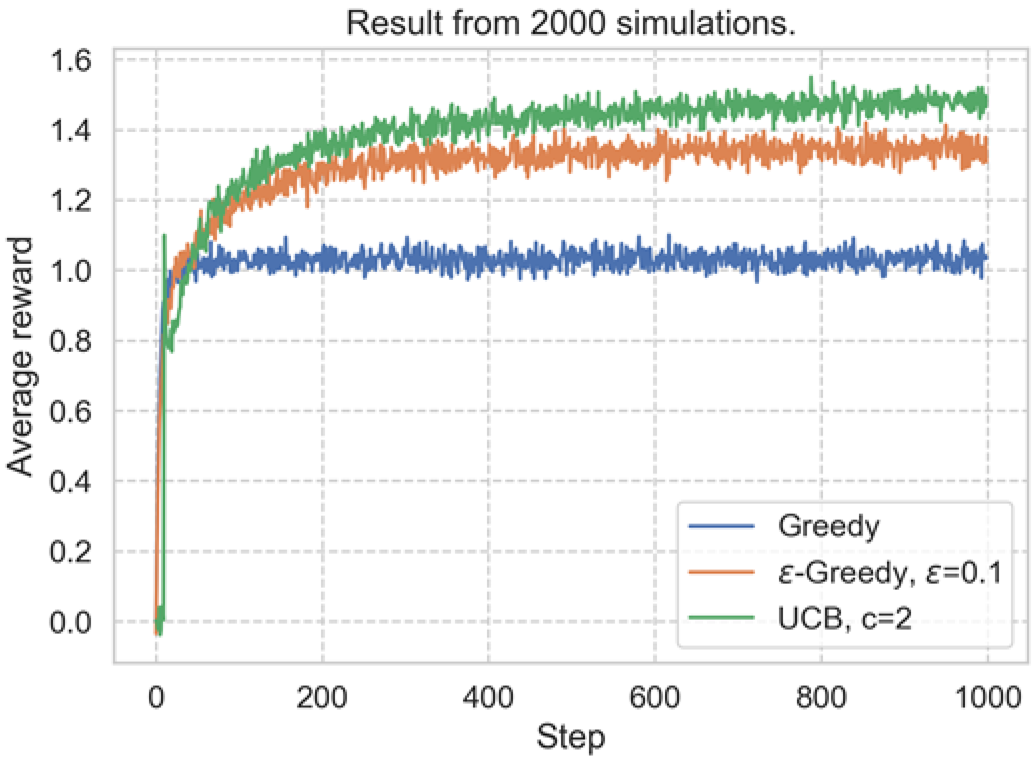
Bandit Problems are essentially MDPs with a single state

Single-armed bandit

Which node to select in MCTS is a Bandit problem!

Each action returns a random result and we'd like to select the best action as frequently as possible.

Performance of Different Exploration Policies in Multi-Armed Bandits



Upper Confidence Bound (UCB)

Select action according to:

$$UCB(t) = \underset{\mathrm{i}}{\operatorname{argmax}} \; \hat{\mu}_i + \sqrt{\frac{C\ln(\mathrm{t})}{n_i}}$$

Some constant times the log of number of rounds

Action at round t

Action i that maximizes…

Empirical mean of that action so far

Number of times action i has been executed

Upper Confidence Bound (UCB)

Select action according to:

$$UCB(t) = \underset{\mathrm{i}}{\operatorname{argmax}} \quad \hat{\mu}_i + \sqrt{\frac{C\ln(t)}{n_i}}$$

