Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search

Rémi Coulom

Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search

Rémi Coulom

LIFL, SequeL, INRIA Futurs, Université Charles de Gaulle, Lille, France

Abstract. Monte-Carlo evaluation consists in estimating a position by averaging the outcome of several random continuations, and can serve as an evaluation function at the leaves of a min-max tree. This paper presents a new framework to combine tree search with Monte-Carlo evaluation, that does not separate between a min-max phase and a Monte-Carlo phase. Instead of backing-up the min-max value close to the root, and the average value at some depth, a more general backup operator is defined that progressively changes from averaging to min-max as the number of simulations grows. This approach provides a fine-grained control of the tree growth, at the level of individual simulations, and allows efficient selectivity methods. This algorithm was implemented in a 9×9 Go-playing program, Crazy Stone, that won the 10th KGS computer-Go tournament.

1 Introduction

When writing a program to play a two-person zero-sum game with perfect information, the traditional approach consists in combining alpha-beta search with a heuristic position evaluator [20]. The heuristic evaluator is based on domain-specific knowledge, and provides values at the leaves of the search tree. This technique has been very successful for games such as chess, draughts, checkers, or Othello.

Although the traditional approach has worked well for many games, it has failed for the game of Go. Experienced human Go players still easily outplay the best programs. So, the game of Go remains an open challenge for artificial-intelligence research [8].

Among the main difficulties in writing a Go-playing program is the creation of an accurate static position evaluator [15, 8]. When played on a 9×9 grid, the complexity of the game of Go, in terms of the number of legal positions, is inferior to the complexity of the game of chess [2, 27], and the number of legal moves per position is similar. Nevertheless, chess-programming techniques fail to produce a player stronger than experienced humans. One reason is that tree search cannot be easily stopped at quiet positions, as it is done in chess. Even when no capture is available, most of the positions in the game of Go are very dynamic.

An alternative to static evaluation that fits the dynamic nature of Go positions is Monte-Carlo evaluation. Monte-Carlo evaluation consists in averaging

the outcome of several continuations. It is an usual technique in games with randomness or partial observability [5, 23, 26, 14, 17], but can also be applied to deterministic games, by choosing actions at random until a terminal state is reached [1, 9, 10].

The accuracy of Monte-Carlo evaluation can be improved with tree search. Juillé [18] proposed a selective Monte-Carlo algorithm for single-agent deterministic problems, and applied it successfully to grammar induction, sorting-network optimization and a solitaire game. Bouzy [6] also applied a similar method to 9×9 Go. The algorithms of Juillé and Bouzy grow a tree by iterative deepening, and prune it by keeping only the best-looking moves after each iteration. A problem with these selective methods is that they may prune a good move because of evaluation inaccuracies. Other algorithms with better asymptotic properties (given enough time and memory, they will find an optimal action) have been proposed in the formalism of Markov decision processes [12, 19, 22].

This paper presents a new algorithm for combining Monte-Carlo evaluation with tree search. Its basic structure is described in Section 2. Its selectivity and backup operators are presented in the following sections. Then, game results are discussed. The conclusion summarizes the contributions of this research, and gives directions for future developments.

2 Algorithm Structure

The structure of our algorithm consists in iteratively running random simulations from the root position. This produces a tree made of several random games. This tree is stored in memory. At each node of the tree, the number of random games that passed through this node is counted, as well as the sum of the values of these games, and the sum of the squares of the values. In Crazy Stone, the value of a simulation is the score of the game.

Our approach is similar to the algorithm of Chang, Fu and Marcus [12], and provides some advantages over Bouzy's method [6]. First, this algorithm is anytime: each simulation brings additional information that is immediately backed up to the root, which is convenient for time management (Bouzy's algorithm only provides information at each deepening iteration). Also, this framework allows algorithms with proved convergence to the optimal move, because selectivity can be controlled at the level of individual simulations, and does not require that complete branches of the tree be cut off.

In practice, not all the nodes are stored. Storing the whole tree would waste too much time and memory. Only nodes close to the root are memorized. This is done by applying the following rules:

- Start with only one node at the root.
- Whenever a random game goes through a node that has been already visited once, create a new node at the next move, if it does not already exist.

As the number of games grows, the probability distribution for selecting a move at random is altered. In nodes that have been visited less than the number

of points of the goban (this threshold has been empirically determined as a good compromise), moves are selected at random according to heuristics described in Appendix A. Beyond this number of visits, the node is called an *internal* node, and moves that have a higher value tend to be selected more often, as described in Section 3. This way, the search tree is grown in a best-first manner.

