CS 380: Artificial Intelligence

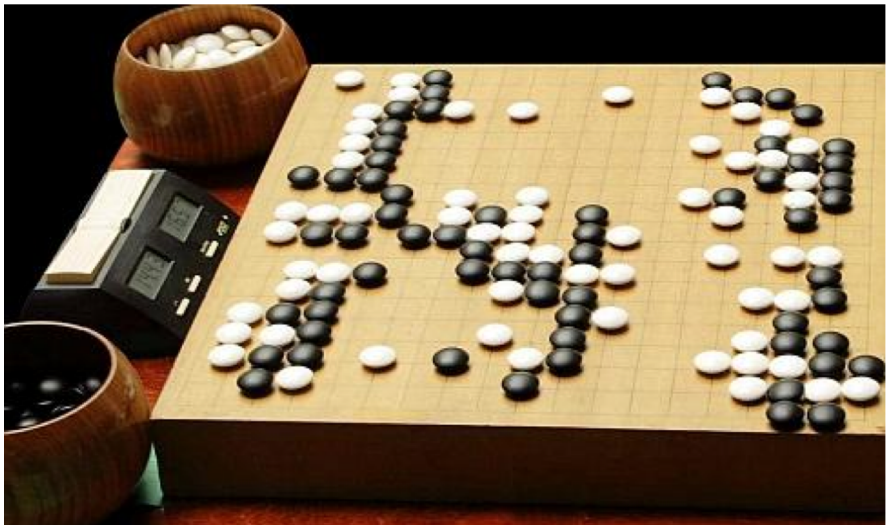
Lecture 7: Adversarial Search II

Recall: Adversarial Search

- Idea:
  - When there is only one agent in the world, we can solve problems using DFS, BFS, ID, A*, etc.
  - When more than one agent, we need adversarial search…
- Last time we saw:
  - Minimax:
    - Systematically expands a "game tree" (can bound depth)
    - Apply eval function at leaves, and then back-propagate values, to find best action
  - Alpha-beta pruning:
    - Improvement over minimax to expand less nodes
  - Expectiminimax:
    - Extend minimax with chance nodes when we have nondeterminism
    - Can still use Alpha-beta pruning

Minimax in Practice

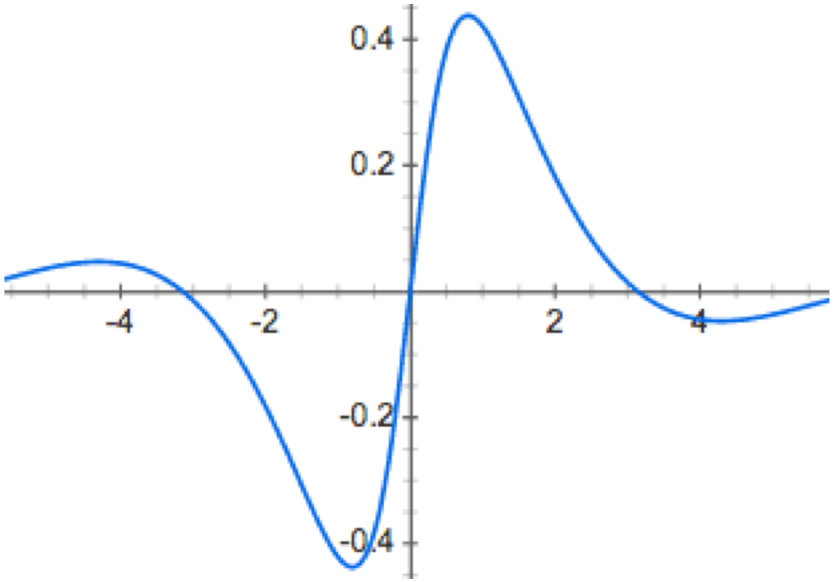
- Minimax (with alpha-beta pruning) can be used to create AI that plays games like Checkers or Chess.
- Problem: There are games that humans can play that are too complex for Minimax.
  - For example: Go



Monte Carlo Methods

- Algorithms that rely on random sampling to find solution approximations.
- Example: Monte Carlo integration
  - Imagine that I ask you to compute the following:

$$A = \int_1^3 sin(x) \frac{1}{1 - x^2} dx$$



Monte Carlo Methods

- Method 1: Symbolic integration
  - You could fetch your calculus book, integrate the function, etc.
  - But you'd need to do this method by hand (did you know that automatic symbolic integration is still unsolved?)
- Method 2: Numerical computations
  - Simpson's method, etc. (recall from calculus?)
- Method 3: Monte Carlo

Monte Carlo Methods

- Method 3: Monte Carlo

$$f(x) = sin(x)\frac{1}{1-x^2}dx$$

  - Repeat *N* times:
    - Pick a random *x* between 1 and 3
    - Evaluate *f*(*x*)
  - Now average and multiply by 2 (due to 3 – 1)
  - Voilà!
  - The larger the *N*, the better the approximation

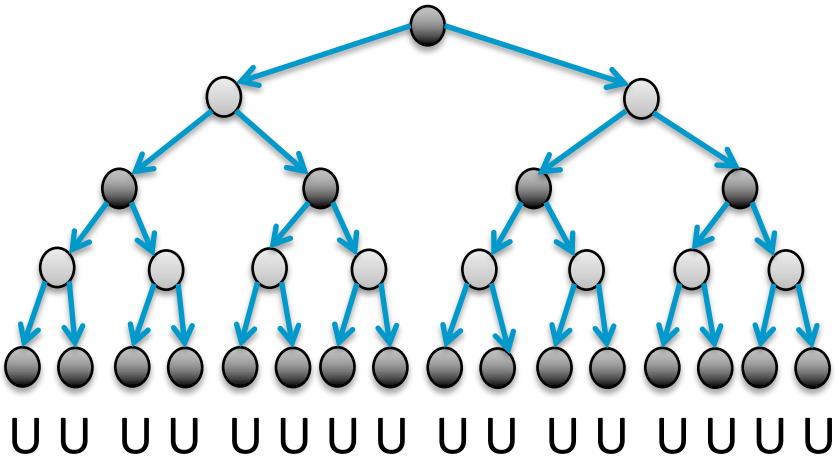
Monte Carlo Methods

- Idea:
  - Use random sampling to approximate the solution to complex problems

- How can we apply this idea to adversarial search?
  - The answer to this question is why we have computer programs that can play Go at a master level.

Minimax vs Monte Carlo Search

Minimax:

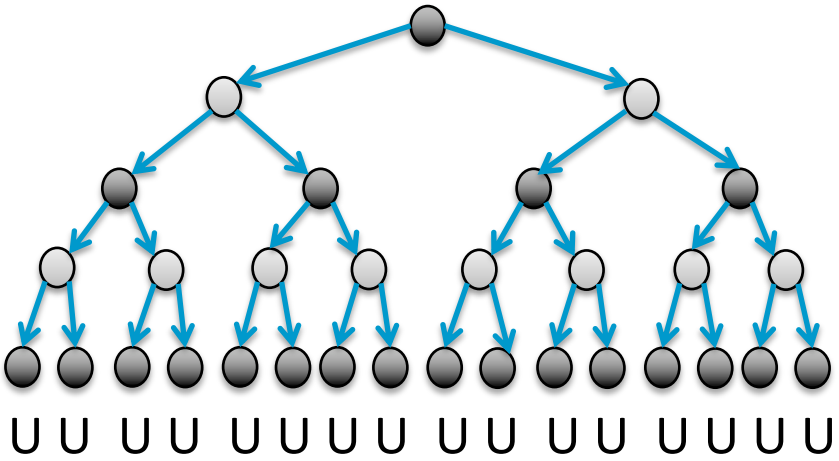


Minimax opens the complete tree (all possible moves) up to a fixed depth.
Then, a Utility function is applied to the leaves.

