Iterative Widening

Tristan Cazenave
Labo IA, Université Paris 8
2 rue de la Liberté, 93526 Saint Denis, France.
cazenave@ai.univ-paris8.fr

Abstract

We propose a method to gradually expand the moves to consider at the nodes of game search trees. The algorithm is an extension of Abstract Proof Search, an algorithm that solves more problem than basic Alpha-Beta search in less time and which is more reliable. Unlike other related algorithms, iterative widening adapts to the game via general game definition functions. In the game of Go, it can solve more problems than the original non widening algorithm in approximately half of the time, as shown by experimental results.

1 Introduction

We propose a method to gradually expand the moves to consider at the nodes of game search trees. The algorithm begins with an iterative deepening search using the minimal set of moves, and if the search does not succeed, iteratively widens the set of possible moves, performing a complete iterative deepening search after each widening. Iterative widening enables to reduce by almost a factor two the time used for solving capture problems in the game of Go and solves more problems than the original non-widening algorithm, as shown by experimental results.

The second section describes the search algorithm and compares it with related existing algorithms. The third section explains the game used to perform our experiments. The fourth section deals with theorem proving in Go. The fifth section gives hints on how to define and combine the gradually expanding sets of moves and defines some of these sets for the capture game in the game of Go. The sixth section details experimental results.

2 The Search Algorithm

2.1 The Basic Search Algorithm

In this subsection, we describe the initial non-widening algorithm.

We use Abstract Proof Search [Cazenave, 2000] to develop AND/OR proof trees for the game of Go. This is an iterative deepening Null Window Search [Marsland and Björnsson, 2000], that uses some game specific functions to efficiently prove theorems about goals in games. This search algorithm is much more efficient than a usual alpha-beta search, and scales well when given more resources.

We do not use forward pruning with null move search because we are looking for exact results, some of our experiments show that null move pruning can speed-up the algorithm with very little drawbacks. However, Abstract Proof Search often stops searching when alpha-beta continues searching: it stops searching at AND nodes when it cannot prove the goal can be reached in less than four plies after a null move. When the goal can be reached in four plies or less, it selects the only relevant moves that can prevent reaching the goal with another small five plies search. So Abstract Proof Search can be considered as performing a kind of forward pruning [Smith and Nau 1994] that improves the quality and the rapidity of the search instead of making it worse as with usual forward pruning.

We use iterative deepening [Korf, 1990], transposition tables, quiescence search, null-window search when not at the root and the history heuristic [Schaeffer, 1989]. We stop search early when the goal is reached. In the experiments of this paper, we stop search after the first winning move.

In our tests on the capture game, 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.

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 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.

In our experiments in Go, 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

[1] atari means only one liberty left

atari. This ensures that the Quiescence search returns correct results on the capture status of the string and quickly reads simple and deep ladders.

Iterative deepening stops when a winning move is found or as soon as the maximum processing time has elapsed.

2.2 Iterative Widening

We now define how we have applied iterative widening to our search algorithm.

We define 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 S1, S2,…,Sn. We have $S1 \subset S2 \subset \ldots \subset Sn$. The algorithm begins with an Abstract Proof Search, trying the moves in the set S1. If this search fails, it then makes another search with the S2 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.

In our first experiments, when a search fails at a given widening threshold, the transposition table is reinitialized and a new search is performed with the next set if possible. Further experiments have shown that this can be improved when reusing the same transposition table for all the widening steps.

2.3 Related Work

After having designed the method, we found that it has links with Iterative Broadening [Ginsberg and Harvey, 1992]. This method is successful in constraint satisfaction search [Meseguer and Walsh, 1998]. However, Iterative broadening is not the same algorithm as ours because it sets an artificial breadth cutoff *c*, and backtracks at most *c* times at any node of the tree. It iteratively increases *c*, and information can be memorized for the next iteration. Experiments by Ginsberg and Harvey in applying Iterative Broadening to Chess gave disappointing results because the move ordering of current Chess program is already near the optimum. Another previous related approach was phased state space search [Marsland and Srimani, 1986], an hybrid algorithm between SSS* and Alpha-Beta that partitions the set of all immediate successors of MAX nodes into k groups, and limits the search to one partition per phase. The originality of our method is that it is more concerned with widening at AND nodes, and that it provides game independent games definition functions that enable large speed-ups, and adapt to the game and the position at hand to select the worthwhile moves, rather than setting an artificial breadth cut-off.

