

# Learning an Evaluation Function for Go

**Yucheng Low**  
ylow@andrew.cmu.edu

March 21, 2008

## Abstract

The Common Fate Graph is a convenient representation of a Go Board due to its transformation invariant properties. I propose and evaluate two methods of applying the Common Fate Graph to learn an evaluation function for Go in a supervised learning setting. The first method uses is derived from NeuroGo and uses a Neural Network with a dynamic topology. The second method involves the application of Markov Random Fields on the Common Fate Graph. We then investigate the use of the evaluator in a Monte-Carlo Go system.

## 1 Problem

The game of Go is practically the Holy Grail for computer game playing due to its massive branching factor and difficult evaluation. The current state of the art computer program is only able to play at an amateur level. Additionally, the computer programs tend to have specific weaknesses which can be targeted by professional human players. Most notably, in 1998, an amateur 6 dan player beat the state of the art program with a 29 stone handicap.

Even If we simplify the game by reducing the size of the board from 19 by 19 to 9 by 9, the number of possible game states fall to around  $3^{81} \approx 10^{38}$  which is around the complexity of international chess ( $\approx 10^{43}$  game states). Even so, the best computer players for 9x9 still only play at an amateur level. This is primarily due to the difficulty of coming up with an effective evaluation function.

## 2 Implementation

A system has been implemented to convert any given board configuration into a transformation (rotation/flip) invariant representation of the board called a Common Fate Graph[5]. This representation is then transformed into a neural network using the same scheme as [3]. Training has been performed using 113 professional games. Details of the procedure are described below. The simple configuration in Figure 1 will be used as an example.

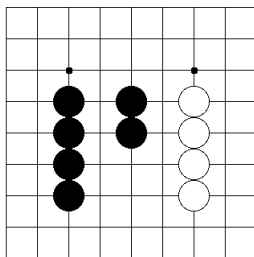


Figure 1: Sample Board

## 2.1 Augmented Common Fate Graph

The intuition behind the common fate graph is to represent the board as an undirected graph and combining vertices which will share a common outcome (life/death/belongs to white/belongs to black). The procedure works as follows:

### 2.1.1 Chain finding

Every piece on the board is tagged with a “chain number”, such that if two pieces of the same color are connected, they will have the same chain number. For instance, in the example, the four black stones on the left will be labeled as “Chain 1”, the two black stones in the middle will be “Chain 2” and the four white stones on the right will be “Chain 3”.

### 2.1.2 Pattern Matching

A small collection of simple hand constructed connecting patterns is used to relate chains of stones on the board. All transformations of the patterns (rotation and flipping) are searched. In the example above, a “Bamboo Joint” pattern will be found connecting the chain on the left and the chain in the middle.

### 2.1.3 Influence

This is a further extension upon the Common Fate Graph scheme where I propose to merge adjacent empty intersections using an influence function. The intuition behind this is that contiguous regions under strong influence from one party, is not likely to be contested and will therefore share a common outcome.

A simple influence function is used called the Bouzy 5/21 algorithm [1]. This algorithm is widely used as it closely matches a human’s intuitive understanding of influence. The basic procedure is the use of mathematical morphology operators. We first set all black pieces to a value of “100”, and all white pieces to a value of “-100”. (These values are arbitrary. Just as long as they are large). We then perform 5 dilations followed by 21 erosions.

The outcome of performing the Bouzy 5/21 algorithm on the board above is shown in Figure 2. Regions falling under strong influence are grouped as a “chain”.

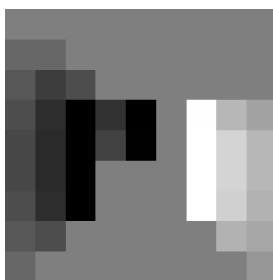


Figure 2: Bouzy 5/21 Influence

### 2.1.4 Basic Tags

Finally, each chain is tagged with the following information:

#	Label	Valid Values
1	Type	Black, White, Empty
2	Eye Count (Both true and false eyes)	1,2,3,4, $\geq 5$
3	Number of intersection in chain	1,2,3,4, $\geq 5$
4	Number of liberties	1,2,3,4,5, $\geq 6$

### 2.1.5 Graph construction

A graph is first constructed where each intersection has an edge to its neighbors. Then the chain information derived in 2.1.1 and 2.1.3 is used to merge vertices. Labels are preserved.

