Multiplayer Go

Tristan Cazenave

LIASD, Dept. Informatique, Université Paris 8
2 rue de la liberté, 93526, Saint-Denis, France
cazenave@ai.univ-paris8.fr

Abstract. Multiplayer Go is Go played with more than two colors. Monte-Carlo tree search is an adequate algorithm to program the game of Go with two players. We address the application of Monte-Carlo tree search to multiplayer Go.

1 Introduction

The usual algorithm for multiplayer games is max^n [11, 13]. We propose alternative UCT (UCT stands for Upper Confidence bounds applied to Trees) based algorithms for multiplayer games, and we test them on multiplayer Go.

Two-player Go is already a complex game which is difficult to program [1], however recent progress in Monte-Carlo tree search makes it easier and gives better results than previous algorithms. Going from two-player Go to multiplayer Go makes the game even more complex. However the simplicity and the strength of Monte-Carlo tree search algorithms help manage multiplayer Go, and result in effective programs for this game.

The second section presents recent research on Monte-Carlo tree search. The third section explain some subtleties of multiplayer Go. The fourth section details the different algorithms that can be used to play multiplayer Go. The fifth section gives experimental results.

2 Monte-Carlo tree search

This section gives an overview of Monte-Carlo Go and its recent improvements. We start with Monte-Carlo tree search and the UCT algorithm then we briefly explain the RAVE algorithm.

2.1 Search and Monte-Carlo Go

Monte-Carlo Go started as a simulated annealing on the list of possible moves [2]. A more simple sampling method consisting in playing random playouts replaced it. Programs that develop a tree before the playouts have then replaced simple sampling [5, 8]. MOGO [6, 7] and CRAZY STONE [5] are good examples of the success of these methods. The UCT algorithm [8] is now the standard algorithm used for Monte-Carlo Go programs.

2.2 RAVE

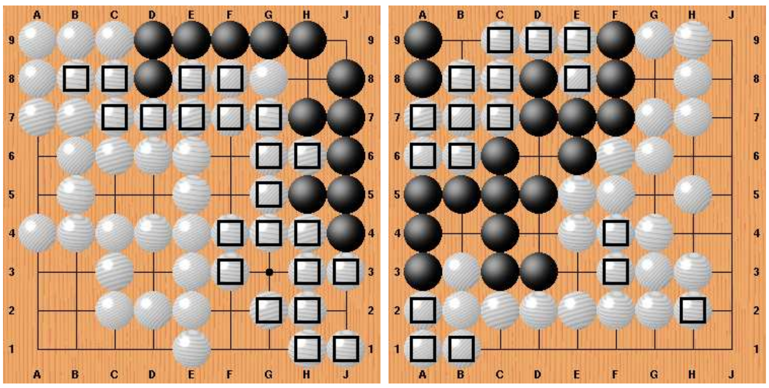
The RAVE (Rapid Action Value Estimation) algorithm [7] improves on UCT by using a rapid estimation of the value of moves when the number of playouts of a node is low. It uses a constant k and a parameter β that progressively switches from the rapid estimation heuristic to the normal UCT value. The parameter β is computed using the formula: $\beta = \sqrt{\frac{k}{3 \times games + k}}$. β is then used to bias the evaluation of a move in the tree with the formula: $val_i = \beta \times heuristic + (1.0 - \beta) \times UCT$. Experiments with MoGo have found that k = 1,000 gives good results.

The rapid estimation consists in computing statistics on all possible moves at every node of the UCT tree. After each playout, every move in the playout, which has the same color in the node and in the playout is updated with the result of the playout. For example, the last move of a playout is used to update the statistics of the corresponding move in the nodes of the UCT tree that start the playout. The value of the heuristic is the mean value of the move computed using these statistics. It corresponds to the mean result of the playouts where the move has been played.

3 Multiplayer Go

Multiplayer Go is sometimes played for fun at Go tournaments or at Go clubs.

In this section we explain some subtleties that are specific to multiplayer Go. We use three colors, Black, White and Red, who play in that order. In the figures, Red stones are the stones marked with squares.



black is alive   implicit coalition

Fig. 1. Some particularities of multiplayer Go

The first board of figure 1 shows a particular way to live in three-player Go. The black stones are alive, provided Black does not play at H8. White cannot play at H8, and if Red plays at H8 and captures the White stone, Black immediately recaptures the Red stone and gets two eyes.

