Virtual Global Search: Application to 9×9 Go

Tristan Cazenave

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

Abstract. In games, Monte-Carlo simulations can be used as an evaluation function for Alpha-Beta search. Assuming w is the width of the search tree, d its depth, and g the number of simulations at each leaf, then 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 a new algorithm, *Virtual Global Search*, that only needs $g \times 2^d$ simulations for a similar number of games per leaf. The algorithm is also applicable to games where moves often but not always permute, such as Go. We specify the application for 9×9 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 when two moves by the same color were switched during 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 efficiently to combine Alpha-Beta search with Monte-Carlo evaluation at the leaves. The algorithm we propose, named *Virtual Global Search*(VGS), has a time complexity proportional to $g \times 2^d$, and a space complexity proportional to w^d. In games where moves always permute, VGS gives results close to the normal best moves within the complexity frameworks mentioned above. We have found that VGS is also interesting in games such as Go, where moves often permute.

The course of the article is as follows. Section 2 describes related work. Section 3 exposes Standard Global Search (SGS). Section 4 presents Virtual Global Search. Section 5 estimates the complexities of the two search algorithms. Section 6 provides details on experimental results. Section 7 outlines future work. Section 8 completes with a conclusion.

H.J. van den Herik et al. (Eds.): CG 2006, LNCS 4630, pp. 62–71, 2007.
© Springer-Verlag Berlin Heidelberg 2007

2 Related Work

Below we briefly discuss related works. They are on Monte Carlo and games (2.1), Standard Monte-Carlo Go (2.2), Monte-Carlo Go enhancements (2.3), and Global search in Go (2.4).

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 [12]. In Poker, Poki uses selective sampling and a simulation-based betting strategy for the game of Texas Hold'em [2]. In Scrabble, Maven controls selective simulations [15]. Moreover, Monte Carlo has also been applied to Phantom Go by randomly putting opponent stones before each random game [6], and to probabilistic combinatorial games [16].

2.2 Standard Monte-Carlo Go

The first Go program based on Monte Carlo techniques is Gobble [5]. It uses simulated annealing on a list of moves. The list is sorted by the mean score of the games in which the move under investigation has been played 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 in which it has been played as a first move. Gobble[5]has a good global sense but lacks tactical knowledge. For example, it often plays useless atari, or tries to save captured strings.

2.3 Monte-Carlo Go Enhancements

An enhancement of Gobble [5] is to combine its Monte-Carlo techniques 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]. A second 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 Bouzy [4] for 9×9 Go. His algorithm associates progressive pruning with Alpha-Beta search to discriminate moves in a global-search tree with a Monte-Carlo evaluation at the leaves.

Crazy Stone [11] uses a back-up operator and biased move exploration in combination with a Monte-Carlo evaluation at the leaves. It finished first in the 9×9 Go tournament in the 2006 Computer Olympiad [10].

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

3 Standard Global Search

In this section we first present how global moves are selected (3.1), and then how they are used for Standard Global Search (3.2).

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 be considered in the search, we use a program that combines tactical search and Monte-Carlo search [7], called TSMC. It gives a static evaluation to each move. In this paper, TSMC only uses connection search in association with Monte-Carlo simulations. The w moves that have the best static evaluation according to TSMC are selected as global moves. They are sorted according to their static evaluation. An alternative is tresholding, i.e., to select the global moves that have a static evaluation above a fixed percentage of the static evaluation of the best move, so far.

It is important to note 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 (SGS) is a 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 a natural idea when combining Monte-Carlo sampling and game tree search [1].

4 Virtual Global Search

The selection of moves for Virtual Global Search (VGS) is the same as for Standard Global Search. The difference with SGS is in the second phase (the search). In this section, we first specify, how the permutation of moves is transformed into sequences of moves (4.1). Then we explain how sequences are evaluated at the end of the random games (4.2). Eventually, we explain how the global search tree is developed (4.3).

4.1 Permutation of Moves

The main idea underlying the algorithm is that a move in a random game has roughly the same influence when it is played (1) at the beginning, (2) in the course of a random game, or (3) at the end. Gobble [5] uses a similar idea when evaluating the moves independently of the depth where they occurred in the random games. In Gobble, a move is evaluated using all the games where it has been played as first move on an intersection, not taking into account whether it has been played at the beginning or at the end of a game.

We extend this idea to sequences of moves. Therefore, we assume that a sequence of moves has roughly the same value even when (in a random game)

the moves of the sequence are played in any order. This leads to count a random game by sequence (instead of by move), i.e., when all the moves of the sequence have been played on their intersection during the random game. (Please note that 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 earlier 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, TSMC may store 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 it is assumed that the same selection of w moves at the root node is used for all positions.

TSMC has a structure which records, for every possible sequence of selected moves, the mean score of the games. We note again that 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. Moreover, each global move is associated to an index in the sorted array of selected global moves. TSMC 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 $(i-1) \times b$.

