MoGo: Improvements in Monte-Carlo Computer-Go using UCT and sequence-like simulations

Sylvain Gelly[1], Yizao Wang[1,2], Rémi Munos[2,3], Olivier Teytaud[1]

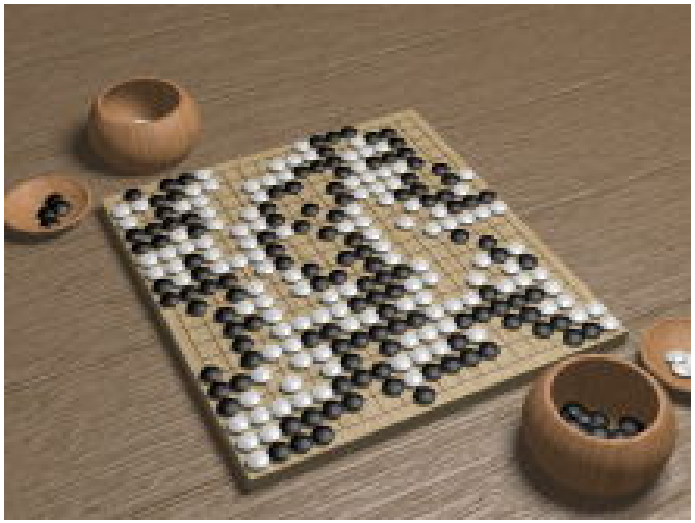
1: Université Paris-Sud, INRIA, CNRS, TAO Group, FRANCE
2: Centre de Mathématiques Appliquées, Ecole Polytechnique, FRANCE
3: INRIA, SEQUEL Group, FRANCE

December 12, 2006

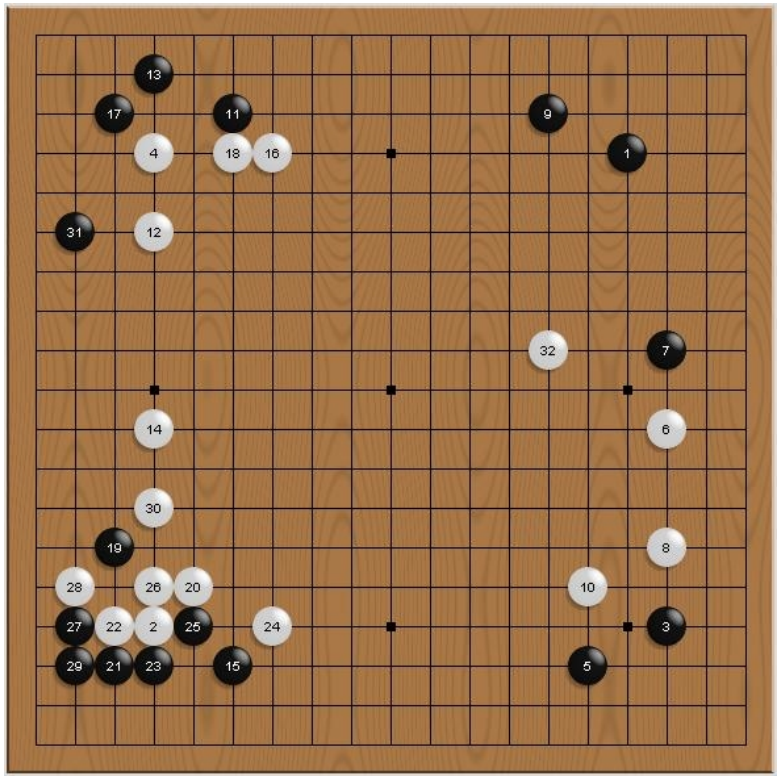
A short history of MoGo

- **July 2006**: first participation in tournaments: 1650 ELO on CGOS (Computer Go Server) (9x9);
- **Aug. 2006 (beg.)**: ranked best program on CGOS: 1920 ELO;
- **Aug. 2006 (end)**: MoGo reached 2000 ELO;
- **Oct. 2006**: MoGo won the 2 KGS tournaments (9x9 and 13x13);
- **Nov. 2006**: MoGo won the 2 KGS tournaments (9x9 and 13x13);
- **Nov. 2006**: MoGo reached 2200 ELO on CGOS;
- **Dec. 2006**: MoGo 2nd on KGS formal tournament (19x19);

A Quick introduction to game of Go



A Quick introduction to game of Go



- Go-board (Goban): 19×19 intersections;
- Back and White play alternatively. Black starts the game;
- Adjacent stones are called a *string*. *Liberties* are the empty intersections next to the string;
- Stones do not move, there are only added and removed from the board. A string is removed iff its number of liberties is 0;
- Score: territories (number of occupied or surrounded intersections).

- Beginning of Computer-Go, 1970s
- Classical methods
  - Expert knowledge based evaluation function;
  - Minimax tree search;
- Comparison with chess
  - Chess: Deeper Blue won against Kasparov, 1997;
  - Go: The strongest programs are about 10kyu in 2006 (amateurs of good level can win with 9 stones handicap)

Difficulties in computer-Go

- Huge branching factor ≈ 200, chess ≈ 40
  (John Tromp and Gunnar Farnebäck, 2006)
  Legal positions number
  2.0×10^{170} on 19×19, 1.0×10^{38} on 9×9
- Good evaluation function difficult to build (Stern et al. 2004, Wu L. and P. Baldi 2006, Silver D. et al. 2007). Must take into account local, and global information.

MoGo player

Two components

- Simple simulation policy finishing the game as an evaluation function;
- bandit based tree search.

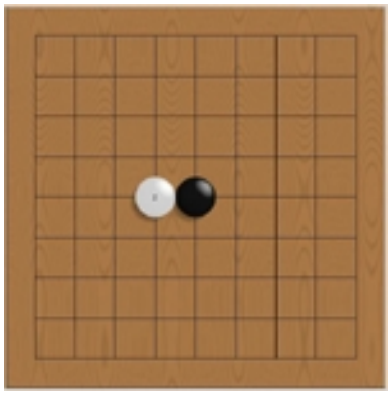
Evaluation function: Outline

- Monte-Carlo Go
- Improvements using **sequences**
- Perspectives in the simulations

Monte-Carlo evaluation function

(B. Bruegmann, 1993)

- Let p the position to evaluate;
- let π a (stochastic) player;
- from p, π plays against itself until the end of the game;
- the final score is then allocated to p;
- possibly iterate and average.




$\longrightarrow$ $\longrightarrow$ score

Monte-Carlo evaluation function

(B. Bruegmann, 1993)

- Let p the position to evaluate;
- let π a (stochastic) player;
- from p, π plays against itself until the end of the game;
- the final score is then allocated to p;
- possibly iterate and average.


$\longrightarrow$ $\longrightarrow$ score

Monte-Carlo evaluation function: Pros

- even if π plays uniformly among legal moves, gives suprising good results;
- very precise for the end of games;
- simple;
- fast;
- easy to improve.