The second board shows an implicit coalition. It is White to play. In normal two-player Go, the semeai between Black and Red in the upper left corner is won by Black. However, here it is profitable to White to kill Black as it will enable him to expand its territory, and it is of course also profitable to Red to kill Black as it will give him chances to live. So in this situation, even if Black is better in its semeai against Red, it will lose it because White will play at G8, and Red will play at D6 just after and capture Black.

If we define the goal of multiplayer Go to be the winner of the game, we define a queer game [10], which means that in some positions, no player can force a win. A problem with this definition of the rule is that a player that has lost the game can decide who is the winner. For example in three player Go, if a player P1 has a small group surrounded by another player P2, and if P2 wins if the small group is captured, but loses if the small group lives, P1 can either always pass and let P2 win, or make his group alive and let P3 win. When we implemented and tested three player Go with this rule, similar situations often happened and P2 was always passing because all the moves were losing moves, letting the weaker player win.

To avoid queer games, we decided to define the goal of multiplayer Go as to score as many points as possible, using Chinese rules. Every stone of a player counts as one point in the end of a game, so as every empty intersection completely surrounded by only one player. A games end when all the players have consecutively passed.

Another possibility would have been to use Japanese rule and to remove a string only if it is surrounded only by one color. Not removing a string surrounded by more than one color would be a very strong bias on the game since it would make weak surrounded strings alive, and the game would be much different than usual Go. Using Chinese rules is natural for Monte-Carlo based programs and Japanese rules would be even more complicated for multiplayer Go.

4 Multiplayer Algorithms

In this section, we describe different algorithms for multiplayer games. We start with max^n, the algorithm currently used in most multiplayer games. Then we overview UCT, Coalitions of players, Paranoid UCT, UCT with alliance and eventually variations on Confident UCT.

4.1 Max^n

The usual algorithm for multiplayer games is max^n [13] which is the extension of Minimax to multiplayer games. Some pruning techniques can be used with max^n [9, 12]. We did not test this algorithm on Go since building an evaluation function for Go is difficult and since the UCT framework is more simple and powerful for Go.

4.2 UCT

Multiplayer UCT behaves the same as UCT except that instead of having one result for a playout, the algorithm uses an array of scores with one result for each player which is the number of points of each player at the end of the playout. When descending the UCT tree, the player use the mean result of its scores, not taking into account the scores of the other players.

4.3 Coalitions of players

In multiplayer games, on the contrary of two-player games, players can collaborate and form coalitions. In order to model a coalition between players, we have used a very simple rule: it consists in not filling an empty intersection that is surrounded by players of the same coalition, provided that none of the surrounding strings is in atari. Of course, this rule only applies for players of the same coalition.

We have also defined the Allies scoring algorithm which is different from the Normal scoring algorithm. In Normal scoring, an empty intersection counts as a point for a player if it is completely surrounded by stones of the player's color. In Allies scoring, an empty intersection is counted as territory for a player if it is surrounded by stones that can be either of the color of the player or of the color of any of its allies.

The third scoring algorithm we have used is the Joint scoring algorithm, it consists in counting allied eyes as well as allied stones as points for players in a coalition.

4.4 Paranoid UCT

The paranoid algorithm consists in considering that all the other players form a coalition against the player. The UCT tree is traversed in the usual way, but the playouts use the coalition rules for the other players.

The algorithm has links with the paranoid search of Sturtevant and Korf [14] as it considers that all the other players are opponents.

In the design of the Paranoid UCT, we choose to model paranoia modifying the playouts. Another possibility is to model it directly in the UCT tree. In this case, the other players choose the move that minimize the player mean instead of choosing the move that maximize their own mean when descending the UCT tree. It could also be interesting to combine the two approaches.

4.5 UCT with alliance

The alliance algorithm models an explicit coalition of players. The playouts of the players in the alliance use the coalition rule. The playouts can be scored either using the Normal, the Allies or the Joint scoring algorithm.

4.6 Confident UCT

Confident UCT algorithms dynamically form coalitions depending on the situation on the board. There are different types of confident algorithms.

The simplest one is the Confident algorithm. At each move it develops as many UCT trees as there are players. Each UCT tree is developed assuming a coalition with a different player. This includes developing a UCT tree with a coalition with itself, which is equivalent to the normal UCT algorithm. Among all these UCT trees, it chooses the one that has the best mean result at the root, and plays the corresponding move. The idea of the confident algorithm is to assume that the player who is the most interesting to cooperate with, will cooperate. In case of the paranoid algorithm, it will never be the case. Even in the case of other confident algorithms, it is not always the case, since the other confident algorithm can choose to cooperate with another player, or not to cooperate.