How good that action is

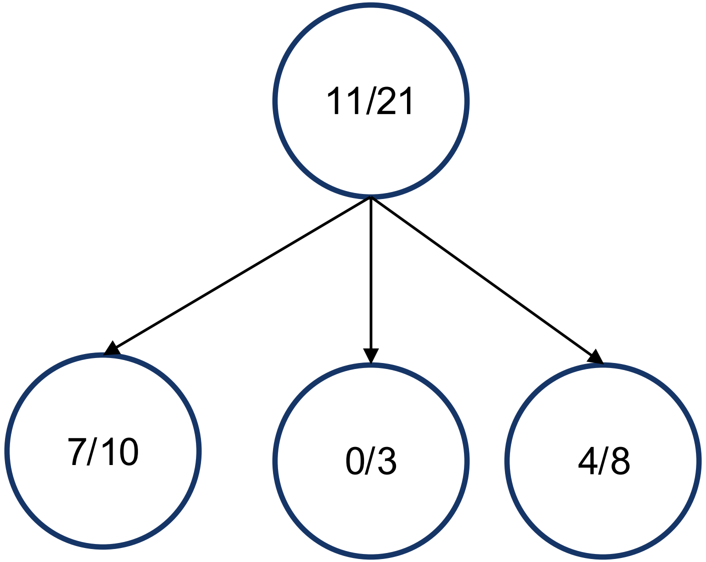
How "explored" that action is compared to other actions

When t is large and n_i is small, this term is larger. Action is more likely to be selected.

UCB is near-optimal for Multi-Armed Bandits problems

Which root node should we select to run simulations on?

Using C=2 (common choice)



UCB score = $7/10 \ + \sqrt{\frac{2 \cdot ln(21)}{10}}$

$= \frac{7}{10} + .61$

UCB score = $3/8 \ + \sqrt{\frac{2 \cdot ln(21)}{8}}$

$= 4/8 + .87$

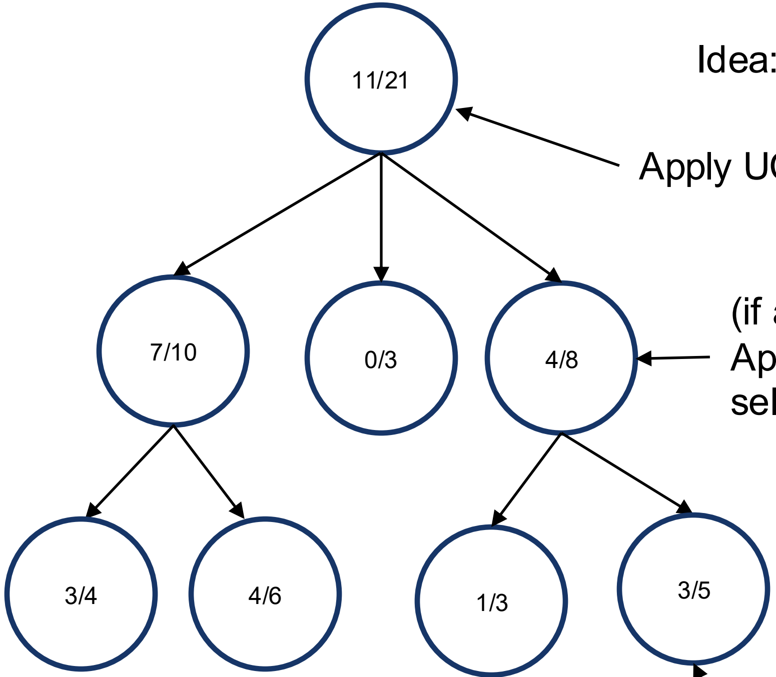
UCB score = $0/3 \ + \sqrt{\frac{2 \cdot ln(21)}{3}}$

$= 0 + 1.42$

UCB selects action 2

MCTS will iteratively grow the search tree, but not every branch will be grown at the same rate.

How can we select between root nodes to run simulations?