To each sequence a structure is associated. This structure records (1) the number of games where the sequence has been played in any order, and (2) the cumulated scores of the games where the sequence 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 completed, TSMC 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 w possible global moves, only $\frac{w}{2}$ have the required color in the random game on average. At the leaves of the search tree, TSMC 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 TSMC 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 sometimes even 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 a global search with a 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 search is used to develop the Virtual Global Search tree. Moves are not played, actually, 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 SGS (5.1) and of VGS (5.2).Then we give a comparison of the sibling leaves (5.3).

5.1 Complexity of Standard Global Search

Let g be the number of random games played at each leaf of the SGS 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 [14]. Therefore, the total number of random games played is $g \times (2 \times w^{\frac{d}{2}})$. The space complexity of the Alpha-Beta search 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:

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

Therefore, TSMC has to play $g \times 2^d$ random games, for 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 VGS 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 this statement we 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 1 and 2, we can see that for a similar precision, VGS with width 8 and at depth 3 takes 0.3 seconds when SGS with a similar setting takes 15.8 seconds.

Table 1. Comparison of times for the first move

Algorithm	w	d	Games	Time
SGS	8	3	$g = 100$	15.8s
VGS	8	3	$g_1 = 800$	0.3s
SGS	16	3	$g = 100$	118.2s
VGS	16	3	$g_1 = 800$	0.3s
VGS	81	3	$g_1 = 800$	0.6s

An interesting result is given in the last line, where VGS only takes 0.6 seconds for a full width depth-3 search, and 800 games played in total for all sequences. This number is equivalent to 100 random games at each leaf according to the description given in Subsection 5.2.

Lines 2 and 4 have the same time since the time used for the tree search is negligible compared to the time used for the random games, and the number of random games needed is not related to the width of the tree.

Table 2 compares the two different algorithms. Each line of the table resumes the result of 100 games between the two programs on 9×9 boards (50 games with Black, and 50 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 for VGS. 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 adjusted by a given percentage. The sixth column gives the average time used for VGS (including the random games). The next columns give similar information for the min player. The last two columns give the average score of the 100 games for Max, and the number of games won by Max out of the 100 games.

For example, the first line of Table 2 shows that VGS, with width eight, depth three, two thousand random games, all moves allowed, takes half a second per move and loses against SGS with similar settings. This experiment shows that

Table 2. Comparison of algorithms

Max	w	d	g_1	%	Time	Min	w	d	g	Time	Result	Won
VGS	8	3	2,000	0%	0.5s	SGS	8	3	250	11.5s	−3.3	42
VGS	8	3	8,000	0%	2.1s	SGS	8	3	100	7.2s	5.6	66
VGS	8	3	8,000	50%	2.2s	SGS	8	3	100	7.0s	7.2	75
VGS	16	3	8,000	50%	1.8s	SGS	8	3	100	6.4s	9.6	70
VGS	8	5	32,000	0%	13.1s	SGS	8	3	100	7.6s	10	73

with equivalent precisions on the evaluation (here the number of games per leaf for VGS is $\frac{2000}{2^3} = 250$, the same as for SGS), the virtual global search takes 23 times less time for an average loss of 3.3 points per game.

The next lines test different options for VGS against a fixed version of SGS (100 games per leaf, width 8, depth 3). A depth-3 VGS, with at most 16 global moves that have a static evaluation which is at least half the best static evaluation, and 8,000 games takes less than 2 seconds per move and wins by almost 10 points against a SGS that takes more than 6 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 3 gives some results of 100-game 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 (Pre) is the number of games played before the search in order to select the moves to try, the fifth 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 (sixth column), the average time of the search per move (seventh column), the mean result of the 100 games against Gnugo 3.6 (eight column), the associated standard deviation (nineth column), and the number of won games (tenth column).

The best number of won games is 31 for VGS with $g_1 = 80,000$ ($g = 10,000$ and $d = 3$). However, the best mean is -11.1 for SGS with $g = 1,000$ and $d = 3$, but it only wins 21 games. The two results for depth 1 are close to a standard Monte-Carlo evaluation without global search. The results show that

Table 3. Results against Gnugo 3.6

Max	w	d	Pre	g	%	Time	Mean	σ	Won
VGS	8	3	100	100	80%	0.4s	−34.4	27.6	4
VGS	8	3	1,000	1,000	80%	3.7s	−26.6	27.7	10
VGS	8	1	1,000	4,000	80%	3.7s	−17.7	28.6	16
VGS	8	3	16,000	2,000	80%	4.7s	−16.1	23.1	17
VGS	8	3	1,000	10,000	80%	37.4s	−14.4	28.5	31
SGS	8	3	100	100	80%	3.3s	−23.9	22.3	10
SGS	8	1	1,000	4,000	80%	4.4s	−17.3	24.7	16
SGS	8	3	1,000	1,000	80%	23.6s	−11.1	23.9	21

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-8 and depth-3 search, standard search is better than virtual search.

Table 4 gives some results of 100 game matches with width-16 global search against GNUGO 3.6. In these experiments, the global search is given more importance since the number of locations investigated is 16 and those locations that are not highly evaluated by the static evaluation are searched.