3 Selectivity

In order not to lose time exploring useless parts of the search tree, it is important to carefully allocate simulations at every node. Moves that look best should be searched deeper, and bad moves should be searched less.

3.1 Background

A large variety of selectivity algorithms have been already proposed in the framework of Monte-Carlo evaluation. Most of them rely on the central limit theorem, that states that the mean of N independent realizations of a random variable with mean μ and variance σ^2 approaches a normal distribution with mean μ and variance σ^2/N. When trying to compare the expected values of many random variables, this theorem allows to compute a probability that the expected value of one variable is larger than the expected value of another variable.

Bouzy [9, 7] used this principle to propose progressive pruning. Progressive pruning cuts off moves whose probability of being best according to the distribution of the central-limit theorem falls below some threshold. Moves that are cut off are never searched again. This method provides a very significant acceleration.

Progressive pruning can save a lot of simulations, but it is very dangerous in the framework of tree search. When doing tree search, the central limit theorem does not apply, because the outcomes of random simulations are not identically distributed: as the search tree grows, move probabilities are altered. For instance, the random simulations for a move may look bad at first, but if it turns out that this move can be followed up by a killer move, its evaluation may increase when it is searched deeper.

In order to avoid the dangers of completely pruning a move, it is possible to design schemes for the allocation of simulations that reduce the probability of exploring a bad move, without ever letting this probability go to zero. Ideas for this kind of algorithm can be found in two fields of research: n-armed bandit problems, and discrete stochastic optimization. n-armed bandit techniques (Sutton and Barto's book [25] provides an introduction) are the basis for the Monte-Carlo tree search algorithm of Chang, Fu and Marcus [12]. Optimal schemes for the allocation of simulations in discrete stochastic optimization [13, 16, 3], could also be applied to Monte-Carlo tree search.

Although they provide interesting sources of inspiration, the theoretical frameworks of n-armed bandit problems and discrete stochastic optimization do not

fit Monte-Carlo tree search perfectly. First, and most importantly, n-armed bandit algorithms and stochastic optimization assume stationary distributions of evaluations, which is not the case when searching recursively. Also, in n-armed bandit problems, the objective is to allocate simulations in order to minimize the number of selections of non-optimal moves during simulations. This is not the objective of Monte-Carlo search, since it does not matter if bad moves are searched, as long a good move is finally selected. The field of discrete stochastic optimization is more interesting in this respect, since its objective is to optimize the final decision, either by maximizing the probability of selecting the best move [13], or by maximizing the expected value of the final choice [16]. This should be the objective at the root of the tree, but not in internal nodes, where the true objective in Monte-Carlo search is to estimate the value of the node as accurately as possible. For instance, with Chen's formula [13], in case the choice is between two moves, and simulations of these two moves have the same variance, then the optimal allocation consists in exploring both moves equally, regardless of their estimated values. This does indeed optimize the probability of selecting the best move, but is not at all what we wish to do inside a search tree: the best move should be searched more than the other, since it will influence the backed-up value more.

3.2 Crazy Stone's algorithm

The basic principle of Crazy Stone's selectivity algorithm is to allocate simulations to each move according to its probability of being better than the current best move. This scheme seems to be sound when the objective is to obtain an accurate backed-up value, since the probability of being best corresponds to the probability that this simulation would have an influence on the final backed up value if the algorithm had enough time to converge.

Assuming each move has an estimated value of μ_i with a variance of σ_i^2, and moves are ordered so that $\mu_0 > \mu_1 > \ldots > \mu_N$, each move is selected with a probability proportional to

$$u_i = \exp\Big(-2.4 \frac{\mu_0 - \mu_i}{\sqrt{2(\sigma_0^2 + \sigma_i^2)}} \Big) + \epsilon_i$$

This formula is an approximation of what would be obtained assuming Gaussian distributions (the 2.4 constant was chosen to match the normal distribution function). This formula is very similar to the Boltzmann distributions that are often used in n-armed bandits problems. ϵ_i is a constant that ensures that the urgency of a move never goes to zero, and is defined by

$$\epsilon_i = \frac{0.1 + 2^{-i} + a_i}{N} \ ,$$

where a_i is 1 when move i is an atari, and 0 otherwise. This formula for ϵ_i was determined empirically by trial and error from the analysis of tactical mistakes of Crazy Stone. It is important to increase the urgency of atari moves, because

they are likely to force an answer by the opponent, and may be underestimated because their true value requires another follow-up move.

