Abstract Proof Search

Tristan Cazenave

Laboratoire d'Intelligence Artificielle
Département Informatique, Université Paris 8,
2 rue de la Liberté, 93526 Saint Denis, France.

cazenave@ai.univ-paris8.fr

Abstract. In complex games with a large branching factor such as Go, programs usually use highly selective search methods, heuristically expanding just a few plausible moves in each position. As in early Chess programs, these methods have shortcomings, they often neglect good moves or overlook a refutation. We propose a safe method to select the interesting moves using game definition functions. This method has multiple advantages over basic alpha-beta search: it solves more problems, the answers it finds are always correct, it solves problems faster and with less nodes, and it is more simple to program than usual heuristic methods. The only small drawback is the requirement for an abstract analysis of the game. We give examples and experimental results for the capture game, an important sub-game of the game of Go. The principles underlying the method are not specific to the capture game. The method can also be used with different search algorithms. This algorithm is important for every Go programmer, and is likely to interest other game programmers.

Key words: Computer Go, Search, Theorem Proving, Capture Game.

1 Introduction

It is very important in complex games where search trees have a large branching factor to safely select the possible moves worth trying. Finding the moves worth trying and the moves that can be eliminated, drastically reduces the search trees [1]. It is important to select the moves safely, which includes not forgetting a possible refutation and not considering as a refutation a useless move. Abstract Proof Search uses game definition functions to safely select complete and minimal sets of moves worth trying. The capture game is used as an illustration of the algorithm, experimental results for this sub-game of Go show that Abstract Proof Search is very efficient: it is more accurate, more safe and faster than basic alpha-beta search for this kind of problems.

The capture game is a fundamental sub-game of the game of Go. All the non-trivial computer Go programs use it. A Go proverb says "If you don't read ladders, don't play Go", its equivalent in computer Go is "if you don't program ladders, don't program Go" as Mark Boon pointed it. The capture game is important by itself, but it is also an important sub-game of other useful sub-games such as the connection, eye and life sub-games.

Proving theorems on the capture game is important because most or even all the other sub-games of Go rely on it. False results of the capture game can invalidate a connection or a life and death analysis, and it often results in the program losing a group or being under severe attack. It is responsible for many lost games.

In our experiments we use a variant of Alpha Beta Null Window Search. However, our method works with other search algorithms, it has also been successfully tested with Proof Number search for example.

Abstract Proof Search improves the speed and the accuracy of Go programs, it is likely that it can also be used to improve search in other games. The difference between our algorithm and other planning approaches to game playing using abstraction [3, 9] is that we concentrate on the classes of states that are worthwhile searching (ip1, ip2 and ip3 states at AND nodes) instead of identifying abstract operators. The word abstract in our algorithm means that the moves are selected using abstract properties of the objects of the games, such as the liberties of the strings.

The second section describes the capture game and its relation to other sub-games of Go. The third section uncovers our search algorithm. The fourth section explains what is the abstract analysis of games that enables Abstract Proof Search. In the fifth section we invalidate the widely accepted knowledge among Go programmers that the number of liberties is a good heuristic for the capture game, we show that the capture game is more subtle and that using too simple heuristics can be harmful. We propose a more accurate classification of situations worthwhile searching as well as a more selective move generator. The sixth section details experimental results and compares Abstract Proof Search to usual alpha-beta search for the capture game on standard test sets.

2 The capture game

The capture game is the most fundamental sub-game of Go. It is usually associated to deep and narrow search trees. It has strong relations with connections, eyes, life and death, safety of groups and many important Go concepts.

Figure 1 gives some examples of the capture game. The first example is called a geta, a white move at A captures the black stone marked with an x, it can be found in 5 plies. The second example is an illustration of the capture game as a sub-game of the connection game, a white move at B captures the marked black stone and connects the two white strings, it requires 9 plies. The third example shows the capture game as a sub-game of the life and death game, white at C can make two eyes by capturing the marked black stone in a simple but 15 plies depth ladder. Note that move B is harder to find than move C, the depth of a problem is not always a good measure of its complexity in Go.