3 The Capture Game

In our experiments, we mainly used the capture game to test the algorithm. The capture game is the most fundamental sub-game of Go. It is usually associated with 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 solved 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.

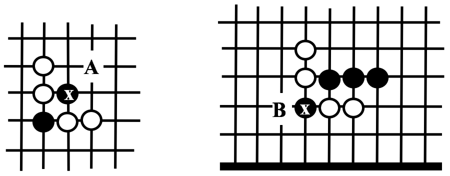

Figure 1. Examples of captures

4 Theorem Proving in Go

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.

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. 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:
$\exists$ 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 white moves in position P if B plays first when ip_k (P,B) is verified.

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

In some previous research [Cazenave, 1998], we have shown it is possible to generate programs for the game definition functions using the rules of the game defined in a logic language, and 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 metaprogramming 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 black stone has only one liberty and can be captured in one white move at A, i.e. in a 1 ply search. The only abstract black moves to prevent the capture are the liberty of the stone and the liberties of its adjacent strings in atari. Based on the logic of our previous system, we have designed complete sets of abstract moves that can prevent one, three and five plies search.

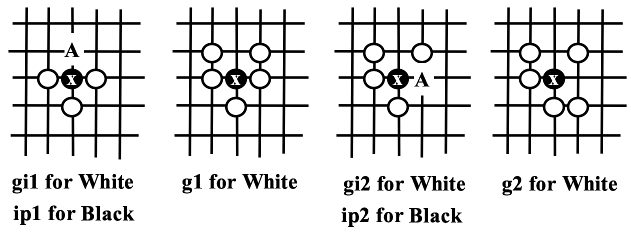

Figure 2. Examples of games

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 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 ipn moves are the moves that prevent a string to be captured in n moves by the opponent. For example, the ip1 moves are the moves that may prevent a string in atari to be captured in one move (i. e. playing the liberty, or capturing an adjacent string).

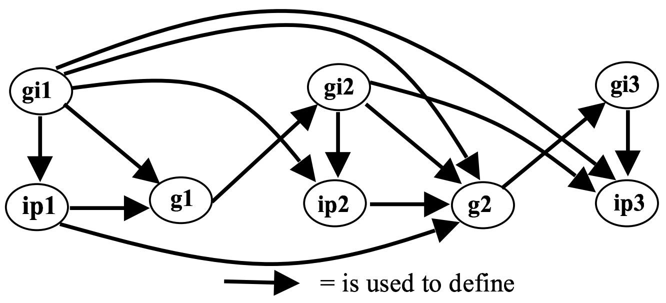

Figure 3. The dependencies between games

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 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 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, for example the move at A in the third diagram of figure 2 is an ip2 move for Black and a gi2 move for White).

Figure 3 gives the dependencies between games definitions. 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 (as in the second diagram of figure 2). So the g1 game is defined using the definitions of the gi1 and of the ip1 games, as it is shown in figure 3 where arrows go from the gi1 and ip1 games to the g1 game. 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. In order to make things clear some examples of games are given in the figure 2.

The gi2 game relies on the g1 game as shown by the arrow between g1 and gi2 in the figure 3. A gi2 game can be tried if the string to capture has two liberties. For each of the two liberties, the program tries to fill the liberty, and verifies that the game is g1 after the liberty is filled, using the g1 game definition function.

The function defining the ip2 game and its associated moves is equivalent to finding 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.

The ip2 game definition function is defined using simple concepts and the functions corresponding to other games. Here again, as shown in the figure 3, the ip2 function relies on the functions defining the gi1 and gi2 games. The function adds the forced moves to prevent a 3 plies capture (the ip2 moves). The function begins with verifying that the string can be captured in two moves if White plays first, by calling the gi2 game definition function. If it is the case, the function finds the complete set of black moves that may change the issue of the gi2 game. Then, for each move of this set, it plays it and checks whether 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.

The g2 game definition is a little more complex than the previous ones because there are two possibilities:

Either the function ip1 is verified, the black string can be captured in one move by White, so it has only one liberty. After playing on its liberty the string can still be captured in two white moves (the gi2 function applies).

Or the function ip2 is verified, but all the moves that could prevent the game to be gi2 do not work, so the ip2 function returns an empty set of forced moves. In that case, the game is won for White because none of the black moves to prevent gi2 works. The rightmost diagram of figure 2 is an example of this kind of g2 game.

The gi3 and ip3 game definition functions use the same kind of definitions as the gi2 and ip2 functions.