Monte-Carlo evaluation function: Cons

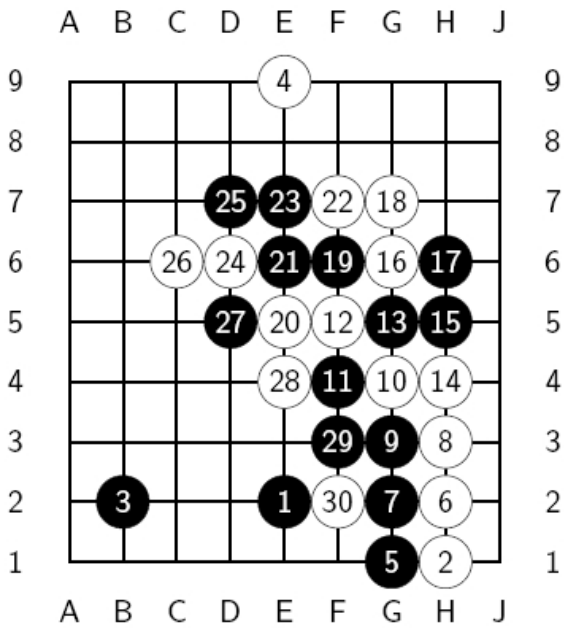
- stochastic evaluation;
- bad for early position;
- precision decrease with game length;

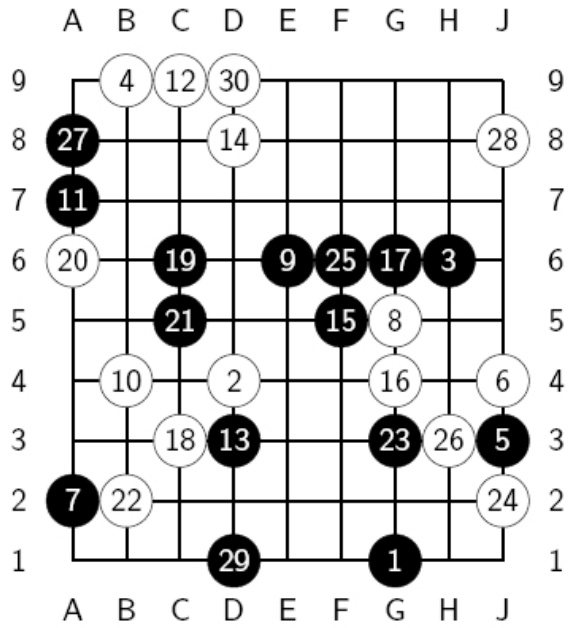
Playing uniformly randomly can't be the best

what can be better?

- derandomize?
- change the distribution to play better local moves?
- change the distribution to play better global moves?

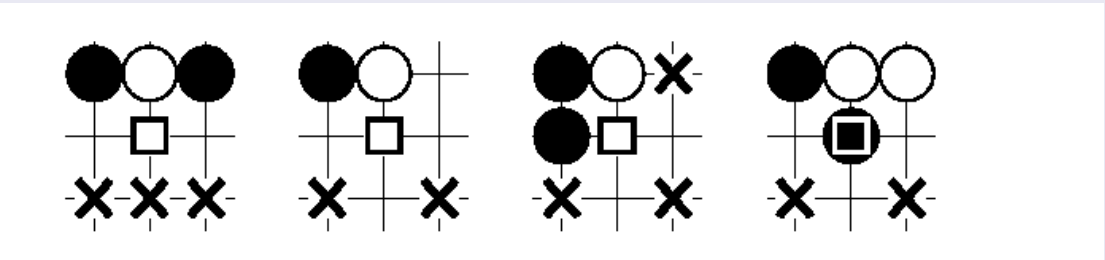
Move *sequences* matter





How it works in MoGo

- Look at the 8 intersections around the previous move;
- for each such intersection, try to match a pattern (including symetries);



- if at least one pattern matched: play uniformly among matching intersections;
- else play uniformly among legal moves.

Some results

Winning rate with 70k simulations per move, against gnugo 3.6, level 8:

simulation mode	winning rate
uniform	46.9%
Aug. 2006	65%
Sept. 2006	70.5%
Oct. 2006	77.5%
Nov. 2006	81%

Reinforcement Learning

RL for the player

- Playing Go is clearly a control problem;
- RL can be used to directly learn a good player;
- However build a good approximation function is difficult (some succeed e.g. **NeuroGo**);

Reinforcement Learning

RL for the player

- Playing Go is clearly a control problem;
- RL can be used to directly learn a good player;
- However build a good approximation function is difficult (some succeed e.g. **NeuroGo**);

What about RL for the *simulation* player?

- Building the player π used in the MC simulations is also a control problem;
- Experiments showed that "improving" π a little improved a lot MoGo;

Learning the *simulation* player

Reward function

Let π a simulation player, and $MoGo(\pi)$ the MoGo player using π.
Some rewards:

- 1 if $MoGo(\pi)$ wins against GNUGo, 0 else:
  - it is exactly what we are looking for;
  - too slow (6 minutes a game, 80h for 800 games...).

In fact it is what he do "by hand".

Learning the *simulation* player

Reward function

Let π a simulation player, and $MoGo(\pi)$ the MoGo player using π.
Some rewards:

- 1 if π wins against a fixed π_0, 0 else:
  - fast;
  - asymptotically great (minimax);
  - not monotonous at all.

Bad estimation players can have a high winning rate against our best player.

Learning the *simulation* player

Reward function

Let π a simulation player, and $MoGo(\pi)$ the MoGo player using π.
Some rewards:

- evaluation precision of some given positions:
  - tractable;
  - asymptotically (in π and number of positions) great;
  - can have overfitting.

Not so bad in practice.

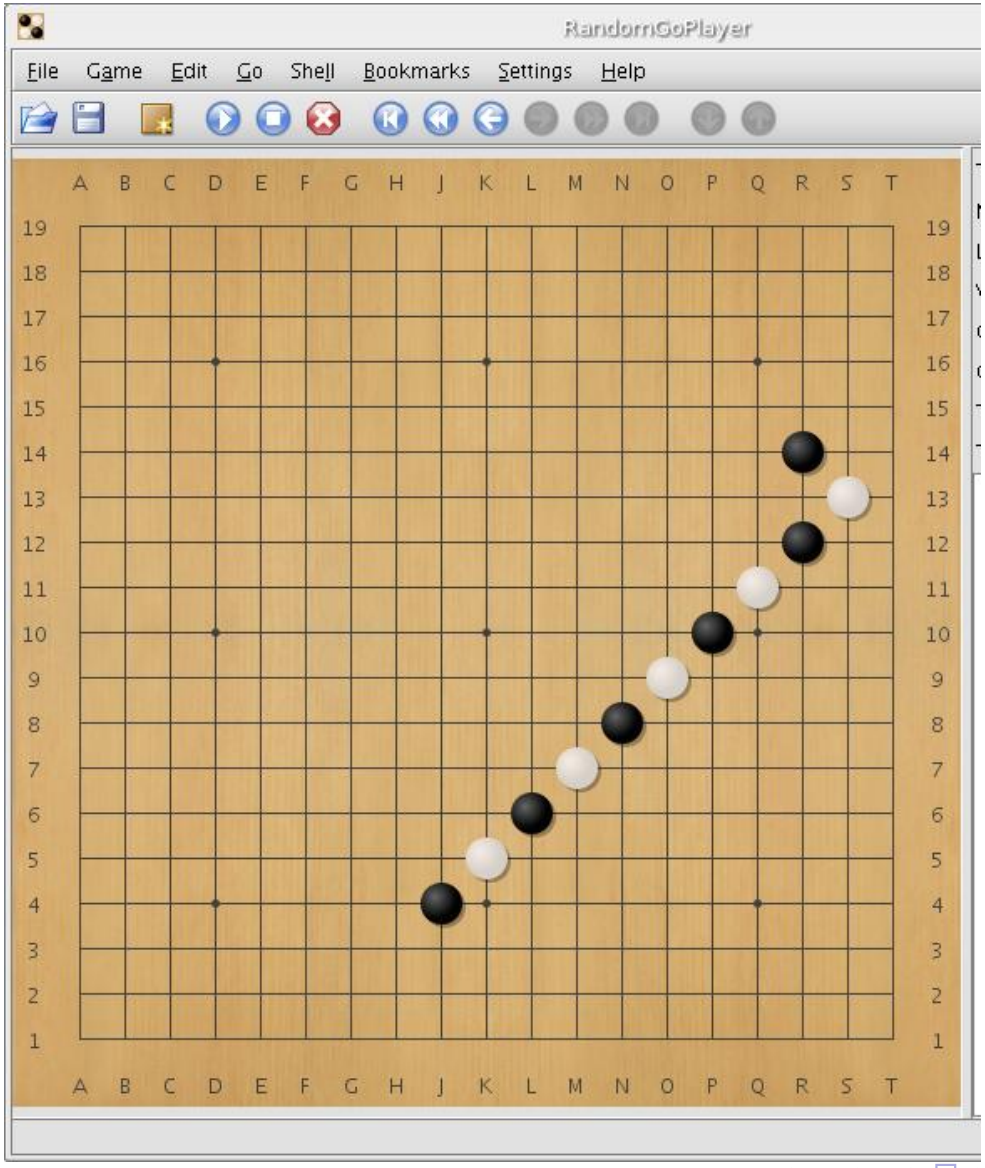
Learning the *simulation* player: what we tried