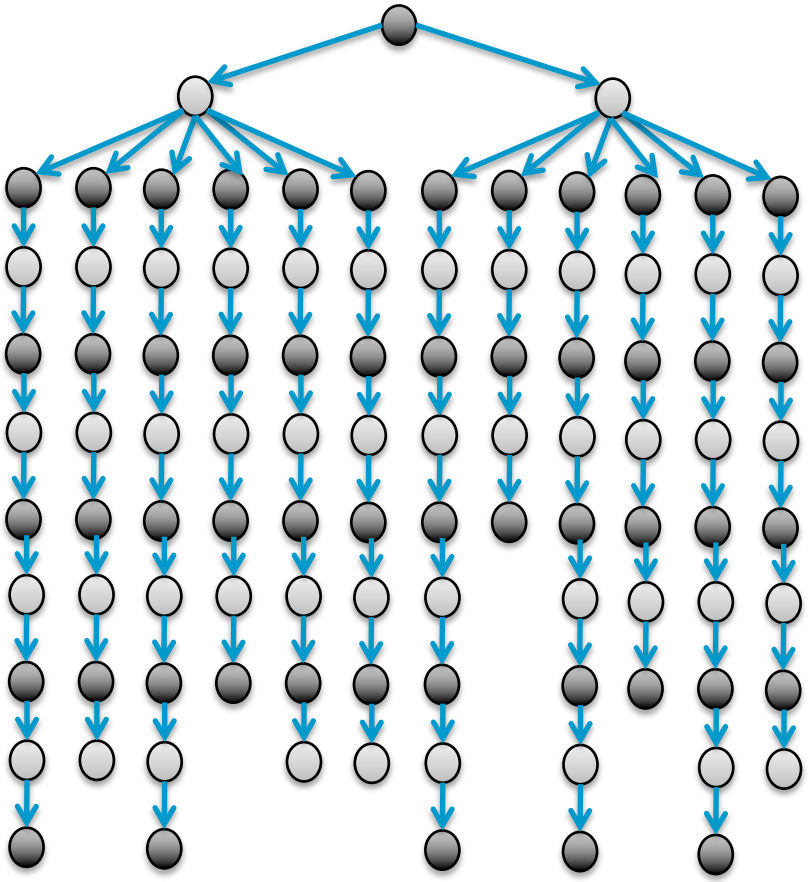
Monte-Carlo:

Minimax vs Monte Carlo Search

Minimax:

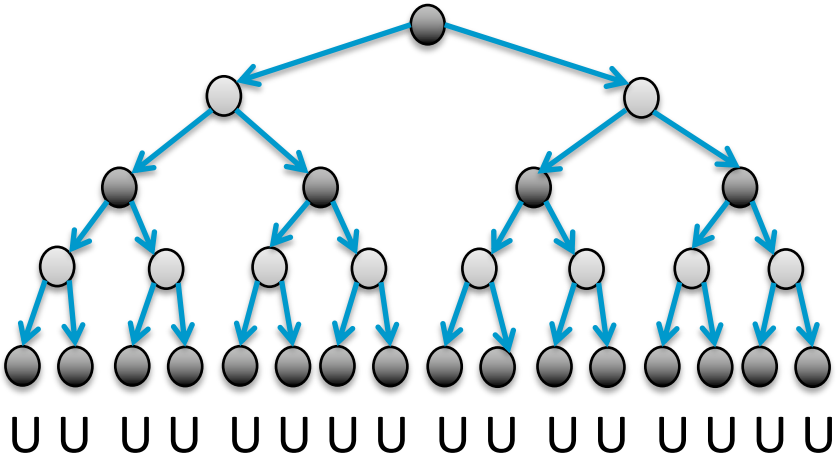


Monte-Carlo:

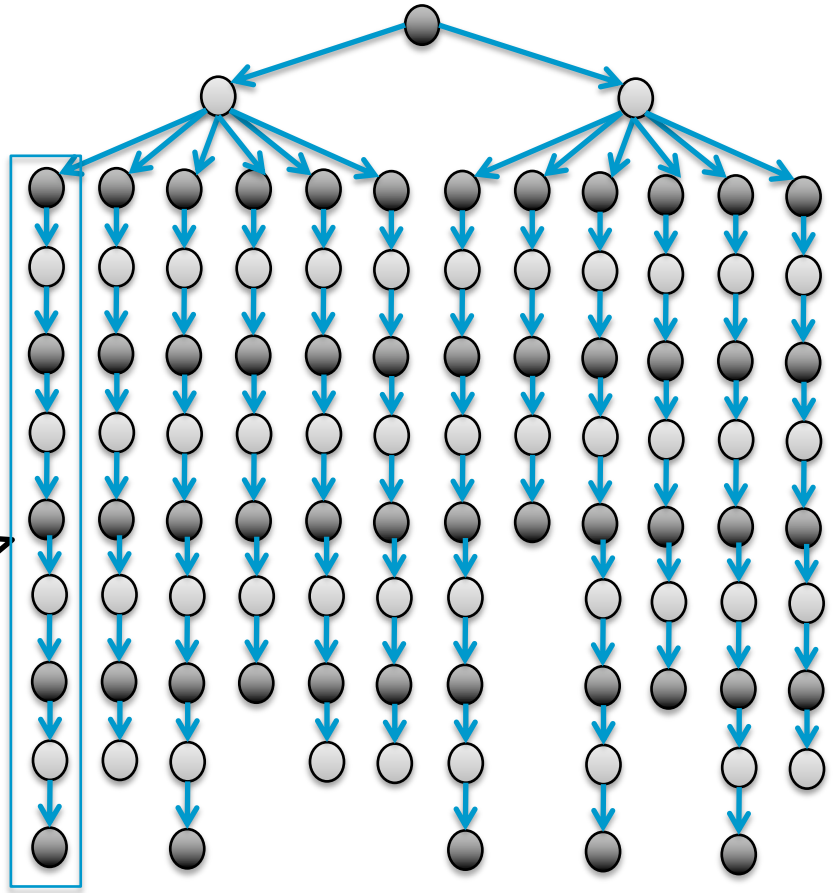


Minimax vs Monte Carlo Search

Minimax:



Monte-Carlo:



Monte-Carlo search runs for each possible move at the root node a fixed number *K* of random complete games.
No need for a Utility function (but it can be used)

Complete Game

Monte Carlo Search

- For each possible move:
  - Repeat *N* times:
    - Play a game until the end, selecting moves at random
  - Count the percentage of wins
- Select the action with the highest percentage of wins.

- Properties:
  - **Complete**: ?
  - **Optimal**: ?
  - **Time**: ?
  - **Memory**: ?

