PLAYING THE RIGHT ATARI

Tristan Cazenave[1]

L.I.A.S.D. Université Paris 8, Saint-Denis, France

ABSTRACT

We experimented a simple yet powerful optimization for Monte-Carlo Go tree search. It consists in dealing appropriately with strings that have two liberties. The heuristic is contained in one page of code and the Go program that uses it improves from 50 % of won games against Gnugo 3.6 to 76 % of won games.

1. INTRODUCTION

Monte-Carlo Go is an active research field. Recent progress makes it a competitive and probably the best algorithm for 9x9, 13x13 and even 19x19 boards. Crazy Stone (Coulom, 2006) and MoGo (Gelly *et al.*, 2006; Gelly and Wang, 2006) are two bright examples of this trend . We propose an improvement of the algorithm used by Crazy Stone. The heuristic is based on simple Go knowledge that is used during random games. The strengths of the heuristic are that it is simple, easy to implement, powerful and is contained in one page of code. Moreover the overhead is small compared to the improvement.

The heuristic we present uses abstract game knowledge (the number of liberties). It is different from the heuristic used in MoGo, which is based on patterns. We advocate that the use of abstract heuristics is an alternative to the use of raw patterns in Monte-Carlo Go.

In the second section, Monte-Carlo Go is presented. In the third section the combination of Monte-Carlo Go with search as in Crazy Stone is recalled. In the fourth section, the heuristics that were used to improve random games are described. The fifth section gives experimental results. The sixth section concludes.

2. MONTE-CARLO GO

Monte-Carlo Go is quite a popular technique. It has the merit of simplicity, and on all boards it can beat programs that are much more complex and difficult to improve. We start with standard Monte-Carlo Go, then we discuss the improvements that can be made to Monte-Carlo Go by adding Go knowledge. We end this section with considerations on the combination of global search with Monte-Carlo Go.

2.1 Standard Monte-Carlo Go

Gobble (Bruegmann, 1993) has been the first program to use Monte-Carlo methods for the game of Go. 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 the following move in the list 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. Standard Monte-Carlo Go has a good global sense but lacks of tactical knowledge. For example, it often plays useless atari, puts itself in a harmful atari, or tries to save captured strings.

[1]email:cazenave@ai.univ-paris8.fr

2.2 Monte-Carlo Go with Go knowledge

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 (Bouzy, 2005). Another use of Go knowledge is to bias the selection of moves during the random games using 3×3 patterns and rules (Bouzy, 2005). Crazy Stone also uses Go knowledge to bias move generation, mainly knowledge on strings in atari (Coulom, 2006). Another combination of Go knowledge with Monte-Carlo Go is to search for unstable connections, and then to evaluate them using the average of the random games where they have succeeded (Cazenave and Helmstetter, 2005). The use of a few well chosen patterns that find moves near the previous move has improved much the level of MoGo (Gelly *et al.*, 2006).

2.3 Combining global search with Monte-Carlo Go

The combination of global search with Monte-Carlo Go has been studied by B. Bouzy (Bouzy, 2006) for 9x9 Go. His algorithm associates progressive pruning to Alpha-Beta search to discriminate between moves in a global search tree with Monte-Carlo evaluation at the leaves.

Crazy Stone (Coulom, 2006) 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.

Another improvement is to use permutations of moves to reduce the number of games that have to be played in a global search with Monte-Carlo evaluation at the leaves (Cazenave, 2006).

Currently, the best Go program is probably MoGo which is based on the combination of global search and Monte-Carlo Go.

3. MONTE-CARLO TREE SEARCH

Our base program is a re-implementation of Crazy Stone (Coulom, 2006) with some modifications. UCT is used as the exploration policy and the program stops searching when the game is evaluated as 100 % won or 100 % lost.

3.1 Re-implementing Crazy Stone

All the Go specific heuristics described in R. Coulom's paper have been reused with exactly the same values. The backup operator has also been reused. Instead of computing the average score of a position, the program computes the percentage of won random games in this position. The evaluation of a leaf is the percentage of won random games. This heuristic was used in the version of Crazy Stone that participated in the 2006 Olympiad.

3.2 UCT

A modification also used by recent versions of Crazy Stone and MoGo is to use the UCT algorithm (Kocsis and Szepesvári, 2006) in order to choose between moves to explore in the global search tree.

When choosing a move to explore, there is a balance between exploitation (exploring the best move so far), and exploration (exploring other moves to see if they can prove better). The UCT algorithm addresses the exploration/exploitation problem. UCT consists in exploring the move that maximizes $\mu_i + C \times \sqrt{log(t)/s}$. The mean result of the games that start with the c_i move is μ_i, the number of games played in the current node is t, and the number of games that start with move c_i is s.

