Theorem Proving in the Game of Go

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. We present a method that enable to efficiently prove tactical theorems in the game of Go. We have experimented theorem proving with different tactical sub-games of the game of Go: the capture game, the connection game, the eye and the life and death games. Theorem proving works very well for tactical Go problem solving. The moves it finds are always correct and it finds the move faster than usual algorithms. In this paper we present our algorithms associated to concrete examples, and we outline the possible applications that might interest Go players: teaching programs for beginner's to learn to capture and connect stones, problem solving programs for intermediate players, and even some applications that might interest good players such as a perfect 5x5 Go problem solver and composer.

Key words: Game of Go, Theorem proving, Artificial Intelligence.

1 Introduction

Proving theorems about games can reduce their complexity. It also enables to solve problems that were not solvable with other algorithms. In our experiments, proving theorems in the game of Go has enabled to improve a lot tactical problem solving. We have experimented with different sub-games of the game of Go: capture, connection, eyes and life and death.

The second section describes how theorem proving is possible in the game of Go. The third section gives hints on how to evaluate the possibility of a connection. The fourth section deals with the non-transitivity of connections. The fifth section concentrates on search algorithms related to computer Go. The sixth sections states future work. The conclusion outlines the possible applications that might interest Go players and teachers.

2 Theorem Proving

In this section, we show how to prove tactical goals in the game of Go. Our method has also proven useful in other games such as Phutball, Gomoku or Hex.

In order to know if a sub-goal of the game of Go can be achieved, a computer develops an AND/OR search tree. Usually, the achievement of the goal is associated to 1, the failing to 0 and the unknown situation to 0,5. Algorithmic improvements over the basic AND/OR tree search algorithm can be discovered thanks to computer Go, such as Abstract Proof Search or Iterative Widening [Cazenave, 2000, 2001a]. These new algorithms have of course applications in computer games, but they can also benefit to other fields of computer science such as mathematical theorem proving for example.

In the first sub-section, we explain the origin of our notations for games. In the second sub-section we give the game definition functions that are the basis of our algorithms. In the third sub-section we outline how complete sets of moves can be found with relevancy zones at the AND nodes of the search trees. In the fourth sub-section, we detail another method for finding complete sets of moves based on abstract properties of the game. In the fifth sub-section, we give examples of the use of the game definition functions to solve connections problems.

2.1 Unknown Status and Game Names

Conway's number theory and combinatorial game theory have already proven useful for Go with the Thermostrat and the Hotstrat strategies for example [Conway & al., 1982]. Combinatorial games can only be computed when all the nodes of the corresponding tree have been assigned correct and precise values. In the course of a game of Go, it is not always possible to compute exactly all the values associated to a combinatorial game. Due to the lack of computing power, some of these values remain unknown. The classical combinatorial game theory does not handle these cases. We propose to define games with missing information at some nodes.

There are basically three values that a leaf of a tree can take: Won, Lost or Unknown. We use the letter 'i' for Unknown ('Inconnu' in French), 'g' for Won ('Gagné' in French) and 'p' for Lost ('Perdu' in French). Therefore a tree of depth 1 that is won if the friend player plays first, and that is Unknown if the opponent player plays first can be represented by the left tree of figure 1.

Figure 1. Simple games

As there is one friend move to win, the game is named a gi game, it is equivalent to { g | i } using Conway's notation. The right tree represents an ip game: a situation where the game is unknown if the friend plays first, and that is won for the opponent if the opponent play first (won for the opponent is lost for the friend). The name we give to the game definition functions of the next sub-section are inspired from this notation.

2.2 Game Definition Functions

Let P be a position and m a move. Let play(P,m) be the function that returns the position after move m on position P. The letter 'W' is used for White, and the letter 'B' for Black. The definitions are still valid with color reversed. We can define games:

gi_k (P,W) = W can capture in k White moves if W plays first in position P and if perfect play and alternated moves are assumed. It is equivalent to: $\exists$ White move { P_1=play(P,move), $g_{k-1}(P_1,W)$ }

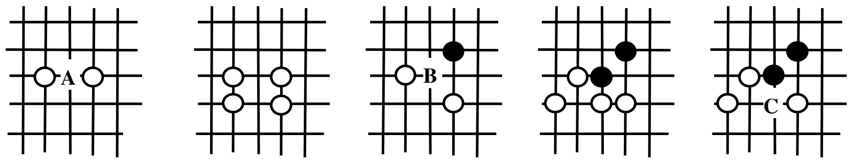
ip_k (P,B) = gi_k (P,W).

S_k (P,B) is the set of all black moves that prevent W from capturing in k or less white moves in position P if B plays first when ip_k (P,B) is verified: { B moves on P / { gi_k (P,W), { P_1=play(P,move), $\forall$ o, o<=k, not ($gi_o(P_1,W)$) }}}.

g_k (P,W) = $\exists$ m { m<=k, ip_m(P,B), $\forall$ move $\in$ S_m (P,B) { P_1=play(P,move), $\exists$ o {o<=k, $gi_o(P_1,W)$ }}}.