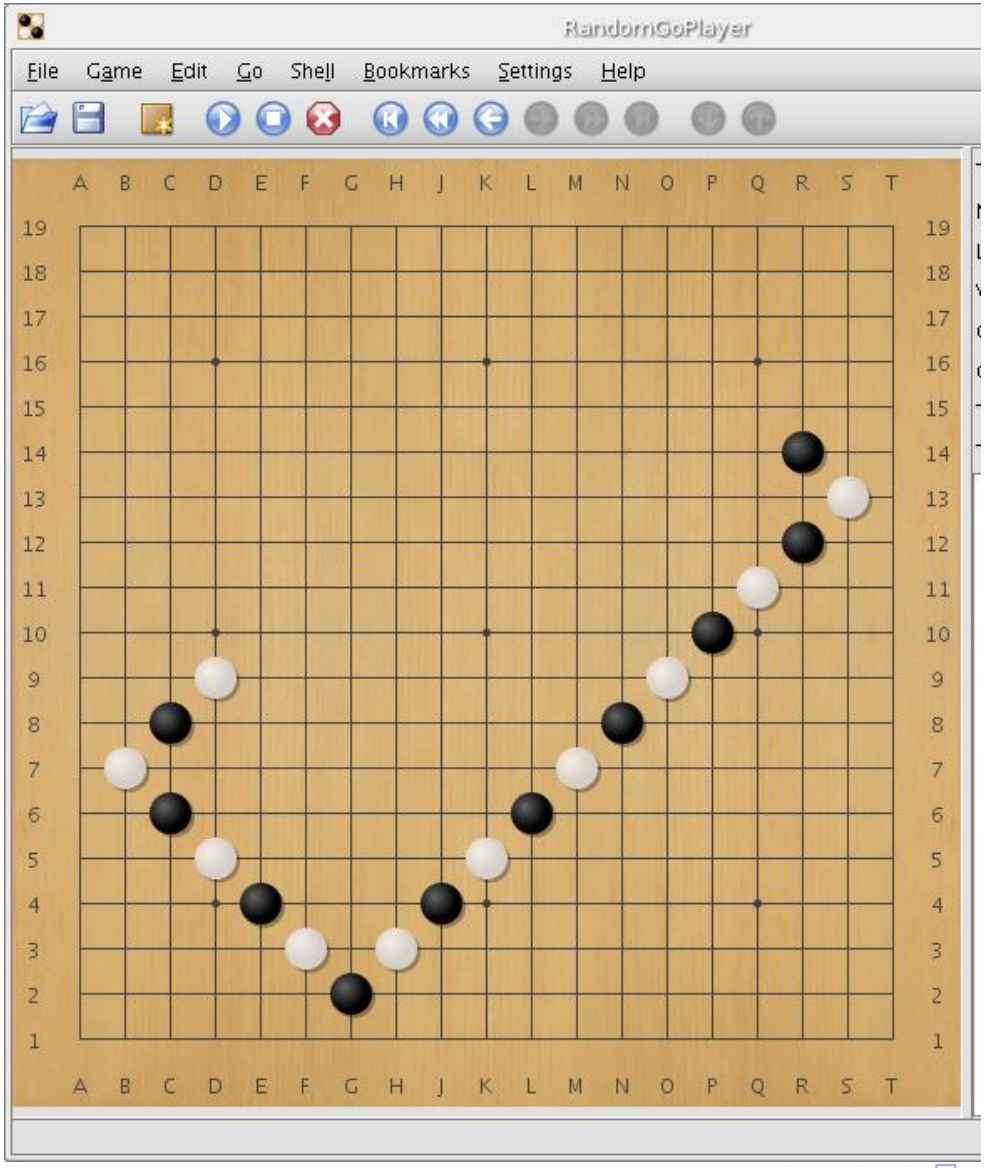
Optimizing π

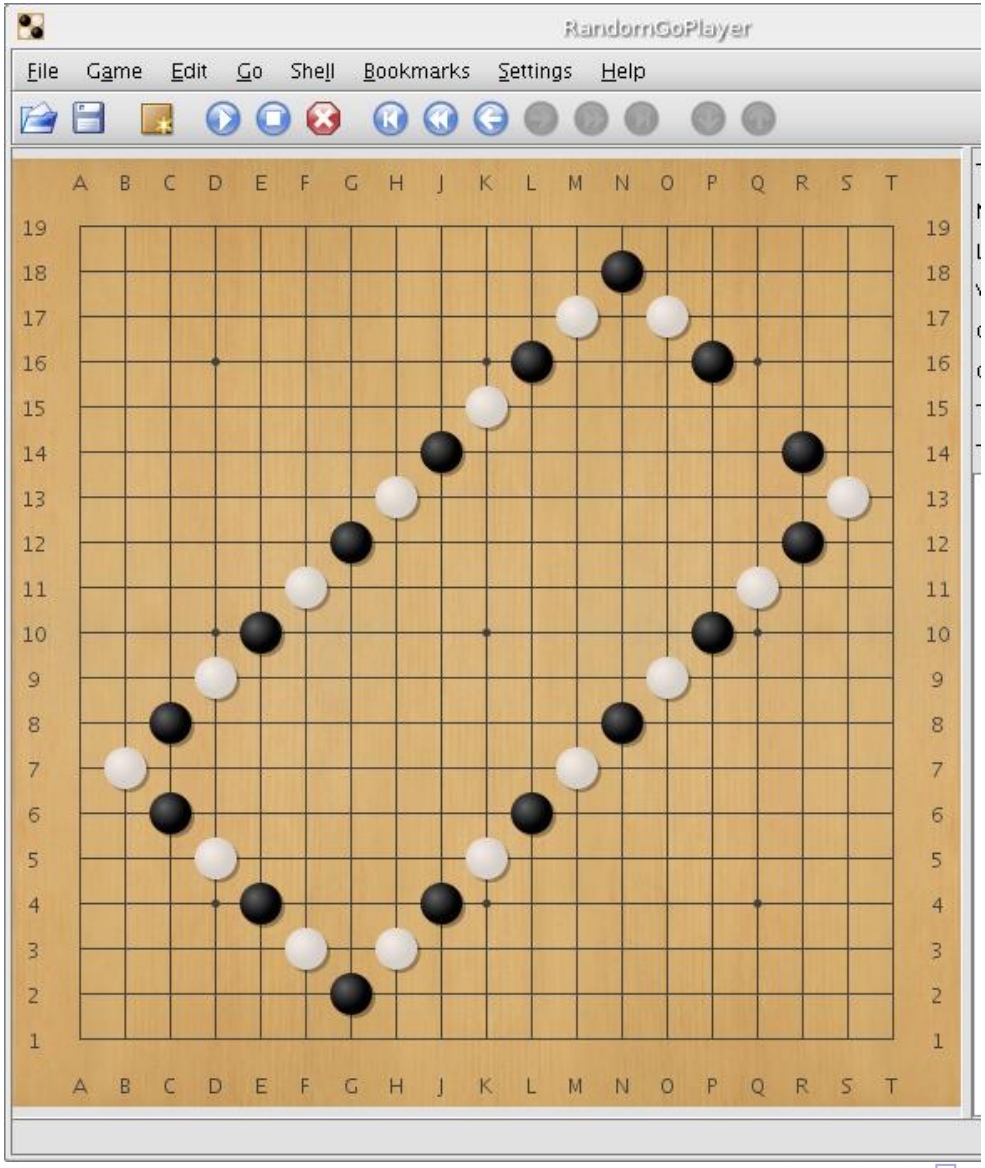
- Non parametric policy search;
- π has almost the same structure as previously described: we optimize the patterns;
- Reward: average on position evaluations.