The C constant can be used to adjust the level of exploration of the algorithm. High values favor exploration and low values favor exploitation.

3.3 The Backup Operator

We have reused the backup operator of Crazy Stone which is given below, *childMaxGames* is the child with the maximum number of games, *childSecondBest* is the child that has the second best mix value after *bestChild* :

```
computeMix () {
  Node *childMaxGames, *childSecondBest, *second;

  if (No Child) {
    mix = mean;
  }
  else {
    if (childMaxGames->games > bestChild->games)
      second = childMaxGames;
    else
      second = childSecondBest;

    float MeanWeight = 2 * sizeGoban;
    if (games > 16 * sizeGoban)
      MeanWeight *= games / (16 * sizeGoban);
    mix = mean;
    if (bestChild->games && second->games) {
      float AveragedValue0 = (bestChild->games * bestChild->mix +
              MeanWeight * mix) / (bestChild->games + MeanWeight);
      float AveragedValue1 = (second->games * bestChild->mix +
              MeanWeight * mix) / (second->games + MeanWeight);
      if (bestChild->games < second->games) {
        if (second->mix > mix)
          mix = AveragedValue1;
        else if (bestChild->mix < mix)
          mix = AveragedValue0;
      }
      else
        mix = AveragedValue0;
    }
    else
      mix = bestChild->mix;
  }
}
```

The backup operator enables to get information from all the tree and not only from leaves with a sufficient number of games as in previous approaches. It gradually changes the evaluation of a node from the mean to the maximum of its children. When a small number of games are available it is better to be close to the mean, but when the children have gained more confidence with more games, it progressively switches to the maximum.

3.4 Stop Searching when the Game is Clearly Over

In order to avoid losing time at the end of a game, Golois stops searching when the Monte-Carlo tree search returns 0 or 1 after 1,000 random games. If the root node has value 0, it passes. If the root node has value 1 after 1,000 random games, it searches over all the children of the root node the one with evaluation 1 and the maximum number of games. It also verifies that this children has more than 50 games. Then, if all the conditions are satisfied, it plays the move that corresponds to the selected children. Otherwise, it continues searching.

This modification enables it to spend more time on the first and more important moves of the game, and not to spend time on trivial moves. Golois has never lost a won game with this optimization (it is important to select a children that has a sufficient number of games or it can play losing moves occasionally).

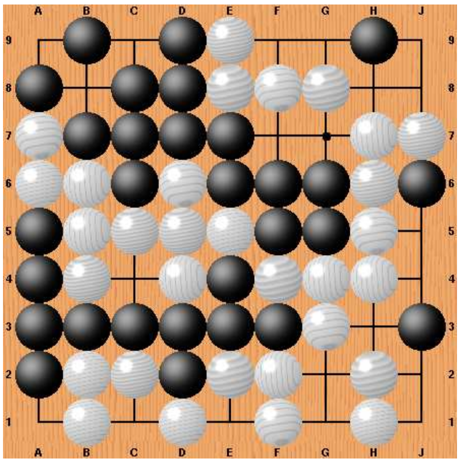

Figure 1: Wrong evaluation due to not using real eyes

4. IMPROVING RANDOM GAMES

This section presents two simple heuristics that improve the playing level of random games, and also the playing level of the Monte-Carlo program. The first one it to use real eyes instead of surrounded intersections. The second one is to handle strings with two liberties.

4.1 Real Eyes

A common implementation of Monte-Carlo Go avoids playing on intersections that are already surrounded by stones of the color to play, and that are not in atari. I believe it is better to use real eyes instead. An example where it is harmful to avoid playing at surrounded intersections is given in figure 1. Black is the Monte-Carlo program, and it believes playing at C4 wins the game because it never considers the White moves at C1 and E1. But when Black plays C4, White answers at C1, and the game is lost. Therefore the Monte-Carlo evaluation is wrong due to the use of surrounded intersections instead of real eyes.

A real eye is an empty intersection that is surrounded by stones of the color to play, that are not in atari. Moreover, it takes into account the status of the diagonal intersections. It computes the sum of the diagonals of the color opposite to the color to play and of the diagonals that are not protected for the color to play. If the intersection has four neighbors, it tests if the sum is one or zero. If the intersection has less than four neighbors, it tests if the sum is zero. We give below the code for real eyes.

```
bool realEye (int Move, int Color) {
  for all neighbors of Move
    if (board [neighbor] == Color)  {
      if (liberties [neighbor] == 1)
        return false;
    }
    else
      return false;

  int nbDiagOther = number of diagonals owned by the opponent;
  int nbDiagNotProtected = number of empty diagonals not protected;
  if (Move has four neighbors) {
    if (nbDiagOther + nbDiagNotProtected > 1)
      return false;
  }
  else {
    if (nbDiagOther + nbDiagNotProtected > 0)
      return false;
  }
  return true;
}
```

