Virtual Global Search: Application to 9x9 Go

Tristan Cazenave

LIASD
Dept. Informatique
Université Paris 8, 93526, Saint-Denis, France
cazenave@ai.univ-paris8.fr

Abstract. Monte-Carlo simulations can be used as an evaluation function for Alpha-Beta in games. If w is the width of the search tree, d its depth, and g the number of simulations at each leaf, the total number of simulations is at least $g \times (2 \times w^{\frac{d}{2}})$. In games where moves permute, we propose to replace this algorithm by another algorithm that only needs $g \times 2^d$ simulations for a similar number of games per leaf. The algorithm can also be applied to games where moves often but not always permute, such as Go. We detail its application to 9x9 Go.

1 Introduction

Monte-Carlo methods can be used to evaluate moves and states in perfect information games. They can be combined with Alpha-Beta search, using Monte-Carlo simulations at the leaves of the search tree to evaluate positions [1]. In the remaining of the paper, w is the width, and d the depth of the global search, and a global move is a move that can be tried in a global search. The time complexity of the combination is at least proportional to $g \times (2 \times w^{\frac{d}{2}})$, and the space complexity is linear in d.

In some games such as Hex, moves always permute. The final position of a game is the same if two moves are switched in the game. It is not true in other games such as Go. However, it is often true in Go, and this property can be used to combine efficiently Alpha-Beta search with Monte-Carlo evaluation at the leaves. The algorithm we propose, named virtual global search, has a time complexity proportional to $g \times 2^d$, and a space complexity proportional to w^d. In games where moves always permute, it gives results close to the usual combination. We have found that it is also interesting in games such as Go, where moves often permute.

Section two describes related work. Section three exposes standard global search. Section four presents virtual global search. Section five estimates the complexities of the two search algorithms. Section six details experimental results. Section seven outlines future work.

2 Related work

Related works are on Monte-Carlo in games, Monte-Carlo Go and selective global search.

2.1 Monte-Carlo and games

Monte-Carlo methods have been used in many games. In Bridge, GIB uses Monte-Carlo to compute statistics on solved double dummy deals [11]. In Poker, Poki uses selective sampling and simulation-based betting strategy for the game of Texas Hold'em [2]. In Scrabble, Maven controls selective simulations [14]. Monte-Carlo has also been applied to Phantom Go, randomly putting opponent stones before each random game [6], and to probabilistic combinatorial games [15].

2.2 Standard Monte-Carlo Go

The first Monte-Carlo Go program is Gobble [5]. It uses simulated annealing on a list of moves. The list is sorted by the mean score of the games each move has been played in first. Moves in the list are switched with their neighbor with a probability dependent on the temperature. The moves are tried in the games in the order of the list. At the end, the temperature is set to zero for a small number of games. After all games have been played, the value of a move is the average score of the games it has been played in first. Monte-Carlo Go has a good global sense but lacks of tactical knowledge. For example, it often plays useless Atari, or tries to save captured strings.

2.3 Monte-Carlo Go enhancements

An enhancement of Monte-Carlo Go is to combine it with Go knowledge. Indigo has been using Go knowledge to select a small number of moves that are later evaluated with the Monte-Carlo method [3]. Another use of Go knowledge is to bias the selection of moves during the random games using patterns and rules [3, 8]. A third enhancement is to compute statistics on unsettled tactical goals instead of only computing statistics on moves [7].

2.4 Global search in Go

The combination of global search with Monte-Carlo Go has been studied by B. Bouzy [4] for 9x9 Go. His algorithm associates progressive pruning to Alpha-Beta search to discriminate moves in a global search tree with Monte-Carlo evaluation at the leaves.

Crazy Stone [10] uses a back-up operator and biased move exploration in combination with Monte-Carlo evaluation at the leaves. It finished first of the 9x9 Go tournament in the 2006 Olympiad.

The analysis of decision errors during selective tree search has been studied by Ken Chen [9].

3 Standard global search

In this section we first present how global moves are selected, and then how they are used for standard global search.