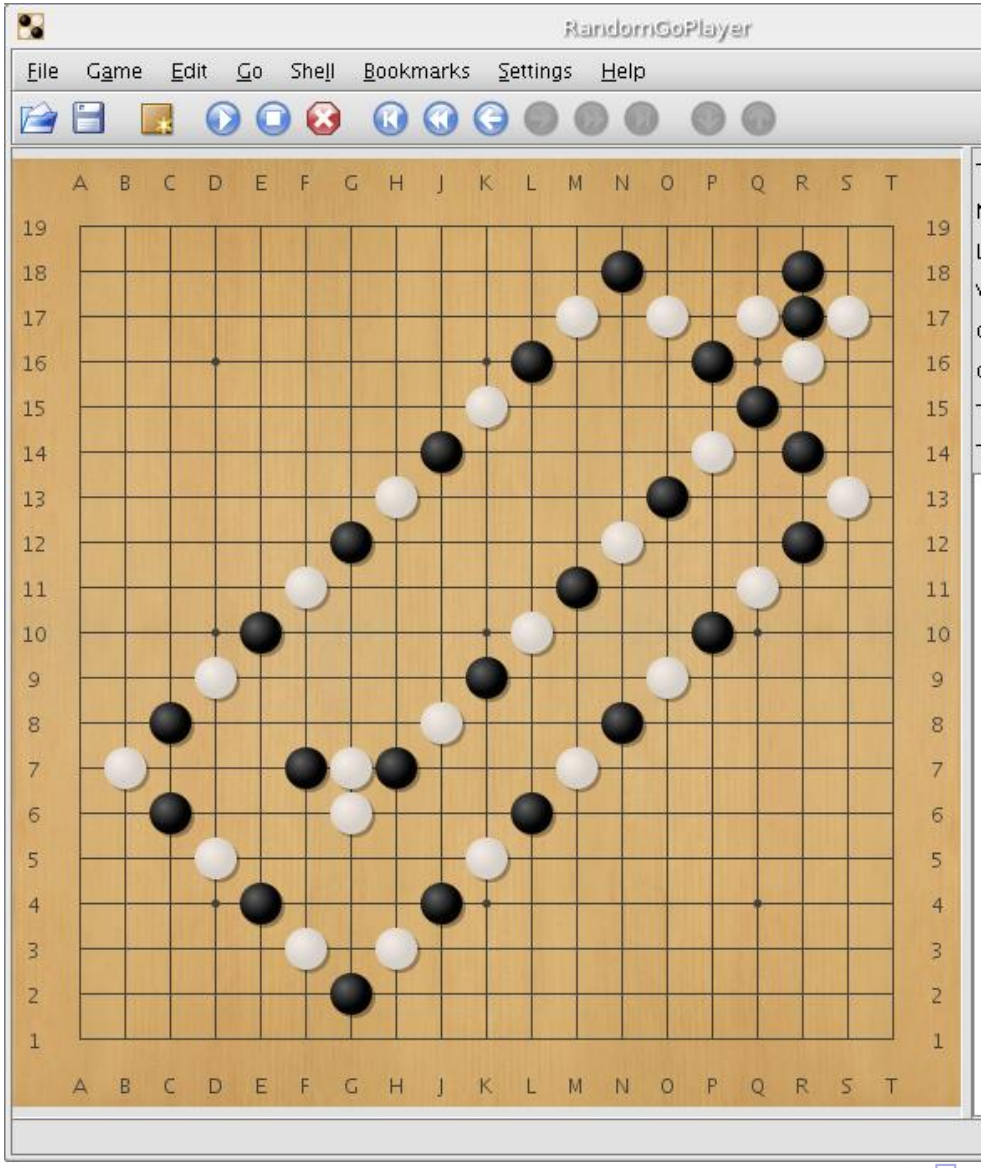
Results

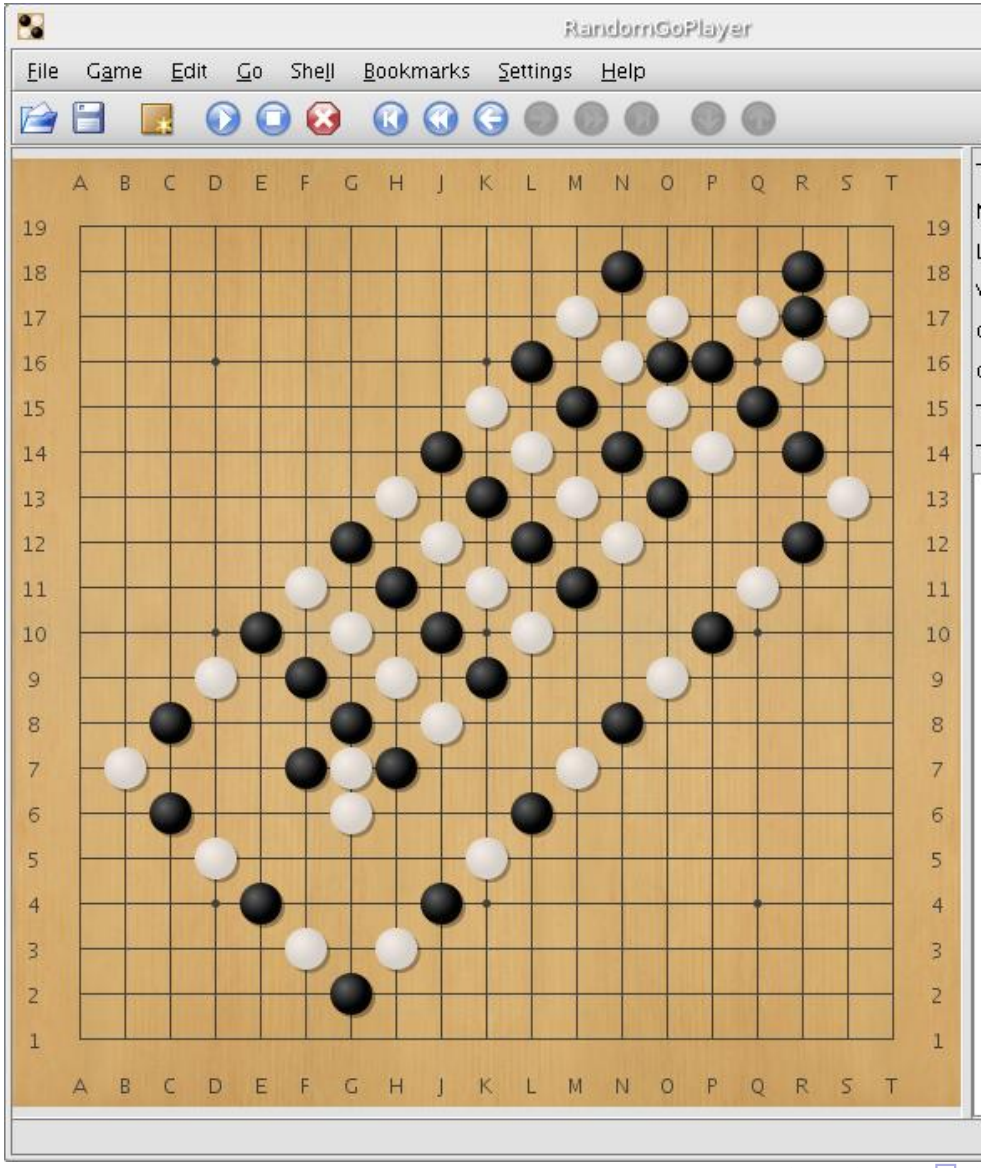
- winning rate of $MoGo(\pi)$ close to w.r. of $MoGo(\pi_{human\ patterns})$;
- we have a strong player with **almost no Go knowledge**!
- sometimes strange results.