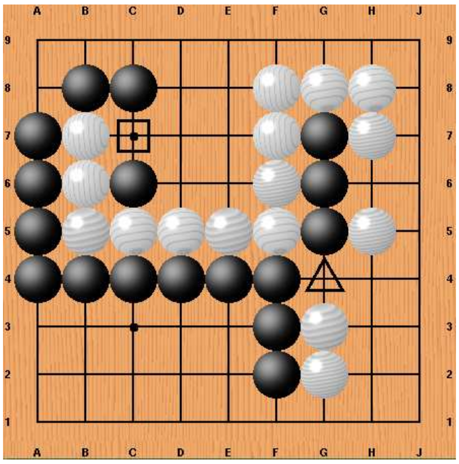

Figure 2: Examples of move urgency

4.2 Playing atari

We say that a string is adjacent to another string if they are of different colors, and there is at least one stone of the first string which is a neighbor of a stone of the other string.

In order to improve pseudo-random games, we consider the case of strings with two liberties. For such strings, a program can efficiently detect cases of simple capture. For example, if the string has no adjacent string in atari and if the attacker plays on one liberty of the string and the defender answers on the other liberty but fails to increase its number of liberties, then the string can be captured. This situation can be detected fast with no search, computing for example the number of liberties if the defender plays on one of its liberties and checking that this number is lower than two. In this case, playing on the other liberty is a good move for the attacker.

Considering the defender point of view, if playing on the other liberty gains more than three liberties, it is an interesting move for the defender since it prevents the attacker from playing a good attacking move.

Even if the defender gets three liberties by playing on one of its liberties, playing on the other liberty is still a threat for the attacker and can be given an urgency greater than a normal move. Symmetrically, playing on the other liberty is also a good move for the defender since it prevents a threat.

In order to be coherent with the values used in Crazy Stone, we have set the urgency $10,000\times$*size of the string* for capturing a string with two liberties, $1,000\times$*size of the string* for defending a string with two liberties, and $200\times$*size of the string* for playing and avoiding threats.

Examples of moves that have a modified urgency are given in figure 2. In the example position, the triangle move for White has an urgency of 30,000, the triangle move for Black has an urgency of 3,000. The square move has an urgency of 200 for Black, and also an urgency of 200 for White. The code that implements the *playing atari* heuristic is given below.

```
updateUrgenciesTwoLiberties (int colorToPlay) {
  for all strings
    if (number of liberties of string == 2) {
      if (no adjacent string in atari) {
        int Lib1 = first liberty of string;
        int Lib2 = second liberty of string;
        int nblib1 = nb liberties of string if colorOfString plays at Lib1;
        int nblib2 = nb liberties of string if colorOfString plays at Lib2;
        if (colorOfString == colorToPlay) {
          if (nblib1 > nblib2) {
            if (nblib2 == 3)
              urgency [Lib1] += 200 * size of string;
            else if (nblib2 <= 2)
              urgency [Lib1] += 1000 * size of string;
          }
          else if (nblib1 < nblib2) {
            if (nblib1 == 3)
              urgency [Lib2] += 200 * size of string;
            else if (nblib1 <= 2)
              urgency [Lib2] += 1000 * size of string;
          }
        }
        else { // color of string != colorToPlay
          if (nblib1 > nblib2) {
            if (nblib2 == 3)
              urgency [Lib1] += 200 * size of string;
            else if (nblib2 <= 2)
              urgency [Lib1] += 10000 * size of string;
          }
          else if (nblib1 < nblib2) {
            if (nblib1 == 3)
              urgency [Lib2] += 200 * size of string;
            else if (nblib1 <= 2)
              urgency [Lib2] += 10000 * size of string;
          }
          else if (nblib1 == 3) {
            urgency [Lib1] += 200 * size of string;
            urgency [Lib2] += 200 * size of string;
          }
          else if (nblib1 <= 2) {
            urgency [Lib1] += 10000 * size of string;
            urgency [Lib2] += 10000 * size of string;
          }
        }
      }
    }
}
```

5. EXPERIMENTAL RESULTS

We made two series of experiments. The first ones are based on the number of games played at each move. The second ones are based on fixing a time for each move. We have also reimplemented the pattern of MoGo (Gelly *et al.*, 2006) in our program, and the program that uses patterns has performances that are very close to the performances of the program that uses the *playing atari* heuristic.

5.1 Experiments with a fixed number of games

For these experiments, the programs run on a Pentium 4, 3 GHz with 1 GB of RAM. Each experiment consists in playing 100 9x9 games, 50 with black and 50 with white. The games were played against Gnugo 3.6 default level. The programs are not multi-threaded.