In some previous research [Cazenave, 1998], we have shown it is possible to automatically generate programs for the game definition functions using the rules of the game defined in a logic language, using a metaprogramming system. The generated programs select the same moves as our search based game definition functions. They can be generated dynamically by safely generalizing trace of proofs on examples [Cazenave, 1996], or statically by specializing the definitions of the games functions on the rules of the game [Cazenave, 1998].

Recently, we defined an equivalent search based algorithm that selects the same moves using small game definition functions based on abstract properties of the games [Cazenave, 2000]. The algorithm is more concise and easier to program than our previous meta-programming system. It needs to know the complete set of abstract moves that can change the outcome of a fixed depth, small search. For example, in the leftmost diagram of figure 2, the two white stones have one common liberty and can be connected in one white move at A, i.e. in a 1 ply search. The only abstract black moves to prevent the connection is the common liberty. Based on the logic of our previous system, we have designed complete sets of abstract moves that can prevent a goal proved in one, three or five plies search.



gi1 for White ip1 for Black | g1 for White | gi2 for White ip2 for Black | g2 for White | gi3 for White ip3 for Black

Figure 2. Examples of games

In the following we will give names to the different games, according to their possible outcomes. The names of the games are usually followed by a number that indicate the minimum number of white moves in order to reach the goal. A game that can be won if White moves first is called 'gi', a game where White wins unless Black plays first is called 'ip', it is the almost the same as 'gi' except that it is associated to a set of black moves. A game that White can win even if Black plays first 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 the opponent from reaching the goal. The gi and ip games are also associated to a set of moves. The ip_n moves are the moves that prevent a string to be captured in n moves by the opponent. For example, the ip_1 moves are the moves that may prevent two strings to be connected in one move (i. e. playing a common liberty).

A *forced move* is a move associated to an ip game. For example, when the program verifies a Black ip2 game, it begins with verifying that White can connect in two moves if he/she plays first (a gi2 game, associated to a three plies search). The forced ip2 moves are the black moves that prevent White from connecting the strings in a tree search of depth three once one of the Black ip2 moves has been played (we can say that the gi2 game has been invalidated by the black move, for example the move at B in the third diagram of figure 2 is an ip2 move for Black and a gi2 move for White).

2.3 Selection of the relevant moves using relevancy zones

A method for the selection of the relevant moves at AND nodes is the use of relevancy zones. A relevancy zone is a set of intersections that supports a proof. In order to prove tactical goals, the search engine has to memorize all the reasons that are responsible for the result of a search. In the leftmost diagram of figure 2, there is a winning connection game for White. Playing on the empty intersection marked with an 'A' connects the two white stones. To prove this, the search engine has plays the move at 'A', and verifies that the stones are connected after the move. During the search, the program has to memorize the conditions involved in this proof (i.e. a set of sufficient conditions that ensure that the strings can be connected as long as they are fulfilled). In this example it is not very hard to find them: the basic operations performed were : verify that the move is legal, play the move and verify that the strings are connected. For each of these basic operations some conditions have to be verified. For example, the verification that the move at A is legal for White tests if there is an empty intersection next to A (this is a sufficient condition for a move to be legal), therefore the first empty intersection next to A will be included in the set of sufficient conditions associated to the proof. Once we have this set of conditions, we can reduce it to a set of conditions on the intersections only. For example, if we have the condition that a black stone has only one liberty, the associated set of corresponding conditions on intersections is: the test on the emptiness of the liberty intersection, and one test for one liberty of each adjacent white string (to ensure that these strings are not captured).

Given the set of conditions on the intersections, it becomes easy to find the only Black moves that can invalidate the proof. A move that does not change any of the sufficient conditions cannot change the issue of the search. So the only Black moves are the moves on the intersections. The Black moves that can modify the issue of a search are the potential forced moves.

More generally, the set of possible moves that can modify the outcome of the search is found dynamically by memorizing the intersections tested during the search. The only moves that can modify the issue of the search are the moves that modify one of the tested intersections. It is similar to keeping an explanation of the search to find the forced moves.

2.4 Selection of the relevant moves using abstract knowledge

A problem related to the use of relevancy zones is that they select many useless intersections. For example, to verify that an adjacent string is not captured, one liberty of this string is included in the relevancy zone. Therefore, the corresponding move will be tried in higher order games. However, if the adjacent string has 6 liberties, it is clearly useless to try this move as it will not capture the adjacent string in one move. The use of goal dependent abstract knowledge enables to discard such moves, and therefore to reduce the combinatorial explosion of game definitions functions.