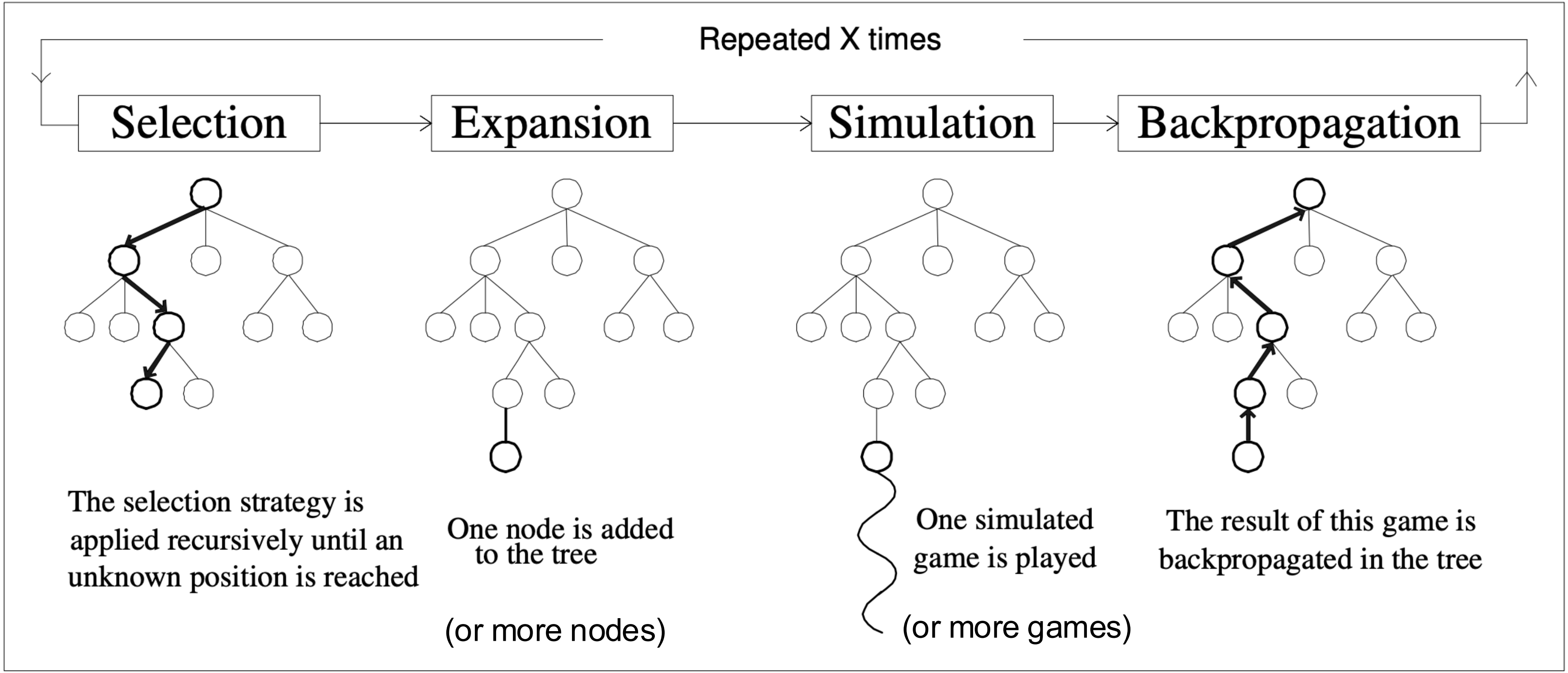


Idea: Recursively apply UCB

Apply UCB to select action from root

(if action 3 is chosen)
Apply UCB at new node to select between children

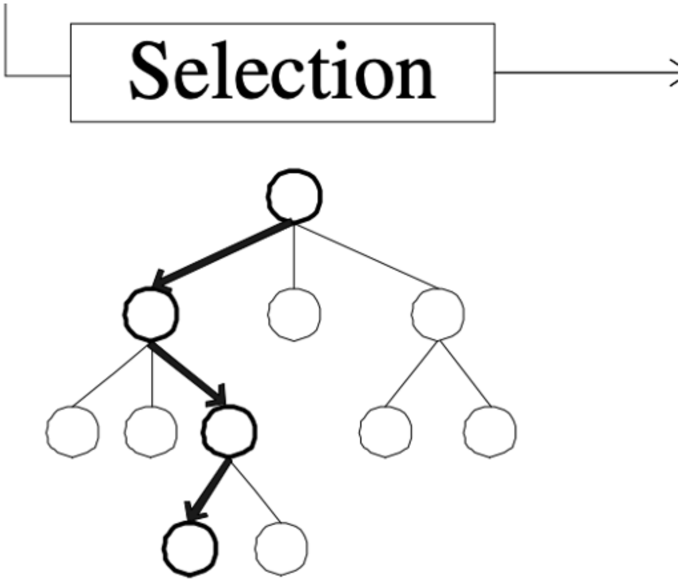
Eventually, a root node is selected



Source: Chaslot, Monte-Carlo Tree Search. 2006

At each node in search tree, what action should we take?