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 three plies deep search tree), and if it is not, it checks if it is gi3 (equivalent to a five plies deep search tree). As soon as one of the gi games is recognized, the program stops searching and returns 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 returns Lost, otherwise it tries the moves associated to the verified ip1, ip2 or ip3 game.

5 Designing the Gradual Sets of Moves

It is quite important to carefully choose the sets of moves. The first set is better if it contains the moves that are likely to reach the goal. Typically, the last set contains all the moves worth trying. We have separated the sets for the OR nodes and the AND nodes of the tree, as they have completely different properties.

We have defined two sets of moves at OR nodes: OR1 is constituted by the liberties of the string to capture only. OR2 is constituted by all the moves worth trying, including the liberties of the string to capture, the liberties of the liberties of the string to capture and the liberties of the adjacent strings that have less liberties than the string to capture.

Similarly, we have defined two sets of moves at AND nodes : AND1 is constituted by the ip1 and ip2 moves, AND2 is constituted by the ip1, ip2 and ip3 moves.

There are different orders in which the widening can be performed. Each order is called a widening strategy, each widening strategy has a number and a name:

1. OR2-2AND2-2: This is the original non-widening, iterative deepening Null Window Search algorithm. The OR2 set of moves is used at OR nodes, and the AND2 set of moves is used at AND nodes.
2. OR1-2AND2-2: The algorithm begins with the OR1 and AND2 sets of moves, and if the search fails, it searches again with the OR2 and AND2 sets of moves.
3. OR2-2AND1-2: The algorithm begins with the OR2 and AND1 sets of moves, and if the search fails, it searches again with the OR2 and AND2 sets of moves.
4. AND1-2OR1-2: The algorithm begins with the OR1 and AND1 sets of moves, and if the search fails, it searches again with the OR1 and AND2 sets of moves. If the search fails again, it searches again with the OR2 and AND2 sets of moves.
5. OR1-2AND1-2: The algorithm begins with the OR1 and AND1 sets of moves, and if the search fails, it searches again with the OR2 and AND1 sets of moves. If the search fails again, it searches again with the OR2 and AND2 sets of moves.
6. ORAND1-2: The algorithm begins with the OR1 and AND1 sets of moves, and if the search fails, it searches again with the OR2 and AND2 sets of moves.
7. OR1-2ANDOR1-2: The algorithm begins with the OR1 and AND1 sets of moves, and if the search fails, it searches again with the OR2 and AND1 sets of moves. If the search fails again, it searches again with the OR1 and AND2 sets of moves. If the search fails again, it eventually searches with the OR2 and AND2 sets of moves.
8. ORAND1-2AND1-2: The algorithm begins with the OR1 and AND1 sets of moves, and if the search fails, it searches again with the OR1 and AND2 sets of moves. If the search fails again, it searches again with the OR2 and AND1 sets of moves. If the search fails again, it eventually searches with the OR2 and AND2 sets of moves.
9. Brute Force : The algorithm always tries all the possible moves. It is intended to compare our selective search algorithm based on game definition with a brute force approach. Note that our quiescence search is responsible for most of the problems solved by the brute force approach.

6 Experimental Results

This section gives experimental results on a standard test set for capturing strings in Go: we call them ggv1 [Kano, 1985a], ggv2 [Kano, 1985b] and ggv3 [Kano, 1987]. These books are regarded by Go players as a well balanced collection of problems that cover the essential problems that arise in real games. The first book contains very simple beginner's problems, the second book requires more knowledge of the game, and the third book contains problems that average players can find interesting. We have selected all the problems involving a capture of a string, including semeai and some connection problems. There are 114 capture problems in ggv1, 144 in ggv2 and 75 in ggv3. Experiments were performed on a Pentium III 600 MHz microprocessor.

Problems are counted as solved only when the search returns Won (in some case it may be useful to consider the preventing moves associated to the Unknown value as they are not refuted due to a lack of search, but may be preventing moves, however the brute force algorithm returns Unknown for many bad moves, therefore they should not be considered as correct answers).

A maximum processing time is set for each search, as soon as the time has elapsed, the search is stopped, and the status is set to 0 (Unknown) on remaining leaves. Two searches are performed for each problem, one with Black playing first, the other with White playing first.

For each book and each maximum processing time per search, a table gives the widening strategy used, the time in seconds used to search all the problems, the number of nodes including leaf nodes, the minimum number of nodes (when the best move is always tried first at each node), and the percentage of solved problems.