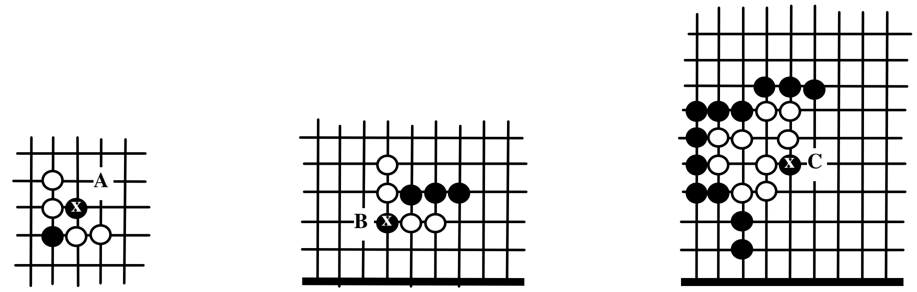

Figure 1. Examples of captures

3 The search algorithm

We use Null Window Search [7] with some modifications tailored to computer Go. We do not use forward pruning with null move search because we are looking for exact results, however some of our experimental results show that null move pruning can speed-up the algorithm with very little drawbacks or even improve it. We use iterative deepening, transposition tables, quiescence search, null-window search when not at the root and the history heuristic. We stop search early when the goal is reached. We search all the moves at the root, even if a previous move at the root has solved the problem, in order to find all the moves that reach the goal. Because it is useful for a Go program to know more than one way to accomplish its goals. Especially when it is useful for the program to reach multiple goals with one move.

To make the explanations easier, from now on, the friend color is black and the enemy color is white. The string that is under attack is black.

In the capture game, the evaluation can take three values : -INFINITY if the string has more than 5 liberties and it is white to play or the string is captured and it is black to play; +INFINITY if the string has more than 5 liberties and it is black to play or the string is captured and it is white to play; 0 when the state of the string is unknown. At the beginning of each node, the evaluation function is called, and the value is returned if it is different from 0.

We also use incrementality so as not to recalculate all the abstract properties of the strings after each move. We keep track of the liberties of the strings, and of the adjacent strings of each string. Each intersection is associated to a bit in a bit array so as to optimize checking of liberties, and the same is done for adjacent strings numbers.

Transposition Tables are used to detect identical positions and return the associated value if the search depth of the stored position is greater than the depth of the node or if the value is +INFINITY or –INFINITY. Transpositions are also used to recall the best move from previous search in the position and try it first when searching deeper so as to maximize cut-off. The size of the transposition table is set to 16384, a larger table could easily contain in memory, but the time to initialize the table before each search becomes too important for large tables. Given the simplicity of some problems and the number of different problems that have to be solved a small table is enough as the threshold for the number of nodes is set to 10 000. Another interesting possibility would be to set a larger size for the table, and to switch it off for the first 100 nodes, keeping the initialization for harder problems that potentially require many nodes.

The History Heuristic is used to order the moves that are not given by the transposition table. When all the moves at a node have been tried, the move that returned the best value, or the one that caused a cut-off, is credited with 2^{Depth}. At each node, the moves are sorted according to their credit, and tried in this order.

A Quiescence Search is performed at leaf nodes. The quiescence search alternatively calls two function QSCapture() that plays on the liberties of the string to capture if it has 2 liberties, and QSSave() that plays the liberty of the string to capture and the liberties of the adjacent strings in atari[1], if the string to capture is in atari. This ensures that the Quiescence search sends back correct results on the capture status of the string and quickly reads simple ladders.

Iterative deepening does not stop after the first winning move, it continues two more plies to find some other working moves. There are multiple stopping criteria to iterative deepening: the time allotted to the search, the number of visited interior nodes, the depth of the search, and the comparison between the depth of the first solution found and the current depth.

In order to find the games status and the moves associated to goals in the test positions, one or two searches may be performed. The first search is made with the player trying to capture playing first. If the goal is to prove that the string can be captured, no more search is performed. Otherwise another search is made with the player trying to save the string playing first, it is useful to know when the string is captured, and when it can be saved and which moves save it.

4 Abstract analysis of games

The possible moves that can modify the outcome of the search can be easily found when the goal is almost reached. However, when the goal is not one or two moves away, it becomes less clear. This section deals with the selection of a complete set of worthwhile possible moves, when the goal cannot be directly reached.

We try to find the complete set of abstract moves that can possibly change the outcome of a game a given number of moves in advance. For example, given that a string can be captured in 5 plies, we want to find all the abstract moves than can possibly prevent it to be captured in 5 plies. An abstract move is a move that is defined using abstract properties either of the strings or of the board. An example of an abstract move is 'a liberty of the string to capture'.

The set of possible moves that can modify the outcome of the search could be found dynamically by simply recalling the intersections tested during the search.

[1] atari means only one liberty left

The only moves that can modify the issue of the search are the moves that modify one of the tested intersections. Selecting forced moves in this way may be more general than an abstract analysis. It is done in [8], and it is similar to keeping an explanation of the search to find the forced moves as in early learning versions of Introspect [1]. Abstract analysis is more related to pre-computation of some parts of the search tree in order to be more efficient, such as in the partial evaluation version of Introspect [2]. Some more tests need to be performed to compare the two approaches and decide which one is the most efficient.