Selection strategy balances learning more about best action and learning more about uncertain actions



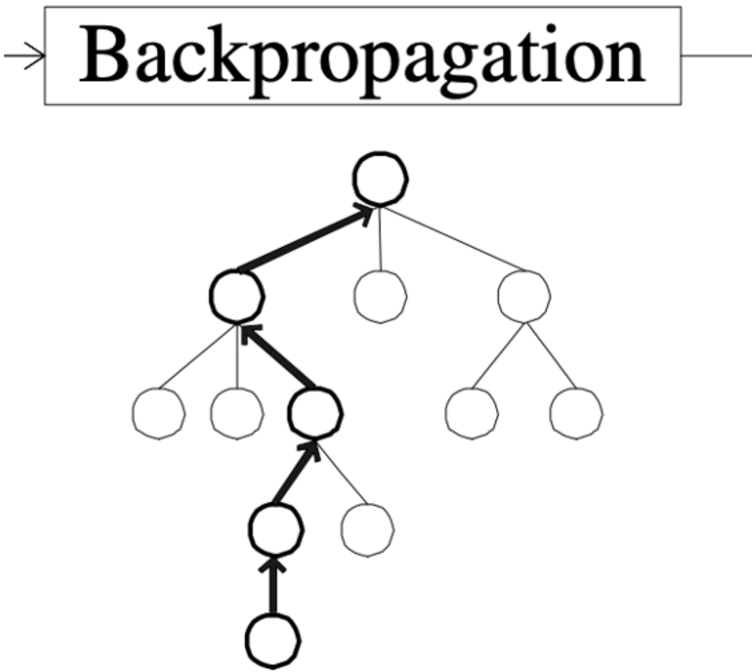
The selection strategy is applied recursively until an unknown position is reached

MCTS Tree Nodes

Each node tracks:

1. Number of simulated wins from node or children of node
2. Total number of simulations
3. Parents and Children of node

Backpropagation updates total number of wins and simulations for parent nodes



The result of this game is backpropagated in the tree

Algorithm 1 Monte-Carlo-Tree-Search(state, π_S) $\rightarrow$ action

$tree \leftarrow MCTSNode(state)$
while time-remaining **do**
    $leaf \leftarrow SELECT(tree, \pi_S)$
    $children \leftarrow EXPAND(leaf)$
    $result \leftarrow SIMULATE(children)$
    $BACKPROPAGATE(results, children)$
end while

Return: Best Action

Algorithm 2 SELECT(state, π_S)

$currNode \leftarrow state$
while $!isLeaf(currNode)$ **do**
    $currNode \leftarrow \pi_S(currNode.children)$
    $currNode.visits \leftarrow currNode.visits + 1$
end while
return $currNode$

Algorithm 3 Expand(leaf)

```
children ← []
state ← leaf.state
actions ← state.legalActions()
for action in actions do
    children.append(transition(state, action))
end for
return children
```

Algorithm 4 SIMULATE(children)

$results \leftarrow [\,]$

for child in children **do**

  $result = rollout(child)$

  $results.append(result)$

end for

return $results$

Algorithm 5 BACKPROPAGATE (results, children)

Input: A new leaf node (children) and simulation result for each new leaf node (results)

```
for child in children, result in results do
    currNode ← child
    while currNode != NULL do
        currNode.visits ← currNode.visits +1
        if result == WHITE–WIN & currNode.player == BLACK then
            currNode.value ← currNode.value +1
        else if result == BLACK–WIN & currNode.player == WHITE then
            currNode.value ← currNode.value +1
        end if
        currNode ← currNode.parent
    end while
end for
```