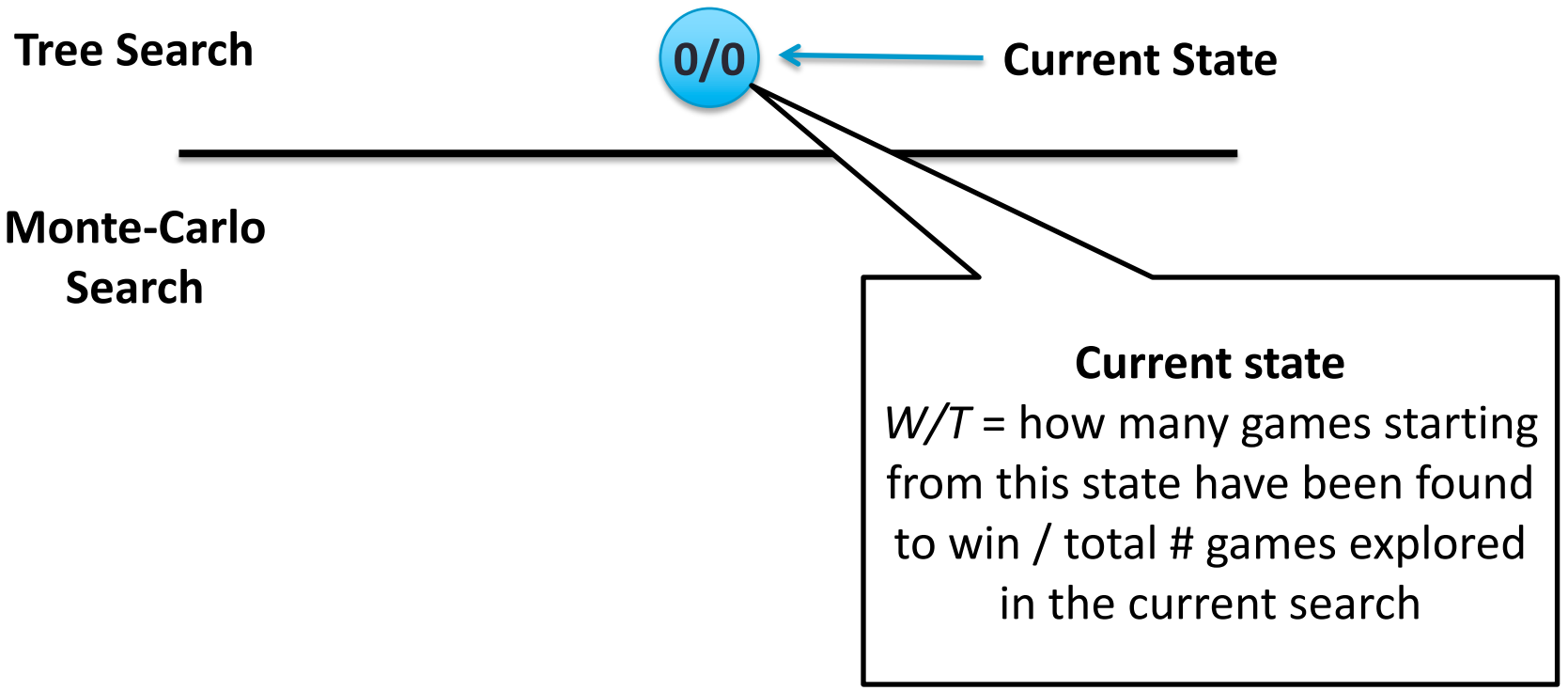
Monte Carlo Search

- For each possible move:
  - Repeat *N* times:
    - Play a game until the end, selecting moves at random
  - Count the percentage of wins
- Select the action with the highest percentage of wins.

- Properties:
  - **Complete**: no
  - **Optimal**: no
  - **Time**: d*b*n
  - **Memory**: b

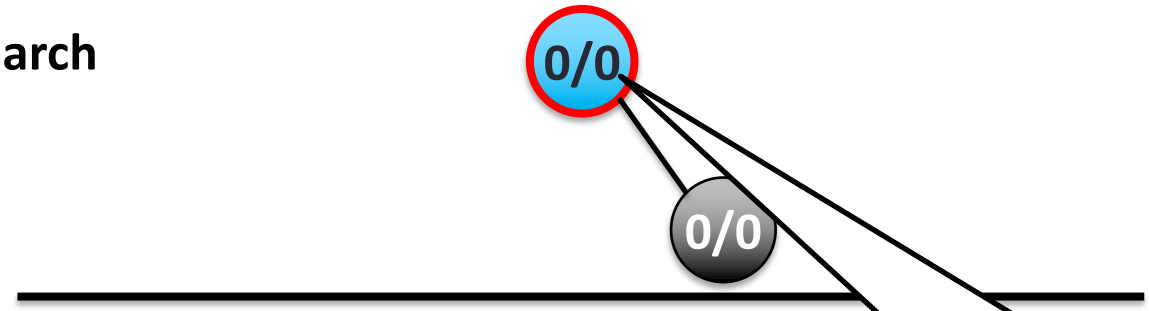
- Works much better than minimax for large games, but has many problems. We can do better.