For each move i, μ_i is the opposite of the value μ of the successor node, and σ_i^2 is its variance σ^2. For internal nodes of the search tree, μ and σ^2 are computed with the backup method described in the next section. For external nodes, that is nodes that have been visited less than the threshold defined in Section 2, μ and σ^2 are computed as $\mu = \Sigma/S$, and

$$\sigma^2 = \frac{\Sigma_2 - S\mu^2 + 4P^2}{S+1} \ ,$$

where P is the number of points of the board, Σ_2 is the sum of squared values of this node, Σ is the sum of values, and S is the number of simulations. The formula for σ^2 does as if a virtual game with a very high variance had been played. This high prior variance is necessary to make sure that nodes that have been rarely explored are considered very uncertain.

4 Backup Method

The most straightforward method to backup node values and uncertainties consists in applying the formula of external nodes to internal nodes as well. As the number of simulations grows, the frequency of the best move will dominate the others, so the mean value of this node will converge to the maximum value of all its moves, and the whole tree will become a negamax tree. This is the principle of the algorithm of Chang, Fu and Marcus [12].

This approach is simple but very inefficient. If we consider N independent random variables, then the expected maximum of these variables is not equal, in general, to the sum of the expected values weighted by the probabilities of each variable to be the best. This weighted sum underestimates the best move.

Backing up the maximum ($\max_i \mu_i$) is not a good method either. When the number of moves is high, and the number of simulations is low, move estimates are noisy. So, instead of being really the best move, it is likely that the move with the best value is simply the most lucky move. Backing up the maximum evaluation overestimates the best move, and generates a lot of instability in the search.

Other candidates for a backup method would be algorithms that operate on probability distributions [21, 4]. The weakness of these methods is that they have to assume some degree of independence between probability distributions. This assumption of independence is wrong in the case of Monte-Carlo evaluation because, as explained in the previous paragraph, the move with the highest value is more likely to be overestimated than other moves. Also, a refutation of a move is likely to also refute other moves of a node.

Since formal methods seem difficult to apply, the backup operator of Crazy Stone was determined empirically, by an algorithm similar to the temporal difference method [24]. In the beginning, the backup method for internal nodes was the external-node method. 1,500 positions were sampled at random from

self-play games. For each of these 1,500 positions, the tree search was run for 2^{19} simulations. The estimated value of the position was recorded every 2^n simulations, along with useful features to compute the backed-up value. Backup formulas were tuned so that the estimated value after 2^n simulations matches the estimated value after 2^{n+1} simulations. This process was iterated a few times during the development of Crazy Stone.

4.1 Value Backup

	Mean		Max		Robust Max		Mix	
Simulations	$\sqrt{\langle\delta^2\rangle}$	$\langle\delta\rangle$	$\sqrt{\langle\delta^2\rangle}$	$\langle\delta\rangle$	$\sqrt{\langle\delta^2\rangle}$	$\langle\delta\rangle$	$\sqrt{\langle\delta^2\rangle}$	$\langle\delta\rangle$
128	6.44	-3.32	41.70	37.00	39.60	35.30	5.29	-1.43
256	7.17	-4.78	25.00	22.00	23.60	20.90	4.72	-1.89
512	7.56	-5.84	14.90	12.70	13.90	11.90	4.08	-1.70
1,024	6.26	-4.86	9.48	7.91	8.82	7.41	3.06	0.13
2,048	4.38	-3.15	6.72	5.37	6.11	4.91	2.63	0.77
4,096	2.84	-1.55	4.48	3.33	3.94	2.91	2.05	0.69
8,192	2.23	-0.62	2.78	1.47	2.42	1.07	1.85	0.32
16,384	2.34	-0.57	2.45	0.01	2.40	-0.30	2.10	-0.19
32,768	2.15	-0.52	2.19	0.10	2.26	-0.12	1.93	-0.02
65,536	2.03	-0.50	1.83	0.23	1.88	0.01	1.70	0.01
131,072	2.07	-0.54	1.80	0.25	1.94	0.02	1.80	-0.02
262,144	1.85	-0.58	1.49	0.25	1.51	0.07	1.39	-0.02

Table 1. Backup experiments

Numerical values of the last iteration are provided in Table 1. This table contains the error measures for different value-backup methods. $\langle\delta^2\rangle$ is the mean squared error and $\langle\delta\rangle$ is the mean error. The error δ is measured as the difference between the value obtained by the value-backup operator on the data available after S simulations, with the "true" value obtained after $2S$ simulations. The "true" value is the value obtained by searching with the "Mix" operator, described in Figure 1.