	Time	Nodes	Minimum	Solved
1	0.63s	4404	4017	99.12%
2	0.54s	2123	1857	99.12%
3	0.45s	4268	3844	99.12%
4	0.65s	2484	2299	99.12%
5	0.70s	2494	2279	99.12%
6	0.48s	2036	1848	99.12%
7	0.80s	2430	2257	99.12%
8	0.79s	2582	2404	99.12%
9	26.40	890256	761161	78.07%

Table 1. Results for ggv1, Search Time < 1 second.

	Time	Nodes	Minimum	Solved
1	11.39s	50968	41751	88.19%
2	9.68s	36619	29909	89.58%
3	7.44s	49304	41479	89.58%
4	10.25s	38476	32132	88.89%
5	8.52s	45365	37788	89.58%
6	7.84s	32102	26588	89.58%
7	9.19s	46859	38785	88.89%
8	10.40s	46540	38244	88.19%
9	102.09s	4533217	4275396	30.56%

Table 2. Results for ggv2, Search Time < 1 second.

	Time	Nodes	Minimum	Solved
1	11.86s	52173	42140	80.00%
2	8.73s	32837	27412	84.00%
3	6.38s	37953	32335	84.00%
4	9.11s	33860	28402	84.00%
5	6.79s	34706	29401	84.00%
6	7.38s	30783	24931	84.00%
7	7.39s	34140	30159	84.00%
8	8.28s	37723	33325	84.00%
9	53.28s	2259720	2140103	29.33%

Table 3. Results for ggv3, Search Time < 1 second.

	Time	Nodes	Minimum	Solved
1	4.40s	22782	20634	75.00%
2	3.63s	13297	11745	83.33%
3	2.92s	25107	22355	85.42%
4	3.75s	12834	11834	84.03%
5	3.41s	16183	14681	85.42%
6	3.02s	13234	11770	84.03%
7	3.40s	15988	14467	86.81%
8	3.58s	14190	12973	85.42%
9	11.20s	488576	465145	25.00%

Table 4. Results for ggv2, Search Time < 0.1 second.

We have also tested the algorithm with a lower maximum processing time, closer to current limitations of Go programs, the one second limit may be interesting for programs that spend more time on tactical analysis than on global search. It can also figure the possible improvements due to search in the near future as computers get faster. Each search was stopped as soon as it took more than 100 ms. The results for ggv1 are not given as they are similar to table one, except for the brute force algorithm that solves 74.56% of the problems in 3.21s and 142276 nodes. The results on more complex problems is different, as shown in tables four and five.

	Time	Nodes	Minimum	Solved
1	3.00s	13005	11474	65.33%
2	2.78s	10776	10050	69.33%
3	2.04s	18817	17207	78.67%
4	2.78s	11681	10862	69.33%
5	2.39s	14023	12909	77.33%
6	2.39s	9626	8858	73.33%
7	2.55s	13280	12328	74.67%
8	2.99s	13439	12730	69.33%
9	5.87s	313009	292004	24.00%

Table 5. Results for ggv3, Search Time < 0.1 second.

The brute force approach is clearly much worse than all other strategies. Almost all the problems it can solve are solved by our quiescence search. Considering than the problems in our test set are setup on small boards (9x9 or 13x13 boards), it would be even much worse in a full Go playing program (19x19 board).

Many widening strategies gives speed-ups compared to the original algorithm, except for some very simple problems of volume one where some widening strategies slightly increase the solving time which is more than compensated by more difficult problems. All the widening strategies both decrease the time to solve problems and increase the percentage of solved problems on average. They do not only reduce significantly the computation time, they also solve more problems than the original non iterative widening algorithm (OR2-2AND2-2).

In the original non widening algorithm, the liberties of the string are tried first, and the order of the moves at each node is the same as in the iterative widening algorithm, therefore the observed speed-ups are due to the iterative widening, not to another factor such as move ordering. Moreover, the number of nodes is close enough to the minimum number of nodes to consider that the move ordering is not bad. And even with the perfect move ordering, we can see that iterative widening is still clearly better than the non widening algorithm.

The OR2-2AND1-2 strategy is quite efficient and it is domain independent. There is no widening at OR nodes, and the widening at AND nodes is performed only by the selection of some specified game definitions functions. As game definition functions can be defined similarly for many games, this widening strategy is both effective and general.