Monte Carlo Tree Search



Monte Carlo Tree Search

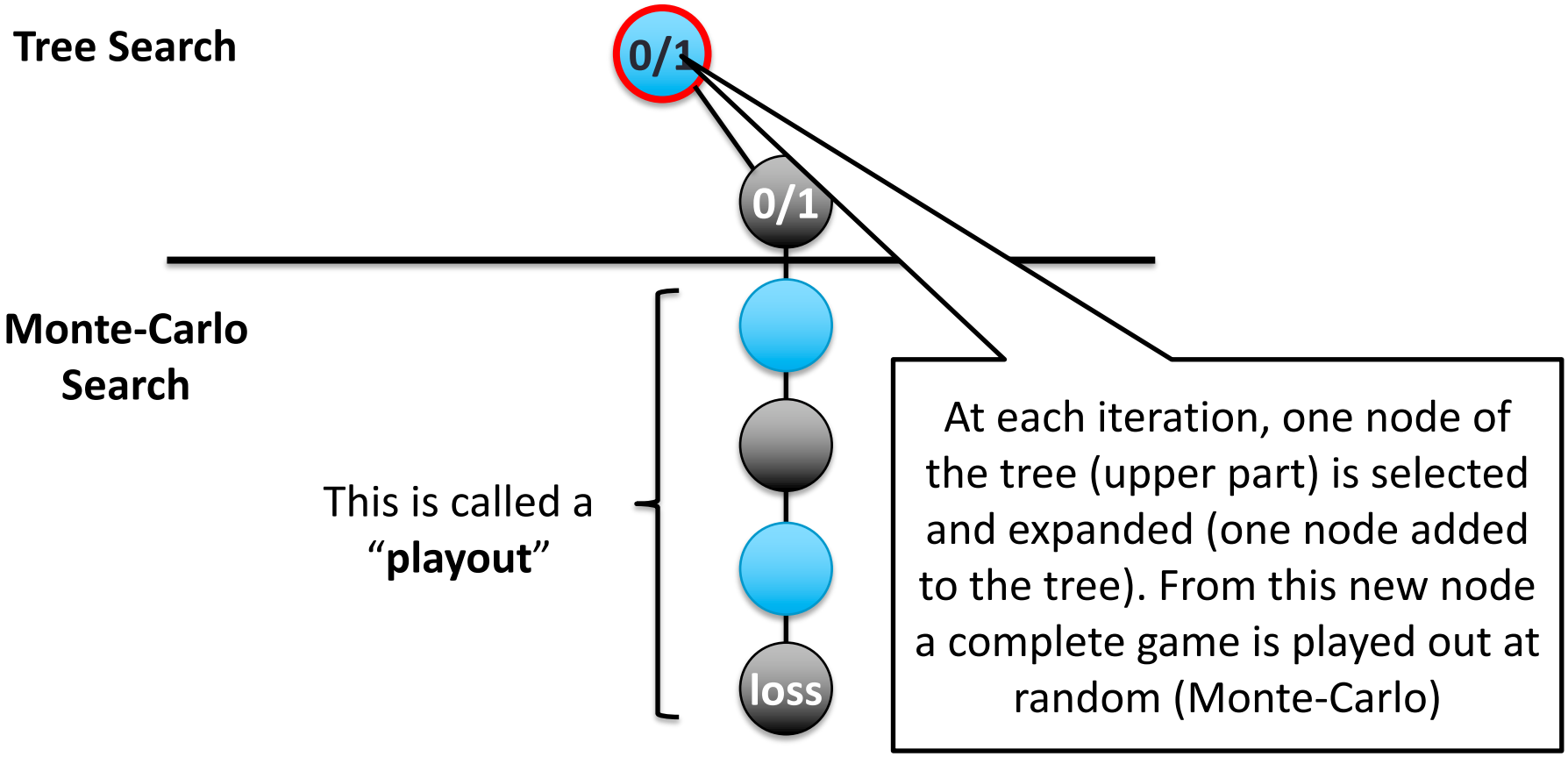
Tree Search



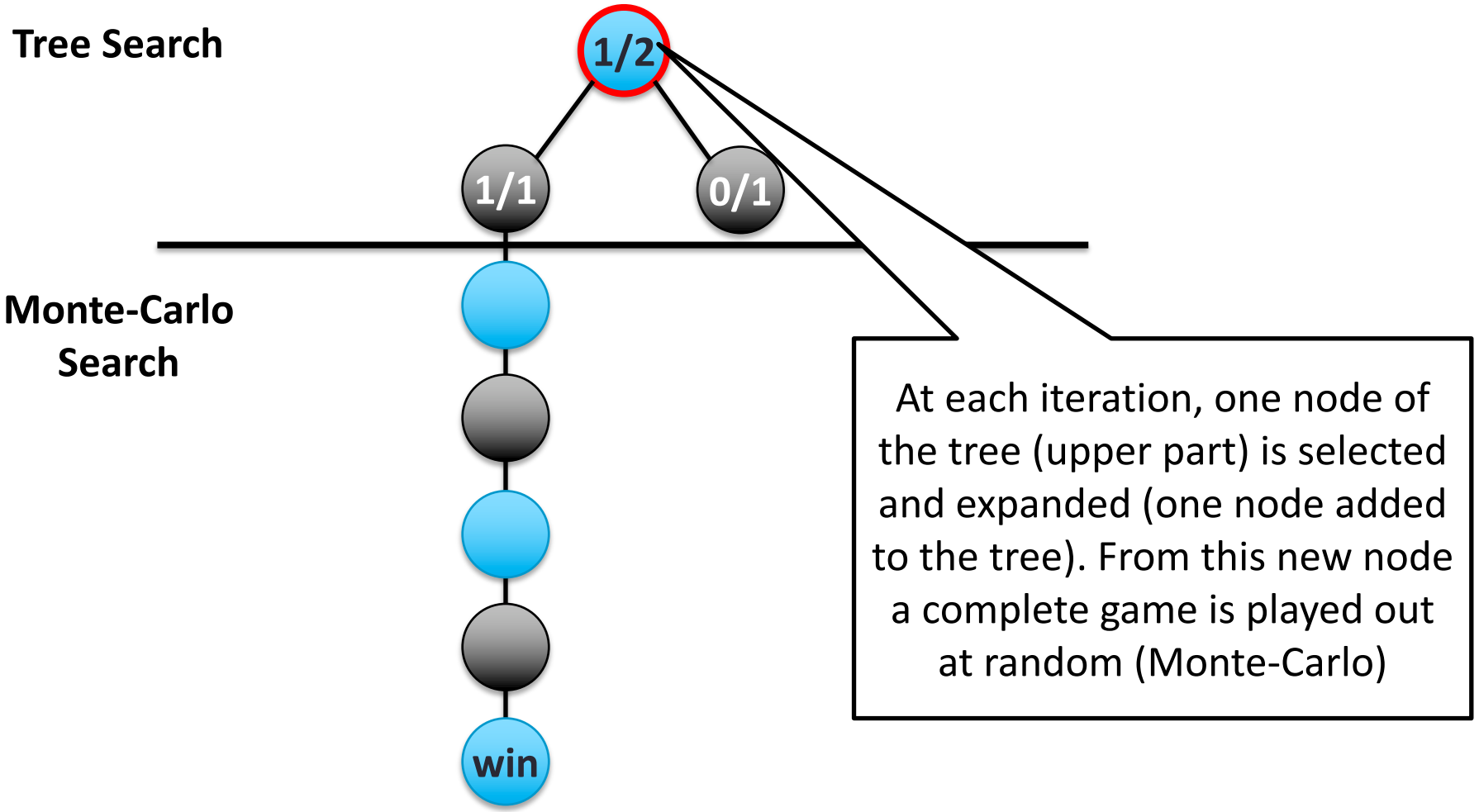
Monte-Carlo Search

At each iteration, one node of the tree (upper part) is selected and expanded (one node added to the tree). From this new node a complete game is played out at random (Monte-Carlo)