These data clearly demonstrate what was suggested intuitively in the beginning of this section: the mean operator (Σ/S) under-estimates the node value, whereas the max operator over-estimates it. Also, the mean operator is more accurate when the number of simulations is low, and the max operator is more accurate when the number of simulations is high.

The robust max operator consists in returning the value of the move that has the maximum number of games. Most of the time, it will be the move with the best value. In case it is not the move with the best value, it is wiser not to back up the value of a move that has been searched less. A similar idea had

```
float MeanWeight = 2 * WIDTH * HEIGHT;
if (Simulations > 16 * WIDTH * HEIGHT)
 MeanWeight *= float(Simulations) / (16 * WIDTH * HEIGHT);

float Value = MeanValue;
if (tGames[1] && tGames[0])
{
 float tAveragedValue[2];
 for (int i = 2; --i >= 0;)
  tAveragedValue[i] =
   (tGames[i] * tValue[i] + MeanWeight * Value) / (tGames[i] + MeanWeight);

 if (tGames[0] < tGames[1])
 {
  if (tValue[1] > Value)
   Value = tAveragedValue[1];
  else if (tValue[0] < Value)
   Value = tAveragedValue[0];
 }
 else
  Value = tAveragedValue[0];
}
else
 Value = tValue[0].
return Value;
```

Fig. 1. Value-backup algorithm. The size of the goban is given by "WIDTH" and "HEIGHT". "Simulations" is the number of random games that were run from this node, and "MeanValue" the mean value of these simulations. Move number 0 is the best move, move number 1 is the second best move or the move with the highest number of games, if it is different from the two best moves. tValue[i] are the the backed-up values of the moves and tGames[i] their numbers of simulations.

been used by Alrefaei and Andradottir [3] in their stochastic simulated annealing algorithm.

Figure 1 describes the "Mix" operator, that was found to provide the best value back up. It is a linear combination between the robust max operator and the mean operator, with some refinements to handle situations where the mean is superior to the max (this may actually happen, because of the non-stationarity of evaluations).

4.2 Uncertainty Backup

Uncertainty backup in Crazy Stone is also based on the data presented in the previous section. These data were used to compute the mean squared difference between the backed-up value after S simulations and the backed-up value after $2S$ simulation. To approximate the shape of this squared difference, the backed-up variance was chosen to be $\sigma^2/\min(500, S)$ instead of σ^2/S. This is an extremely primitive and inaccurate way to backup uncertainty. It seems possible to find better methods.

5 Game Results

As indicated in the abstract, Crazy Stone won the 10th KGS computer-Go tournament, ahead of 8 participants, including GNU Go, Neuro Go, Viking 5, and Aya [28]. This is a spectacular result, but this was only a 6-round tournament, and luck was probably one of the main factor in this victory.

In order to test the strength of Crazy Stone more accurately, 100-game matches were run against GNU Go, and the latest version of Indigo (I thank Bruno Bouzy for providing it to me), performing a search at depth 3, with a width of 7. Games were run on an AMD Athlon 3400+ PC running Linux. Results are summarized in Table 2.

Player	Opponent	Winning Rate	Komi
CrazyStone (5 min / game)	Indigo 2005 (8 min / game)	61% (±4.9)	6.5
Indigo 2005 (8 min / game)	GNU Go 3.6 (level 10)	28% (±4.4)	6.5
CrazyStone (4 min / Game)	GNU Go 3.6 (level 10)	25% (±4.3)	7.5
CrazyStone (8 min / Game)	GNU Go 3.6 (level 10)	32% (±4.7)	7.5
CrazyStone (16 min / Game)	GNU Go 3.6 (level 10)	36% (±4.8)	7.5

Table 2. Match results, with 95% confidence intervals

These results show that Crazy Stone clearly outperforms Indigo. This is a good indication that the tree search algorithm presented in this paper is more efficient than Bouzy's algorithm. Nevertheless, it is difficult to draw definitive conclusions from this match, since Indigo's algorithm differ from Crazy Stone's

in many points. First, it relies on a knowledge-based move pre-selector, that Crazy Stone does not have. Also, random simulations are different. Crazy Stone's simulations probably have better handling of the urgency of captures. Indigo's simulations use patterns, while Crazy Stone is based on an uniform distribution. All in all, this victory is still a rather convincing indication of the power of the algorithm presented in this paper.