To try to address the shortcomings of the Confident algorithm, we devised the Symmetric Confident algorithm. It does run the confident algorithm for each player, so for each player we have the mean result of all coalition with all players. We can then find for each player the best coalition. If the best coalition for the player to move is also the best coalition for the other player of the coalition, then the player to move chooses the move of the UCT tree of the coalition. If it is not the case, the player chooses the move of a coalition that is better than no coalition if the other player of the coalition has its best coalition with the player. In all other cases, it chooses the move of the UCT tree without coalition.

The last confident algorithm is the Same algorithm. On the contrary of the two previous algorithm, it assumes that it knows who are the other Same algorithms. It consists in choosing the best coalition among the other Same players, including itself (choosing itself is equivalent to no coalition).

5 Experimental Results

The experiments consist in playing the algorithms against each other. All our results use 200 9x9 games experiments.

5.1 UCT

The *Surrounded* algorithm always avoids playing on an intersection which has all its adjacent intersections of the same color as the player to move, it is a very simple rule use in the playouts. The *Eye* algorithm also verifies the status of the diagonals of the intersection in order to avoid playing on virtual eyes during playouts.

Table 1 gives the results of three players games. Each line gives the mean number of points of each algorithm playing 200 9x9 games against the other algorithms.

Given the first line that only contains Surrounded algorithms, and the lines that only contain Rave algorithm, we can see that the value of komi is low in three-player Go (approximately one point for White and two points for Red given the Rave results for 10,000 playouts). This is our first surprising result, we expected the third player to be more at a disadvantage than only two points.

The first three lines of table 1 compare algorithms that use surrounded intersections as the avoiding rule, and algorithms that use real eyes as the avoiding rule. In two player Go, using virtual eyes is superior to using surrounded intersections [3]. Surprisingly, it

Table 1. Results for different UCT algorithms

	Playouts	Black	Black points	White	White points	Red	Red points
1	1,000	Surrounded	29.78	Surrounded	25.02	Surrounded	26.20
2	1,000	Eye	26.80	Surrounded	27.76	Surrounded	26.42
3	1,000	Surrounded	27.91	Eye	28.66	Eye	24.44
4	1,000	Pattern	24.46	Surrounded	33.31	Surrounded	23.23
5	1,000	Surrounded	23.63	Pattern	23.82	Surrounded	33.56
6	1,000	Rave	36.40	Surrounded	21.34	Surrounded	23.26
7	1,000	Rave	27.94	Rave	26.74	Rave	26.32
8	10,000	Rave	28.27	Rave	27.13	Rave	25.60

is not the case in three player Go as can be seen from the first three lines of the table. I feel uneasy giving explanations about the strength of playout policies as it is a very non-intuitive topic, it has been observed that a *better* playout player (i.e. one that often wins playouts against a simple one) can give worse results than very simple playout players when used in combination with tree search. In all the following algorithms we have used the *Surrounded* rule.

The fourth and fifth lines of table 1 test an algorithm using MOGO patterns in the playouts [6]. It uses exactly the same policy as described in the report. Friend stones in the patterns are matched to friend stones on the board, and enemy stones in the patterns can be matched with all colors different from friend on the board. On the contrary of two player Go, MOGO's patterns give worse results than no pattern, and tend to favor the player following the player using patterns. A possible explanation for the bad results of patterns may be that in multiplayer Go moves are less often close to the previous move than in usual Go. The following algorithms do not use patterns in the playouts.

The next lines of table 1 test the *Rave* algorithm. When matched again two *Surrounded* UCT based algorithms, it scores 36.40 points which is much better than standard UCT. When three *Rave* algorithms are used, the results come back close to the results of the standard UCT algorithms. All the following algorithms use the *Rave* optimization.

5.2 Paranoid UCT

Table 2 gives the results of the Paranoid algorithm matched against the Rave algorithm. Paranoid against two Rave algorithms has better results than both of them. Two Paranoid algorithms against one Rave algorithm have also better results. When three Paranoid algorithms are matched, the scores come back to the equilibrium. So in these experiments, Paranoid is better than Rave.

5.3 Alliance UCT

Table 3 gives the results of the the Alliance algorithm with different scoring systems for the playouts. When using Normal scoring, Alliance is not much beneficial against Paranoid, but has better results against Rave.