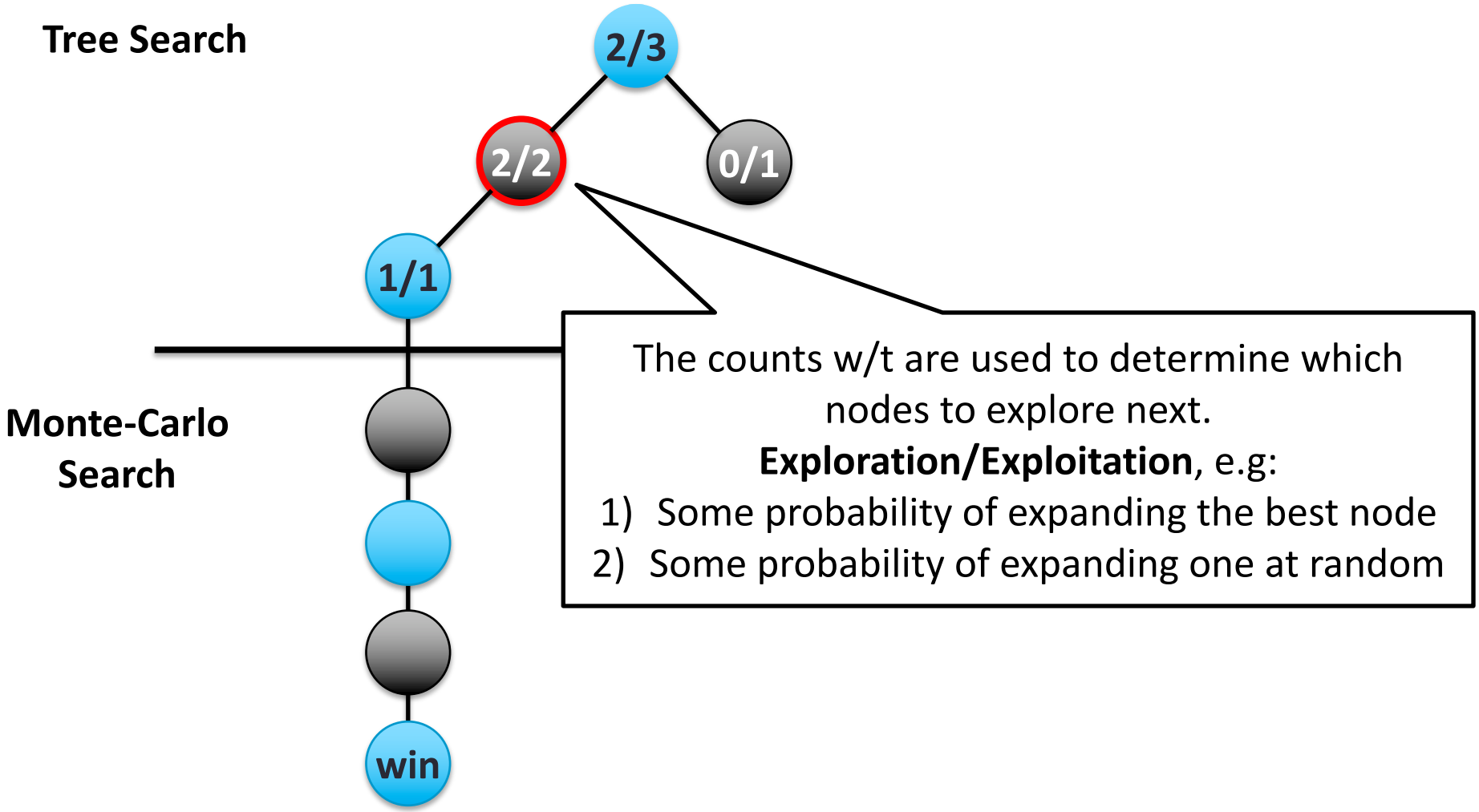
Monte Carlo Tree Search



Monte Carlo Tree Search

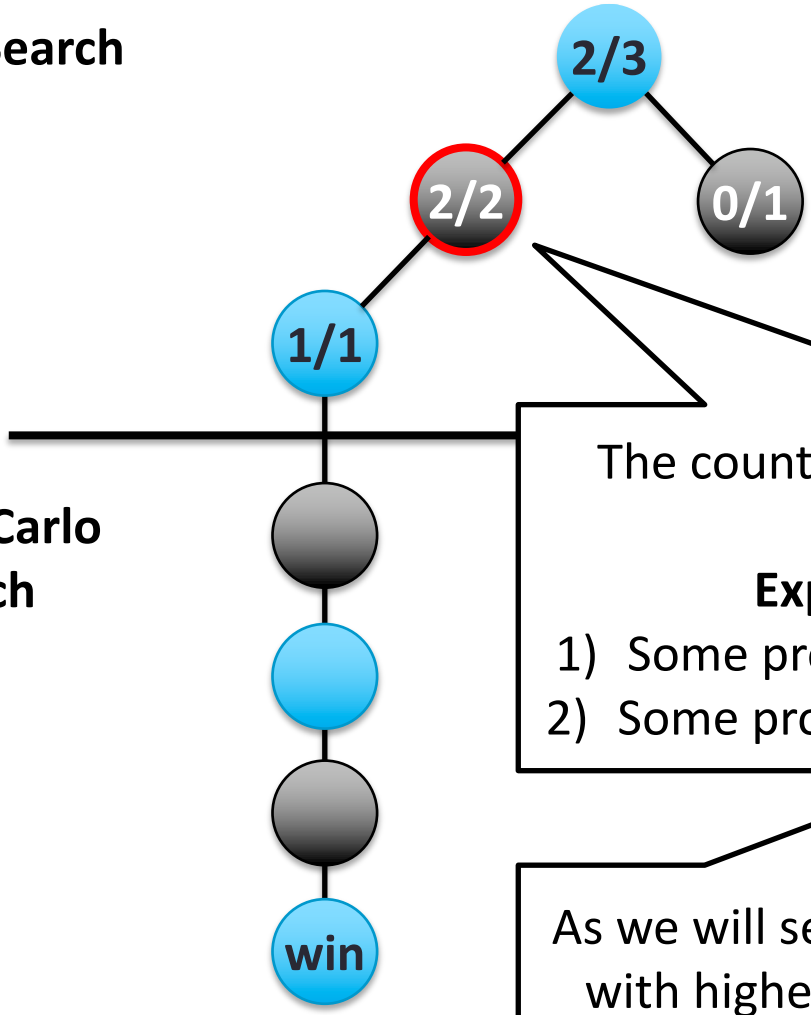


Monte Carlo Tree Search



Monte Carlo Tree Search

Tree Search

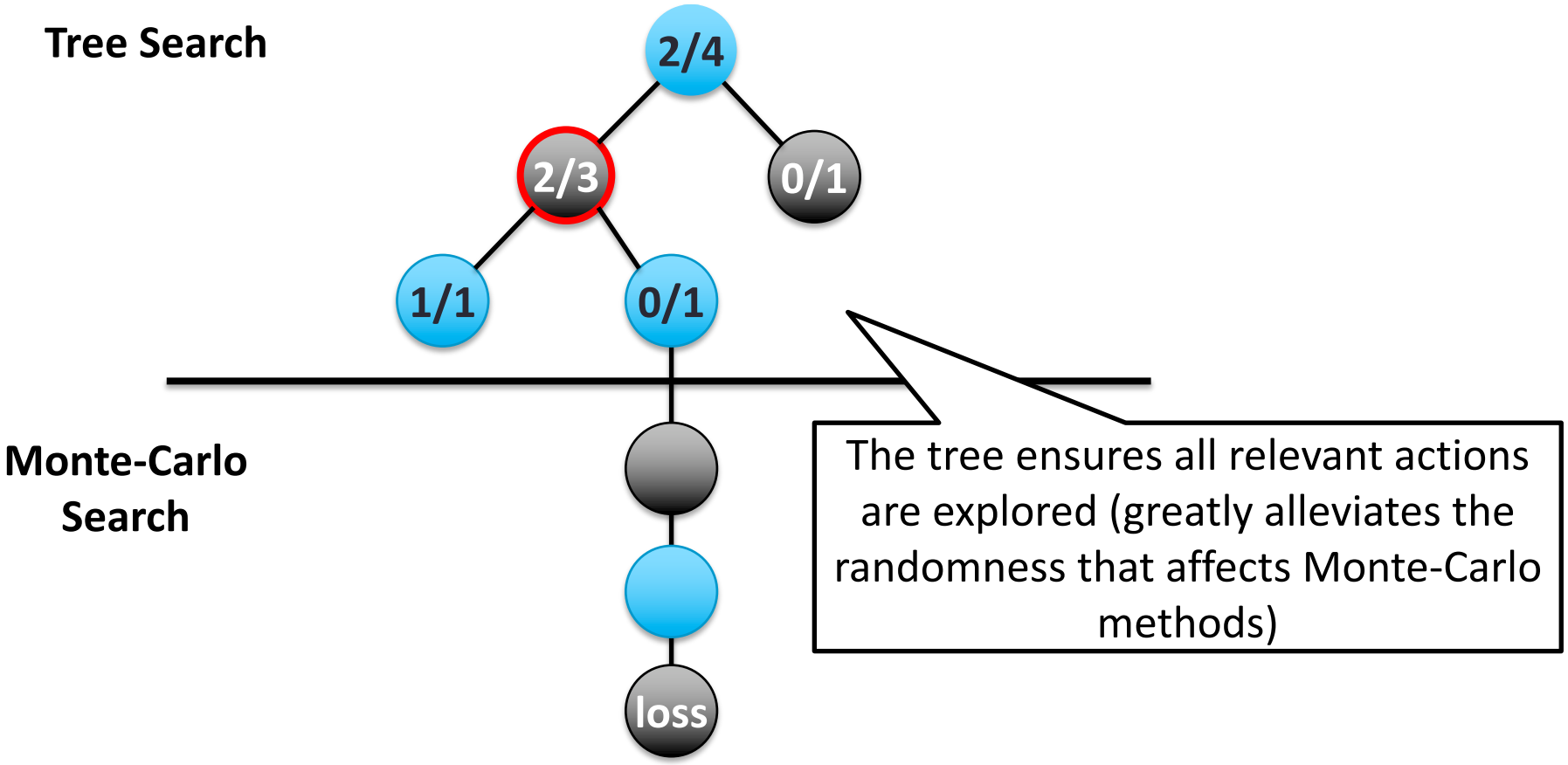


Monte-Carlo Search

The counts w/t are used to determine which nodes to explore next.
Exploration/Exploitation, e.g:
1) Some probability of expanding the best node
2) Some probability of expanding one at random

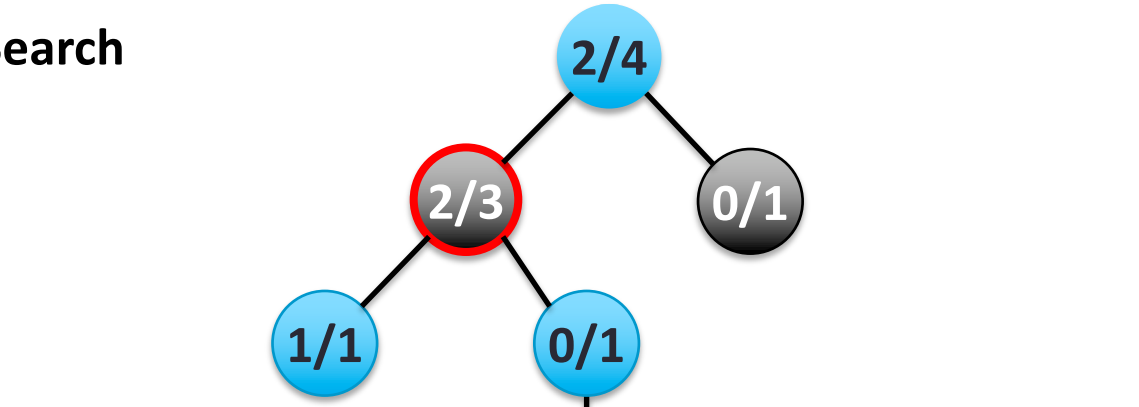
As we will see, we want to expand the best node with higher probability than any of the others

Monte Carlo Tree Search



Monte Carlo Tree Search

Tree Search



Monte-Carlo Search

The random games played from each node of the tree serve to estimate the Utility function.
They can be random, or use an opponent model (if available)

MCTS Algorithm

```
MCTS(state, player)
  tree = new Node(state, player)
  Repeat until computation budget is exhausted
    node = treePolicy(tree)
    if (node.isTerminal) child = node
                         else child = node.nextChild();
    R = playout(child)
    child.propagateReward(R)
  Return tree.bestChild();
```