An example of a useful abstract knowledge for connections is that it is useless to try to verify a white gi_n games if n white moves in a row cannot connect the two strings. An even better example is that it is useless to try capturing a common adjacent string of n+1 liberties to prove a gi_n connection game (this knowledge saves more time than the n moves in a row knowledge).

Another kind of abstract knowledge for connections is that the only relevant moves to prevent the connection of two strings in a gi_2 are the common liberties of the two strings, the liberties of one string that are also second order liberty of the other string. The only adjacent strings of interest are the strings which are adjacent to both strings, and the strings adjacent to one string and which have a common liberty with the other string. The relevant moves to prevent a gi_2 connection are also composed of the liberties of the adjacent strings of interest, and the liberties of the strings in atari which are adjacent to the adjacent strings of interest.

The abstract knowledge on relevant moves can be automatically found using the rules of the game expressed in first order logic and a meta-programming system such as Introspect [Cazenave, 1996,1998].

2.5 Examples Using Connections

We give examples of solving connections using the games definition functions.

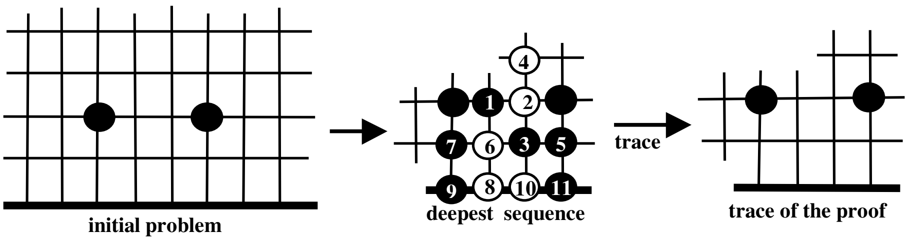


Figure 3. Memorizing relevant intersections

An example of a small proof of connection between two stones is given in figure 3. When proving a connection, all the intersections that have been tested during the search can be memorized. We have to memorize the intersections where some moves have been played, but also the intersections that have been tested to verify that a move is legal, or to find the forced moves. The only white moves that can prevent the connection between the two black stones are the moves on the intersections tested during the search. So here, we have only 14 possible white moves to disconnect black. Most of these moves are quickly refuted.

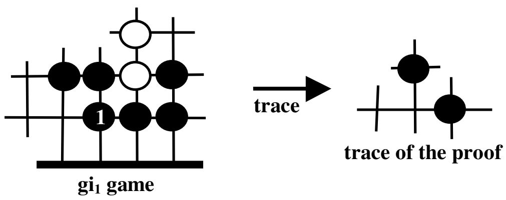


Figure 4. Finding the ip1 moves using the trace of the gi_1 proof

For example, after move 5 in figure 3, the game is gi_1 for Black (i.e. Black can connect in on move as shown in the left of figure 4). So there are only two White moves to consider. They are given by the empty intersections of the trace of the proof in figure 4.

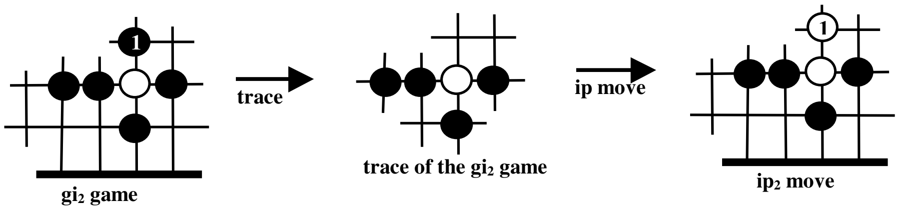


Figure 5. Finding the ip_2 moves using the trace of the gi_2 proof

Figure 5 gives an example of the finding of some ip_2 moves. First, the program finds that the sub-game is gi_2, then using the trace of the proof, it selects a few ip_2 moves. Reducing a lot the number of relevant moves.

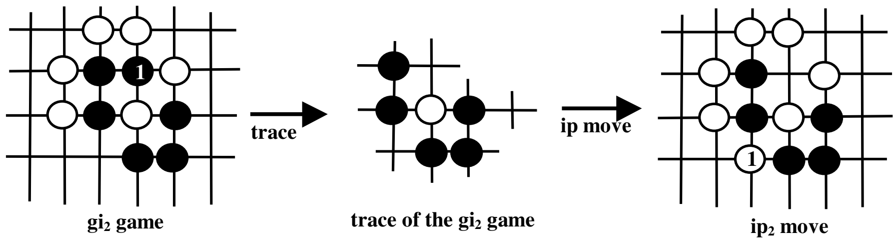


Figure 6. The importance of finding all the ip_2 moves.

Figure 6 gives an example where it is important to find all the relevant moves. Our method, based on theorem proving does not overlook moves. It is very important for simple sub-games such as capture, connection or eyes to consider all the relevant possible refutations. Overlooking a single move can invalidate the whole search.

