Artificial Intelligence – CM8

Monte-Carlo Tree Search

Arthur Bit-Monnot

INSA Toulouse – 4IR

Monte-Carlo Tree Search

Minimax:

- systematically explores a game-tree
- up to certain depth
- returns the evaluation that maximizes our estimated utility at this depth
  - assuming the adversary minimizes this evaluation

Minimax limitations

- complexity exponential in depth: $O(b^d)$
  - typical branching factors:
    - checkers: 8
    - chess 31
    - go: 250
- symmetric: all branches explored up to the max depth
  - no preference for promising states
- highly-dependent on the evaluation function

Minimax limitations

- complexity exponential in depth: $O(b^d)$
  - typical branching factors:
    - checkers: 8
    - chess 31
    - go: 250
- symmetric: all branches explored up to the max depth
  - no preference for promising states
- highly-dependent on the evaluation function

⇨ Many mitigations (alpha-beta pruning, ...) but hard to bypass

Monte-Carlo methods

Monte-Carlo methods

Broad class of computational algorithms that rely on repeated random sampling ♡

General schema:

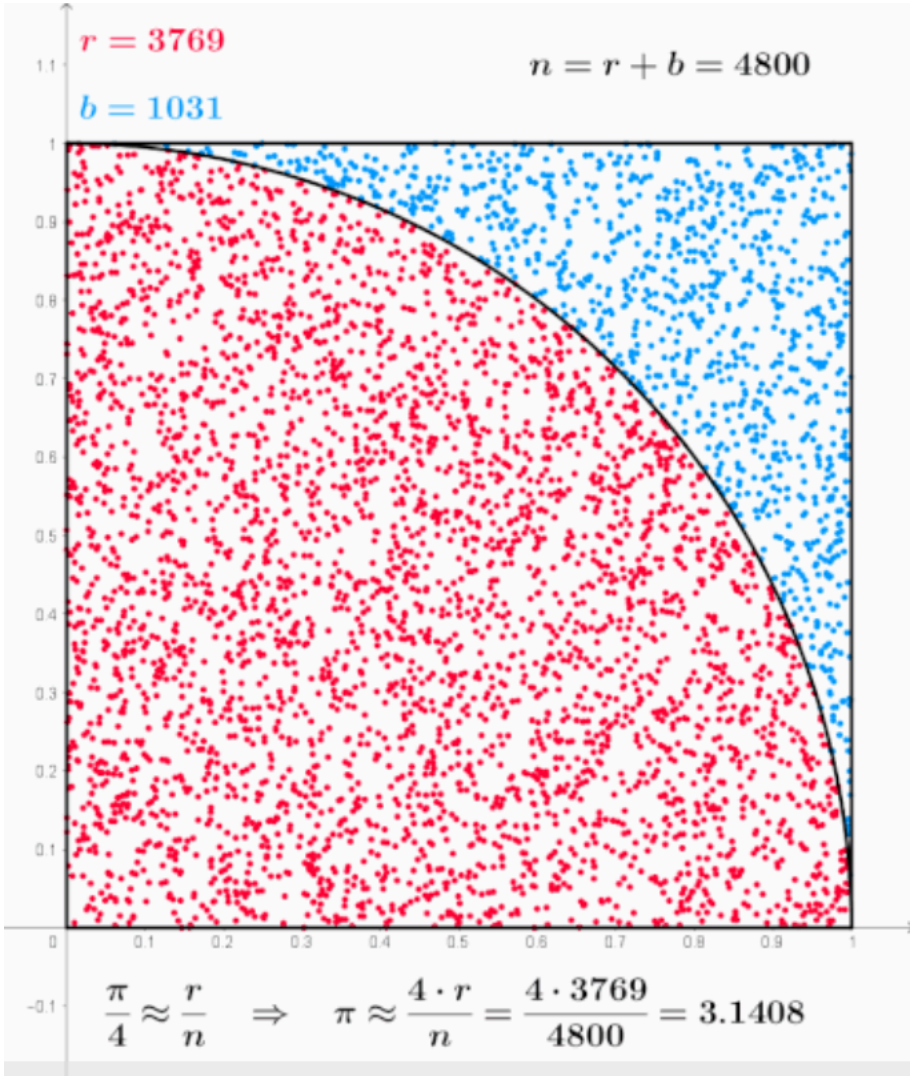
- Define a domain of possible inputs.
- **Generate inputs randomly** from a probability distribution over the domain.
- Perform a deterministic computation of the outputs.
- Aggregate the results.