## 2.2 Neural Network

### 2.2.1 Construction

A 3 layer (1 hidden layer) network is constructed, where each layer is a copy of the graph constructed in 2.1 excluding edges. An edge is added between two vertices of two adjacent layers, if there is an edge between those two vertices in the Common Fate Graph.

In other words. Let the Common Fate Graph be represented by  $G = \{V, E\}$ , where  $V = \{v_1, v_2 \dots v_n\}$  and the the Neural Net be represented by  $\{V^1, V^2, V^3, V^4, E\}$ , where  $V^i$  consists of the vertices in layer  $i$ , and  $V^i = \{v_1^i, v_2^i, v_3^i \dots v_n^i\}$

Then there is a edge between  $v_j^1 \in V_1$  and  $v_k^2 \in V_2$  iff there is an edge  $(v_j, v_k) \in E$ .

Edge weights  $W(v_i, v_j)$  are identified by the tuple  $(\text{Label}(v_i), \text{Label}(v_j), \text{Layer}(v_i), \text{Layer}(v_j), \text{Label}((v_i, v_j)))$ . (i.e. Layer and label of the source vertex, layer and label of the destination vertex, and label of the edge)

Vertex biases  $W(v)$  are identified by the tuple  $(\text{Label}(v), \text{Layer}(v))$ .

By construction there are as many output values as input values.

### 2.2.2 Execution

The neural network is executing by feeding in a vector of all 0.5's. The reason behind this is to use the neural network to learn the input necessary through the biases in the first layer, rather than require a separate training scheme.

There is a reasonable interpretation of this scheme. We can think of the input values as the initial "beliefs" regarding the ownership of particular vertices. Then during the neural network forward propagation, each vertex will take the ownership "beliefs" of its neighbors and use them to determine its own belief value. This value is then propagated forward down the neural network.

This interpretation allows me to treating the network as a recurrent network. That is, on execution, feed the output back into the input for some predetermined number of times or until convergence.

This has not been evaluated completely, but preliminary results are available.

### 2.2.3 Training

Training is performed by playing against Training is done by standard backpropagation. However, since two edges might share the same weights, shared weights must be taken into account.

## 2.3 Pairwise Markov Random Field

Another simple method of analysing the augmented common graph is to interpret it as a pairwise Markov Random Field,

### 2.3.1 Construction

The construction method is very similar to that of the neural network construction in 2.2.1. Let the graph be represented by  $G = \{V, E\}$  where  $V$  is the set of vertices and  $E$  is the set of edges. Every vertex can take on one of two possible values (0 or 1, where 1 means that the vertex classified as Black's territory). The unary potentials  $\psi_i(v_i)$  are uniquely identified by  $\text{Label}(v_i)$ , and the binary potentials  $\phi_{ij}(v_i, v_j)$  are uniquely identified by the tuple  $(\text{Label}(v_i), \text{Label}(v_j), \text{Label}((v_i, v_j)))$ .

Then the joint probability of any given value assignments to the vertices/variables  $v_1 \dots v_{|V|}$  is

$$P(v_1 \dots v_{|V|}) = \frac{1}{Z} \left( \prod_{v_i \in V} \psi_i(v_i) \right) \left( \prod_{(v_i, v_j) \in E} \phi_{ij}(v_i, v_j) \right) \quad (1)$$

### 2.3.2 Execution

Given the potential values, evaluation is performed by evaluating the marginal probabilities  $P(v_i)$ . This can then be directly interpreted as the probability that  $v_i$  is black's territory. The marginal probabilities were originally computed using Loopy Belief Propagation. This was later changed to Damped Belief Propagation [6] due to convergence concerns. Other Double Loop algorithms [6, 7] with guaranteed convergence were considered but not implemented due to their relative complexity.

During the period of testing, no non-convergent situations were met. There is also no guarantee that Belief Propagation will return the true values of the marginals even when convergent. However, solving for the exact marginal probabilities is known to be an NP-Hard problem.

### 2.3.3 Training

Training is complicated by the inability to produce a sufficiently large training set representative of the problem; therefore batch methods cannot be used. An incremental training scheme is necessary to allow for gradual improvisation while training data is generated on the fly (through self-play or by playing against another AI). Additionally, assumptions cannot be made regarding the class of potential functions used, therefore requiring the system to store and train the full parameterization of the potentials (i.e.  $\psi_i \in \mathbb{R}^2, \phi_{ij} \in \mathbb{R}^{2 \times 2}$ )

Training is performed through incremental gradient descent on single examples, with the gradient direction aiming to maximize the joint probability of the example. A simplifying assumption is made that every potential is only used once in each graph even though this is evidently not true.