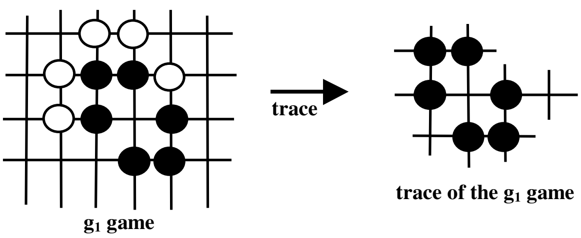


Figure 7. The trace of a g_1 game, used to find the trace of the gi_2 game of figure 4.

Figure 7 gives the g_1 game that enables to find the ip_2 move of figure 6. In practice, we have included moves that save a common adjacent string in atari in the ip_1 set of forced moves (strictly using our definitions, they would be ip_2 moves).

3 Heuristically evaluating the difficulty of a connection

A good heuristic to evaluate the difficulty of a connection is the minimal number of moves in a row to connect the two strings. This number is the minimal order of the ip game associated to the connection. If the player has to play three moves to connect, the associated ip game is at least ip_3.

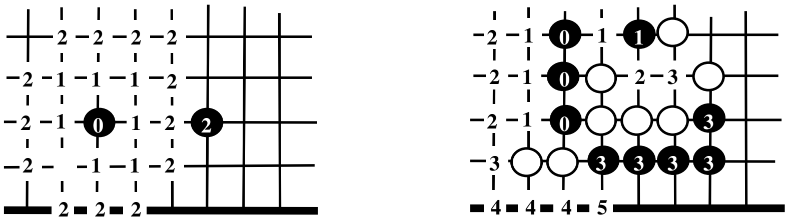


Figure 8. Evaluating the difficulty of a connection.

Figure 8 gives an example of such computations. These numbers can be computed incrementally after each move, so that the evaluation of the difficulty of the connection can be performed at a low cost. Another optimisation in the case of the connection sub-game is to consider these numbers for the two strings to connect. So that the order of the computation for one string is divided by two. It makes the computations faster as the computation of the order is exponential with its length, but it also find connections that would not be found with an approch based on only one string.

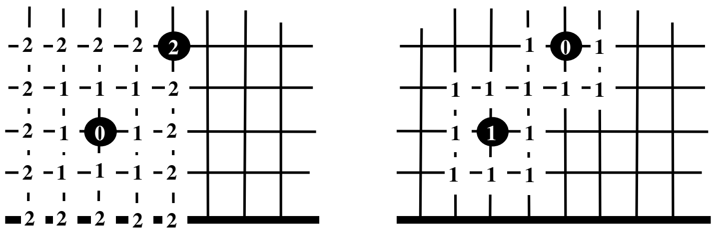


Figure 9. Computing connections from the two strings instead of one.

Figure 9 gives an example where it is more efficient to compute the sets associated to the number of moves for each string, and then to stop the search as soon as the two sets have an intersection. In the left diagram, the computation is stopped after the set issuing from the first string includes the second string. The result is that the heuristic evaluation of the connection is two moves. If instead we use the method alternatively for the two stones, as shown in the right diagram of figure 9, the heuristic connection is now evaluated to one move, which is a better result, and the result is found faster.

4 The Non-Transitivity of Connections

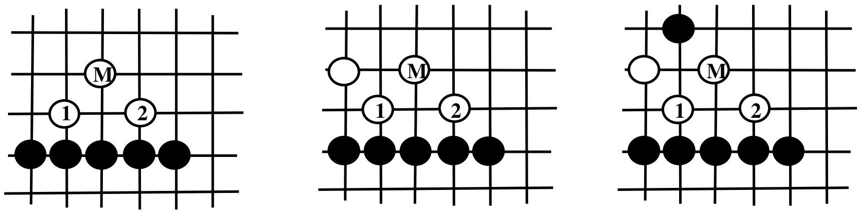


Figure 10. A frequent non-transitive connection

Figure 10 gives an example of non transitivity that arises quite often during games. We note by '1' and '2' the two strings that are not connected. However, '1' is connected to another string in the middle 'M', and 'M' is also connected to '2'. This non-transitivity of connection could be detected by a special pattern, but even to treat this simple case, many patterns could be necessary. The second diagram in figure 10 shows a case where '1' and '2' are connected with the same pattern, and the second diagram shows another case where they are not connected despite a pattern similar to the second diagram occurs. Discussions with other Go programmers such as D. Fotland [Fotland, 2001], tend to account for the fact that today's top programs use patterns to discriminate the cases of transitivity and non-transitivity. We think that this method is not general enough and that too many patterns are needed to judge accurately the non-transitivity. A general seach based method is preferable, this is the method we advocate for in this paper.