Casino of Monte-Carlo (Monaco)

Monte-Carlo methods (π estimation)

Process:

- Define a square of side length 1
- randomly sample n points in it
- let r be the number of points within distance 1 of a corner
- let b be the others
- $\frac{r}{n}$ estimates the area of the quarter-circle ($\frac{\pi}{4}$)



Monte-Carlo Tree Search

Sampling in tree search

Rollout

From given state s of the game, perform one complete game with randomized actions until the game is decided (win, draw, loose).

♡

Complexity: $O(n \times b)$

- n: average number of actions until the decision
- b average branching factor

Rollouts as utility estimation

Given:

- a set R of rollouts from state s
- the utility $u(r)$ of the rollout
  - (e.g. 1 for win, 0 for lost, 0.5 for draw)

$$U(s) = \frac{1}{|R_s|} \times \sum_{r \in R_s} u(r)$$

$\Rightarrow U(s)$ approximates the utility of s if the two players act randomly[1]

[1] According to the randomized playout policy, often as simple as unirform random among all possible actions.

Playout Policy

Move in the rollout selected according to the *playout policy*

- **uniform random** among legal move
  - simplest, no game-specific knowledge
  - utility for complety random play

Playout Policy

Move in the rollout selected according to the *playout policy*

- **uniform random** among legal move
  - simplest, no game-specific knowledge
  - utility for complety random play
- **informed randomized**
  - bias towards seemingly good move
  - require prior knowledge (heuristic, learning)

O	0	X
X	0.5	0.2
O	0.05	0.25

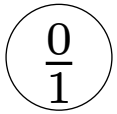
Example move distribution for a playout policy of "X" that favors blocking opponents and aligning pieces

MCTS: Monte-Carlo for Tree-Search

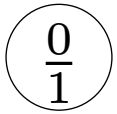
Monte-Carlo Tree Search is an algorithm that will iteratively build a game tree by repeatidely:

1. **select** a node at the frontier
2. **expand** this node with a new untried action
3. **simulate** with a rollout the outcome
4. **backpropagate** this information up the tree

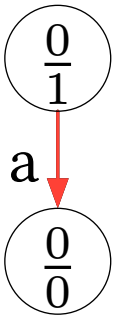
Example



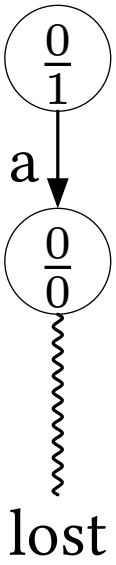
Example (selection)



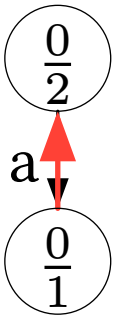
Example (expansion)



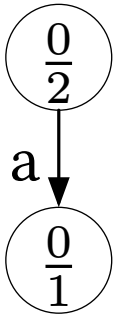
Example (simulation)



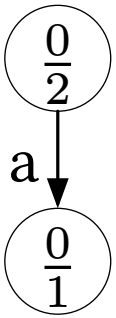
Example (backpropagation)



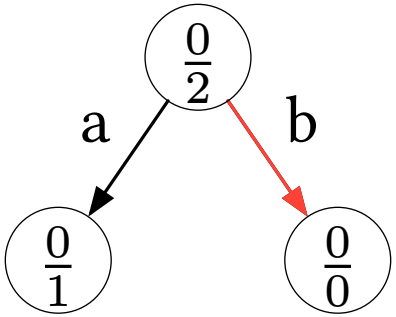
Example



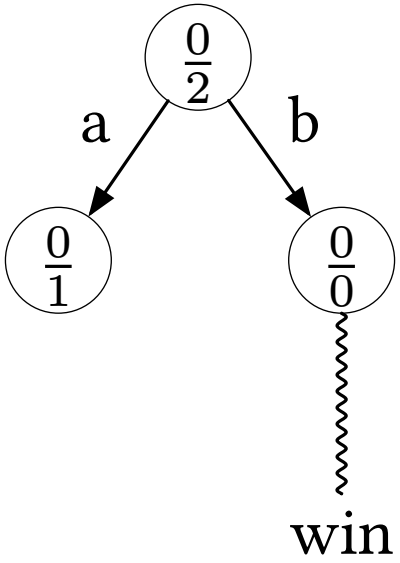
Example (selection)