Table 2. Results for the Paranoid algorithm

Playouts	Black	Black points	White	White points	Red	Red points
1,000	Paranoid	31.71	Rave	26.89	Rave	22.38
1,000	Paranoid	29.82	Paranoid	28.48	Rave	22.67
1,000	Paranoid	29.11	Paranoid	26.85	Paranoid	25.04

Using the Allies scoring gives better results for the Alliance and much worse results for Paranoid and Rave. The best results for the alliance are obtained with Joint scoring, in these case Paranoid only scores 1.57 on average and Rave 0.30.

Table 3. Results for the Alliance algorithms

Playouts	Black	Black points	White	White points	Red	Red points
1,000	Alliance(Normal)	28.69	Alliance(Normal)	25.09	Paranoid	24.58
1,000	Alliance(Normal)	32.15	Alliance(Normal)	30.44	Rave	14.56
1,000	Alliance(Allies)	30.78	Alliance(Allies)	29.85	Paranoid	6.91
1,000	Alliance(Allies)	32.88	Alliance(Allies)	30.55	Rave	2.64
1,000	Alliance(Joint)	34.38	Alliance(Joint)	33.12	Paranoid	1.57
1,000	Alliance(Joint)	35.61	Alliance(Joint)	34.01	Rave	0.30

5.4 Confident UCT

In this section, we address the various confident algorithms. In our tests, we used 1,000 playouts for each tree developed by the various confident algorithms.

Table 4 gives the results of the Confident algorithms against the Rave and the Paranoid algorithms. The Paranoid and the Rave algorithms are clearly better than the confident algorithm, and Paranoid is slightly better than Rave.

Table 4. Results for the Confident algorithm

Playouts	Black	Black points	White	White points	Red	Red points
1,000	Confid(Normal)	18.08	Confid(Normal)	23.84	Paranoid	39.08
1,000	Confid(Normal)	17.75	Confid(Normal)	24.86	Rave	38.40
1,000	Confid(Allies)	14.22	Confid(Allies)	17.67	Paranoid	48.11
1,000	Confid(Allies)	10.99	Confid(Allies)	22.24	Rave	45.97

Table 5 gives the results of the Symmetric Confident algorithms against the Rave and the Paranoid algorithms. Here again, Paranoid and Rave are better that Symmetric Confident, and Paranoid is slightly better than Rave.

Table 5. Results for the Symmetric Confident algorithm

Playouts	Black	Black points	White	White points	Red	Red points
1,000	Sym(Normal)	19.94	Sym(Normal)	16.67	Paranoid	42.97
1,000	Sym(Normal)	18.97	Sym(Normal)	19.54	Rave	39.85

In table 6 two types of Same algorithms are tested against Paranoid and Rave algorithms. When the Same algorithm uses Normal scoring at the end of the playouts, playing two Same algorithms against a paranoid algorithm is not much more beneficial and even slightly worse than using paranoid algorithms as can be seen by comparing with table 2. So cooperation with Normal playout scoring is not much beneficial. However, when the playouts are scored according to the Allies scoring, cooperation becomes much more interesting and the paranoid algorithm gets much worse results (10.28 instead of 27.79). The Rave algorithm is in this case even worse with only 4.73 points. For all these results, the games were scored with the Normal scoring algorithm. Only the playouts can be scored with the Allies scoring algorithm. This is why the games played by the Same(Allies) algorithms do not sum up to 81 points, the allies algorithm stops playing when empty intersections are surrounded by allies, but the game is scored with the normal scoring and the empty intersection that are surrounded by more than one color are not counted.

When three Same(Allies) algorithms play together, the results favor the first player.

The next two lines of table 6 gives the results of the Same algorithm with Joint scoring of the playouts. The Same algorithm gives even better results in this case, and Paranoid and Rave often end the games with no points at all.

When Same(Allies) plays against/with Same(Joint) and Paranoid, it scores extremely well. Paranoid is beaten when the two Same algorithms decide to make an alliance, and Same(Allies) beats Same(Joint) since it continue to make territory for itself at the end of the game when Same(Joint) considers it common territory.