In the following we will use names for the different games states. The names of the games are usually followed by a number that indicate the minimal number of white moves in order to reach the goal. A game that can be won if white moves is called 'gi', a game where black has to play otherwise white wins the game by playing in it is called 'ip', it is the almost the same as 'gi' except that it is associated to black moves. A game that is won for white whatever black plays is called 'g'. A game is always associated to a player, the g and gi games are associated to the player that can reach the goal, the ip games are associated to the player that tries to prevent to reach the goal. Here the goal is to capture strings, it can be easily defined as: removing the last liberty of the string to capture. A forced move is a move associated to an ip game. For example, when the program checks whether a game is ip2, it begins with verifying that white can capture in two moves if it plays first (a gi2 game). The forced ip2 moves are the black moves that prevent white from capturing the string in two moves once one of the black ip2 moves has been played (we can say that the gi2 game has been invalidated by the black move).

It is quite simple to find forced moves, one move from the goal: when the string has only one liberty, the only moves to save the string are the moves that directly increase the number of liberties. There are only two ways to increase the number of liberties of a string: play one of its liberties, or remove an adjacent string in atari. These moves are associated to the ip1 game.

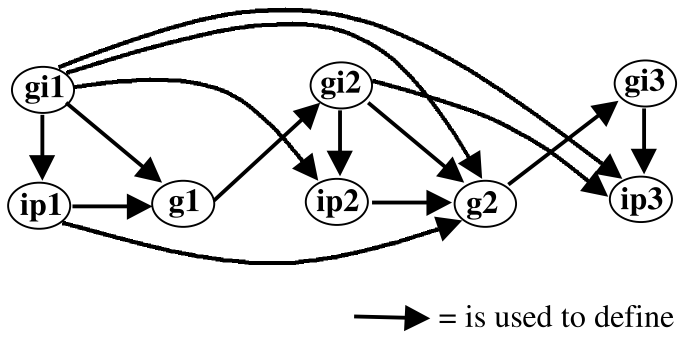

Figure 2. The dependencies between games

Figure 2 gives the dependencies between games. A game can be defined using the games for the lower number of plies, for example, the g1 game for white is defined as: the game is ip1 for black, and all the forced black moves lead to a gi1 game for white after the black move. So the g1 game is defined using the definitions of the gi1 and of the ip1 games, as it is shown in figure 2 where the g1 game depends on the gi1 and ip1 games. Another example is the gi3 game for white: a white move leads to a g2 game for white. So the gi3 game depends on the g2 game only. Some more detailed game definition functions are given in the next section on the selection of moves. In order to make things clear some examples of games are given in the figure 3.

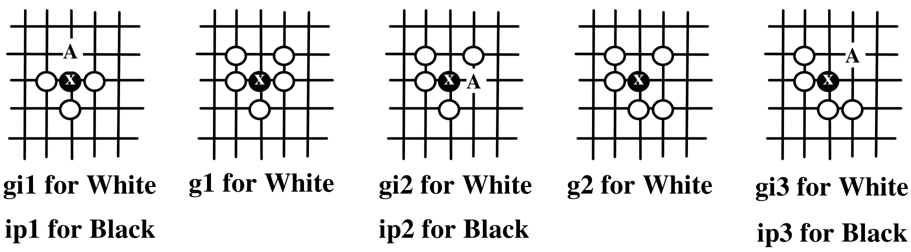

Figure 3. Examples of games

The only abstract moves that can change an ip1 game are the liberty of the string and the liberties of the adjacent strings in atari. A g1 game is defined as an ip1 game for black that is still gi1 for white after each of the forced black move is played. The set of intersections that are responsible for the state of a g1 game are the intersections involved in the corresponding ip1 game, and the intersections involved in the gi1 games following each forced black move. But as we know the abstract set of intersections for the ip1 and the gi1 games, we can deduce that the intersections responsible for a g1 game are the liberty of the string, the liberties of the adjacent strings in atari, the liberty after a black move is played on the liberty of the string, or on the liberties of the adjacent strings in atari.

Another example of how the abstract sets of moves can be calculated is the transition from a gi set of moves to an ip set of move: the only move that can modify an empty intersection is to play on this intersection, therefore if some empty intersections are involved in the definition of a gi game, the set of moves that can prevent it (the corresponding possible ip moves) contains all these empty intersections.

More detailed explanations of how this knowledge can be automatically generated can be found in [2], but more formal and easy to use tools for analyzing games need to be investigated.