Results against GNU Go indicate that Crazy Stone is still weaker, especially at equal time control (GNU Go used about 22 second per game, on average). The progression of results with longer time control indicates that the strength of Crazy Stone scales well with the amount of CPU time it is given.

Beyond the raw numbers, it is interesting to take a look at the games, and the playing styles of the different players[1]. Most of the losses of Crazy Stone against GNU Go are due to tactics that are too deep, such as ladders, long semeais, and monkey jumps, that GNU Go has no difficulty to see. The wins of Crazy Stone over GNU Go are based on a better global understanding of the position. Because they are based on the same principles, the styles of Crazy Stone and Indigo are very similar. It seems that the excessive pruning of Indigo cause it to play tactical errors that Crazy Stone knows how to exploit.

6 Conclusion

In this paper was presented a new algorithm for Monte-Carlo tree search. It is an improvement over previous algorithms, mainly thanks to a new efficient backup method. It was implemented in a computer-Go program that performed very well in tournaments, and won a 100-game match convincingly against a state-of-the-art Monte-Carlo Go-playing program. Directions for future research include

- improving the selectivity algorithm and uncertainty-backup operator. In particular, it might be a good idea to use stochastic optimization algorithms at the root of the search tree.
- trying to overcome tactical weaknesses by incorporating game-specific knowledge into random simulations.
- scaling the approach to larger boards. For 19x19, an approach based on a global tree search does not seem reasonable. Generalizing the tree search with high-level tactical objectives such as Cazenave and Helmstetter's algorithm [11] might be an interesting solution.

Acknowledgements

I thank Bruno Bouzy and Guillaume Chaslot, for introducing me to Monte-Carlo Go. A lot of the inspiration for the research presented in this paper came from our discussions. I also thank the readers of this paper for their feedback that helped to improve this paper.

[1] Games of the matches are available at `http://remi.coulom.free.fr/CG2006/`

References

1. Bruce Abramson. Expected-outcome: A general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):182–193, February 1990.
2. L. Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Universiteit Maastricht, 1994.
3. Mahmoud H. Alrefaei and Sigrún Andradóttir. A simulated annealing algorithm with constant temperature for discrete stochastic optimization. *Management Science*, 45(5):748–764, May 1999.
4. Eric B. Baum and Warren D. Smith. A bayesian approach to relevance in game playing. *Artificial Intelligence*, 97(1–2):195–242, 1997.
5. Darse Billings, Denis Papp, Lourdes Peña, Jonathan Schaeffer, and Duane Szafron. Using selective-sampling simulations in poker. In *Proceedings of the AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*, 1999.
6. Bruno Bouzy. Associating shallow and selective global tree search with Monte Carlo for 9×9 Go. In H. J. van den Herik, Y. Björnsson, and N. S. Netanyahu, editors, *Fourth International Conference on Computers and Games*, Ramat-Gan, Israel, 2004.
7. Bruno Bouzy. Move pruning techniques for Monte-Carlo Go. In *Advances in Computer Games 11*, Taipei, Taiwan, 2005.
8. Bruno Bouzy and Tristan Cazenave. Computer Go: an AI-oriented survey. *Artificial Intelligence*, 132:39–103, 2001.
9. Bruno Bouzy and Bernard Helmstetter. Monte Carlo Go developments. In H. J. van den Herik, H. Iida, and E. A. Heinz, editors, *Proceedings of the 10th Advances in Computer Games Conference*, Graz, 2003.
10. Bernd Brügmann. Monte Carlo Go, 1993. Unpublished technical report.
11. Tristan Cazenave and Bernard Helmstetter. Combining tactical search and Monte-Carlo in the game of go. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2005.
12. Hyeong Soo Chang, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus. An adaptive sampling algorithm for solving Markov decision processes. *Operations Research*, 53(1):126–139, Jan.–Feb. 2005.
13. Chun-Hung Chen, Jianwu Lin, Enver Yücesan, and Stephen E. Chick. Simulation budget allocation for further enhancing the efficiency of ordinal optimization. *Journal of Discrete Event Dynamic Systems: Theory and Applications*, 10(3):251–270, July 2000.
14. Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte-Carlo planning in RTS games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2005.
15. Markus Enzenberger. Evaluation in Go by a neural network using soft segmentation. In *Proceedings of the 10th Advances in Computer Games Conference*, Graz, 2003.
16. Andreas Futschik and Georg Ch. Pflug. Optimal allocation of simulation experiments in discrete stochastic optimization and approximative algorithms. *European Journal of Operational Research*, 101:245–260, 1997.
17. Matthew L. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 584–593, Sweden, 1999.