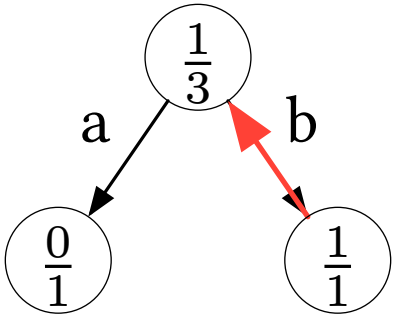
Example (expansion)



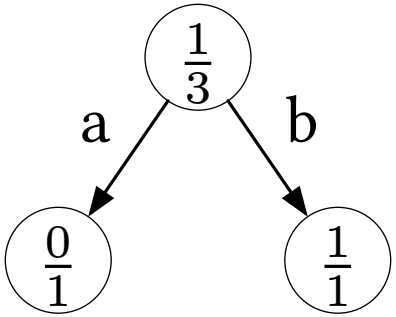
Example (simulation)



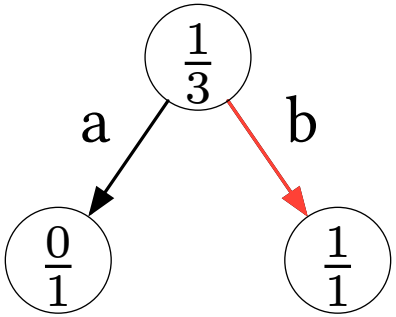
Example (backpropagation)



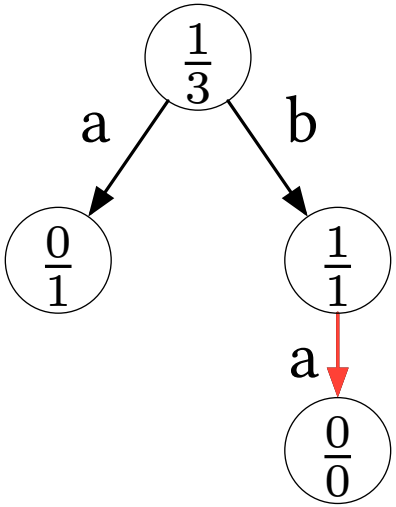
Example



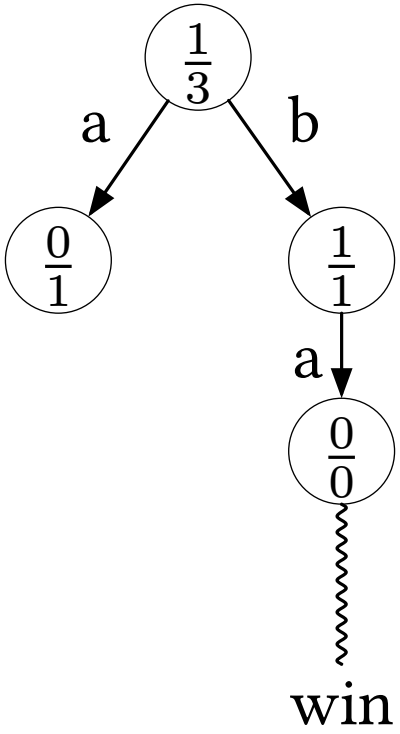
Example (selection)



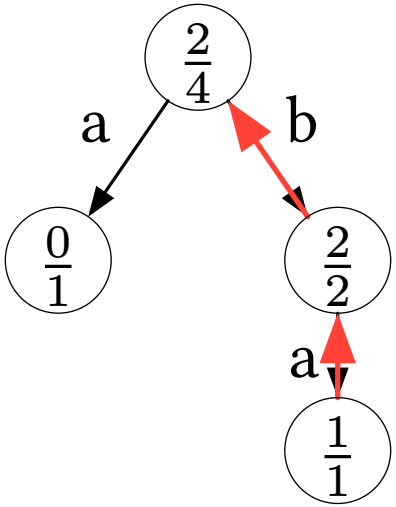
Example (expansion)



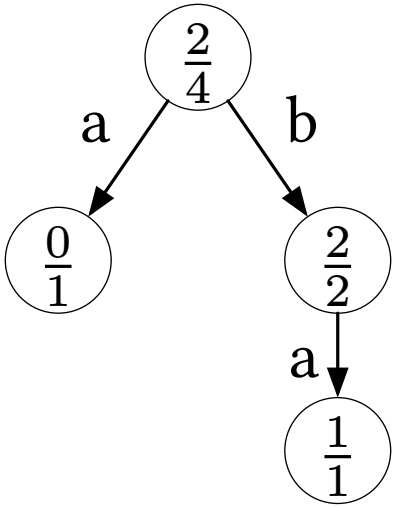
Example (simulation)