A more practical example of such a set of abstract moves is the function that finds the complete set of abstract moves for the ip2 game. The function begins with adding the liberties of the string to the set of moves to prevent gi2, then for each liberties, it plays a black move on it, and adds the liberties of the string after the move. It also adds the liberties of the white strings adjacent to the string to capture that have strictly less than three liberties after the black move. Then it adds the liberties of all the adjacent string that have strictly less than four liberties (because a gi2 string has two liberties and the only adjacent strings that can be captured to save it have strictly less than four liberties: any four liberties adjacent string cannot be captured before the two liberties gi2 string). The code for CompleteSetOfMovesTo-Preventgi2 is quite simple, requiring only 16 lines of C. Similarly the code for CompleteSetOfMovesToPreventgi3 is 23 lines of C.

Here is the CompleteSetOfMovesToPreventgi2 function in pseudo-code that finds the complete set of abstract moves for the ip2 game:

```
CompleteSetOfMovesToPreventgi2(S) {
  for each liberty l {
    add l to S // add the liberty to the set of moves
    if (LegalMove (l,StringColor)) {
      MakeMove (l,StringColor);
      // add the liberties of liberties
      add the liberties after the move to S;
      // liberties of adjacent strings after the move
      for each adjacent string adj
        if (number of liberties of adj < 3)
          add the liberties of adj to S;
      UndoMove();
    }
  }
  // liberties of adjacent strings < 4 liberties
  for each adjacent string adj
    if (number of liberties of adj < 4)
      add the liberties of adj to S;
}
```

A property of the game of Go is that the minimum number of moves to take a string is its number of liberties. As a consequence, it is often useless to try to increase the number of liberties of a string by capturing an adjacent string that has more liberties or to play the external liberties of an adjacent string that has many liberties trying to make a seki[2] with it. There are exceptions to this rule when the adjacent string and the string to capture share some liberties or when the string to save has protected liberties and a sufficient number of liberties. Only in this case, it can be useful to fill the external liberties of the adjacent string in order to obtain a seki or to capture it so as to save the string under attack.

[2] Seki: Two strings that are mutually alive. One string cannot capture the other by playing a common liberty because it will be captured itself first. However as the pass move is legal in Go, the two strings of a seki are safe provided all the adjacent strings are also safe.

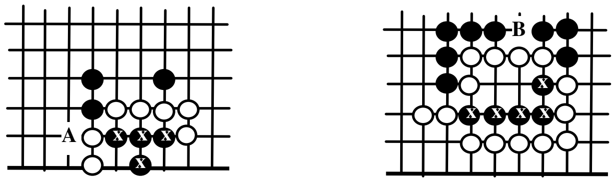

Figure 4. Playing liberties of adjacent strings with more liberties

The figure 4 gives illustrations of these two cases. In the left position, playing at A, one of the three liberties of a string adjacent to the string to save that has two liberties, enables to save it by capturing the adjacent string in 5 plies. The reason is that the string to save has two protected liberties after the move. In the right position, playing at B saves the marked string by making a seki between the black and the white strings. In the cases of ip2 and ip3 games, the string to capture has only two or three liberties and can be captured in 3 or 5 plies. These limitations ensure that looking at the adjacent strings that have less than one or two liberties more than the string to capture, is enough. Strings that become adjacent after a move can also be taken into account as shown in the CompletSetOfMovesToPreventgi2 function, where the abstract properties of the string are taken into account after some black moves are tried. Improvements could be made by also counting shared liberties between the string and its adjacent strings so as to be more selective on the adjacent strings to consider.

5 Selection of moves

The functions that safely select moves use practically very little knowledge, they are quite simple to program and are based on the abstract analysis of the game and the definition of games values. This way of coding the functions is more simple than explicitly programming all the interesting case. Experiments with coding all the cases related to the Preventip3 knowledge show that it needs 22034 lines of C for the Preventip3 function itself and some more lines to write the functions associated to the definitions of high level and abstract concepts which are called by the main function [1, 2].

Instead of explicitly coding all the cases, either in patterns or in complex programs, it is better to rely on the definition of games, and to rely on simple concepts only, simulating the playing of moves.

At each node and at each depth of the Abstract Proof Search, the game definition functions are called, they are equivalent to the development of small search trees. So Abstract Proof Search is a search algorithm that can be considered as developing small specialized search trees at each node of its search tree. At OR nodes, the program first checks if the position is gi1, if it is not, it checks if it is gi2 (equivalent to a depth 3 search tree), and if it is not, it checks if it is gi3 (equivalent to a depth 5 search tree). As soon as one of the gi games is checked, the program stops searching and sends back Won. Otherwise it tries the OR node moves associated to the position. At AND nodes, the same thing is done for ip1, ip2 and ip3 games, if none of them is verified, the programs sends back Lost, otherwise it tries the moves associated to the verified ip1, ip2 or ip3 game. Note that the game definition functions are equivalent to the programs generated by the Introspect system to safely select moves in games search trees [1, 2].