Algorithm 1 Monte-Carlo-Tree-Search(state, π_S) $\rightarrow$ action

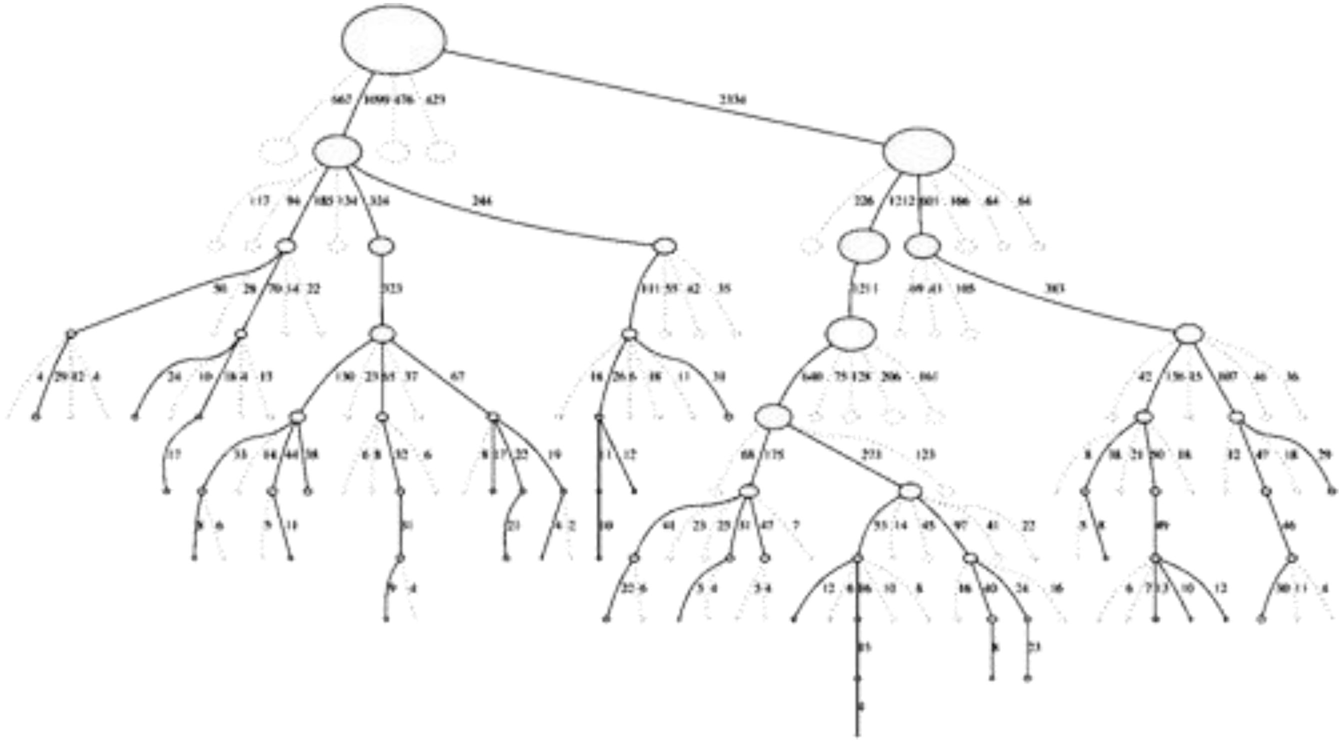
$tree \leftarrow MCTSNode(state)$
while time-remaining **do**
    $leaf \leftarrow SELECT(tree, \pi_S)$
    $children \leftarrow EXPAND(leaf)$
    $result \leftarrow SIMULATE(children)$
    $BACKPROPAGATE(results, children)$
end while
return The action of the node in children(tree) with the highest number of visits

MCTS Search Tree

(Ideally)