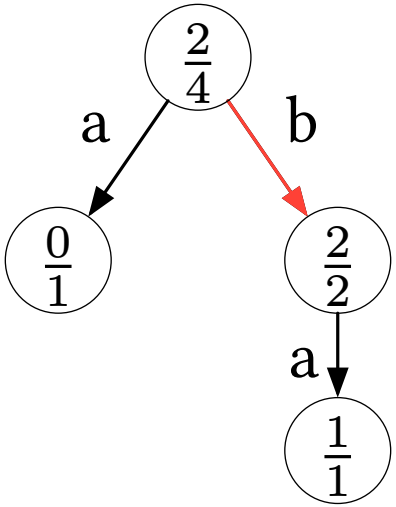
Example (backpropagation)



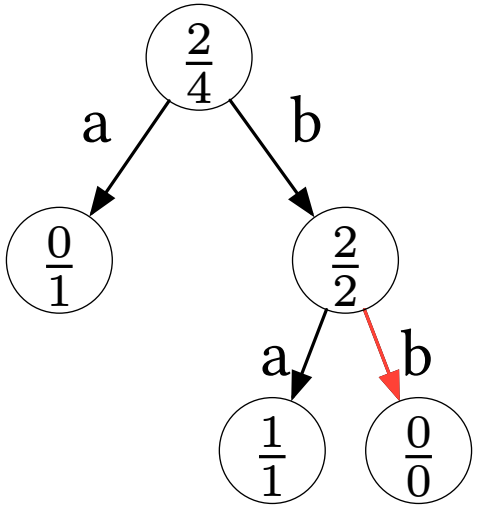
Example



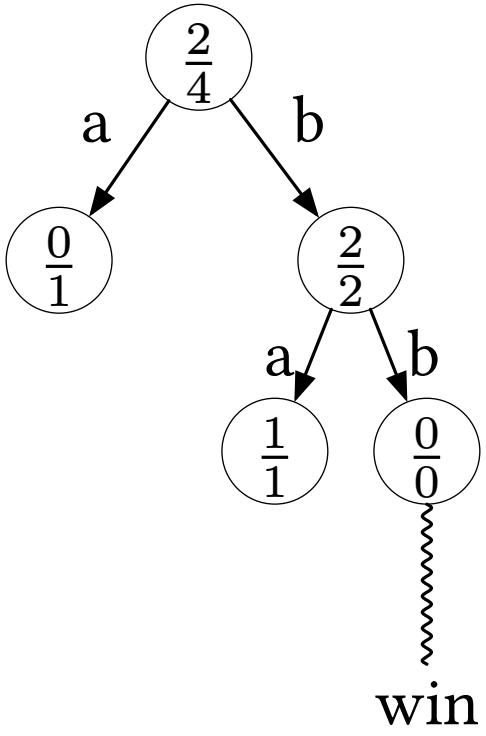
Example (selection)



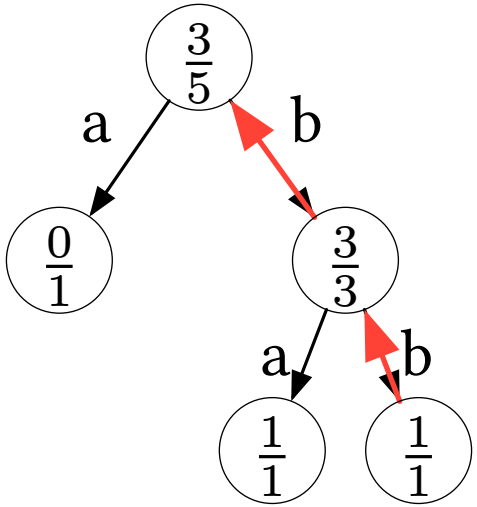
Example (expansion)



