Biased exploration for Monte-Carlo Tree Search Applied to the game of Go

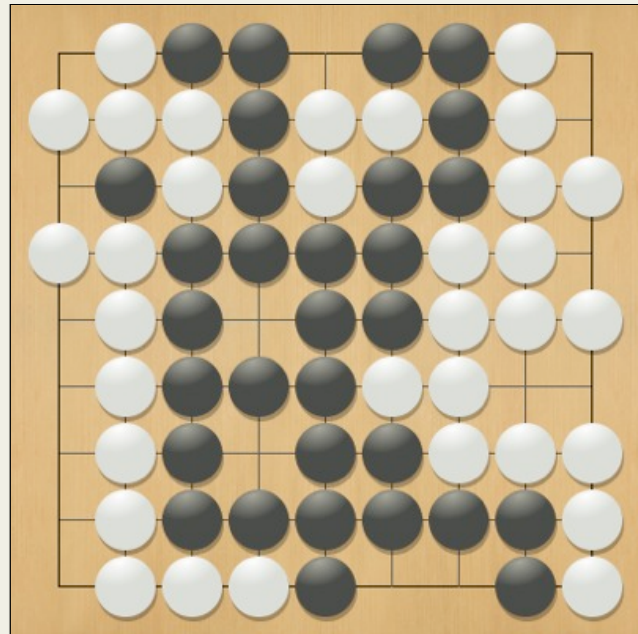
Frédéric Maillasson
Research student, Imai Laboratory, Department of Computer Science, University of Tokyo
<frederic.maillasson@lager.is.s.u-tokyo.ac.jp>

The Game of Go

Popular board game in Asia, regarded as **difficult for computers**

Rules

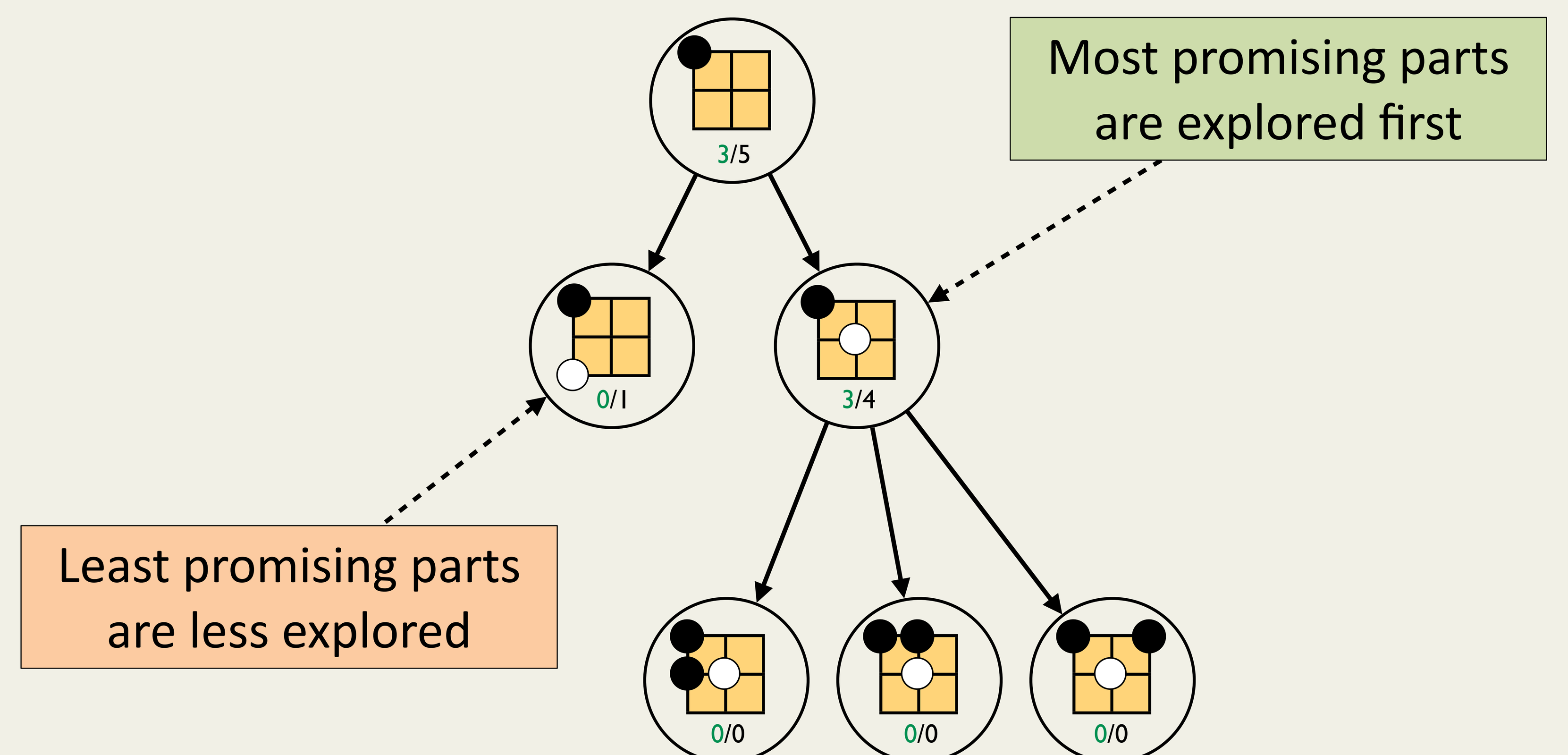
1. Black and White place *stones* on intersections in turns
2. Stones connects in 4 directions and form blocks
3. Win by **surrounding greater area**
4. *Blocks* gets **captured** if completely surrounded