Assume a single training example  $(x_1 \dots x_n)$  with labels  $(y_1 \dots y_n)$ . Then from 1

$$P(x_1 \dots x_n) = \frac{1}{Z} \left( \prod_{v_i \in V} \psi_i(x_i) \right) \left( \prod_{(v_i, v_j) \in E} \phi_{ij}(x_i, x_j) \right) \quad (2)$$

$$\log P(x_1 \dots x_n) = -\log Z + \left( \sum_{v_i \in V} \log \psi_i(x_i) \right) + \left( \sum_{(v_i, v_j) \in E} \log \phi_{ij}(x_i, x_j) \right)$$

Taking derivatives about a single potential value  $\psi_k(x_k)$

$$\frac{d \log P(x_1 \dots x_n)}{d\psi_k(x_k)} = -\frac{1}{Z} \frac{dZ}{d\psi_k(x_k)} + \frac{1}{\psi_k(x_k)} \quad (3)$$

Consider  $\frac{dZ}{\psi_k(x_k)}$ . Let  $W$  be the set of neighbors of  $v_k$ .

$$\begin{aligned}
Z &= \sum_{\text{all } \vec{v}} \left( \prod_{v_i} \psi_i(v_i) \right) \left( \prod_{(v_i, v_j) \in E} \phi_{ij}(v_i, v_j) \right) \\
\frac{dZ}{d\psi_k(x_k)} &= \sum_{\text{all } \vec{v}} \mathbf{1}(v_k = x_k) \left( \prod_{v_i / (\text{excluding } v_k)} \psi_i(v_i) \right) \left( \prod_{(v_i, v_j) \in E} \phi_{ij}(v_i, v_j) \right) \\
&= \frac{1}{\psi_k(x_k)} \sum_{\text{all } V} \mathbf{1}(v_k = x_k) \left( \prod_{v_i = V} \psi_i(v_i) \right) \left( \prod_{(v_i, v_j) \in E} \phi_{ij}(v_i, v_j) \right) \\
\therefore \frac{\frac{dZ}{\psi_k(v_k)}}{Z} &= \frac{P(v_k = x_k)}{\psi_k(x_k)} \\
\frac{dZ}{\psi_k(v_k)} &= P(v_k = x_k) \times \frac{Z}{\psi_k(x_k)} \tag{4}
\end{aligned}$$

Substituting back into 3

$$\frac{d \log P(x_1 \dots x_n)}{d\psi_k(x_k)} = \frac{P(v_k = x_k) + 1}{\psi_k(x_k)}$$

Then, to increase the log-likelihood of the example, the gradient update is  $\psi_k(x_k) \leftarrow \psi_k(x_k) + \epsilon \frac{P(v_k = x_k) + 1}{\psi_k(x_k)}$ .

A similar argument can be made for the pairwise potentials  $\phi_{ij}$ .

## 3 Training using Gnugo

### 3.1 Training

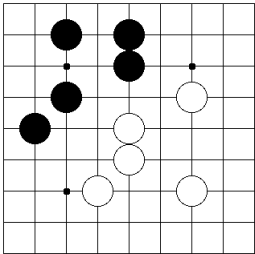
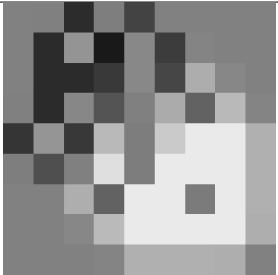
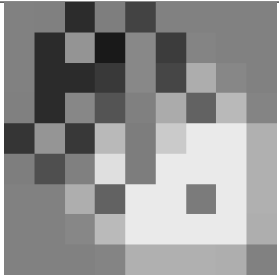
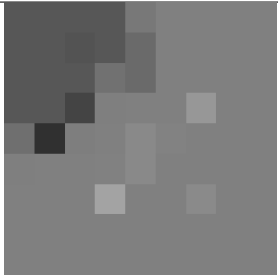
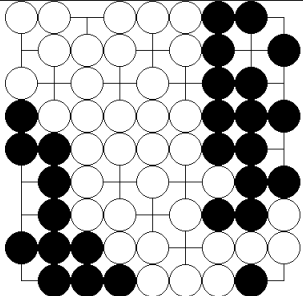
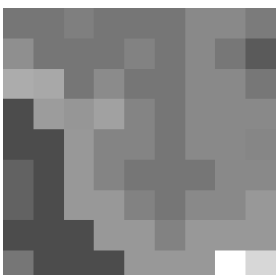
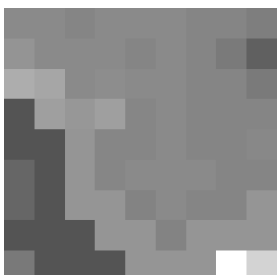