For example the pseudo-code that finds whether the string can be captured in 3 plies at each OR node is:

```
Capturegi2 () {
  res = 0;
  if (number of liberties == 2)
    for each liberty l
      if (res == 0)
        if (LegalMove (l, Opposite(StringColor))) {
          MakeMove (l, Opposite(StringColor));
          if (Captureg1())
            res = 1;
          UndoMove();
        }
  return res;
}
```

It relies on the Captureg1 function as shown by the arrow between gi1 and gi2 in the figure 2. The functions begins with verifying that the string to capture has two liberties. Then for each of the two liberties, and if the results has not been proved yet (res==0), it tries to fill the liberty, and verifies that the game is g1 after the liberty is filled, using the Captureg1 game definition function.

The function defining the ip2 game and its associated moves is equivalent to find the forced moves that prevent the string to be captured in 3 plies. It is checked at every AND nodes of the Abstract Proof Search tree provided the ip1 function has not been verified before:

```
Captureip2 (S) {
  res = 0;
  if (Capturegi2()) {
    res = 1;
    CompleteSetOfMovesToPreventgi2 (S1);
    for each move m of S1
      if (LegalMove (m, StringColor)) {
        MakeMove (m, StringColor);
        if (!Capturegi1())
          if (!Capturegi2())
            add move m to S;
        UndoMove();
      }
  }
  return res;
}
```

Again it is defined using simple concepts and the functions corresponding to other games. Here again, as shown in the figure 2, the ip2 game definition function relies on the functions defining the gi1 and gi2 games. The function adds the forced moves to prevent capture in 3 plies (the ip2 moves). The function begins with verifying that the string can be captured in two moves if white plays first, by calling the Capturegi2 game definition function. If it is the case, the function finds the complete set of black moves that may change the issue of a gi2 game by calling the function CompleteSetOfMovesToPreventgi2. Then, for each move of this set, it plays it and verifies that the game is not gi1 and not gi2 after the move. If it is the case, then the move has been successful in preventing the gi2 game, and is therefore an ip2 black move, so it adds the move to the set of forced ip2 moves.

In order to give an example for each kind of game, here is the pseudo-code that detects situations won 4 plies ahead:

```
Captureg2 () {
  res = 0;
  if (Captureip1(S)) {
    res = 1;
    for each move m of S
      if (LegalMove (m, StringColor)) {
        MakeMove (m, StringColor);
        if (!Capturegi1())
          if (!Capturegi2())
            res = 0;
        UndoMove();
      }
  }
  else if (Captureip2(S))
    res = S is empty;
  return res;
}
```

The Captureg2 game definition is a little more complex than the previous ones because there are two possibilities:

- Either the black string can be captured in one move by white, so it has only one liberty, and the Captureip1 function fills the set S with it. And after playing on its liberty the string can still be captured in two white moves (the Capturegi2 function matches).
- Or the function Captureip2 is verified, but all the moves that could prevent the game to be gi2 do not work, so the Captureip2 function sends back an empty set in S for the preventing moves. In that case, the game is won for white because none of the black moves to prevent gi2 works.

Again, as shown in the figure 2, the g2 game is defined using the ip1, ip2, gi1 and gi2 games.