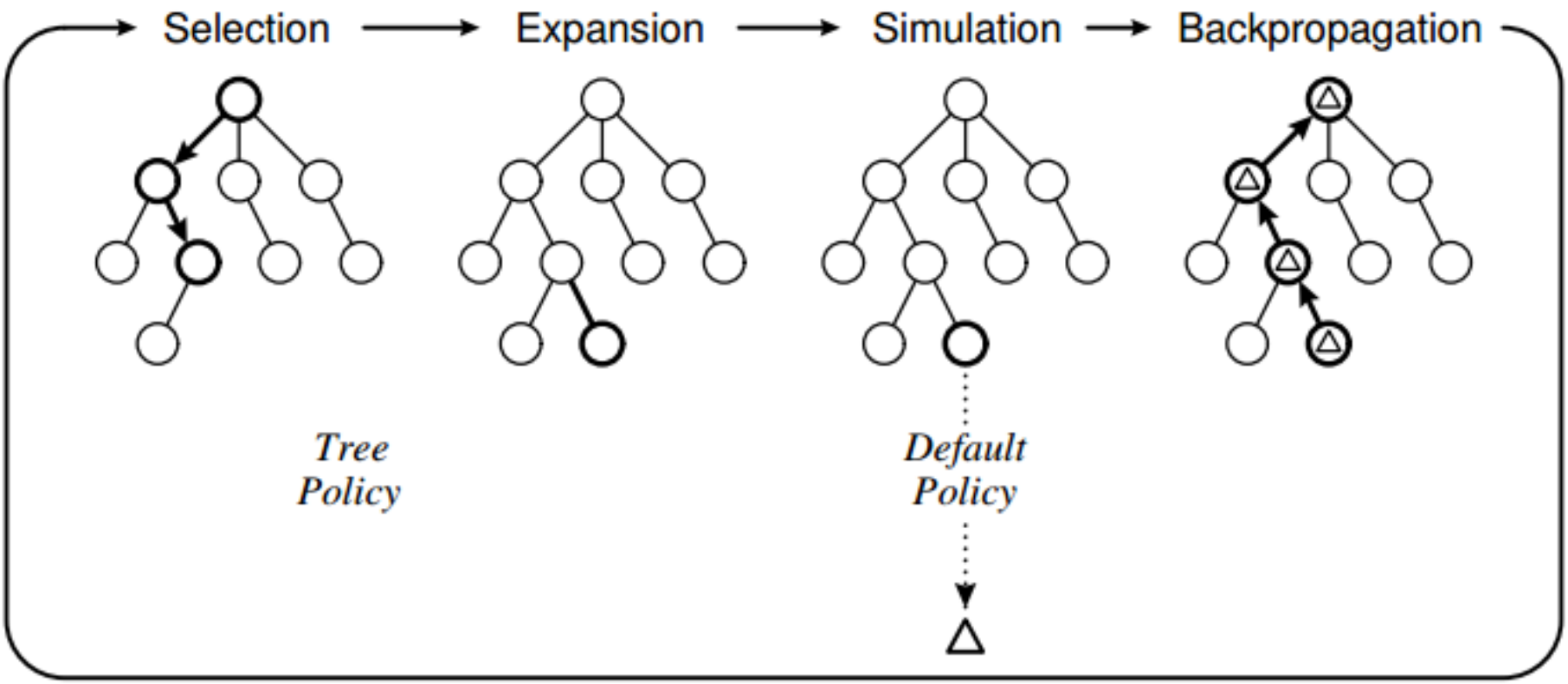
Monte Carlo Tree Search

- Question: How do we choose the next node to be added to the tree?
  - Start at the root.
  - Descend the tree choosing actions according to the current probability estimates. (Assume a uniform probability distribution for anything you haven't seen before)
  - Add to the tree the first node that you reach that isn't already in it.
- Or we could use something other than just the current probability estimates.
- This looks like the "multi-armed bandit" problem.

Tree Policy: ε-greedy

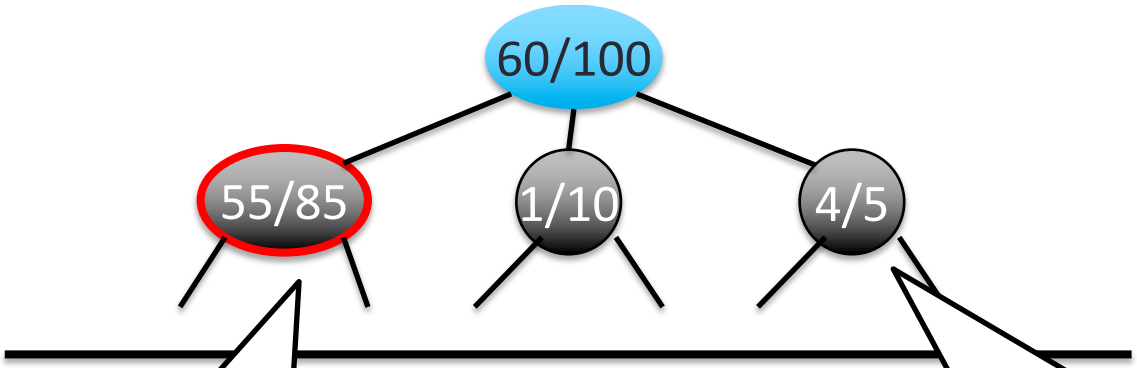
- Given a list of children, which one do we explore in a given iteration of MCTS?
- Ideally, we want:
  - To spend more time exploring **good** children (no point wasting time on bad children)
  - But spend some time exploring **bad** children, just in case they are actually good (since evaluation is stochastic, it might happen that we were just unlucky, and a child we thought was bad is actually good)
- Simplest idea: ε-greedy
  - With probability 1-ε: choose the current best
  - With probability ε: choose one at random

Monte Carlo Tree Search



Which is the best child?

Tree Search

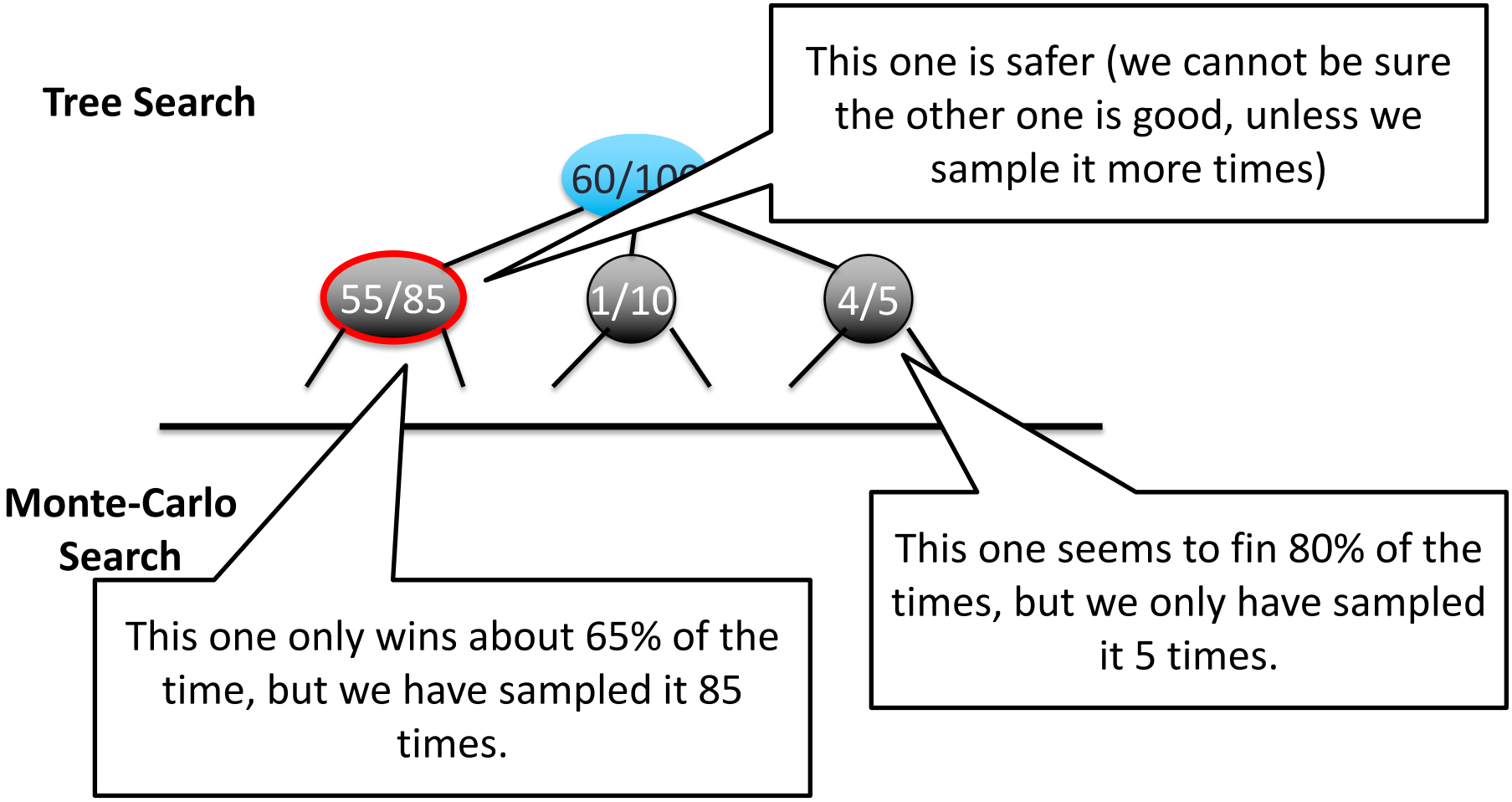


Monte-Carlo Search

This one only wins about 65% of the time, but we have sampled it 85 times.

This one seems to fin 80% of the times, but we only have sampled it 5 times.

Which is the best child?