Gnugo’s territorial evaluation is taken as the ground truth. 113 professional games taken from the GoGod database are used. For each game, the last 10 positions of the game are taken and the final territory division as evaluated by Gnugo is used as the expected output.

It is noted that the models above require very large number of weights to be created, and this training set is clearly insufficient. Therefore, further training is performed by playing it against Gnugo at level 1 and training it against Gnugo’s evaluator two moves into the future. (That is, the current board position is taken as input, and Gnugo’s evaluator two moves later is taken as output.) To prevent overfitting to small number of positions, a probabilistic scheme is used where each move is weighted by its value using the formula  $e^{\frac{\text{value}}{0.35}}$ .

The Neural Network is trained to 4000 iterations, while the Markov Random Field is trained to 2200 iterations (due to limited time, and that MRF evaluation is extremely slow).

### 3.2 Evaluation

A few simple hand constructed positions are tested. The board is first evaluated relative to Black, then relative to White, and the results averaged. Image contrast has been increased to facilitate viewing.

Input	NN Output	Recurrent NN output	MRF Output
			
			

The neural network works well for the relatively simple first position, but performs poorly for the second position. If the second position is simplified slightly by removing some white seeds from the center of the board, the output is improved significantly. Its failure to evaluate this position properly is perhaps caused by the use of some rare weights.

The MRF performs much better on the second position, accurately determining territory. But it performed poorly in the first position: incorrectly classifying white's probably territory in the bottom right.

The use of recurrency (3 iterations) improve the results of the the NN evaluation, now correctly classifying the centre group (albeit with a very small weight).

Both schemes have a tendency to classify occupied intersections as neutral (or even opponent's) territory. This can be explained by a limitation of the training scheme. Since the evaluator loses large groups to Gnugo frequently during training, it may end up believing that pieces can be captured easily and downgrades its confidence that they are actually territory.

### 3.3 Testing

The Neural Network is tested by using it as an evaluator in a 2-ply search against Gnugo at level 1. The MRF is tested similarly, but with a 1-ply search. 60 games are played, 30 as white and 30 as black. Since Gnugo has a small random factor built in into its engine, not all games are identical. The x-axis is in hundreds of iterations.

A trend can be seen for the Neural Network, but the MRF is not very convincing.

## 4 Training using Self-Play

### 4.1 Training

The use of self-play through temporal difference learning (typically TD(0)) to train an evaluator has began with Tesauro's TD-Gammon. This was also used in a variety of systems for training a Go evaluator ([4, 3]). the use of TD(0) makes the training scheme extremely simple: The network is given the output of the evaluator one move later. At a terminal position (end of game), the network is given the ground truth (as evaluated using Gnugo).