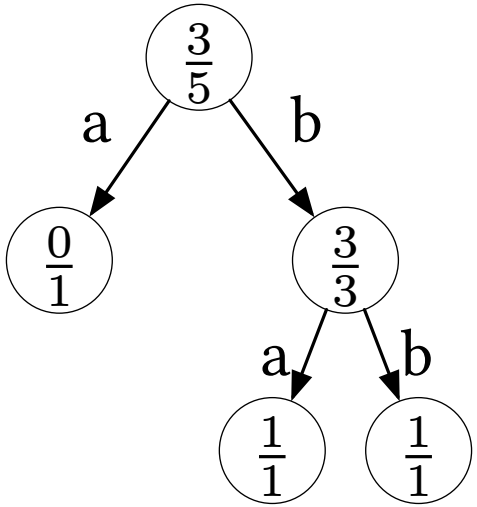
Example (simulation)



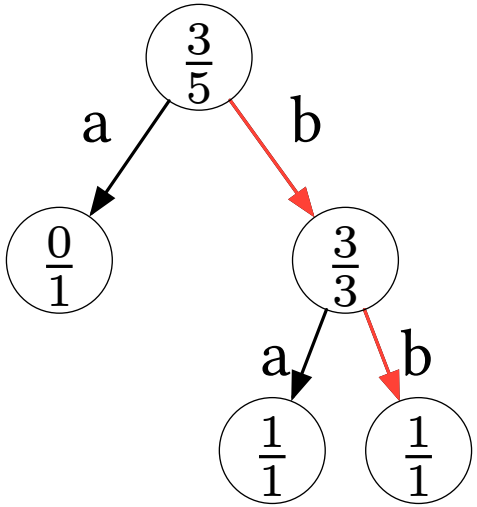
Example (backpropagation)



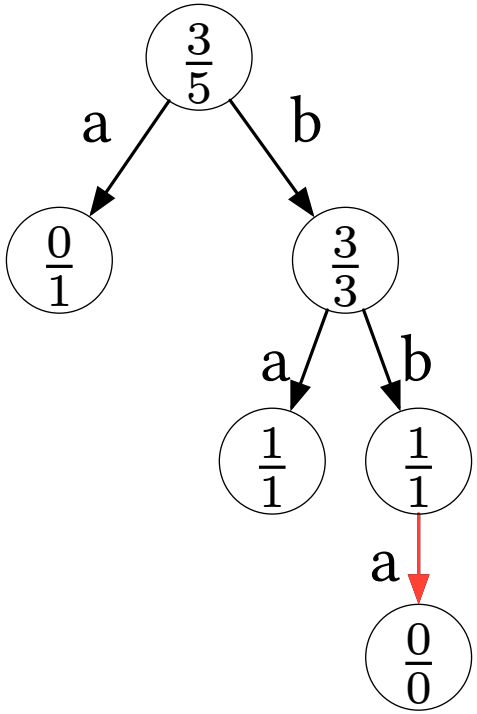
Example



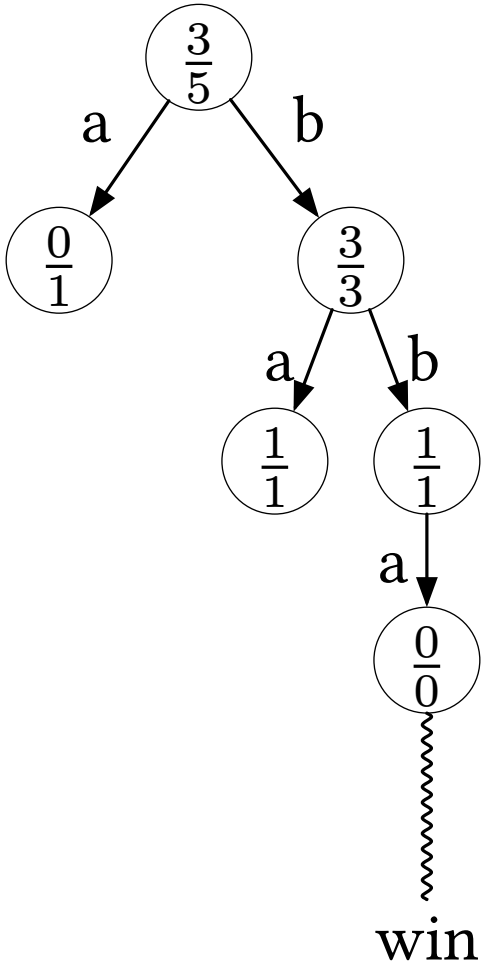
Example (selection)