MCTS automatically searches branches with better expected outcomes

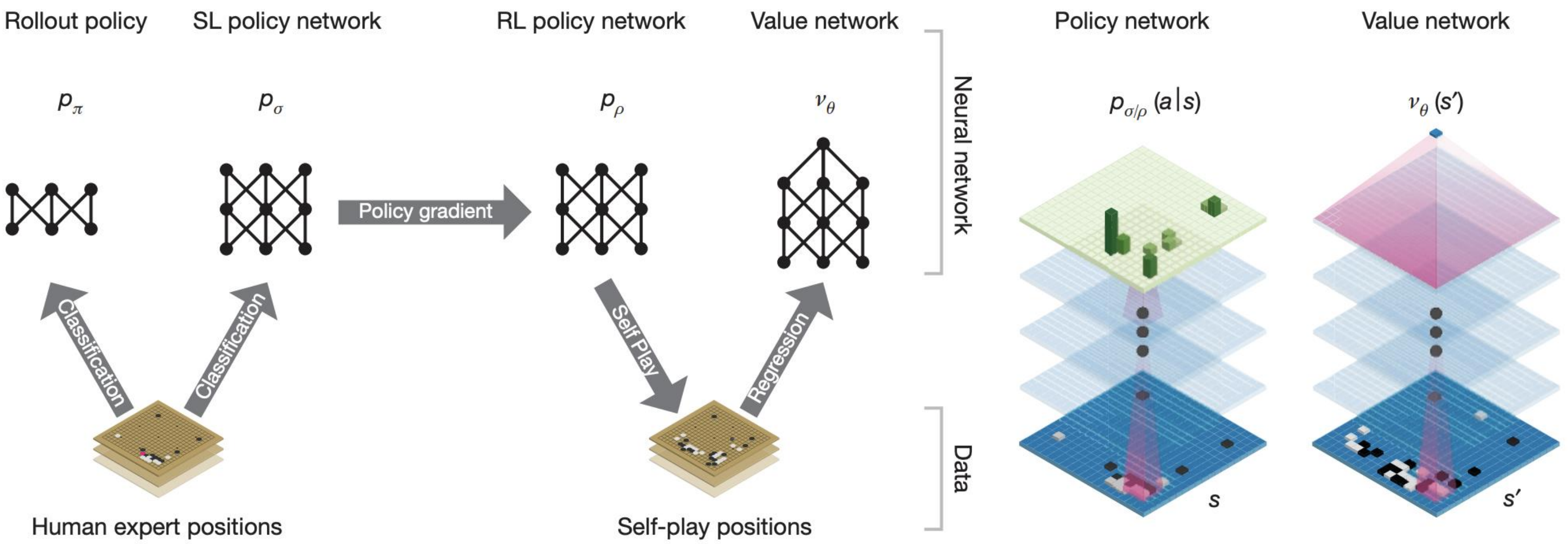
MCTS trees go deeper in promising directions and remain shallow in poor quality branches