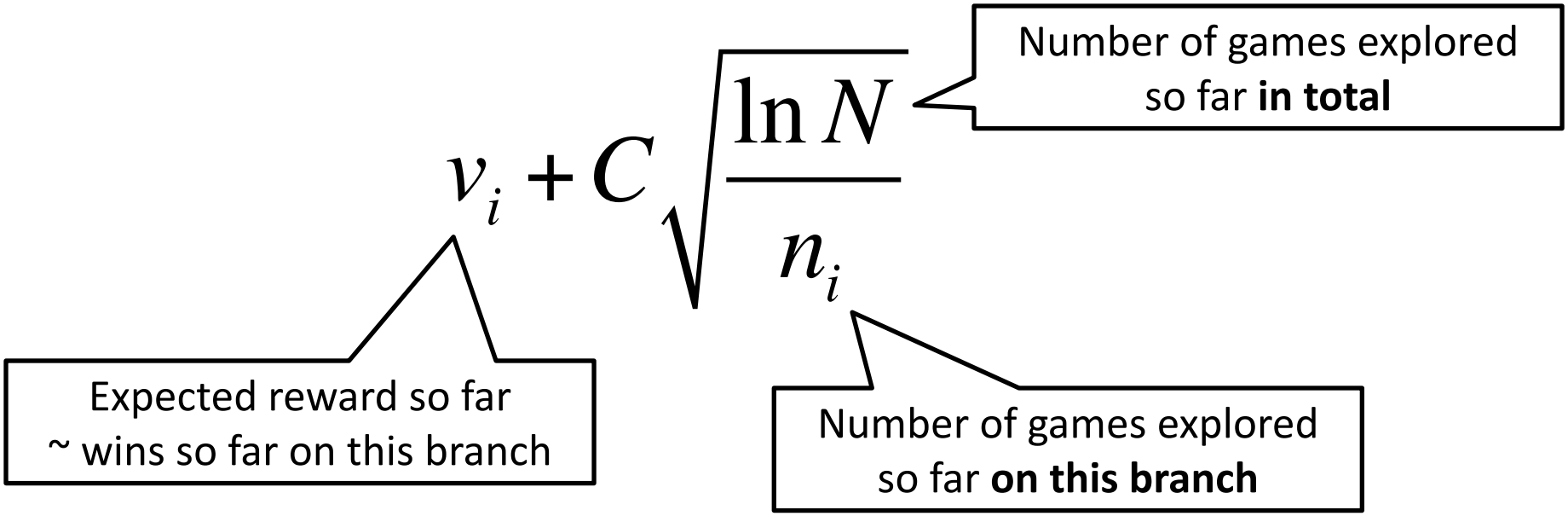


Tree Policy: Can we do Better?

- We just mentioned the ε-greedy policy… but is there a way to do better?
- ε-greedy is robust, but, for example:
  - If there are 3 children:
    - A: 40/100
    - B: 39/100
    - C: 2/100

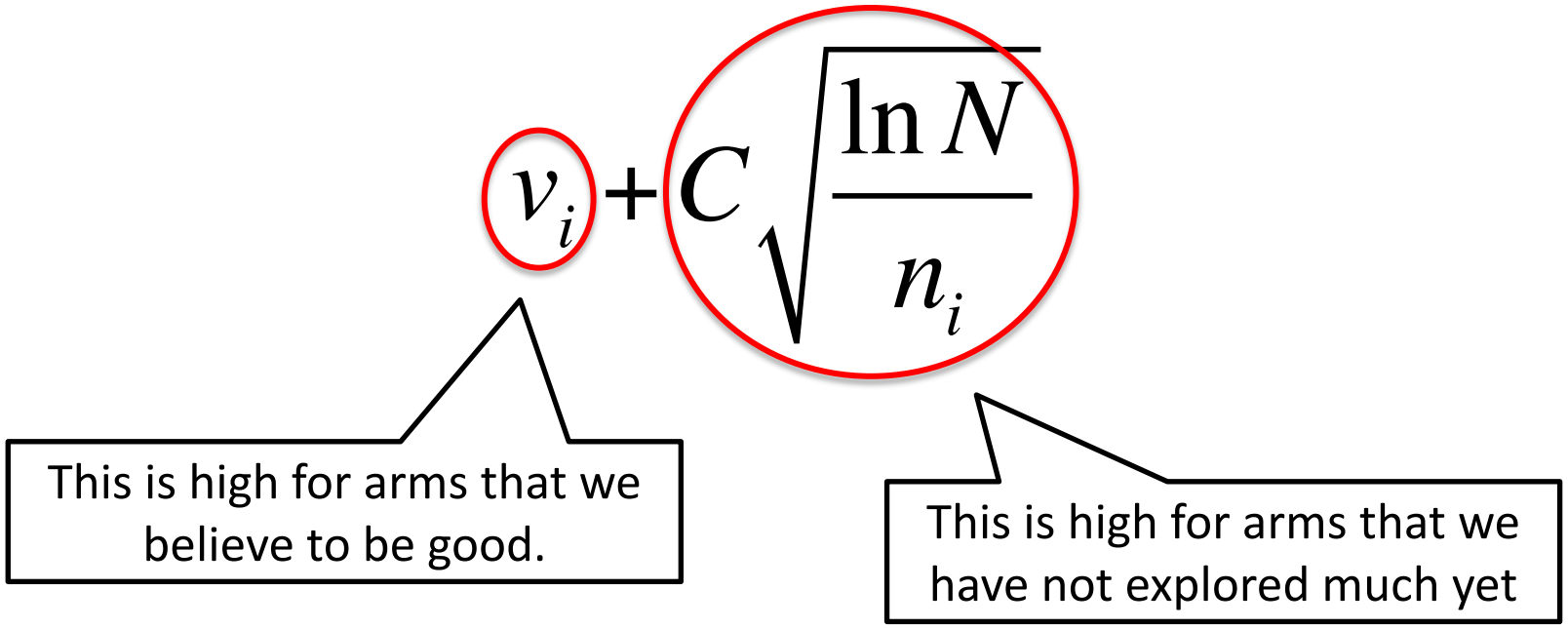
UCB1

- Upper Confidence Bounds
- Balance between **exploration** and **exploitation**
- UCB value for a given "arm" (or tree branch) *i*



UCB1

- Upper Confidence Bounds
- Balance between **exploration** and **exploitation**
- UCB value for a given arm i:

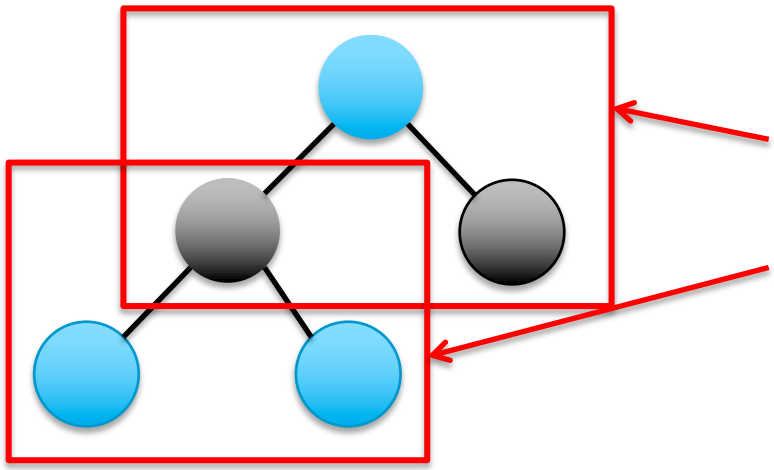


Upper Confidence Tree Algorithm

```
UCT(state, player)
  tree = new Node(state, player)
  Repeat until computation budget is exhausted
    node = selectNodeViaUCB1(tree)
    if (node.isTerminal) child = node
                         else child = node.nextChild();
    R = playout(child)
    child.propagateReward(R)
  Return tree.bestChild();
```

Upper Confidence Tree Algorithm

```
UCT(state, player)
  tree = new Node(state, player)
  Repeat until computation budget is exhausted
    node = selectNodeViaUCB1(tree)
    if (node is not terminal) child = node
```



This means using UCB1 at the root to select a child, and then recursively repeat, until we reach a leaf.