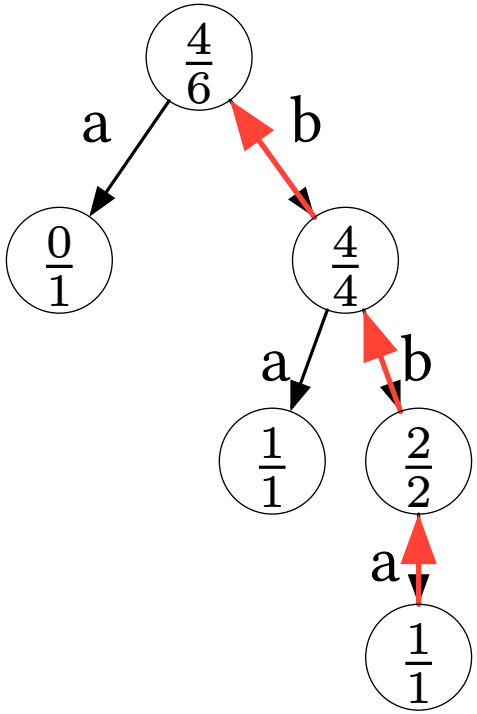
Example (expansion)



Example (simulation)



Example (backpropagation)



MCTS Key Components

- $U(s)$: utility of the first rollout
- $N(s)$: numbers of times s was selected
- $N(s, a)$: number of times the action a was selected in s
- $Q(s)$: estimated utility of state s
- $Q(s, a)$: estimated utility of taking a in s $(= Q(\text{result}(s, a)))$

MCTS Key Components

- $U(s)$: utility of the first rollout
- $N(s)$: numbers of times s was selected
- $N(s, a)$: number of times the action a was selected in s
- $Q(s)$: estimated utility of state s
- $Q(s, a)$: estimated utility of taking a in s $(= Q(\text{result}(s, a)))$

$$Q(s) = \frac{U(s)}{N(s)} + \sum_a \frac{N(s, a)}{N(s)} \times Q(s, a)$$

MCTS Key Components

- $U(s)$: utility of the first rollout
- $N(s)$: numbers of times s was selected
- $N(s, a)$: number of times the action a was selected in s
- $Q(s)$: estimated utility of state s
- $Q(s, a)$: estimated utility of taking a in s $(= Q(\text{result}(s, a)))$

$$Q(s) = \frac{U(s)}{N(s)} + \sum_a \frac{N(s, a)}{N(s)} \times Q(s, a)$$

Special case: (win: 1, loss: 0)

$\Rightarrow Q(s)$ ratio of succesful rollouts at or below s

Samples as a policy

When $N(s) \gg 1$

$$Q(s) \approx \sum_a \frac{N(s,a)}{N(s)} \times \underbrace{Q(s,a)}_{\substack{\text{utility of} \\ \text{resulting state}}}$$

Samples as a policy

When $N(s) \gg 1$

$$Q(s) \approx \sum_a \frac{N(s,a)}{N(s)} \times \underbrace{Q(s,a)}_{\substack{\text{utility of} \\ \text{resulting state}}}$$

Expected utility of a player that in a state s would select action a with probability $\frac{N(s,a)}{N(s)}$

Players policy

$$\pi(s,a) = \frac{N(s,a)}{N(s)}$$

$$Q(s) \approx \sum_a \pi(s,a) \times Q(s,a)$$

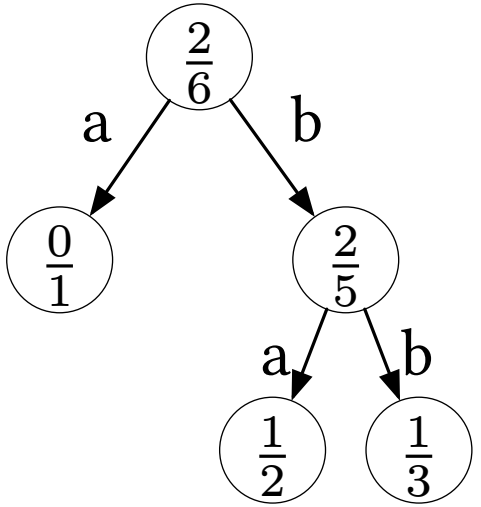
Accurate when:

- $\pi(s,a)$ is near-optimal
  - selects the best action with probability ≈ 1
- $Q(s,a)$ is accurate
  - result(s,a) has seen many rollouts
  - with a near-optimal policy π

Selection in MCTS

Selecting a node s has two effects on Q:

- it **improve its estimate**
  - additional rollout
  - tree expansion
- it **biases the policy** towards it