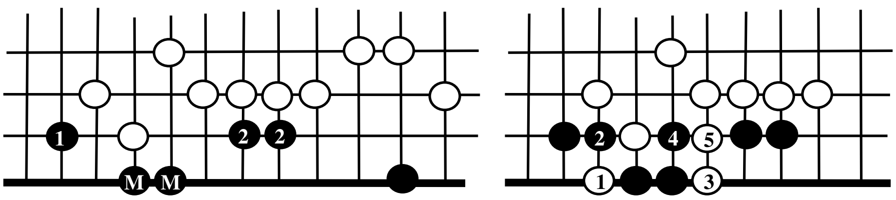


Figure 11. Non-transitivity in a real tournament game.

This position is from a game between Go4++ (Black) and Golois during the 2000 Computer Olympiad in London. Nick Wedd already pointed out the oversight of the two programs regarding connections in this game [Wedd, 2000]. This problem is typically a problem related to the non-transitivity of connections. At that time, Golois did not use the search based mechanism to find non transitivities, so it did not play the right move to disconnect Go4++. Given the moves in the game, it is also quite sure that Go4++ did not see the problem either. In this example as well as in the previous example, the string '1' is connected to the string 'M', the string 'M' is connected to the string '2', but '1' is not connected to '2'.

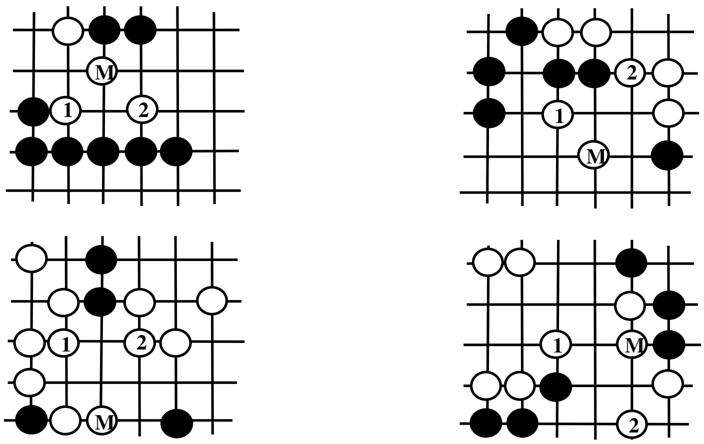


Figure 12. Other examples adapted from Golois games.

Figure 12 gives other examples of non transitivity that have arised during Golois games against other programs. A rule of thumb that can be devised from the examples in figure 10 is that the connection is transitive when one of the two empty intersection, which is not a common liberty is a protected liberty. However this rule is false as the first example of figure 12 shows.

From a more general point of view, it appears that expert Go players and programmers have problems in finding the cases where the rules they write do not work. This is a shortcoming that has already been reported in similar problems: for example in the western game of Chess, the international Chess federation did change the rules of the game so that the king should be in check only when being attacked by one or two opponent pieces. The cause of this new rule was that an opponent move cannot put the king under the attack of more than two pieces. However, given this rule, a friend move can put the friend king under the attack of three opponent pieces, and under the new rules it is not in check [Pitrat, 1998] ! The FIDE came back to the original rule, but it took four years to the world of Chess to find the error. It illustrates well how difficult it is to think about all the cases that may invalidate a rule.

Coming back to our problem, the task of finding the patterns of non transitivity is a tedious and error-prone task, and its result is not 100% reliable.

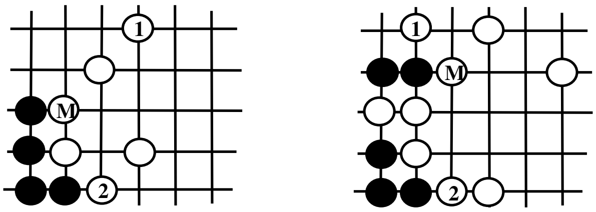


Figure 13. Playing useless moves.

A general and simple method to ensure transitivity would be to prove that ‘1’ and ‘2’ are connected. However, in many interesting cases, it takes too much time to prove. Figure 13 gives examples of bad moves that can be played due to the limitations of the connection theorem prover. In the first diagram, Black can play a bad move if he thinks he can disconnect ‘1’ and ‘2’. In the second diagram, White can add an unnecessary move to connect ‘1’ and ‘2’ even when they are connected. Therefore, a fast approach to verifying transitivity is needed.

A *threat move* is a move that works when followed by another move of the same color. In order to find potential problems in transitivity, we compute the threat moves to disconnect ‘1’ and ‘M’ if two black moves in a row are played, as well as the threat moves to disconnect ‘2’ and ‘M’. The moves that we consider as potential transitivity breaker are the moves that are both threat moves to disconnect ‘1’ and ‘M’ and threat moves to disconnect ‘2’ and ‘M’. If a move disconnect ‘1’ and ‘2’, and if it is also a potential transitivity breaker, ‘1’ and ‘2’ are considered as not connected. If none of the moves to disconnect ‘1’ and ‘2’ are transitivity breakers, and if ‘1’ is connected to ‘M’ and ‘M’ is connected to ‘2’, the connection is considered transitive.