18. Hugues Juillé. *Methods for Statistical Inference: Extending the Evolutionary Computation Paradigm*. PhD thesis, Brandeis University, Department of Computer Science, May 1999.
19. Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *Proceedings of the Sixteenth Internation Joint Conference on Artificial Intelligence*, pages 1324–1331. Morgan Kaufmann, 1999.
20. Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.
21. Andrew J. Palay. *Searching with Probabilities*. Pitman, 1984.
22. Laurent Péret and Frédérick Garcia. On-line search for solving large Markov decision processes. In *Proceedings of the 16th European Conference on Artificial Intelligence*, Valence, Spain, August 2004.
23. Brian Sheppard. Efficient control of selective simulations. *ICGA Journal*, 27(2):67–79, June 2004.
24. Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
25. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.
26. Gerald Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134:181–199, 2002.
27. John Tromp and Gunnar Farnebäck. Combinatorics of Go. In P. Ciancarini and H. J. van den Herik, editors, *Proceedings of the Fifth International Conference on Computer and Games*, Turin, Italy, 2006.
28. Nick Wedd. Computer Go tournaments on KGS. `http://www.weddslist.com/kgs/`, 2005.

A Random Simulations in Crazy Stone

The most basic method to perform random simulations in computer-Go consists in selecting legal moves uniformly at random, with the exception of eye-filling moves that are forbidden. The choice of a more clever probability distribution can improve the quality of the Monte-Carlo estimation. This section describes domain-specific heuristics used in Crazy Stone.

A.1 Urgencies

At each point of the goban, an urgency is computed for each player. The urgency of the black player on a particular point is computed as follows:

- If playing at this point is illegal, or this point is completely surrounded by black stones that are not in atari, then the urgency is set to zero, and processing of this urgency is stopped. This rule will prevent some needed connection moves, but distinguishing false eyes from true eyes was found to be too difficult to be done fast enough during simulations.
- Otherwise, the urgency is set to 1.

- If this point is the only liberty of a black string[2] of size S, then $1,000 \times S$ is added to the urgency, unless it is possible to determine that this point is a hopeless extension. A point is considered a hopeless extension when
  - there is at most one contiguous empty intersection, and
  - there is no contiguous white string in atari, and
  - there is no contiguous black string not in atari.
- If this point is the only liberty of a white string of size S, and it is not considered a hopeless extension for white, then $10,000 \times S$ is added to the urgency. Also, if the white string in question is contiguous to a black string in atari, then $100,000 \times S$ is added to the urgency (regardless of whether this point is considered a hopeless extension for white).

The numerical values for urgencies are arbitrary. No effort was made to try other values and measure their effects. They could probably be improved.

A.2 Useless Moves

Once urgencies have been computed, a move is selected at random with a probability proportional to its urgency. This move may be undone and another selected instead in the following situations:

- If the move is surrounded by stones of the same color except for one empty contiguous point, and these stones are part of the same string, and the empty contiguous point is also contiguous to this string, then play in the contiguous point instead. Playing in the contiguous point is better since it creates an eye. With this heuristic, a player will always play in the middle of a 3-point eye (I thank Peter McKenzie for suggesting this idea to me).
- If a move is surrounded by stones of the opponent except for one empty contiguous point, and this move does not change the atari status of any opponent string, then play in the empty contiguous point instead.
- If a move creates a string in atari of more than one stone then
  - if this move had an urgency that is more than or equal to 1,000, then this move is undone, its urgency is reset to 1, and a new move is selected at random (it may be the same move);
  - if this string had a contiguous string in atari before the move, then capture the contiguous string in atari instead (doing this is very important, since capturing the contiguous string may not have a high urgency);
  - otherwise, if the string had two liberties before the move, play in the other liberty instead.

A.3 Performance

On an AMD Athlon 3400+, compiled with 64-bit gcc 4.0.3, Crazy Stone simulates about 17,000 random games per second from the empty 9×9 goban.

[2] a string is a maximal set of orthogonally-connected stones of the same color