Monte Carlo Tree Search

- After a fixed number of iterations *K* (or after the assigned time is over), MCTS analyzes the tree and picks an action with the highest win ratio.
- MCTS algorithms do not explore the whole game tree!
  - They ***sample*** the game tree
  - They spend more time in moves that are more promising
  - They are "***any-time* algorithms**" (can be stopped at any time)
- It can be shown theoretically that when *K* goes to infinity, the values assigned to each action in the MCTS tree converge to those computed by minimax.
- MCTS algorithms are the standard algorithms for modern Go playing programs

Go

- Basics of the game:

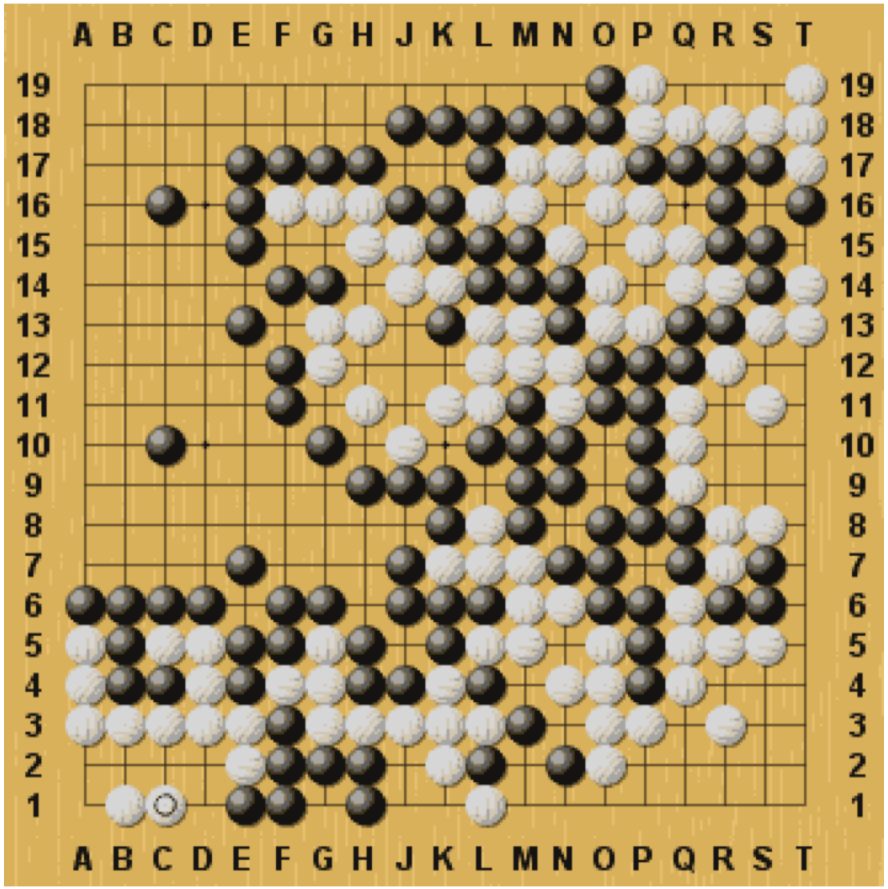
https://www.youtube.com/watch?v=gECcsSeRcNo

Go

- Board is 19x19
- Branching factor:
  - Starts at 361
  - Decreases by (more or less) 1 after every move
- Compare Go and Chess:
  - Chess, branching factor around 35:
    - Search at depth 6: 37,515,625 nodes
  - Go, branching factor around 300:
    - Search at depth 6: 729,000,000,000,000 nodes
- What can we do?

Games using MCTS Variants

- Go playing programs:
  - AlphaGo
  - MoGo
  - CrazyStone
  - Valkyria
  - Pachi
  - Fuego
  - The Many Faces of Go
  - Zen
  - …



AlphaGo

- Google's AlphaGo defeated Lee Sedol in 2016, and Ke Jie in May 2017
- How?
  - Integrated **MCTS** with **deep convolutional neural networks**
  - Data set of 30 million positions from the KGS Go Server
  - Trained a collection of neural networks to predict the probability of each move in the dataset and the expected value of a given board
  - Used neural networks to inform MCTS search

AlphaGo

4 Deep Neural Networks trained:

p_σ

- Trained via **Supervised Learning** from 30million positions from the KGS Go server.
- Predicts expert moves with 57% accuracy.

p_π

- Simplification of p_σ (runs faster)
- Predicts expert moves with 24.2% accuracy

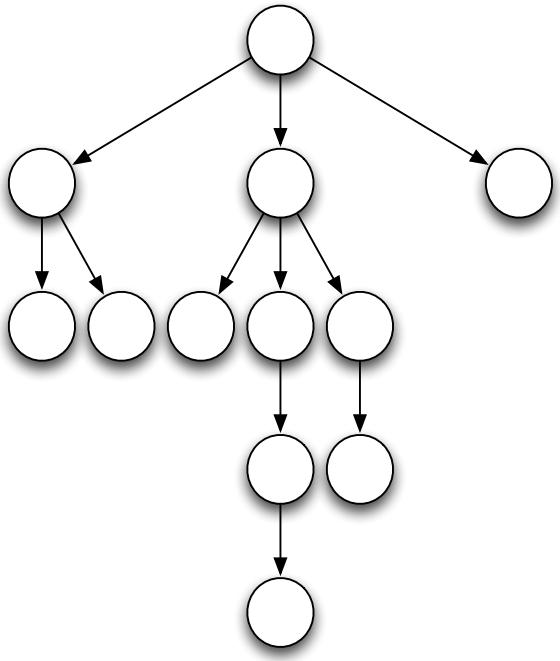
p_ρ

- Starts with p_σ and improves it via self-play (reinforcement learning)
- 80% win ratio against p_σ
- 85% win ratio against Pachi (MCTS with 100K rollouts)

v_ρ

- Predicts the winner given a position
- Trained from 30million positions using p_ρ
- Almost as accurate as rollouts with 15K times less CPU.

Integrated into MCTS:



AlphaGo

4 Deep Neural Networks trained:

p_σ

- Trained via **Supervised Learning** from 30million positions from the KGS Go server.
- Predicts expert moves with 57% accuracy.

p_π

- Simplification of p_σ (runs faster)
- Predicts expert moves with 24.2% accuracy

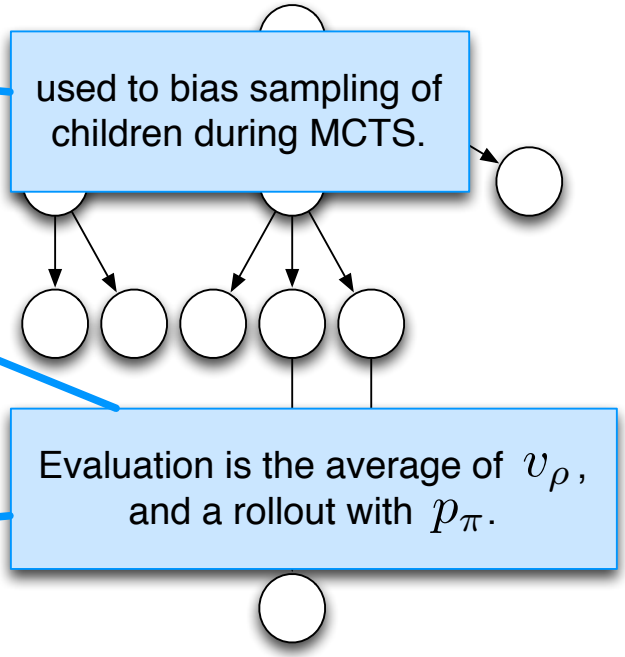
p_ρ

- Starts with p_σ and improves it via self-play (reinforcement learning)
- 80% win ratio against p_σ
- 85% win ratio against Pachi (MCTS with 100K rollouts)

v_ρ

- Predicts the winner given a position
- Trained from 30million positions using p_ρ
- Almost as accurate as rollouts with 15K times less CPU.

Integrated into MCTS:



Go → Chess

- **AlphaZero**: AlphaGo-like system for chess
- **Stockfish**: standard best chess-playing engine
- "The program had four hours to play itself many, many times, thereby becoming its own teacher."
- AlphaZero won the 100-game match vs. Stockfish with 28 wins, 72 draws, and 0 losses (!)
- See:

  https://www.chess.com/news/view/google-s-alphazero-destroys-stockfish-in-100-game-match

Adversarial Search Summary

- For when there's more than one agent in the world
- Search on a **Game Tree**
  - Max layers: for our moves
  - Min layers: for our opponent's moves
  - "Average" layers: for chance elements (e.g. dice rolls)
- Algorithms:
  - Minimax
    - with Alpha-Beta pruning
  - Expectiminimax
  - Monte Carlo Search
  - Monte Carlo Tree Search