This approach based on search is more general and easier to program than an approach based on patterns. The pattern based approach may be faster. However, it is possible to transform automatically the search based approach into a pattern based approach using program transformation [Cazenave, 1996, 1998] and pattern learning techniques [Cazenave, 1996, 2001b]. A more reliable approach would be to verifiy transitivity with a search mechanism taking into account the double connection. However, it would be slower.

5 Search

In this section, we give search algorithms that can reduce significantly the amount of time needed to solve Go problems. When it is possible, we try to give mathematical definitions of the objects used in the algorithms. In the last two sub-sections, we outline some promising new algorithms.

5.1 Improvements related to Alpha-Beta

The basic search algorithm to solve Go problems is Alpha-Beta. There are many improvements related to Alpha-Beta, we use iterative deepening, transposition tables, quiescence search, null-window search and the history heuristic. We stop search early when the goal is reached. In the experiments of this paper, we stop search after the first winning move.

On simple problems [Kano, 1987], with a Pentium III 600 MHz microprocessor, an Alpha-Beta with all these improvements, that tries all possible moves at each node solves 24.00% of the problems in 5.87s, when the maximum amount of search time per problem is set to 100 ms.

5.2 Abstract Proof Search

The Abstract Proof Search algorithm is based on the definitions of games given in section 2.2. An Abstract Proof Search of order n is composed only of moves associated to ip_k games, with $k \leq n$.

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 verifies if the position is gi_1, if it is not, it verifies if it is gi_2 (equivalent to a three plies deep search tree), and so on until the gi_n game. As soon as one of the $gi_{k,\, k \leq n}$ games is verified, the program stops searching and returns Won. Otherwise, if none of the $gi_{k,\, k \leq n}$ functions is verified, it tries the OR node moves associated to the position. At AND nodes, the same thing is done for ip_1, ip_2 … ip_n games, if none of them is verified, the programs returns Lost. Otherwise, it tries the moves associated to the first verified $ip_{k,\, k \leq n}$ game (it is useless to verify the $ip_{m,\, m>k}$ games when the ip_k game is verified because the ip functions find the complete sets of forced moves, so no move other than those returned by the ip_k game is needed).

When verifying a g_k game, the ip_m games are tested in ascending order (beginning with ip_1, then if ip_1 is not verified, testing ip_2 etc…). As soon as an ip_m game is verified, the remaining ip games are useless to test. Because an ip game provides a complete set of forced moves. Therefore, if all these moves are tested, it is useless to look for other sets of moves associated to the remaining ip games (as the final set of forced moves is an intersections of all the sets of forced moves).

Abstract game knowledge for selecting relevant moves is well adapted to Abstract Proof Search because, the size of the small trees developed when verifying game definition functions is known. Therefore, abstract knowledge enables a high selectivity.

On simple problems [Kano, 1987], with a Pentium III 600 MHz microprocessor, Abstract Proof Search using ip_1, ip_2 and ip_3 games solves 65.33% of the problems in 3.00s, when the maximum amount of search time per problem is set to 100 ms.

5.3 Iterative Widening

The iterative widening algorithm is an extension of the Abstract Proof Search algorithm.

Iterative Widening is based on sets of abstract possible moves, that can be tried at the node of the search tree at a given widening threshold. Sets are numbered, the following set always contains the previous set. The algorithm tries the sets of moves in the same order as their numbers. For example, if the sets of possible moves to be tried at different widening threshold are the sets S_1, S_2,…,S_n. We have $S_1 \subset S_2 \subset \ldots \subset S_n$.

The algorithm begins with an Abstract Proof Search, trying the moves in the set S_1. If this search fails, it then makes another search with the S_2 set. And so on until all the possible sets have failed, or the allotted time has elapsed.

For each problem, two search trees are usually expanded. The first one with White playing first and the second one with Black playing first. However, we discard the search with Black trying to prevent the goal, if the search with White playing first does not find a winning move. Similarly, if the problem consists only in finding a winning move, the search to find the preventing move is not performed.

The iterative widening algorithm consists in calling first the iterative deepening search algorithm with the first move function that returns the moves of the first set. If the search does not succeed, it continues with the following sets until the search succeeds or the time has elapsed, or the search eventually fails with the ultimate set.

When a search fails at a given widening threshold, the transposition table keeps the information related to the search for the next widening steps.

Experimental results show that a good and general, widening strategy is to put in S_1 all the OR node moves, and only the ip_1 and ip_2 moves at AND nodes. Then to put in S_n the all the OR node moves, and all the ip_1, ip_2 … ip_n moves at AND nodes.