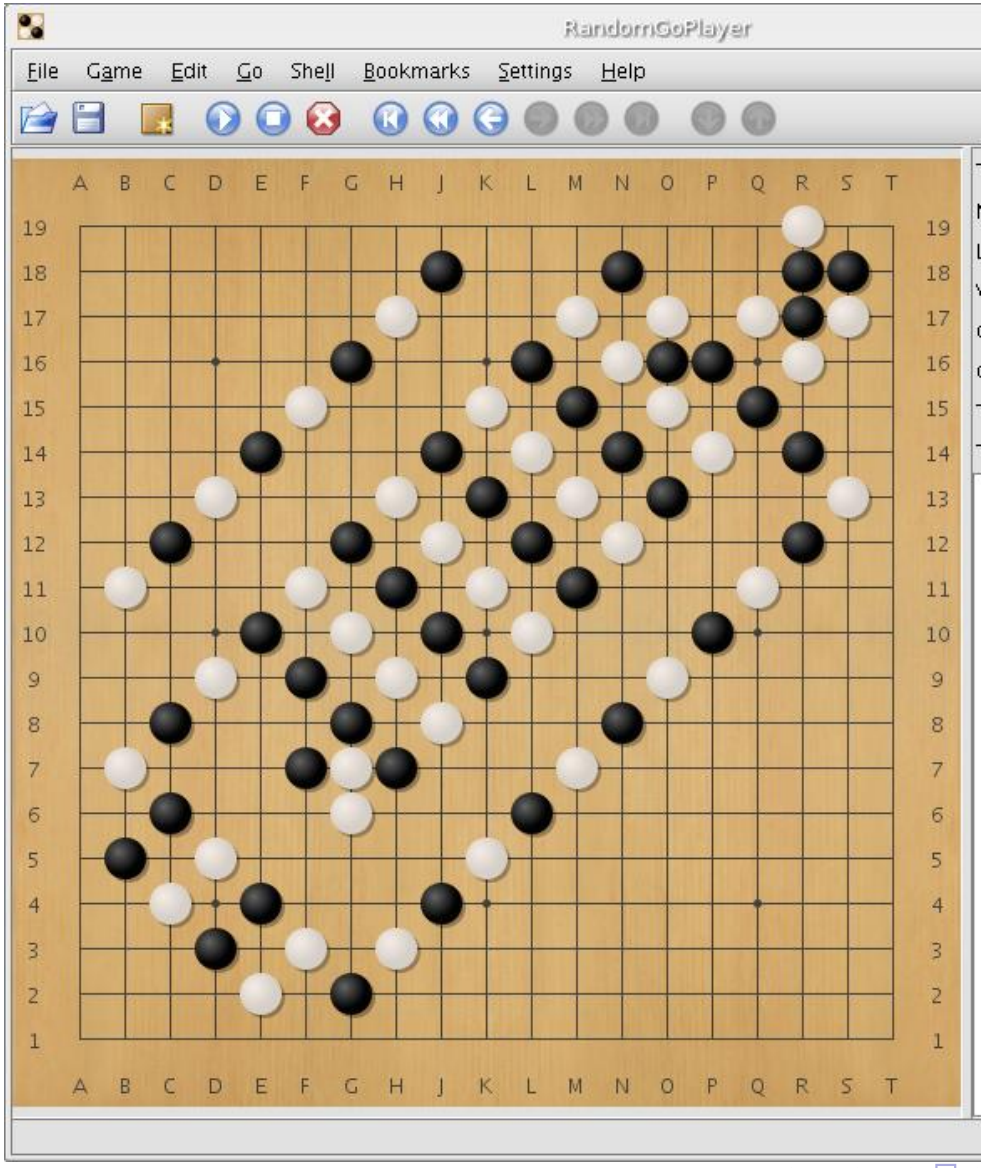

RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

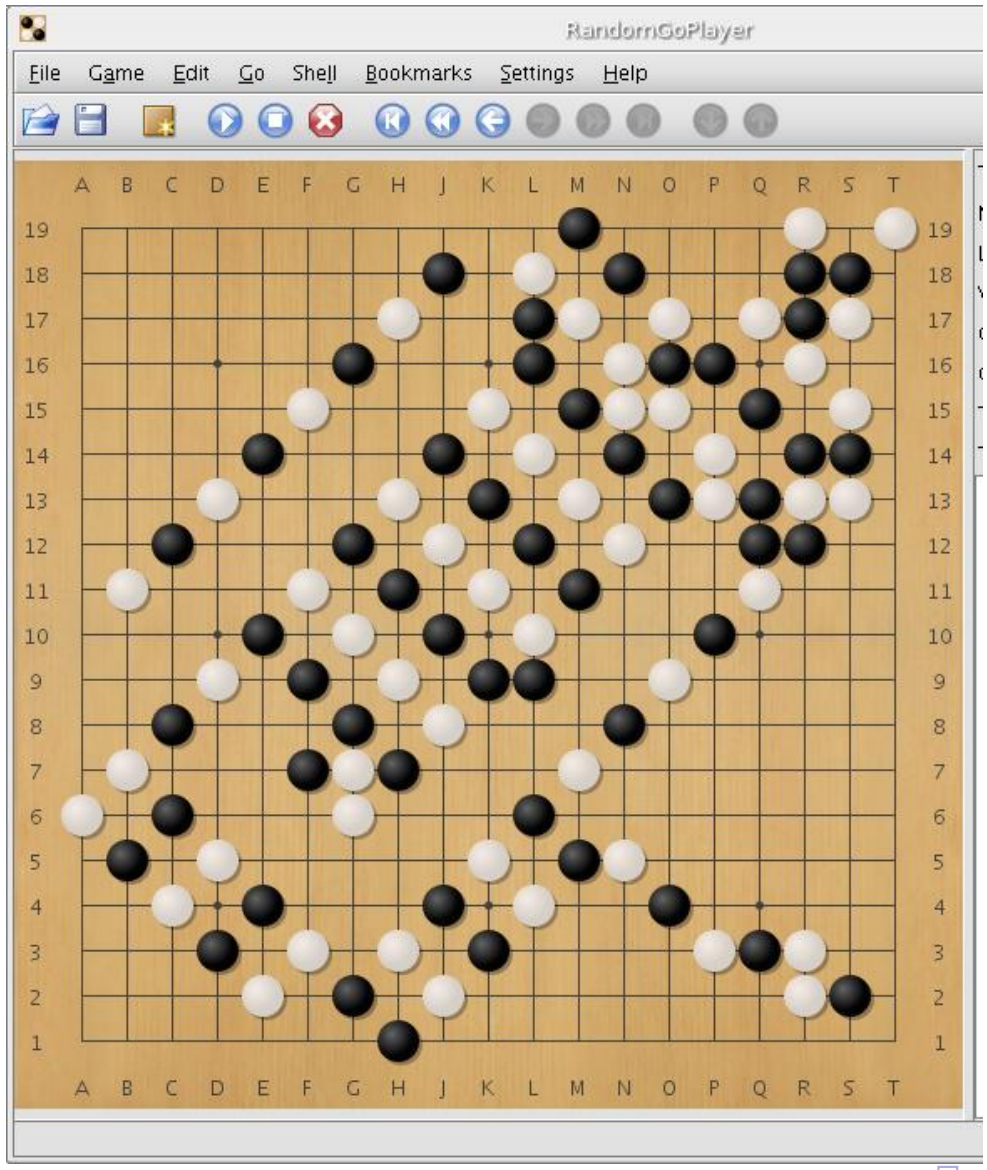

RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

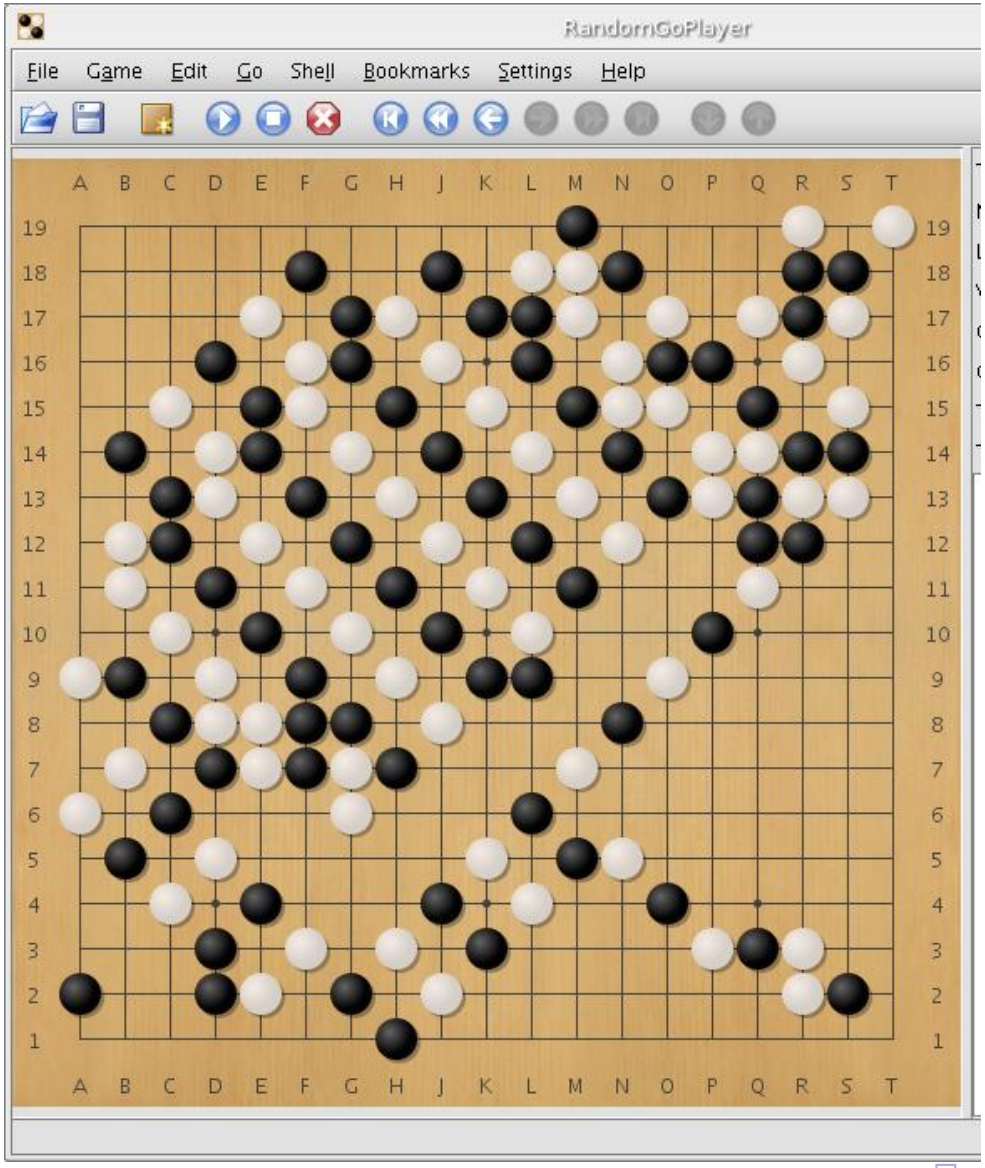

RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

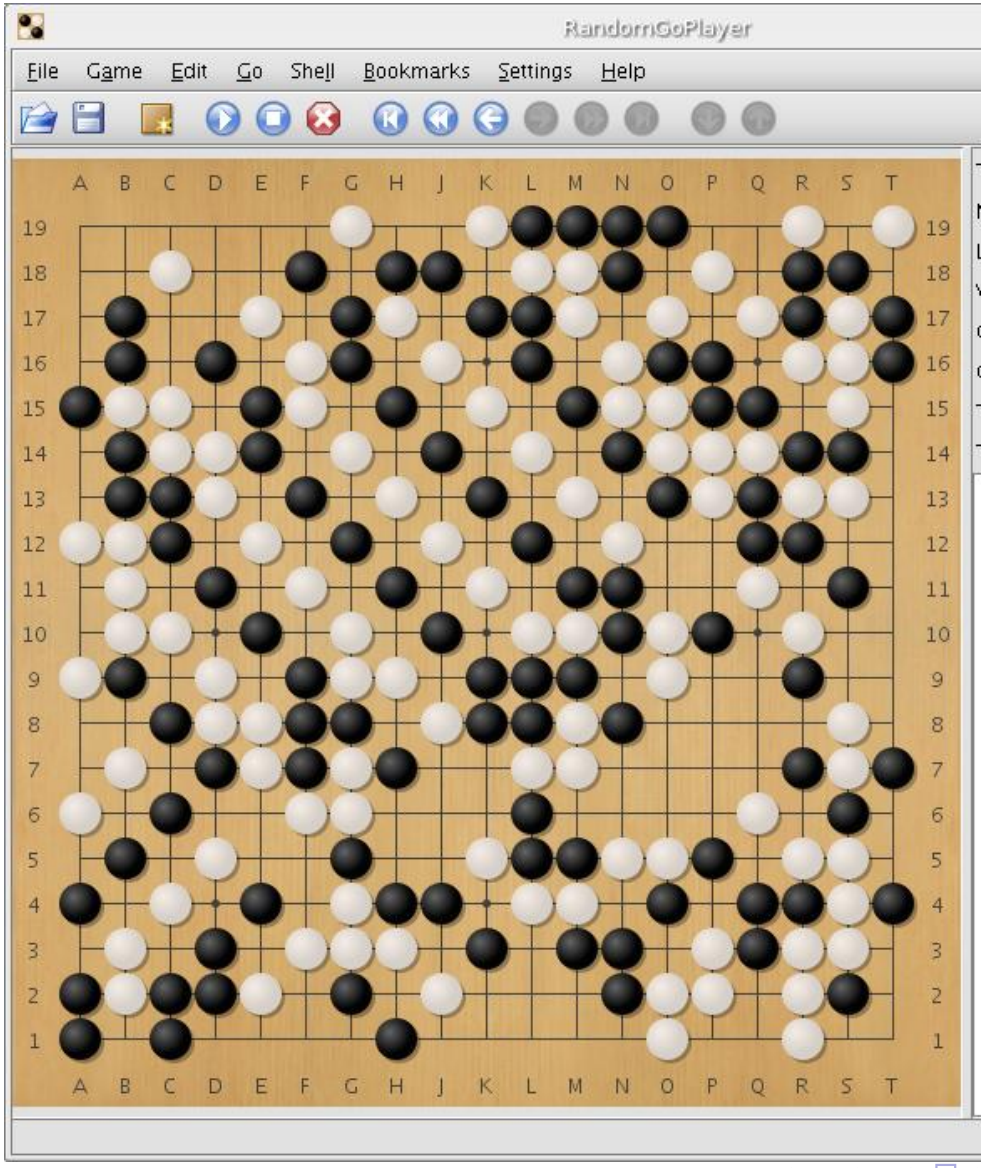

RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


RandomGoPlayer
File Game Edit Go Shell Bookmarks Settings Help
A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Beyond these results

Other approaches to get sequences

Position contains all information. But: Instead of searching $\pi : \mathit{position} \longrightarrow \mathit{move}$, we could think at:

- $\pi : \mathit{position} \times \mathit{moveSequence} \longrightarrow \mathit{move}$;
- $\pi : \mathit{startingPosition} \times \mathit{position} \times \mathit{moveSequence} \longrightarrow \mathit{move}$;
- $\pi : \mathit{position} \longrightarrow \mathit{sequenceOfMoves}$;
- any mixing combination.

It is also not mandatory to play legal moves, as long as the result is consistent with the starting position.