100,000 random games are played for each move of a game. UCT uses $\mu_i + \sqrt{\frac{log(t)}{10 \times s}}$ to explore moves. There is no opening book.

The first experiment tests the re-implementation of Crazy stone with UCT and the percentage of won games as an evaluation function. This program is called our *base* program.

The second experiment was to test the base program enhanced with real eyes and stop search. The third experiment was to test the base program enhanced with real eyes, *playing atari*, and stop search if the game is clearly over.

The results of these three experiments are given in the table 1.

Algorithm	time for first move	% of won games
base	22.7 s.	52 %
base + eye + stop search	23.4 s.	57 %
base + eye + stop search + playing atari	27.7 s.	78 %

Table 1: Results of 100 games against Gnugo 3.6.

Using the eye heuristic, the *playing atari* heuristic, and 100,000 random games, Golois wins 78% of its games against Gnugo 3.6, which is a nice improvement over the 52 % obtained for the base program.

The time required by the *playing atari* heuristic is relatively very low compared to the time that would be required to reach the same level only increasing the number of random games. In another experiment, $base + eye$ played 1,000,000 random games per move, and it scored 66 % against Gnugo 3.6.

Algorithm	time for 50 games
base + eye + playing atari	50,727 s.
base + eye + playing atari + stop search	36,985 s.

Table 2: Average time per game with and without stop search.

Table 2 gives the time used for fifty games (100,000 random games per move) with and without the *stop search* heuristic. It is clear that the heuristic gains time in the endgame. This time can be used in the beginning and middle game to improve playing strength.

5.2 Experiments with a fixed time

For these experiments, the programs run on a Pentium D 2.8 GHz with two cores and 1 GB of RAM. The games were played against Gnugo 3.6 default level. Each result correspond to playing 200 9x9 games, 100 with black and 100 with white. On this machine Gnugo 3.6 plays 9x9 games in approximately 73 seconds, but it does not scale as well as Monte-Carlo tree search with more time.

The programs use two threads, there is a mutual exclusion on the modification and the descent of the search tree.

Algorithm	2s.	4s.	8s.	16s.	32s.	64s.	128s.
base	8 %	12 %	30 %	36 %	41 %	46 %	50 %
base + eye + stop search	6.5 %	11 %	24 %	36 %	39.5 %	54.5 %	57.5 %
base + eye + stop search + playing atari	13 %	25.5 %	44 %	59 %	62 %	72 %	76 %

Table 3: Results of 200 games against Gnugo 3.6 with different times per move.

The time fixed for the moves is the CPU time, it means that for a fixed time of 2 CPU seconds, it plays moves in 1 real second.

Table 3 gives the percentage of won games for different fixed CPU times. The program using the *playing atari* heuristic clearly outperforms the other two programs for each fixed time. The *eye* heuristic gives better results than the base algorithm only for times greater than 64 seconds.

6. CONCLUSION

The *playing atari* heuristic is simple but it improves much the playing level of a Monte-Carlo tree search program. Playing 100,000 random games per move, it scores 78 % against Gnugo 3.6, compared to 57 % when not using it. With a fixed time of 128 seconds per move, it scores 76 % against Gnugo 3.6 compared to 50 % for the base program. The low time overhead of the heuristic, and its simplicity makes it a worthy heuristic for Monte-Carlo Go programs.

7. REFERENCES

Bouzy, B. (2005). Associating domain-dependent knowledge and Monte Carlo approaches within a go program. *Information Sciences*, Vol. 175, No. 4, pp. 247–257.

Bouzy, B. (2006). Associating shallow and selective global tree search with Monte Carlo for 9x9 go. *Computers and Games: 4th International Conference, CG 2004* (ed. N. S. N. H. Jaap Herik, Yngvi Björnsson), Volume 3846 of LNCS, pp. 67–80, Springer-Verlag, Ramat-Gan, Israel.

Bruegmann, B. (1993). Monte Carlo Go. ftp://ftp-igs.joyjoy.net/go/computer/mcgo.tex.z.

Cazenave, T. (2006). An efficient combination of global search and Monte-Carlo for games with permuting moves: application to 9x9 Go. *Proceedings Computers and Games 2006*, Torino, Italy.

Cazenave, T. and Helmstetter, B. (2005). Combining tactical search and Monte-Carlo in the game of Go. *CIG'05*, pp. 171–175, Colchester, UK.

Coulom, R. (2006). Efficient Selectivity and Back-up Operators in Monte-Carlo Tree Search. *Proceedings Computers and Games 2006*, Torino, Italy.

Gelly, S. and Wang, Y. (2006). Exploration exploitation in Go: UCT for Monte-Carlo Go. *NIPS-2006 : On-line trading of Exploration and Exploitation Workshop*, Whistler Canada.

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modification of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA.

Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. *ECML-06*.