Further Reading

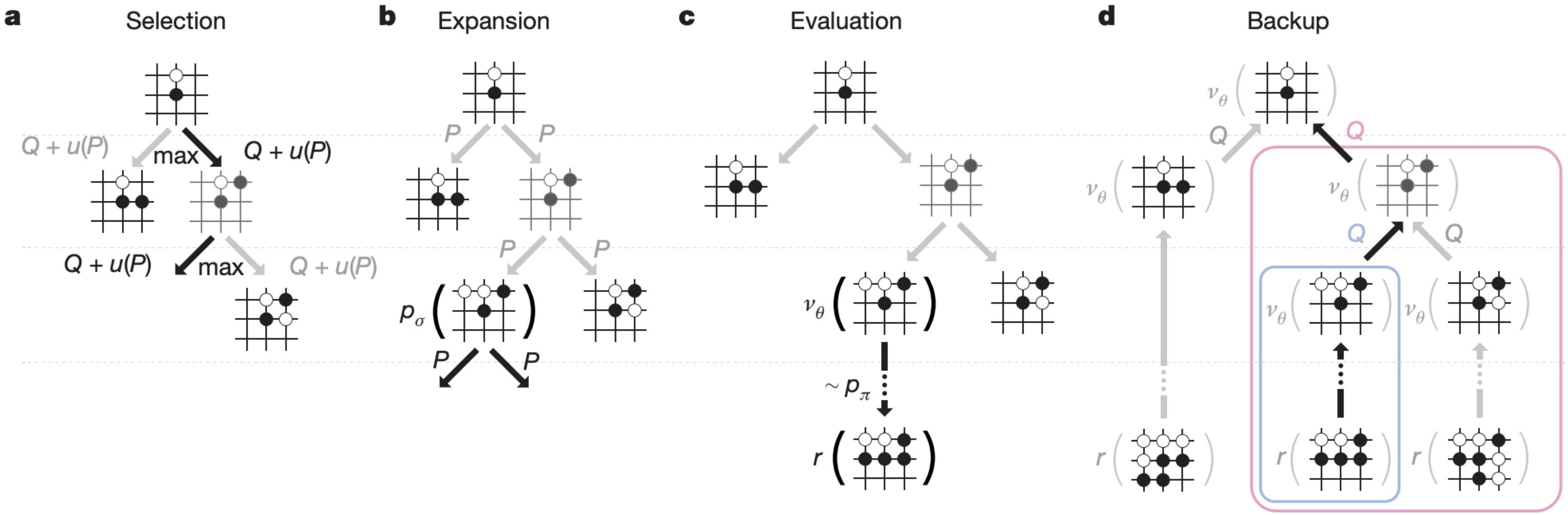
Monte-Carlo Tree Search, Guillaume Chaslot’s Dissertation, 2006

AlphaGo

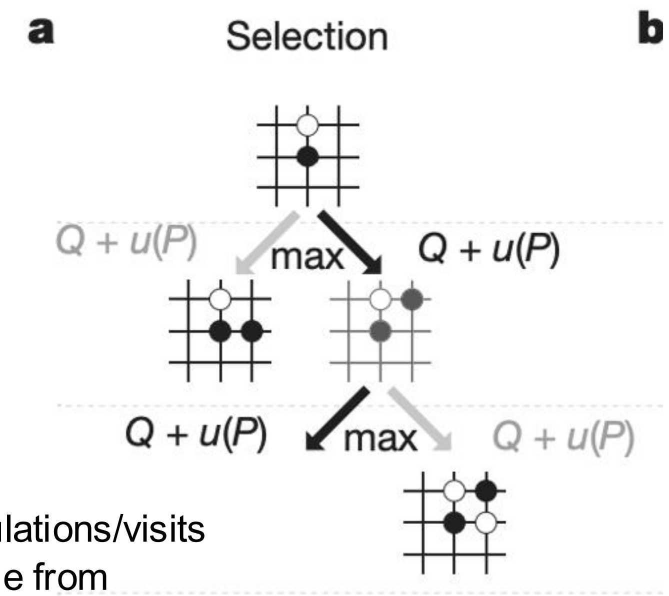


Learn a Value function and a Policy function, use in MCTS

MCTS in AlphaGo



Selection



$$a_t = \operatorname*{argmax}_a (Q(s_t, a) + u(s_t, a))$$

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$$

Policy Network

Total number of simulations/visits

$$N(s,a) = \sum_{i=1}^{n} 1(s,a,i)$$

Average value from that state (including value estimated with Value network and result of simulations)

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s_L^i)$$

Value Network

Simulation result

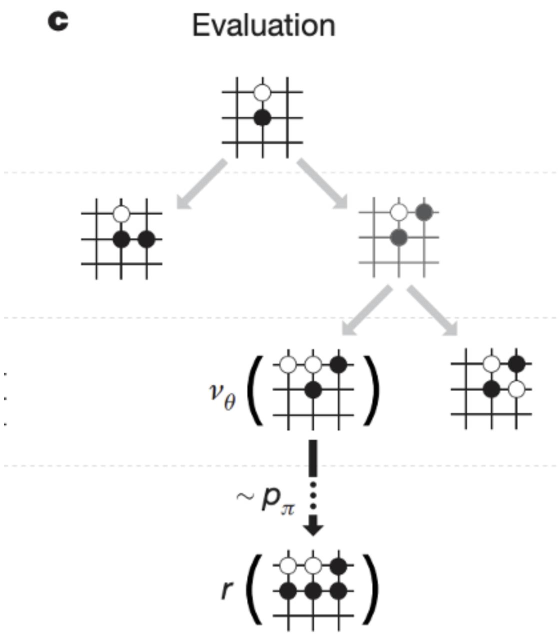
$$V(s_L) = (1-\lambda) v_\theta(s_L) + \lambda z_L$$