3.1 Selection of moves

Global search in Go is highly selective due to the large number of possible moves. In order to select the global moves to consider in the search, we use a program that combines tactical search and Monte-Carlo [7]. This program gives a static evaluation to each move. In this paper, the program that gives static evaluations to move only uses connection search in association with Monte-Carlo simulations. The w moves that have the best static evaluation according to the program are selected as global moves.

The w selected global moves are sorted according to their static evaluation. An option is to select the global moves which have a static evaluation above a fixed percentage of the static evaluation of the best move.

It is important to notice that only the locations of the selected global moves are used for the search and not the moves themselves. The w locations are used in the global search to generate possible moves for both colors.

3.2 Standard global search

Standard global search is global search with Monte-Carlo evaluation at the leaves. Performing Monte-Carlo simulations at the leaves of the tree in order to evaluate them is the natural idea when combining Monte-Carlo sampling and game tree search [1].

4 Virtual global search

The selection of moves for virtual global search is the same as for standard global search. In this section, we first detail the permutation of moves in sequences of moves. Then we explain how sequences are evaluated at the end of the random games. Eventually, we explain how the global search tree is developed.

4.1 Permutation of moves

The main idea underlying the algorithm is that a move in a random game has more or less the same influence if it is played at the beginning, or in the course of a random game. Gobble [5] uses a similar idea in order to evaluate the moves and order them. In Gobble, a move is evaluated using all the games where it has been played first, not taking into account if it has been played at the beginning or at the end of a game.

We extend this idea to sequences of moves. We can consider that a sequence of moves has more or less the same value when the moves of the sequence are played in any order in a random game. This leads to count a random game for the sequence when all the moves of the sequence have been played on their intersection during the random game (the program only takes into account the first move played on an intersection, it does not take into account moves that are played on an intersection where a string has been captured before during the random game).

4.2 Random games with evaluation of the sequences at the end

If we consider that a sequence is valid when its moves have been played in any order during a random game, the program can memorize at the end of the game the score of the game and associate it to the sequence.

The length of the sequences considered is d, and that the same selection of w moves at the root node is used for all positions.

The program has a structure which records for every possible sequence of selected moves, the mean score of the games where the sequence has been played in any order. The size of the memory used to record mean scores is proportional to w^d as it is approximately the number of possible sequences.

A sequence of moves is associated to an index. Each global move is associated to an index in the sorted array of selected global moves. The program allocates b bits for representing the index of a move. If d is the maximum depth allowed, a sequence is coded with $b \times d$ bits. The move at depth i in the sequence is coded starting at the bit number $(i - 1) \times b$.

A structure is associated to each sequence. This structure records the number of games where the sequence has been played in any order, the cumulated scores of the games where it has been played. These two kinds of data are used at the end of the simulations to compute the mean score of the games where the sequence has been played in any order. The size of the array of structures is $2^{b \times d}$ entries, each entry has the size of two integers.

When a random game is finished, the program develops a search tree using the selected global moves. We call this search tree a virtual search tree since it does not really play moves during the expansion of the tree, but only updates the index of the sequence, for each global move chosen (it also forbids to choose a move that is already played in the sequence).

In the virtual search tree, a global move of a given color is played only if it has the same color as the first move on the intersection, in the random game. On average, the move in the random game is of the same color half of the time. Out of the n possible global moves, only $\frac{w}{2}$ have the required color in the random game on average. At the leaves of the search tree, the program updates the score of the sequence that has been played until the leaf: it increments the number of times the sequence has been played, and it adds the score of the game to the cumulated score. Given that the program tries $\frac{w}{2}$ moves at each node, and that it searches to depth d, the number of leaves of the search tree is roughly $(\frac{w}{2})^d$ (a little less in fact since moves cannot be played twice on the same intersection, and that sometimes less than $\frac{w}{2}$ moves have been played with a color).

4.3 Search after all the random games are completed

Virtual global search is global search with virtual pre-computed evaluation at the leaves. It is performed after all the random games have been played, and after all possible sequences have been associated to a mean score.

An alpha-beta is used to develop the virtual global search tree. Moves are not played, but instead the index of the current sequence is updated at each move. The evaluation at the leaves is pre-computed, it consists in returning the mean of the random games where the current sequence has been detected. It only costs an access to the array of structures at the index of the current sequence. Developing the search tree of virtual global search takes little time in comparison to the time used for the random games.