Monte-Carlo Tree Search

• Without a good **evaluation function**, alpha-beta search does not work well

• Monte-Carlo Tree Search evaluates a position by **sampling a random simulation** and looking at the final result



• When a leaf seems promising, we expand the tree at this node : **the most promising moves are explored first**

• Dilemma exploration / exploitation :

$$\text{BestSon}(n) = \underset{m \text{ son of } n}{\text{argmax}} \left\{ \frac{\text{w}(m)}{\text{s}(m)} + \sqrt{\frac{2\log(t)}{\text{s}(m)}} \right\}$$

UCB1 (Upper Confidence Bound) proposed in 2002 [Kocsis]

UCT = MCTS + UCB1

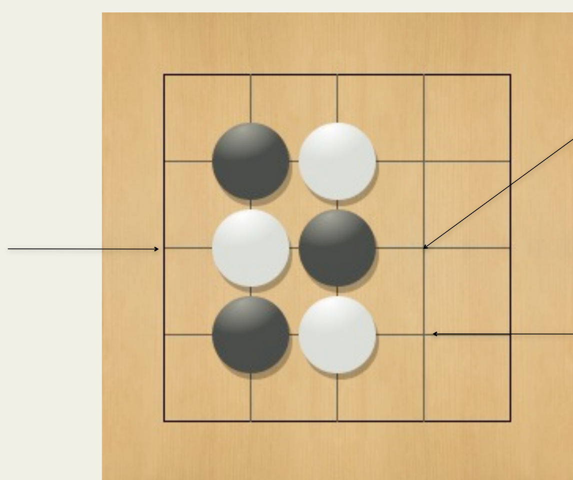
• **Proved to converge** to the optimal answer [Kocsis]
• Easy to implement, **no domain-specific knowledge**
• Converges too slowly : to be used practically, *we need to add domain-specific knowledge*

Biased Softmax policy

• Idea : use domain-specific knowledge to **bias the random simulation** in order to make it converge faster [Coulom]

• Move is chosen according to the (**domain-specific**) features it matches :

This move :
• saves a stone which could be captured
• captures an enemy stone
$\phi = (1, 1, 0)$



Black to play

This move saves a stone which could be captured
$\phi = (1, 0, 0)$

This move threatens to capture an enemy stone
$\phi = (0, 0, 1)$

• The *weight of a move* is the sum of the weights of the features it matches :

$$w(m) = \phi(m) \cdot \theta$$

• The move is selected according to a **softmax policy** :

$$\pi(\text{play } m) \propto e^{w(m)}$$

Simulation Balancing

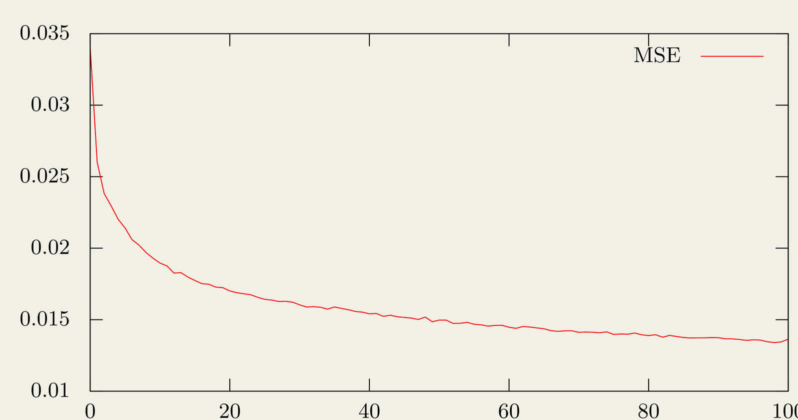
• We can set the features weights manually

• Or we can try to find the weights **minimizing an objective function** (the *imbalance*) [Silver] :

$$[\underbrace{\mu^*}_{} - \underbrace{\bar{\mu}(\theta)}_{}]^2$$

target win rate (pre-computed / estimated) — statistical win rate (by playing according to the softmax policy)

• Best weights found by **gradient-descent algorithm**



Mean-Squared Error decreasing with the number of iterations of the gradient-descent algorithm [Huang]

Experimental results [Huang]

• Implemented in 2010 on the program *Erica* (9x9 board)

• **Outperforms the reinforcement-learning approach** (which tries to maximize the strength instead of minimizing the balance)

• **Promotes the best features** by itself

• Also promotes **some bad features** : it doesn't matter if the moves played in the simulation are bad, as long as errors are played for both sides !

[Kocsis] L. Kocsis, C. Szepesvári, « Bandit-Based Monte-Carlo Planning », ECML 2006, pp. 282-293, 2006
[Silver] D. Silver, G. Tesauro, « Monte-Carlo Simulation Balancing », Proceedings of the 26th Annual International Conference on Machine Learning, pp. 945-952, 2009
[Coulom] R. Coulom, « Computing Elo Ratings of Move Patterns in the Game of Go », ICGA Computer Games Workshop, 2007
[Huang] S.-C. Huang, R. Coulom, S.-S. Lin, « Monte-Carlo Simulation Balancing in Practice », Lecture Notes in Computer Science, Volume 6515, pp. 81-92, 2011