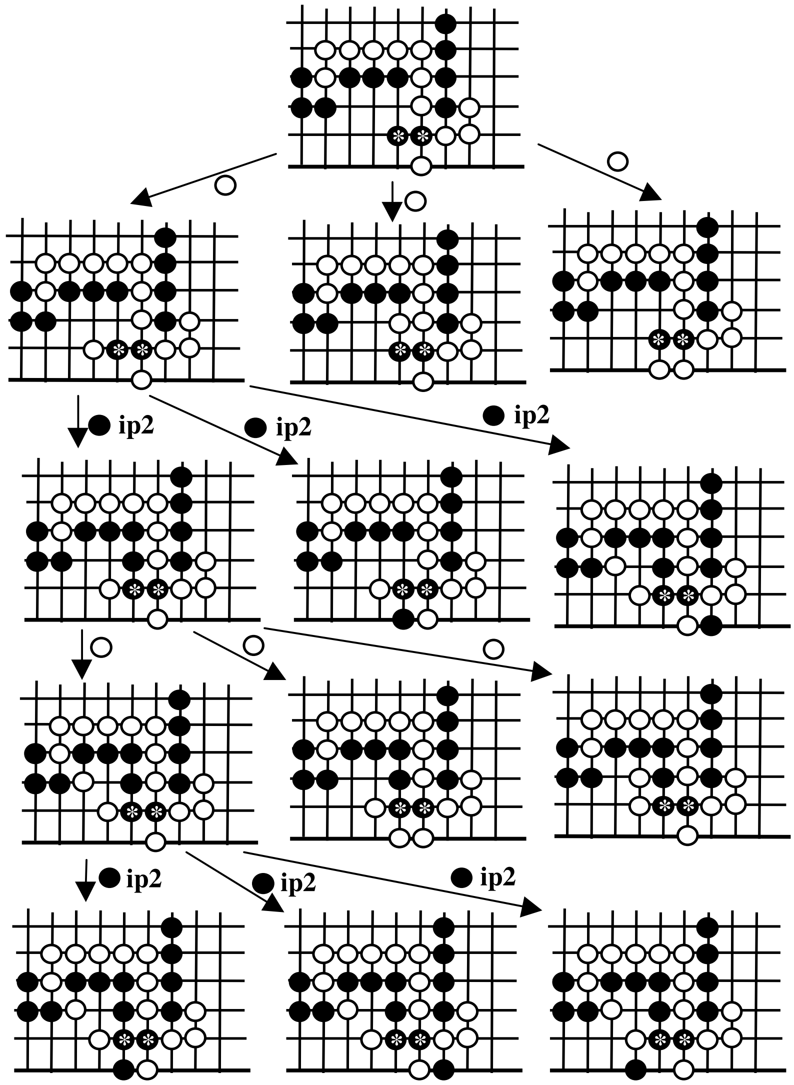

Figure 5. A part of an Abstract Proof Search tree

Figure 5 gives a part of an Abstract Proof Search tree, some moves at OR nodes (white moves) have been omitted. Each move is labeled with its color and for forced moves (black moves) with the name of the game that found it.

A widely accepted knowledge among Go programmers is that the number of liberties is a good heuristic for the capture game. In this paper we show that the capture game is more subtle and that using too simple heuristic can be harmful. We propose a finer classification of situations worthwhile searching, by considering forced moves only when a position can be proved to be winning a given number of plies in advance (gi games that enable to define ip ones).

Forgetting a move at an OR node can lead a program to miss a winning move, however it does not invalidate the result of the search: the result will be Unknown (0) instead of Won (+INFINITY). In the capture game OR node moves are moves that try to capture the string. On the contrary, forgetting an AND node move can make the result of a search wrong by missing a refutation. Our approach to games enable to be sure of not forgetting any move. Moreover it also enables to select only a subset out of all the possibly refuting moves. Selecting the minimal number of moves is as important as selecting all the necessary moves. Because, if a move does not interfere with the result, but the associated search returns Unknown or Lost, it is considered as a refutation and the program gives a false result.

Finding the complete set of forced moves, enables to prove theorems about games by not forgetting to consider some moves, and also by not considering moves that are proved not to have influence on the result of the game.

6 Experimental results

This section gives the results and the analysis of some experiments on a standard test set. We begin with describing how we have managed to compare basic alpha-beta search and Abstract Proof Search. At the end of the section some of the results are detailed and discussed. We give experimental results on a standard test set for capturing strings in Go: we call them ggv1 [4], ggv2 [5] and ggv3 [6]. We have selected all the problems involving a capture of a string, including semeai[3] and some connection problems. There are 114 capture problems in ggv1, 144 in ggv2 and 72 in ggv3. Experiments were performed with a K6-2 450 MHz microprocessor.

In order to compare Abstract Proof Search with the basic alpha-beta search usually performed in Go programs, we choose to use as basic alpha-beta search the same search algorithm with a different move generation function. The basic alpha-beta search calls the function CompleteSetOfMovesToPreventgi3 to generate the set of possible moves at AND nodes. It uses the same function for move selection as Abstract Proof Search at OR nodes. This way, we are fair to basic alpha-beta search, as it uses exactly the same move generation function as Abstract Proof Search, except that Abstract Proof Search uses games definition functions so as to be more selective and to ensure the validity of the results of the search. The CompleteSetOfMovesToPreventgi3 function verifies that a string has strictly less than four liberties, and if it is the case, it returns the liberties of the string, the liberties of the string if black moves are played on its liberties, the liberties of the adjacent strings that have strictly less than four liberties after the black moves on a liberty of the string to capture is played, the liberties of the string and the liberties of the adjacent strings that have strictly less than three liberties if two black moves are played on

[3] Semeai: race to capture between two or more strings.

the liberties of the string, and finally all the liberties of the adjacent strings that have strictly less liberties than the number of liberties of the string to capture plus two.