5 Estimated complexities

In this section, we give estimations of the complexity of standard global search, and of virtual global search.

5.1 Complexity of standard global search

Let g be the number of random games played at each leaf of the standard global search tree in order to evaluate the leaf. The alpha-beta algorithm, with an optimal move ordering, has roughly $2 \times w^{\frac{d}{2}}$ leaves [13]. Therefore the total number of random games played is $g \times (2 \times w^{\frac{d}{2}})$. The space complexity of the alpha-beta is linear in the depth of the search.

5.2 Complexity of virtual global search

There are a little less than w^d possible global sequences. At the end of each random game, approximately $(\frac{w}{2})^d$ sequences are updated. Let g_1 be the number of random games necessary to have a mean computed with g random games for each sequence. We have:

$$g_1 = \frac{g \times w^d}{(\frac{w}{2})^d} = g \times 2^d \quad (1)$$

Therefore the program has to play $g \times 2^d$ random games, before the virtual global search, in order to have an equivalent of the standard global search with g random games at each leaf.

The space complexity of the virtual global search is w^d, as there are w^d possible sequences and the program has to update statistics on each detected sequence at the end of each random game.

5.3 Comparison of sibling leaves

Two leaves that have the same parent share half of their random games. To prove it, let's consider all the random games where the move that links the parent and one of the leaves has been played. Half of these random games also contain the move that links the parent and the other leaf. Therefore, in order to compare two leaves based on g different games as in standard global search, $g \times 2^{d+1}$ random games are necessary instead of $g \times 2^d$ random games.

6 Experimental results

Experiments were performed on a Pentium 4 3.0 GHz with 1GB of RAM.

Table 1 gives the time used by different algorithms to generate the first move of a game. It is a good approximation of the mean time used per move during a game. From lines one and two, we can see that for a similar precision, virtual global search with a width of eight and at depth three takes 0.3 seconds when standard global search with a similar setting takes 15.8 seconds.

An interesting result is given in the last line, where virtual global search only takes 0.6 seconds for a full width depth three search, and an equivalent of one hundred random games at each leaf.

Lines two and four have the same time since the time used for the tree search is negligible compared to the time used for the random games, and that the number of random games needed is not related to the width of the tree.

Table 1. Comparison of times for the first move.

$algorithm$	w	d	$games$	$time$
$standard$	8	3	$g = 100$	15.8s
$virtual$	8	3	$g_1 = 800$	0.3s
$standard$	16	3	$g = 100$	118.2s
$virtual$	16	3	$g_1 = 800$	0.3s
$virtual$	81	3	$g_1 = 800$	0.6s

Table 2 compares different algorithms. Each line of the table resumes the result of one hundred games between two programs on 9x9 boards (fifty games with black, and fifty with white). The first column gives the name of the algorithm for the max player. The second column gives the maximum number of global moves allowed for max. The third column gives the maximum global depth for max. The fourth column gives the total number of random games played before the virtual global search. The fifth column gives the minimum percentage of the best move static evaluation required to select a global move: a move is selected if its static evaluation is greater than the static evaluation of the best move multiplied by the percentage. The sixth column gives the average time used for virtual global search (including the random games). The next columns give similar information for the min player. The last two columns give the average score of the one hundred games for max, and the number of games won by max out of the one hundred games.

For example the first line of table 2 shows that virtual global search, with width eight, depth three, two thousand random games, all moves allowed, takes half a second per move and loses against standard global search with similar settings. This experiment shows that with equivalent precisions on the evaluation (here the number of games per leaf for virtual global search is $\frac{2000}{2^3} = 250$, the same as for standard global search), the virtual global search takes twenty three times less time for an average loss of 3.3 points per game.

The next lines test different options for virtual global search against a fixed version of standard global search (one hundred games per leaf, width eight, depth three). A depth three virtual global search, with at most sixteen global moves that have a static evaluation which is at least half the best static evaluation, and eight thousand games takes less than two seconds per move and wins by almost 10 points against a standard global search that takes more than six seconds per move.