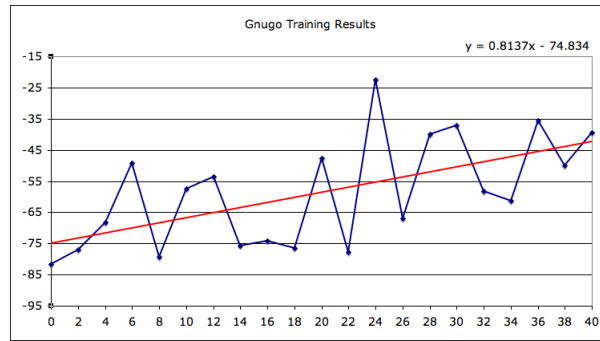


Figure 3: Neural Network Test Results

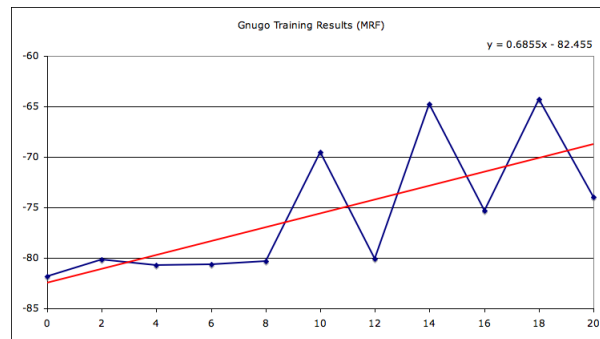
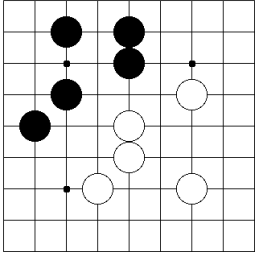
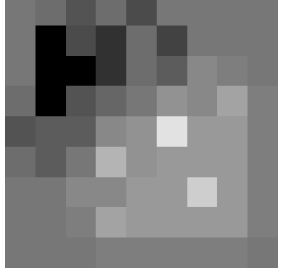
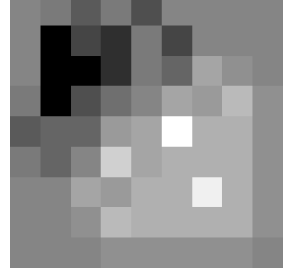
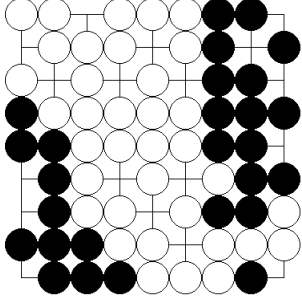

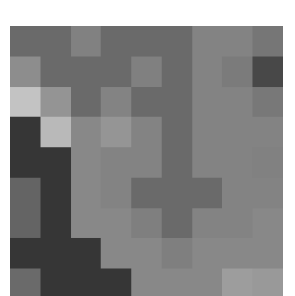


Figure 4: MRF Test Results

The Neural Network is trained to 5500 iterations. Self play using the Markov Random Field scheme has not been evaluated.

## 4.2 Evaluation

A few simple hand constructed positions are tested. The board is first evaluated relative to Black, then relative to White, and the results averaged. Image contrast has been increased to facilitate viewing.

Input	NN Output	Recurrent NN output
		
		

Results are similar to Gnugo evaluation. Except that the recurrent output for the second case is worse.

## 5 Conclusion

Neural Networks perform reasonably well with a good learning trend up to about 1500 iterations. It is not certain why performance decreases after 1500 iterations, but it may be due to a subtle form of overfitting caused by the differences between training scheme (randomized play) and the testing scheme (deterministic play). Selfplay using TD(0) as suggested by NeuroGo [4] should be able to prevent this and is a direction to be investigated in the future.

Increasing the depth of the neural network from 3 to 4 improves performance though complete results are not available at time of writing.

MRF's produce decent performance but due to its poor evaluation speed, a deeper search cannot be performed. This limits the use of MRFs as an evaluation function. Additionally, the number of free parameters in the MRF scheme is much larger than the number of free parameters in the Neural Network scheme; therefore more training iterations will be necessary to achieve reasonable results. Additionally, possible non-convergence of belief propagation is a concern. The MRF scheme however has a much stronger theoretical basis than the Neural Network scheme.

Also deterministic play as method of evaluating the strength of an evaluation function has limited value, as it is possible for a stronger evaluation function to lose more frequently than a weaker evaluation function due to particular subtleties in its play which Gnugo is able to exploit. Randomized play for testing makes sense only if both players are using randomized play. Otherwise the deterministic player has an extremely large advantage.

### 5.1 Testing

The Neural Network is tested by using it as an evaluator in a 2-ply search against Gnugo at level 1. 60 games are played, 30 as white and 30 as black. Since Gnugo has a small random factor built in into its engine, not all games are identical. The x-axis is in hundreds of iterations.





Figure 5: Self Play Neural Network Test Results

As expected, the rate of improvement of self-play is slower than the rate of improvement of training against Gnugo.

## 6 Extending Monte-Carlo Go

It is clear that a search tree scheme is highly inappropriate due to the depth and breadth of the Go. Recent work in 9x9 Go suggested the use of Monte-Carlo search. A specific Monte-Carlo scheme called Upper Confidence Trees (UCT) was implemented [8].