In the tables below, the basic alpha-beta search that stops whenever the time allotted to the search exceeds 1 second or the number of interior nodes exceeds 10,000 is called Preventip3-1s-10000N. Similarly, Preventip3-1s is the basic alpha-beta search that stops when the time exceeds 1 second. The Abstract Proof Search, using moves that prevent a goal up to 5 plies in advance is called ip3-1s-10000N. We do not give the results for ip3-1s since they are the same as for ip3-1s-10000N. In the number of nodes, we only count the interior nodes where some moves have been played. We do not count the leaf nodes (nodes where a transposition has occurred and has directly returned +INFINITY or –INFINITY are considered as leaf nodes).

Algorithm	Total time	Number of nodes	% of problems
Preventip3-1s-10000N	19.79	109117	99.12%
Preventip3-1s	19.79	109117	99.12%
ip3-1s-10000N	11.82	10340	99.12%

Table 1. Results for ggv1

Algorithm	Total time	Number of nodes	% of problems
Preventip3-1s-10000N	113.20	836387	78.47%
Preventip3-1s	118.60	870938	77.78%
ip3-1s-10000N	34.13	42382	88.19%

Table 2. Results for ggv2

Algorithm	Total time	Number of nodes	% of problems
Preventip3-1s-10000N	65.61	449987	65.28%
Preventip3-1s	74.25	483390	65.28%
ip3-1s-10000N	21.13	27283	73.61%

Table 3. Results for ggv3

The problem number 172 in volume 1 is not solved with our algorithm but is solved with the basic alpha-beta algorithm. Problem ggv1_172 can be considered as a mix of capture and life and death. It involves a nakade[4] shape that cannot be a part of the capturing game three moves ahead. The basic problem solver continues to play AND nodes moves even if the moves are not forced, provided the number of liberties is small enough. For these kinds of problems only, the basic algorithm can give better results. However, given the experimental results, the drawbacks of the basic algorithm are more important than its gains. As we can see with the different

[4] Nakade: a shape of string that makes an unsettled life shape when captured.

tables, the basic alpha-beta search method is not selective, and spends more time in useless branches of the tree.

One of the proposed metrics for performance is the percentage of solved problems, this percentage corresponds to the number of problems with a correct game value and a correct move, however on some problems, the basic alpha-beta algorithm sometimes also gives moves that do not work. They are not counted as wrong answers, so the metric favors the basic algorithm.

Surprisingly, there is one more solved problem in the ggv2-Preventip3-1s-10000N test than in the ggv2-Preventip3-1s, this is due to the complexity of some problems. When trying to find a move that saves the black stones, the algorithm does not stop until the Alpha-Beta returns –INFINITY or +INFINITY or one of the stopping criterion is met. So a move that returns 0, can be considered as a move that saves the endangered stones if the search threshold corresponding to the number of nodes is passed over. However, if more search is performed and the algorithm does not send back correct results, as Preventip3 does, the saving move can then be associated to –INFINITY, in other words as not saving the string. This is what happens here for one problem in ggv2, where more search with a basic algorithm leads to worse results.

Another problem related to the basic alpha-beta search is the treatment of sekis. There are seki positions that are correctly assessed by Abstract Proof Search in a natural way and incorrectly assessed by basic alpha-beta search. To prevent basic search from failing in these situations, some special code has to be added or pass moves may be considered. These possible solutions may be search time and/or programming time consuming.

Algorithm	Total time	Number of nodes	% of problems
Preventip3-10s-100000N	635.20	4607171	79.17%
ip3-10s-100000N	63.57	81302	90.28%

Table 4. Results for ggv2 with more time and nodes

Algorithm	Total time	Number of nodes	% of problems
Preventip3-10s-100000N	726.40	4319840	70.83%
ip3-10s-100000N	23.97	33936	73.61%

Table 5. Results for ggv3 with more time and nodes

In order to check whether the algorithm scales well, we also did some experiments with relaxed controls, which are unrealistic for today's technology, but that show the evolution of the problem solving when more computation is available. The results are summarized in tables 4 and 5. With stopping criteria of 10 seconds and 100,000 nodes, the interest of Abstract Proof Search increases. It solves much more problems in the tenth of the time of basic alpha-beta search for ggv2, and for even more complex problems such as in ggv3, it still solves more problem in 1/30th of the time of basic alpha-beta search. We can note that giving more time and more nodes to Abstract Proof Search for ggv3 does not change much the results, because for complex problems, Abstract Proof Search stops searching early as it does not find forced moves, whereas basic alpha-beta search, that relies on the number of liberties of strings is inaccurate in establishing the complexity of some problems and spends much time searching complex and useless sub-trees.