In these experiments the number of games used to select the moves to search is the same as the number of games per leaf. In the next experiments, we have decorrelated these two numbers.

Table 2. Comparison of algorithms.

max	w	d	g_1	$\%$	$time$	min	w	d	g	$time$	$result$	won
$virtual$	8	3	2,000	0%	0.5s	$standard$	8	3	250	11.5s	-3.3	42
$virtual$	8	3	8,000	0%	2.1s	$standard$	8	3	100	7.2s	5.6	66
$virtual$	8	3	8,000	50%	2.2s	$standard$	8	3	100	7.0s	7.2	75
$virtual$	16	3	8,000	50%	1.8s	$standard$	8	3	100	6.4s	9.6	70
$virtual$	8	5	32,000	0%	13.1s	$standard$	8	3	100	7.6s	10	73

Table 3 gives some results of one hundred games matches against gnugo 3.6. The first column is the algorithm used for the max player, the second column the maximum width of the search tree, the third column the depth, the fourth column is the number of games played before the search in order to select the moves to try, the fourth column is the number of games for each leaf of the search tree, then comes the minimum percentage of the best move used to select moves, the average time of the search per move, the mean result of the one hundred games against gnugo 3.6, the associated standard deviation, and the number of won games.

The best number of won games is thirty one for virtual global search with $g_1 = 80,000$ ($g = 10,000$ and $d = 3$). However, the best mean is -11.1 for standard search with $g = 1,000$ and $d = 3$, but it only wins twenty one games. The two results for depth one are close to a standard Monte-Carlo evaluation without global search. The results show that more accuracy (4,000 games instead of 1,000) may be more important in some cases than more depth when comparing lines two and three of the table.

A result of this table is that for the same number of games per leaf and for a width eight and depth three search, standard search is better than virtual search.

Table 4 gives some results of one hundred games matches with width sixteen global search against gnugo 3.6. In these experiments, the global search is given more importance since the number of locations investigated is sixteen and that locations that are not highly evaluated by the static evaluation are searched.

Table 3. Results against gnugo 3.6.

max	w	d	pre	g	%	$time$	$mean$	σ	won
$virtual$	8	3	100	100	80%	0.4s	-34.4	27.6	4
$virtual$	8	3	1,000	1,000	80%	3.7s	-26.6	27.7	10
$virtual$	8	1	1,000	4,000	80%	3.7s	-17.7	28.6	16
$virtual$	8	3	16,000	2,000	80%	4.7s	-16.1	23.1	17
$virtual$	8	3	1,000	10,000	80%	37.4s	-14.4	28.5	31
$standard$	8	3	100	100	80%	3.3s	-23.9	22.3	10
$standard$	8	1	1,000	4,000	80%	4.4s	-17.3	24.7	16
$standard$	8	3	1,000	1,000	80%	23.6s	-11.1	23.9	21

Table 4. Results of width sixteen against gnugo 3.6.

max	w	d	pre	g	%	$time$	$mean$	σ	won
$virtual$	16	3	1,000	2,000	0%	7.6s	-12.2	25.4	29
$virtual$	16	3	1,000	10,000	0%	23.7s	-16.1	26.1	23
$virtual$	16	1	1,000	8,000	0%	4.8s	-23.0	32.4	18
$standard$	16	3	1,000	2,000	0%	515.7s	-15.0	23.9	19

The results of table 4 show that virtual global search outperforms standard global search for width sixteen and depth three. It plays moves in 7.6 seconds instead of 515.7 seconds, scores -12.2 points instead of -15.0, and wins 29 games instead of 19.

7 Future work

In Go, permutation of moves do not always lead to the same position. For example in figure 1, White 1 followed by Black 2 does not give the same position as Black 2 followed by White 1.

In other games, such as Hex for example, moves always permute. Using virtual global search in such games is appropriate.

Concerning Go, an improvement of our current program would be to differentiate between permuting and non permuting moves. For example two moves at the same location and of the same color can be considered different if one captures a string, and the other not. This would enable to detect problems such as the non permutation of figure 1. It is linked with some recent work on single agent search [12]. Another possibility is to match the order of the moves in the random game with the order of the moves in the sequence to evaluate.