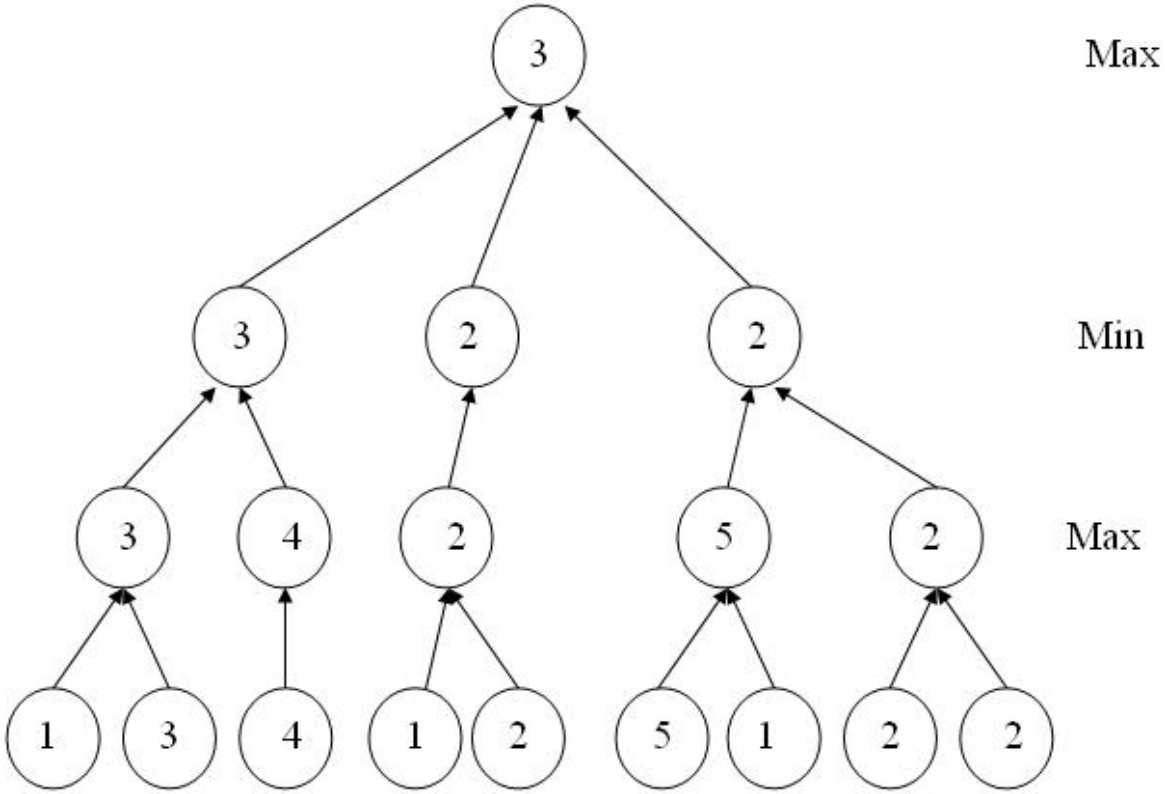
Tree based search: Outline

- Exploration-Exploitation: from alpha-beta to UCT
- Extensions to UCT
- Algorithmic issues

Tree based search

Minimax

We want to approximate the min-max value of the position.
Not necessarily the best strategy: we could try to model the opponent.



Alpha-Beta algorithm

Alpha-Beta computes the minimax value in a tree (exact given an evaluation function).

UCT algorithm

Game as a multi-armed bandit

- each position is a bandit;
- each move is an arm;
- play the best move $\longleftrightarrow$ maximize the reward.

UCT in Go

MoGo was the first Go program to use UCT (July 2006).

UCB and UCT algorithms

UCB algorithm (P. Auer et al.) 2002

- let $\hat{X}_i$ the empirical average rewards for ith arm;
- let T_i the number of trials for arm i;
- let $T = \sum_i T_i$

Then iteratively:

- if one arm has not been played, play it;
- else, play the arm maximizing $\hat{X}_i + \sqrt{2\frac{\log T}{T_i}}$.

UCB and UCT algorithms

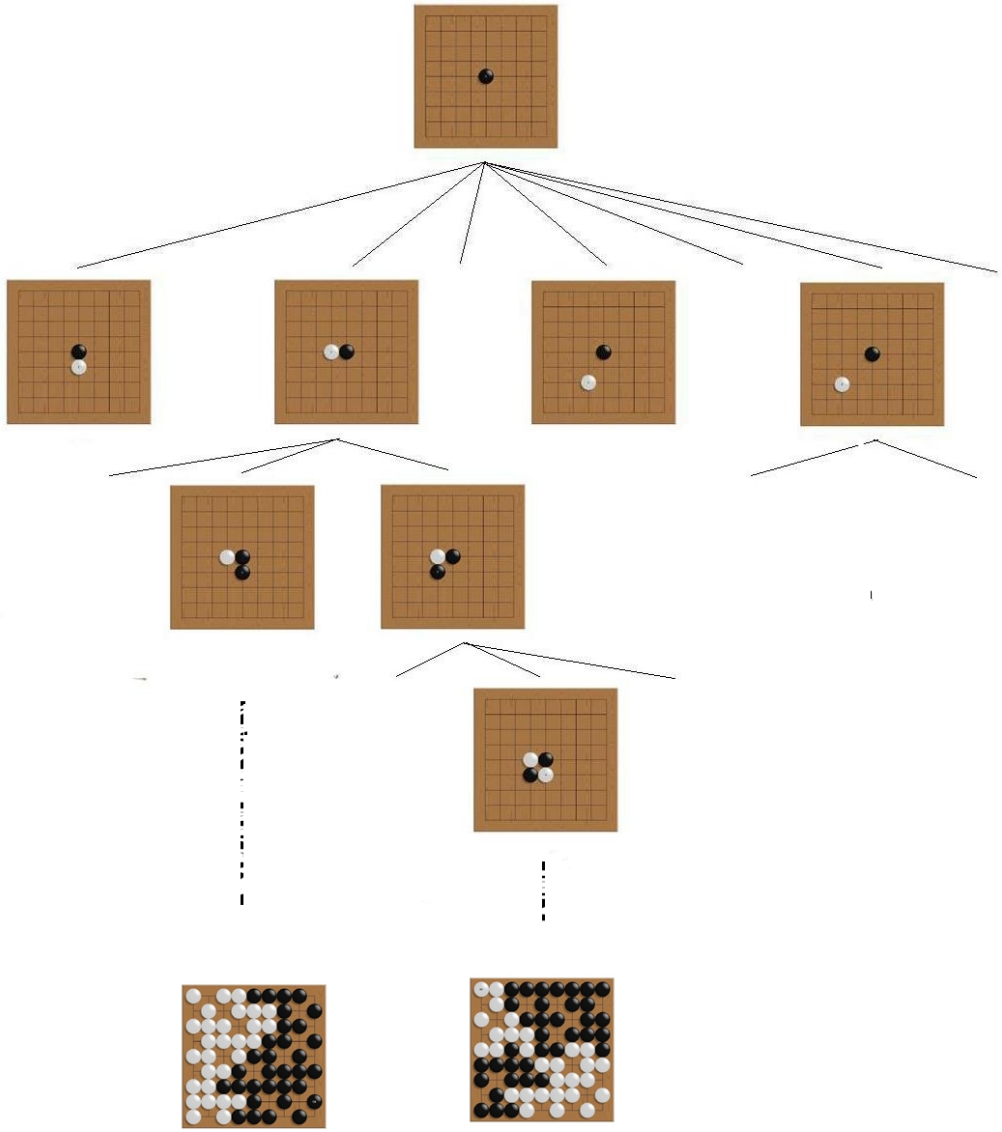
UCT algorithm (L. Kocsis and C. Szepesvari. 2006)