We can also observe in table four and five that the OR2-2AND1-2 strategy searches more nodes that the non widening algorithm, but only takes two third of its time and solves significantly more problems. This apparent anomaly may be due to the cost of the ip3 game definition function, that is higher that the cost of the ip1 and ip2 functions. Therefore searching less nodes using the ip3 function can take more time than searching more nodes only using the ip2 function. In other experiments [Cazenave, 2000], we have already shown that the non widening algorithm based on the ip3 function solves more problems in much less time than the same selective Alpha-Beta Null-Window Search algorithm that uses the same set of moves, without performing the ip3 tests.

We performed tests in another game to assess the generality of iterative widening. We chose Phutball [Conway & al., 1982], as it also falls in the class of games that benefit from theorem proving and selectivity. Other games in this class are Gomoku, mate search in Chess and Hex for example. Generally these games have a high branching factor and a simple and well defined goal. Phutball is played on a Go board, a black stone represents the ball, and players are represented by white stones. A player tries to put the ball on the first line of its opponent. A move consists in moving the ball by jumping over a player, or in putting a new stone on an empty intersection. In Phutball, using iterative widening with the OR2-2AND1-2 combination enabled a speed-up by a factor greater than two.

In these experiments, the transposition table was completely initialized before each widening. It would be more clever to keep the same transposition table, and to put a flag on the transpositions, memorizing the widening step of the transposed board, in order to reuse the information from the previous and narrower search so as to save computation time. Another optimization is to reuse the stored score at OR nodes to update alpha. The results of the experiments that use the enhanced transposition tables, performing the two previously mentioned optimizations, are given in table 6. It appears that it enables to solve roughly 1% more problems in 5/6th of the time. The OR2-2AND1-2 (3rd) widening strategy is used.

Book	MaxTime	Time	Nodes	Minimum	Solved
ggv1	1s	0.30s	4469	4342	99.12%
ggv2	1s	7.48s	63266	58024	88.89%
ggv3	1s	5.10s	43166	38909	84.00%
ggv1	0.1s	0.30s	4469	4342	99.12%
ggv2	0.1s	2.41s	29038	27159	86.81%
ggv3	0.1s	1.74s	21822	20225	80.00%

Table 6. OR2-2AND1-2 with an enhanced transposition table

7 Conclusion

Gradually widening the sets of moves in some complex game search trees enables to reduce significantly the search time when performing an iterative deepening Null Window Search. It also appears that more problems can be solved by using this technique. A general widening strategy relevant to many complex games such as Go, Phutball, Hex and Chess has been experimentally proven useful. It consists in iteratively increasing the order of the game definitions functions used to select forced moves at the AND nodes of the search trees. Results are slightly better when reusing transposition table information from the previous and less wide search.

Acknowledgements

Thank you to J. Méhat and J.-P. Vesinet for proof reading.

References

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

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

[Cazenave, 2000] Cazenave T.: *Abstract Proof Search.* Proceedings of CG2000. Published in LNCS 2001.

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

[Ginsberg and Harvey, 1992] Ginsberg M. L., Harvey W. D.: *Iterative Broadening.* Artificial Intelligence 55 (2-3), pp. 367-383. 1992.

[Kano, 1985a] Kano Y.: *Graded Go Problems For Beginners. Volume One.* The Nihon Ki-in. 1985.

[Kano, 1985b] Kano Y.: *Graded Go Problems For Beginners. Volume Two.* The Nihon Ki-in. 1985.

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

[Korf, 1990] Korf R. : *Depth-first iterative-deepening : An optimal admissible tree search*. Artificial Intelligence 27, N°1, pp 97-109, North-Holland 1990

[Marsland and Srimani, 1986] Marsland T. A., Srimani N.: *Phased State Space Search.* ACM/IEEE Fall Joint Computer Conference, Dallas, Nov. 1986, pp 514-518.

[Marsland and Björnsson, 2000] 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.

[Meseguer and Walsh, 1998] Meseguer P., Walsh T. : *Interleaved and Discrepancy Based Search.* Proceedings ECAI98 (ed. H. Prade). John Wiley & Sons Ltd., Chichester, England. ISBN 0-471-98431-0. 1998.

[Schaeffer, 1989] Schaeffer J.: *The History Heuristic and Alpha-Beta Search Enhancements in Practice.* IEEE Transactions on Pattern Analysis and Machine Intelligence 11, N° 11, pp. 1203-1212, 1989.

[Smith and Nau, 1994] S.J.J. Smith and D.S. Nau. *An Analysis of Forward Pruning.* In *AAAI-94*, 1994.