The selection of moves can be improved, currently the moves are chosen according to their static evaluation at the root, not taking into account moves that are answers to other moves for example. Improving the evaluation at the leaf can also be interesting. A

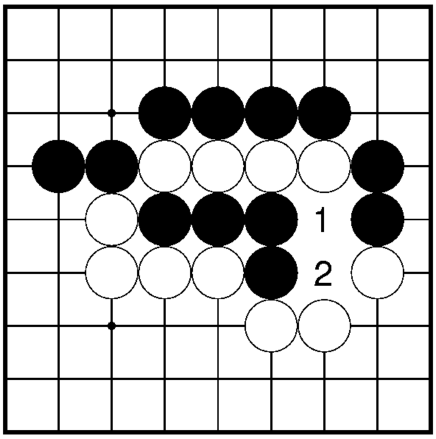

Fig. 1. The order of moves can be important

possibility is to add Go knowledge to play better moves during the random games such as in Indigo [3, 8].

Another idea is to combine global search with tactical goals, playing a global tree of goals instead of a global tree of moves, and evaluating a sequence of goals instead of a sequence of moves [7].

It could also be of interest to combine virtual global search with progressive pruning [4], looking at the global search tree while playing the random games, and stopping as soon as one move is clearly superior to the others.

The combination of virtual search with new back-up operators and biased move exploration such as in Crazy Stone [10] also looks promising.

8 Conclusion

We have presented an algorithm that combines Alpha-Beta search with Monte-Carlo simulations. It takes $g \times 2^d$ simulations and w^d memory instead of more than $g \times (2 \times w^{\frac{d}{2}})$ simulations and linear memory in d for the usual combination of Alpha-Beta and Monte-Carlo simulations. In games where moves permute it gives similar results. In 9x9 Go, it also gives good results even if moves do not always permute.

References

1. B. Abramson. Expected-outcome: a general model of static evaluation. *IEEE Transactions on PAMI*, 12(2):182–193, 1990.
2. D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):210–240, 2002.

3. B. Bouzy. Associating domain-dependent knowledge and Monte Carlo approaches within a go program. *Information Sciences*, 175(4):247–257, November 2005.
4. B. Bouzy. Associating shallow and selective global tree search with Monte Carlo for 9x9 go. In Nathan S. Netanyahu H. Jaap Herik, Yngvi Björnsson, editor, *Computers and Games: 4th International Conference, CG 2004*, Volume 3846 of LNCS, pages 67–80, Ramat-Gan, Israel, 2006. Springer-Verlag.
5. B. Bruegmann. Monte Carlo Go. ftp://ftp-igs.joyjoy.net/go/computer/mcgo.tex.z, 1993.
6. T. Cazenave. A Phantom Go program. In *Advances in Computer Games 11*, Taipei, Taiwan, 2005.
7. T. Cazenave and B. Helmstetter. Combining tactical search and Monte-Carlo in the game of go. In *CIG'05*, pages 171–175, Colchester, UK, 2005.
8. G. Chaslot. Apprentissage par renforcement dans une architecture de Go Monte Carlo. Mémoire de DEA, Ecole Centrale de Lille, Septembre 2005.
9. K. Chen. A study of decision error in selective game tree search. *Information Science*, 135(3-4):177–186, 2001.
10. R. Coulom. Efficient selectivity and back-up operators in monte-carlo tree search. In *Proceedings Computers and Games 2006*, 2006.
11. M. L. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *IJCAI-99*, pages 584–589, Stockholm, Sweden, 1999.
12. B. Helmstetter and T. Cazenave. Incremental transpositions. In Nathan S. Netanyahu H. Jaap Herik, Yngvi Björnsson, editor, *Computers and Games: 4th International Conference, CG 2004*, Volume 3846 of LNCS, pages 220–231, Ramat-Gan, Israel, 2006. Springer-Verlag.
13. D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
14. B. Sheppard. Efficient control of selective simulations. *ICGA Journal*, 27(2):67–80, June 2004.
15. L. Zhao and M. Müller. Solving probabilistic combinatorial games. In *Advances in Computer Games 11*, Taipei, Taiwan, 2005.