On simple problems [Kano, 1987], with a Pentium III 600 MHz microprocessor, Iterative Widening with S_1 and S_2 solves 80.00% of the problems in 1.74s, when the maximum amount of search time per problem is set to 100 ms.

5.4 Lambda Search

Lambda Search was designed by T. Thomsen [Thomsen, 2000]. λ search can be defined as follows:

λ^n-tree = Search tree consisting only of λ^n-moves.

λ^n-move = If the *attacker* is to move, it is a move that implies – if the defender passes - that there exists at least one subsequent λ^i-tree with value 1 (Won), $0 \leq i \leq n-1$. If the *defender* is to move, it is a move that implies that there does not exist any subsequent λ^i-tree with value 1, $0 \leq i \leq n-1$.

Lambda Search can solve difficult problems, but it can very quickly exhaust time constraints, as it each move in a λ^n-tree is found by a λ^{n-1}-tree. Relevancy zones are well adapted to Lambda Search as the size of a lambda tree can vary a lot, and cannot be known in advance on the contrary of Abstract Proof Search.

5.5 Gradual Proof Search

In order to efficiently prove theorems on connections, we have designed a new algorithm, which can be seen as being able to solve very complex problems in the same fashion as Lambda Search [Thomsen, 2000], and also having a more gradual increase in complexity and a more fine-grained control such as Abstract Proof Search [Cazenave, 2000].

We can define trees and moves:

γ^1_m-tree = Search tree consisting only of ip_j moves, $0 \leq j \leq m$.

γ^k_m-tree = Search tree consisting only of γ^k_m moves.

γ^k_m move = If the *attacker* is to move, it is a move that implies – if the defender passes - that there exists at least one subsequent γ^i_j-tree with value 1, $0 \leq i \leq k-1$, $0 \leq j \leq m$. If the *defender* is to move, it is a move that implies that there does not exist any subsequent γ^i_j-tree with value 1, $0 \leq i \leq k-1$, $0 \leq j \leq m$.

$\gamma^k_{m,n}$-tree = Search tree consisting only of γ^k_m and ip_j moves, $0 \leq j \leq n$.

A $\gamma^1_{1,1}$-move is a forced move that has been discovered by performing a tree search containing only ip_1 moves. For example, a ladder is proved to work for White, and all the black moves in the trace of the ladder are the forced moves associated to the ladder, that might invalidate it. The invalidating moves of a ladder are γ^1_1-moves. We can make a parallel with lambda search, as we state that $\gamma^n_{1,1}=\lambda^n$ and that $\gamma^1_{n,1}$=Abstract Proof Search of order n . However, moves associated to $\gamma^n_{2,1}$ traces are different from Lambda Search, and from Abstract Proof Search. $\gamma^k_{m,n}$ is more general than Abstract Proof Search and it is more gradual than Lambda Search: the size of the number of moves grows slower. For the connection game, it may be more adapted than Abstract Proof Search, as it enables to solve more complex problems such as the problem of figure 3, and also more adapted than Lambda Search as it permits more control on the size of the search trees. In order to prove that Black can connect in the problem of figure 3, only ip_1 and ip_2 moves are needed, therefore a $\gamma^0_{1,2}$ search is enough. Whereas, proving that the two black stones are connected when White plays first is more difficult. Moves issuing from ladder searches have to be used, and ip_3 moves are also needed, therefore a $\gamma^1_{2,1}$ search is adapted (it consists of forced moves issuing form search trees based on ip_1 and ip_2 games definitions functions).

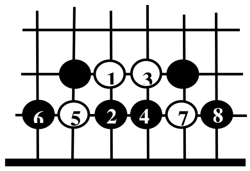


Figure 14. White to play a $\gamma^{1}_{2,1}$ move.

The drawback of Gradual Proof search may be that it is less suited to very fast problem solving. Abstract Proof Search, may be better for real-time game programs, whereas Gradual Proof Search may be more suited for off-line problem and game solving. An open problem is the order in which the gradual proof search trees should be developed (is it better to search $\gamma^{2}_{2,3}$ before or after $\gamma^{3}_{1,3}$?) and its relation to Iterative Widening. For example the forced white move in figure 14 can be found with a $\gamma^{1}_{1,6}$ move, with a $\gamma^{1}_{2,1}$ move, or with a $\gamma^{2}_{1,1}$ move

This algorithm still requires some experiments to verify its interest.

5.6 Gradual Abstract Proof Search

Similarly we could extend Abstract Proof Search by defining an $ip_{m,n}$ game where the maximum number of White moves is m as in the ip_m game, but where only ip_j games $1 \leq j \leq n$, are used to verify it. We would then have:

$gi_{k,n}$ (P,W) = $\exists$ White move { P_1=play(P,move), $g_{k-1,n}(P_1,W)$ }

$ip_{k,n}$ (P,B) = $gi_{k,n}$ (P,W).