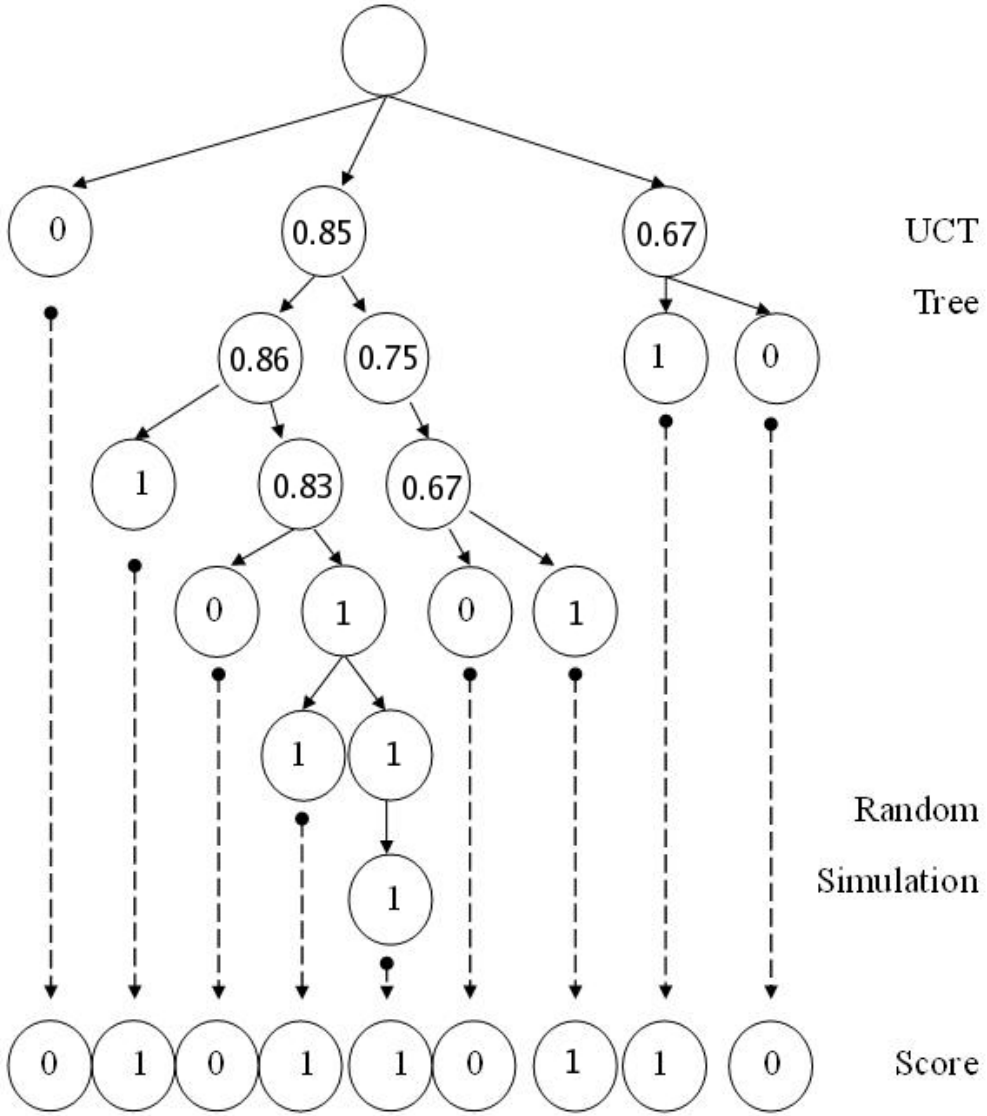
- start from the root;
- until stopping criterion (e.g. the end of the game):
  - choose a move according to UCB;
  - update the position.
- score the game;
- update all visited nodes with this score (without discount).

Efficient memory management

Tree management after CrazyStone (R. Coulom 2006).

- Stop as soon as UCT gets an unseen position;
- add this node to the tree;
- evaluate the position.




0
0.85
0.67
UCT
Tree
0.86
0.75
1
0
1
0.83
0.67
0
1
0
1
1
1
1
Random
Simulation
0
1
0
1
1
0
1
1
0
Score

Value of a position/move

$value(move) = value(position_t + move)$
$value(position) = \frac{1}{T} \sum_i T_i value(move_i)$
$value(position) \rightarrow value(bestMove)$

Why it is efficient compared to alpha-beta?

- Alpha-Beta never reconsider a cut; dangerous with random non accurate evaluation function;
- value(node) $\rightarrow$ max(node) as confidence increases.
- efficient tree exploiration
  - breadth first search;
  - move ordering efficiently managed;
  - asymmetric growing;
- anytime.

Stationarity

Bandit problems are not stationary in the tree:
distribution of each arm change after each addition of a descendant.

- leaf node: reward $\sim$ evaluation function distribution
- addition of a subnode $\rightarrow$ draws the distribution toward max;
- we should use at least a discount factor;
- that gives bad results.

Stationarity: results with discount

Some explanations:

- we iteratively find a good move, then its refutation: algorithm must be robust to that;
- if we took a long time to find a refutation, then it is difficult for the opponent too;
- at a level we compare rewards with subtrees of different depths $\longrightarrow$ prevent give advantage to shallower subtrees.

Improvements

Specific extensions required

- from asymptotic quality to efficient move selection;
- # trials $<$ # moves;
- arms are not independent.

Improvements in UCT when # trials $\approx$ nb arms

First Play Urgency

Starting with playing all arms is not optimal; Let c a default constant. Let X_i' such that

- $X_i' = \hat{X}_i + \sqrt{2\frac{\log T}{T_i}}$ if $T_i > 0$;
- $X_i' = c$ if $T_i = 0$;

Choose the highest X_i'.
Empirically $c \approx 1 \rightarrow +50$ ELO.

Variant in UCB

Tested formulas

With previous notation and $\hat{\sigma}_i$ the empirical standard deviation of rewards for arm i, we experimented:

- $\hat{X}_i + p\sqrt{\frac{\log T}{T_i}}$
- $\hat{X}_i + max(p\hat{\sigma}_i\sqrt{\frac{\log T}{T_i}}, \epsilon)$
- $\hat{X}_i + p\sqrt{\frac{\log T}{T_i}\min\{1/4, V_i\}}$, with $V_i = \hat{\sigma}_i{}^2 + q\sqrt{\frac{\log T}{T_i}}$
- $\hat{X}_i + p\hat{\sigma}_i\sqrt{\frac{\log T}{T_i}} + q\hat{\sigma}_i\frac{\log T}{T_i}$

Exploiting dependencies

Share information between arms

There is no independence between arms vertically and horizontally.
The goal is to improve the performance when T is small.

- Average results from neihbor moves (add a term $\frac{1}{|\mathcal{N}_i|}\sum_{j\in\mathcal{N}_i}\hat{X}_j$);
- use results from ancestors: set a c_i for each move according to $\hat{X}_i$ of its grandfather.

Parallelization

Parallelization

Results showed that MC/UCT algorithm scales well.
Parallelization open great perspectives in Go and other applications.
We tried two kind of parallelizations:

- multi-processors computer with shared memory
- parallelization on a cluster

Parallelization

Shared memory parallelization

Straightforward parallelization:
3 methods:

- DescendTheTreeUsingUCT
- MCSimulation
- UpdateTheTree

Algorithm loop:

```
mutex.lock(); DescendTheTreeUsingUCT(); mutex.unlock();
MCSimulation();
mutex.lock(); UpdateTheTree(); mutex.unlock();
```

Shared memory parallelization: Results

- Not equivalent to the "mono-threaded UCT";
- with the same number of simulations, same performance (4 threads);
- as it allows more simulations per second $\longrightarrow$ +100 ELO on CGOS;
- what happens with many more processors?

Parallelization on a cluster

- Time of one simulation $<$ communication time
  $\longrightarrow$ regroup nodes;
- We tried several algorithms but not significant results so far;
- However there are great perspectives.

Conclusion

Summary

- Monte-Carlo Go program with *sequence like* simulations
- MoGo first Go program using UCT
- Specific adaptation of UCT improving non asymptotic behavior
- Algorithm issues: parallelization of UCT (12000 nodes/second, 400000 nodes/move)

Perspectives

Further works

- Improvements in the simulation policy
- Shifting towards exploitation for 19×19 Go boards
- Exploiting arm dependencies

Thank You

Thank you

- play against MoGo on KGS