Upper confidence applied to trees (UCT)

$$\text{select}(s) = \text{argmax}_a \ \text{UCB1}(s, a)$$

$$\text{UCB1}(s, a) = \underbrace{T \times Q(s, a)}_{\text{exploitation}} + C \times \underbrace{\sqrt{2 \times \frac{\log(N(s))}{N(s, a)}}}_{\text{exploration}}$$

Where

- $T = 1$ if it is my turn and $T = -1$ if it the opponents turn
  - models maximization/minimization of Q
- C is a game-specific constant balancing exploration and exploitation
  - theoretical arguments that it is optimal when the utility is in $[0, C]$

Upper confidence applied to trees (UCT)

$$\text{select}(s) = \text{argmax}_a \ \text{UCB1}(s, a)$$

$$\text{UCB1}(s, a) = \underbrace{T \times Q(s, a)}_{\text{exploitation}} + C \times \underbrace{\sqrt{2 \times \frac{\log(N(s))}{N(s, a)}}}_{\text{exploration}}$$

The exploration term:

- is ∞ when the action has never been selected ($N(s, a) = 0$)
- tends towards 0 when the number of playout grows

MCTS Algorithm

To determine the best action in a state s, Monte-Carlo Tree Search

1. does a number of playouts (a time allows)
   - selection, expansion, simulation, backpropagation
2. selects the node that has been *selected* the most
   - highest ranked by the policy

```
MCTS (s):
1 while IsTimeRemaining ()
2    Playout (s)
3 return argmax_a N(s, a)
```

Playout algorithm

```
PLAYOUT (s):
 1 if N(s) = 0:                                  // Node was just expanded
 2    U(s) ← simulate(s')                        // rollout
 3    N(s) ← 1
 4    Q(s) ← U(s)
 5 else
 6    a ← argmax_a UCB1(s, a)                    // selection
 7    s' ← result(s, a)
 8    PLAYOUT (s')
 9    // Update counts and utility estimates
10    N(s) ← N(s) + 1
11    N(s, a) ← N(s, a) + 1
12    Q(s) ← U(s)/N(s) + Σ_x N(s,x)/N(s) × Q(s, x)
```

MCTS Properties

- **Converges towards optimal policy** (minimax like)

MCTS Properties

- **Converges towards optimal policy** (minimax like)
- Initially focuses on **exploration** (breadth-first like)

MCTS Properties

- **Converges towards optimal policy** (minimax like)
- Initially focuses on **exploration** (breadth-first like)
- Explore promising move more thoroughly
  - **asymetric** (potentially more sample efficient)

MCTS Properties

- **Converges towards optimal policy** (minimax like)
- Initially focuses on **exploration** (breadth-first like)
- Explore promising move more thoroughly
  - **asymetric** (potentially more sample efficient)
- no game specific knowledge (with uniform random rollouts)
  - rules of the games are sufficient to play

MCTS In the Wild

State-of-the-art in many situation:

- Go (AlphaGo, AlphaZero)
- Video games (Atari, StarCraft)
- General Game Playing (game-independent)
- many optimisation problems (routing, scheduling, ...)

MCTS In the Wild

State-of-the-art in many situation:

- Go (AlphaGo, AlphaZero)
- Video games (Atari, StarCraft)
- General Game Playing (game-independent)
- many optimisation problems (routing, scheduling, ...)

Beaten by minimax variants:

- when branching factor is low (checkers)
- when computational resources are limited ("slow" convergence)

Extensions in Go

- RAVE: Rapid Action Value Estimation[1]
  - approximate the value of action independently of their context
  - fast and approximate bootstraping
- AlphaGo[2] exploits ML to learn:
  - a value function to replace the rollouts
  - a prior action distribution to bias exploration towards likely nodes
  - learned from examples (grand master games) + self-play (RL)
- AlphaZero: similar but learning entierly through self play
  - applied beyond Go (chess, ...)

[1] Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go

[2] Mastering the game of Go with deep neural networks and tree search

The Bitter Lesson (Rich Sutton)

The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin.

...

One thing that should be learned from the bitter lesson is the great power of general purpose methods, of methods that continue to scale with increased computation even as the available computation becomes very great. The two methods that seem to scale arbitrarily in this way are *search* and *learning*.

...

Rich Sutton – The Bitter Lesson[1]

[1] http://www.incompleteideas.net/IncIdeas/BitterLesson.html