$S_{k,n}$ (P,B) is the set of all black moves that prevent $gi_{k,n}$ (P,W) to be verified after a move if B plays first when ip_k (P,B) is verified.

$g_{k,n}$ (P,W) = $\exists$ m { m<=n, ip_n (P,B), $\forall$ move $\in$ S_m (P,B) { P_1=play(P,move), $\exists$ o {o<=k, $gi_{o,n}(P_1,W)$ }}}.

This algorithm too requires some experiments to verify its interest. It can also be mixed with Gradual Proof Search to make it even more gradual.

6 Future Work

It is possible to improve our analysis of non-transitivity. We could generate automatically patterns or programs to speed it up, we could also find more selective knowledge to reduce the number of moves to look at when combining the double goal search.

From a more general point of view, we have implemented and tested some knowledge to solve double goals. Some more work is needed here in our program as well as the construction of a problem library, and the evaluation of the performance of different algorithm on this problem library. For example, such double goals as Connect or Live are not well handled by our program.

Another path of research is to find methods to minimize the traces of the searches. It would enable to reduce the size of the relevancy zones and to improve Gradual Proof Search. Some combination of relevancy zones and abstract knowledge is probably interesting. These methods have some strong links with partition search.

An interesting application of Abstract Proof Search and its variants is life and death. The combination of Abstract Proof Search and pattern generation [Cazenave, 2001b] will probably give good results.

Gradual Proof Search could help solve 5x5 Go problems, and possibly solve 5x5 Go. Some experiments are also needed to test the orders for expanding the gradual games definitions. Other games such as Ponnuki Go (or AtariGo, i.e. the first to capture wins), Connecticut (connect the two opposite side of a Go board) will also benefit of progress in the search algorithms for computer Go.

In order to improve the general level of Go programs, and to enable more communication between programmers without giving away well-hidden secrets, the construction of an open library of problems for computer Go would also be an advancement.

7 Conclusion

The tools and algorithms we have presented in this paper can be useful to create programs to teach Go to people that do not know the rules. For example, writing good programs to play games related to Go but more simple such as connect the two sides of a small board, or be the first to capture an opponent stone.

They can also be used to solve Go for small sizes (4x4 or 5x5), generating and solving difficult small board problems that can be challenging even for good players [Hukui, 2000]. A problem solver can be used to play the problems against the computer or to discover automatically new and interesting problems.

Finally, they can improve programs that excel in a subpart of the game such as life and death [Wolf, 1994, 1996, 2000] [Cazenave, 2001b], endgame [Berlekamp & Wolfe, 1994], connection or capture and that are able to automatically create difficult and interesting problems for these sub-games.

References

Berlekamp E., Wolfe D.: *Mathematical Go Endgames, Nightmares for the Professional Go Player*. Ishi press international, San Jose, London, Tokyo, 1994.

Cazenave T.: *Système d'Apprentissage par Auto-Observation. Application au Jeu de Go*. Ph.D. diss., Université Paris 6. 1996.

Cazenave T.: *Metaprogramming Forced Moves*. Proceedings ECAI98 pp. 645-649, Brigthon, 1998.

Cazenave T.: *Abstract Proof Search*. Computers and Games 2000. To appear in LNCS.

Cazenave T.: *Iterative Widening*. Proceedings of IJCAI-01, Seattle, 2001.

Cazenave T.: *Generation of Patterns with External Conditions for the Game of Go*. To appear in Advance in Computer Games 9, 2001.

Conway J., Berlekamp E., Guy R.: *Winning ways*, Tome 1 and 2, Academic Press, 1982.

Fotland D.: Personal Communication, Seoul March 2001.

Hukui M.: 5x5 Go. Masaaki Hukui 2000. In japanese, Tokyo, 2000. ISBN 4-8182-0478-1.

Kano Y.: Graded Go Problems For Beginners. Volume Three. The Nihon Ki-in. 1987.

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.

Pitrat J.: Games: The Next Challenge. in: ICGA Journal, Vol. 21, N° 3, September 1998, pp. 157-156. 1998.

Thomsen T.: *Lambda-search in game trees – with application to go.* Computers and Games 2000. To appear in LNCS.

Wedd N.: *Goemate wins Go tournament*, in: ICGA Journal, Vol. 23, N° 3, September 2000, pp. 175-177. 2000.

Wolf T.: *The program GoTools and its computer-generated tsume-go database*, in: Proceedings of the First Game Programming Workshop in Japan, Hakone, 1994.

Wolf T.: *About problems in generalizing a tsumego program to open positions*, in: Proceedings of the 3rd Game Programming Workshop in Japan'96, Hakone, pp. 20-26, 1996.

Wolf T.: *Forward pruning and other heuristic search techniques in tsume go*, Information Sciences, 122, pp. 59-76, 2000.