Table 6. Results for the Same algorithm

Playouts	Black	Black points	White	White points	Red	Red points
1,000	Same(Normal)	25.38	Same(Normal)	32.21	Rave	23.34
1,000	Same(Normal)	25.61	Same(Normal)	27.59	Paranoid	27.79
1,000	Same(Allies)	32.53	Same(Allies)	37.76	Rave	4.73
1,000	Same(Allies)	30.83	Same(Allies)	34.03	Paranoid	10.28
1,000	Same(Allies)	35.78	Same(Allies)	22.74	Same(Allies)	17.92
1,000	Same(Joint)	35.32	Same(Joint)	34.32	Rave	0.58
1,000	Same(Joint)	33.62	Same(Joint)	34.06	Paranoid	1.70
1,000	Same(Allies)	64.96	Same(Joint)	7.95	Paranoid	4.76

5.5 Techniques used in the algorithms

Table 7 gives the techniques used for each algorithm.

Table 7. Techniques used for each algorithm

Algorithm	Playout policy	Search	Scoring function
Surrounded	Surrounded by player	UCT	stones + eyes
Eye	Eye by player	UCT	stones + eyes
Pattern	Surrounded by player + Patterns	UCT	stones + eyes
Rave	Surrounded by player	Rave	stones + eyes
Paranoid	Surrounded by coalition	Rave	stones + eyes
Alliance(Normal)	Surrounded by coalition	Rave	stones + eyes
Alliance(Allies)	Surrounded by coalition	Rave	stones + common eyes
Alliance(Joint)	Surrounded by coalition	Rave	common stones + common eyes
Confid(Normal)	Surrounded by coalition	Rave	stones + eyes
Confid(Allies)	Surrounded by coalition	Rave	stones + common eyes
Sym(Normal)	Surrounded by coalition	Rave	stones + eyes
Same(Normal)	Surrounded by coalition	Rave	stones + eyes
Same(Allies)	Surrounded by coalition	Rave	stones + common eyes
Same(Joint)	Surrounded by coalition	Rave	common stones + common eyes

6 Conclusion

We addressed the application of UCT to multiplayer games, and more specifically to multiplayer Go. We defined a simple and effective heuristic, used in the playouts, that models coalitions of players. This heuristic has been used in the Alliance algorithm and can be very effective when used with the appropriate scoring of the playouts. The Paranoid algorithm, which assumes a coalition of the other players plays better than the usual Rave algorithm. Confident algorithms, such as Confident and Symmetric Confident, that choose to cooperate when it is most beneficial are worse than the Paranoid algorithm when they are not aware of the other players algorithms. However, when the players algorithms are known, as in the Same algorithm, they become better than the Paranoid algorithm.

If a multiplayer Go tournament were organized, the best algorithm would be dependent on the available information on the competitors. Future work will address the extension of coalition algorithms to more than three players, the effects of communications between players as well as the parallelization of the algorithms [4].

References

1. B. Bouzy and T. Cazenave. Computer Go: An AI-Oriented Survey. *Artificial Intelligence*, 132(1):39–103, October 2001.

2. B. Bruegmann. Monte Carlo Go. Technical Report, http://citeseer.ist.psu.edu/637134.html, 1993.
3. T. Cazenave. Playing the right atari. *ICGA Journal*, 30(1):35–42, March 2007.
4. T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Computer Games Workshop 2007*, pages 93–101, Amsterdam, The Netherlands, June 2007.
5. R. Coulom. Efficient selectivity and back-up operators in monte-carlo tree search. In *Computers and Games 2006*, Volume 4630 of LNCS, pages 72–83, Torino, Italy, 2006. Springer.
6. S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in monte-carlo go. Technical Report 6062, INRIA, 2006.
7. Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *ICML*, pages 273–280, 2007.
8. L. Kocsis and C. Szepesvàri. Bandit based monte-carlo planning. In *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.
9. Richard E. Korf. Multi-player alpha-beta pruning. *Artif. Intell.*, 48(1):99–111, 1991.
10. D. E. Loeb. Stable winning coalitions. In *Games of No Chance*, volume 29, pages 451–471, 1996.
11. Carol Luckhart and Keki B. Irani. An algorithmic solution of n-person games. In *AAAI*, pages 158–162, 1986.
12. Nathan R. Sturtevant. Last-branch and speculative pruning algorithms for $\max^n$. In *IJCAI*, pages 669–678, 2003.
13. Nathan R. Sturtevant. Current challenges in multi-player game search. In *Computers and Games 2004*, volume 3846 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2006.
14. Nathan R. Sturtevant and Richard E. Korf. On pruning techniques for multi-player games. In *AAAI/IAAI*, pages 201–207, 2000.