Table 4. Results of width 16 against GNUGO 3.6

Max	w	d	Pre	g	%	Time	Mean	σ	Won
VGS	16	3	1,000	2,000	0%	7.6s	−12.2	25.4	29
VGS	16	3	1,000	10,000	0%	23.7s	−16.1	26.1	23
VGS	16	1	1,000	8,000	0%	4.8s	−23.0	32.4	18
SGS	16	3	1,000	2,000	0%	515.7s	−15.0	23.9	19

The results of Table 4 show that VGS outperforms SGS for width 16 and depth 3. 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 does not always lead to the same position. For example, in Fig. 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, moves always permute. Using VGS in such games is appropriate.

Concerning Go, we list five potential improvements below. A first improvement of our current program would be to differentiate between permuting and non-permuting moves. There are two implementation possibilities. First, two moves at the same location and of the same color may be considered different if one captures a string, and the other not. This would enable to detect problems such as the non permutation of Fig. 1. The idea is linked with some recent work on single agent search [13]. Second,the order of the moves in the random game should be matched with the order of the moves in the sequence to evaluate.

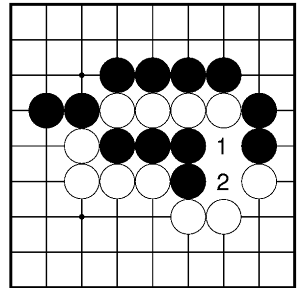

Fig. 1. The order of moves can be important

A second improvement is in selection of moves. Currently the moves are chosen according to their static evaluation at the root, not taking into account moves that are answers to other moves. Three straightforward implementations are: (1) investigate whether the move played is a "forced"' answer, (2) improve the evaluation at the leaf; (3) add Go knowledge such as in INDIGO [3,8].

A third improvement is to combine global search with tactical goals. This means searching in 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].

A fourth improvement is to combine Virtual Global Search with progressive pruning [4]. We keep investigating the global search tree while playing the random games, and stop as soon as one move is clearly superior to the others.

A fifth improvement may be to combine the virtual search with (1) new backup operators and (2) biased move exploration, such as in Crazy Stone [10].

8 Conclusion

We presented a new algorithm VGS that combined 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 (SGS). In games where moves permute VGS gives better results than SGS. In 9×9 Go, it also gives good results even through the moves do not always permute. Hence, we may conclude that VGS is a viable idea to elaborate upon even in games where the moves not always permute.

References

1. Abramson, B.: Expected-Outcome: a General Model of Static Evaluation. IEEE Transactions on PAMI 12(2), 182–193 (1990)
2. Billings, D., Davidson, A., Schaeffer, J., Szafron, D.: The Challenge of Poker. Artificial Intelligence 134(1-2), 210–240 (2002)
3. Bouzy, B.: Associating Domain-Dependent Knowledge and Monte Carlo Approaches Within a Go Program. Information Sciences 175(4), 247–257 (2005)
4. Bouzy, B.: Associating Shallow and Selective Global Tree Search with Monte Carlo for 9x9 Go. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 67–80. Springer, Heidelberg (2006)
5. Brügmann, B.: Monte Carlo Go (1993),`ftp://ftp-igs.joyjoy.net/go/computer/mcgo.tex.z`
6. Cazenave, T.: A Phantom Go Program. In: van den Herik, H.J., Hsu, S.-C., Hsu, T.-s., Donkers, H.H.L.M. (eds.) CG 2005. LNCS, vol. 4250, pp. 120–126. Springer, Heidelberg (2006)
7. Cazenave, T., Helmstetter, B.: Combining Tactical Search and Monte-Carlo in the Game of Go. In: CIG'05, pp. 171–175 (2005)
8. Chaslot, G.: Apprentissage par Renforcement dans une Architecture de Go Monte Carlo. Mémoire de DEA, Ecole Centrale de Lille (September 2005)
9. Chen, K.: A Study of Decision Error in Selective Game Tree Search. Information Science 135(3-4), 177–186 (2001)
10. Coulom, R., Chen, K.: Crazy Stone wins 9×9 Go Tournament. Note. ICGA Journal 29(2), 92 (2006)
11. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M. (eds.) 5th Computers and Games Conference (CG 2006). LNCS, vol. 4630, pp. 73–84. Springer, Heidelberg (2007)

12. Ginsberg, M.L.: GIB: Steps Toward an Expert-level Bridge-playing Program. In: IJCAI-99, pp. 584–589, Stockholm, Sweden (1999)
13. Helmstetter, B., Cazenave, T.: Incremental Transpositions. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 220–231. Springer, Heidelberg (2006)
14. Knuth, D.E., Moore, R.W.: An Analysis of Alpha-Beta Pruning. Artificial Intelligence 6(4), 293–326 (1975)
15. Sheppard, B.: Efficient Control of Selective Simulations. ICGA Journal 27(2), 67–80 (2004)
16. Zhao, L., Müller, M.: Solving Probabilistic Combinatorial Games. In: van den Herik, H.J., Hsu, S.-C., Hsu, T.-s., Donkers, H(J.) H.L.M. (eds.) CG 2005. LNCS, vol. 4250, pp. 225–238. Springer, Heidelberg (2006)