### 6.1 Summary of UCT

UCT describes an online learning scheme for the K-armed bandit problem. The K-arm bandit problem is an analogy to a slot machine with K arms. Pulling an arm results in a reward with some probability associated with the arm. The aim is to maximize sum of rewards. The reward probability for each arm is unknown.

The paper [8] describes such an algorithm for determining the best move.

A search tree is grown iteratively. In each iteration, one leaf of the tree is expanded.

UCT Loop Body:

```

Let  $c$  = root of search tree
While  $c$  is not a leaf
  Let  $c = \max_{V_c(s)}(Children(c))$ 
End While
Insert all possible moves after  $c$  as children of  $c$ 
Pick a random child of  $c$ 
Play a random game, returning 0 or 1.

```

$$V_c(s) = \bar{X}_s + \sqrt{\frac{2 \log T_c}{T_s}}$$

where  $T_s$  is the number of times node  $s$  was played  
and  $\bar{X}_s$  is the expectation that  $s$  results in a win.

(number of times  $s$  resulted in a win, divided by number of times  $s$  was played)

Other subtleties such as the minimax scheme are left out of the description above. Additionally much work was done to require the “random game” to play only “reasonable” moves. This was improved the playing standard of the basic Monte-Carlo implementation significantly.

## 6.2 UCT Issues

UCT plays well only when the game is roughly balanced (both sides have a chance of winning). This is due to the use of the 0/1 valuation of the random game (Territorial values are too unstable). Therefore if the program believes it is winning, it will start to make weak moves. And if the program believes it is losing, it will begin to make unreasonable moves.

## 6.3 Application of Evaluator to Prune UCT Search

We will now concentrate upon the use of the final neural network trained using Gnugo, and applying it. A simple way of using the evaluator is to use it to prune moves and get the UCT search to concentrate on only reasonable moves. Due to the slow performance of the evaluator, we can only perform pruning on the first few levels of the search tree.

Additionally, the first move has a high likelihood of being played poorly.

## 6.4 Evaluation and Results

Since this is a modification on the UCT scheme, we use the basic UCT implementation as the baseline for evaluation. 20,000 iterations are used for both the basic UCT scheme and the Pruned UCT scheme. For each variation, 20,000 games are played. We only prune the top level of the tree. We are currently investigating pruning a few more levels. Initial experiments suggest good results.

Variation Parameters:

**Random Opening** To eliminate the problem of a poor first move, activation of this parameter causes the first move to be played in a random position in the  $3 \times 3$  grid in the center of the board

**$x$  Ply Eval** How many plys deep do I search using the evaluator to get a score for the position. if  $x = 0$ , we directly evaluate the position to obtain a score. if  $x = 1$ , we do a 1-ply search to get a score for the position.

Variation	Win Rate
Random Opening 0 Ply Eval	75%
No Random Opening 0 Ply Eval	85%
Random Opening 1 Ply Eval	70%
No Random Opening 1 Ply Eval	80%

It is interesting to note that that the 0 Ply Eval performs better on average than the 1 Ply Eval. Investigations in progress.

As expected. the program performs better without random openings as the pruning scheme will automatically prioritize better moves, where the straight UCT scheme has difficulties. Random openings “level the ground” for the opening allowing the naive UCT to play better openings.

## 7 Future Work

- Deeper investigation in MCGo variations
- Experiment with a scheme which uses subtrees as features for boosting (As opposed to paths in [5]).

## References

- [1] B. Bouzy (2003). Mathematical Morphology Applied to Computer Go
- [2] E. de Groot (2005). Machine Learning in Go.
- [3] Markus Enzenberger (1996). The Integration of A Priori Knowledge into a Go Playing Network
- [4] Nicol N. Schraudolph, Peter Dayan and Terrence J. Sejnowski (1994). Temporal Difference Learning of Positional Evaluation in the Game of Go.
- [5] Thore Graepel et al (2000). Go, SVM Go.
- [6] T. Heskes. "Stable Fixed Points of Loopy Belief Propagation Are Minima of the Bethe Free Energy"
- [7] A. L. Yuille. "A Double-Loop Algorithm to Minimize the Bethe and Kikuchi Free Energies". In Neural Computation 2001.
- [8] Sylvain Gello, et al (2006). "Modifications of UCT with Patterns in Monte-Carlo Go"