The average speed of basic alpha-beta search is approximately 7,000 nodes per second. It is much faster than Abstract Proof Search that only develops approximately 1,200 nodes per second. However, Abstract Proof Search finds the solutions to the problem in much less nodes than basic alpha-beta search. The verification of the game definitions functions at each nodes explains the relatively small speed of Abstract Proof Search. Each game definition function is equivalent to a small tree search. Transpositions Tables are not currently used in the game definition functions, their proper use may well speed-up Abstract Proof Search.

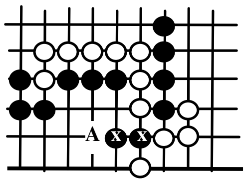

Figure 6. A problem solved by Abstract Proof Search, not by basic alpha-beta search

Figure 6 gives an example of a problem that is solved in 443 nodes and 0.33 seconds with Abstract Proof Search and which is not solved in 8,844 nodes and 1.21 seconds with basic alpha-beta search. The search is stopped as soon as it exceeds one second or 10,000 nodes. When more time and nodes are given to basic alpha-beta search, it solves the problem in 46,227 nodes and 6.59 seconds.

Algorithm	Book	Total time	nodes	%
Preventip3-1s-10000N-NM	ggv1	13.34	69582	98.25%
Preventip3-1s-10000N-NM	ggv2	66.55	518398	77.08%
Preventip3-1s-10000N-NM	ggv3	30.50	230724	65.28%
ip3-1s-10000N-NM	ggv1	10.58	9401	99.12%
ip3-1s-10000N-NM	ggv2	31.57	39220	88.89%
ip3-1s-10000N-NM	ggv3	16.93	20902	73.61%

Table 6. Results with null move forward pruning

Null move forward pruning has been tried with a reduction factor of four. The results shows that it is beneficial to the search algorithm, despite that forward pruning may alter the result of the search in some cases.

7 Conclusion

The theorem proving approach to the capture game in Go gives excellent results. It solves more problems than the basic alpha-beta approach, the answers it finds are always correct on the contrary of heuristics, it solve problems faster and with less nodes, and it is more simple to program than other approaches. The only small drawback is the requirement for an abstract analysis of the game. Note that the abstract analysis is not sufficient by itself, it has to be used with game definition functions for selecting moves.

The principles underlying the method are not specific to the capture game. They can also be used with different search algorithms. In the near future, we will try this method in other sub-games of Go and in other games. The games and sub-games that are concerned with this method are the games where a simple definition of the goal to reach can be given. Such games are for example the connection sub-game of Go, the virtual connections at Hex, or the five in a row game. Some sub-games of other difficult games such as mate search in Chess or Shogi may also benefit of our method. Improvements can be made by using transposition tables in the game definition functions, and by being even more accurate on the complete sets of moves to prevent gi games (for example taking into account the number of shared liberties between strings). Another important improvement to our current approach is the development of tools in order to facilitate the abstract analysis of games and the comparison between a dynamic selection of forced moves by analyzing the set of intersections tested during a search, and a selection based on abstract analysis. A combination of the two may well be the best alternative.

8 References

1. Cazenave T.: *Metaprogramming Forced Moves*. Proceedings ECAI98 (ed. H. Prade), pp. 645-649. John Wiley & Sons Ltd., Chichester, England. ISBN 0-471-98431-0. 1998.

2. Cazenave T.: *Generating Search Knowledge in a Class of Games*. submitted. http://www.ai.univ-paris8.fr/~cazenave/papers.html. 2000.

3. Frank I.: *Search and Planning under Incomplete Information: a Study using Bridge Card Play.* PhD Thesis, Department of Artificial Intelligence, University of Edinburgh. 1996.

4. Kano Y.: *Graded Go Problems For Beginners. Volume One.* The Nihon Ki-in. ISBN 4-8182-0228-2 C2376. 1985.

5. Kano Y.: *Graded Go Problems For Beginners. Volume Two.* The Nihon Ki-in. ISBN 4-906574-47-5. 1985.

6. Kano Y.: *Graded Go Problems For Beginners. Volume Three.* The Nihon Ki-in. ISBN 4-8182-0230-4. 1987.

7. Marsland T. A., Björnsson Y.: *From Minimax to Manhattan.* Games in AI Research, pp. 5-17. Edited by H.J. van den Herik and H. Iida, Universiteit Maastricht. ISBN 90-621-6416-1. 2000.

8. Thomsen T.: *Lambda-search in game trees – with application to go.* CG 2000. This volume.

9. Willmott S., Richardson J. D. C., Bundy A., Levine J. M.: *Applying Adversarial Techniques to Go.* Journal of Theoretical Computer Science. 2000.