Evaluation

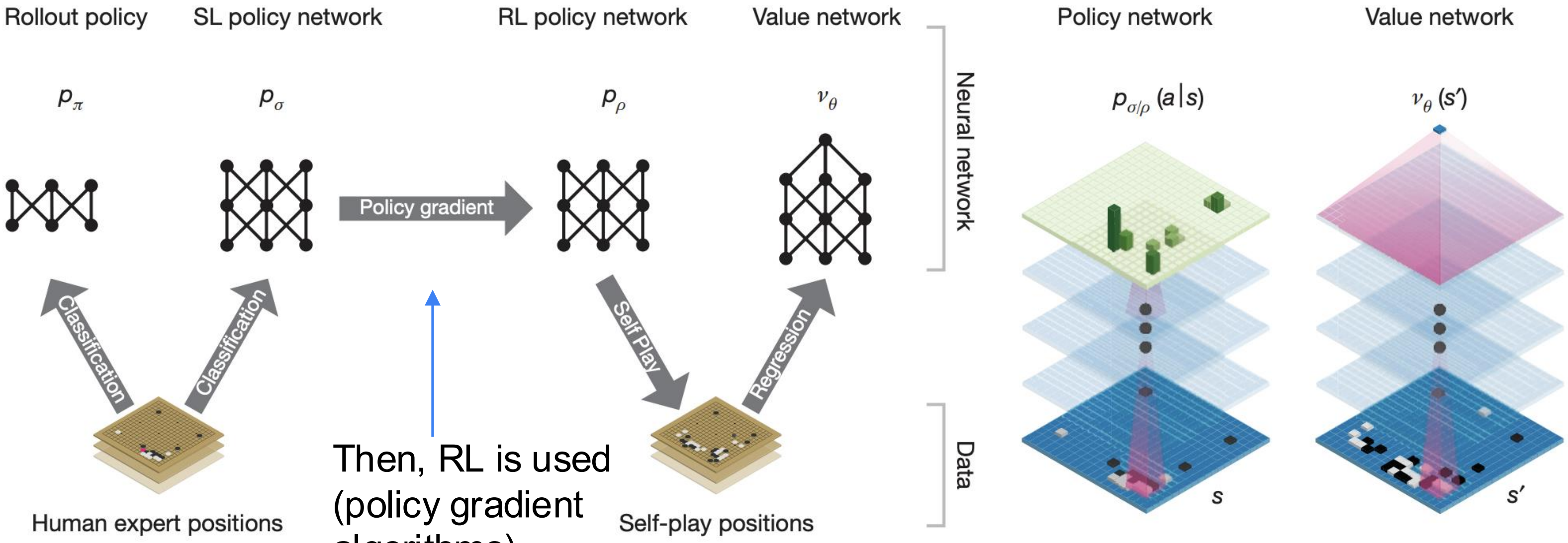
Instead of running random simulations, use a rollout policy p_π.

This is a **simple and fast** rollout policy.

It doesn't need to be perfect, it just needs to be fast. Anything better than random will help.



Training Value and Policy Networks



Then, RL is used (policy gradient algorithms)

Initially trained with database of human games

Policy Network: Predict Actions
Value Network: Predict Values (win/lose from position)

Self-Play

How do you set up a RL environment for Go? Who is your opponent?

Yourself!

AlphaGo repeatedly played against itself and improved its parameters using Reinforcement Learning

AlphaGo

Could defeat the European Champion!