Search for transitive connections

Tristan Cazenave, Bernard Helmstetter

Labo IA, Université Paris 8
2 rue de la Liberté, 93526, St-Denis, France
fax: 33 1 49 40 64 10
e-mail: {cazenave,bh}@ai.univ-paris8.fr

1 Introduction

In this paper, we explore a way to reduce the complexity of a search on a double connection by searching both connections separately as much as possible. In the best case the two connections are independent, and a complex search that would cost $O((2p)^{2d})$ can be reduced to two searches of complexity $O(p^d)$, the variable p being the average number of possible moves in a connection game, and d the depth of the search for solving one of the connection game. In practice, the two connections are seldom perfectly independent. Even when they are not independent, it is useful to search the two connections separately as it helps finding sets of relevant moves.

Programs have weaknesses at finding non transitivities as can be seen in tournament games [1]. Some strong Go programs handle non transitivity using hand-coded patterns. For example, Many Faces of Go uses a few hundred such patterns. Since there are too many cases of non transitivity, this approach is limited.

The second section describes the problem of the non transitivity of connections. The third section outlines the adaptation of Generalized Threats Search to the connection game. The fourth section details the evaluation function and the selection of moves used in our transitive connection search algorithm. The fifth section gives experimental results. Eventually, the last section concludes and outlines future work.

2 Non-transitivity of connections

In this section we define the problem of the non transitivity of connections. Even the best computer Go programs have problems dealing with non transitivity. This problem is a special case of the more general problem of the dependence between two or more different sub-games.

Let A, B, C be three stones of the same color, C_{AB} the connection between A and B and C_{BC} the connection between B and C. We call max player the player who wants to connect and min player the player who wants to disconnect. In the starting position, the connections C_{AB} and C_{BC} must be won; otherwise the transitive connection would not be won a fortiori.

The easiest case is when the connections C_{AB} and C_{BC} are independent. Then the transitive connection is won. Figure 1 shows an example.

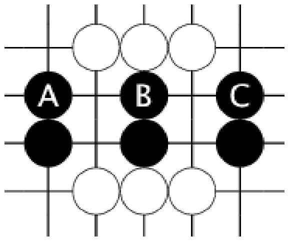


Fig. 1. Two independent connections

The interesting cases are when the connections are not independent, that is to say when there is at least a move by min player which threatens to break both connections. Then the transitive connection may or may not be won. Figure 2 shows an example where the transitive connection is lost if min player moves first: the move that disconnects is white $\triangle$. Now figure 3 shows an example of two connections that are not independent but which make a transitive connection nonetheless. The connections C_{AB} and C_{BC} are not independent because a white move at 1 threatens to break both; but then a black move at 2 would repair both connections.

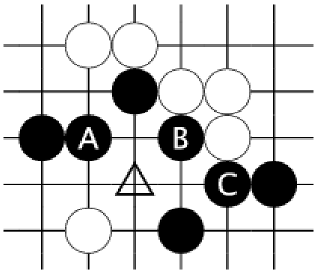


Fig. 2. Non transitive connections

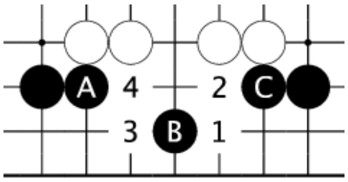


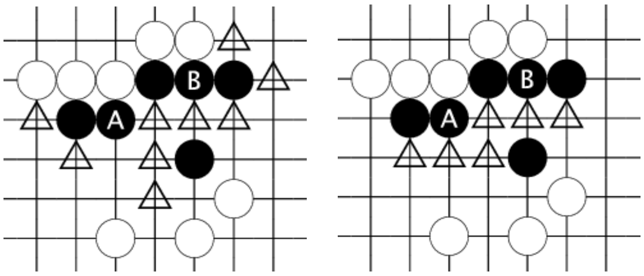
Fig. 3. transitive connections

3 Generalized threats connection search

In order to find transitive connections, we have to solve the problem of finding direct and single connections. This section is about the single connection game. We have used Generalized Threats Search [2] to solve the single connection game. The threat used for these experiments is the (8, 5, 2, 0) general threat. Moves in a generalized threats are associated to an order. The order of a position is the number of moves in a row by the same player that are required to win the game. Moves of order n are moves associated to positions of order less or equal to n. In the (8, 5, 2, 0) threat, eight is the number of order one moves allowed in the generalized threat, five the number of order two moves and two the number of order three moves. Only threats that have less moves for each order are verified at min nodes.

It is not mandatory to use Generalized Threat Search; however, whatever algorithm we use to find direct connections, it has to send back what we call a *trace*. Roughly speaking; the trace is a set of intersections that may change the result.

Precisely, the trace is a set of empty intersections such that if none is modified, the result of the search associated to the trace is not modified. In order to compute the trace of a connection, we have applied the following principle: for each test in the search, add to the trace the intersections that enable the test to be true. In most cases, we have to choose some intersections among many to be included in the trace. For example if the test is that the string has at least two liberties, any pair of liberties could be added. In practice, there are different possible traces that can be associated to a given search. In figure 4 we can see the difference between the trace found by our program for a connection, and a minimal trace obtained by hand.



(a) (b)

Fig. 4. Two possible traces for a connection

The evaluation function for the single connection returns:

- Lost if one of the two strings to connect is captured in a ladder,
- Lost if no max moves have been found, usually because the path between the two strings is too long and the common adjacent strings have too many liberties,
- Won if the two stones to connect are in the same string,
- Unknown otherwise.

We use specialized functions to find the max moves. The order of a threat is the number of moves in a row the max player has to play in order to win the game [2]. The functions called for finding the max moves depend on the order of the threat. We have different specialized and heuristic functions for finding possible moves that connect in one move, in two moves or in three moves. When there is a possibility for one of the two strings to be captured, the only max moves considered are the moves that save the threatened string. Concerning the min moves, the Generalized Threat Search algorithm uses the trace of the verified threat to find the relevant min moves.

4 Search for transitivity

We use an Alpha-Beta algorithm with transposition tables, two killer moves and the history heuristic. The game specific functions of our Alpha-Beta are: the evaluation function, and two functions *minMoves* and *maxMoves*, which return sets of relevant moves for the players min or max.

4.1 Evaluation function

The evaluation function searches the connections C_{AB} and C_{BC} in isolation using the Generalized Threats Search algorithm, and tries to deduce from this the status of the transitive connection. The situation is different depending on who is to play.

We consider first the case where max player is to play. The connections C_{AB} and C_{BC} are first searched with max player playing first, then with min player playing first. Besides the results, the searches also return the traces of all intersections on which the results depend. There are two cases where we can be sure of the status of the transitive connection:

- If C_{AB} or C_{BC} is lost, assuming max player plays first, then the transitive connection is lost.
- If one of the connections, say C_{AB}, is won (assuming min player plays first), if the other, C_{BC}, is winnable (*i.e.* it can be won if max player plays first), and if the traces on which those two results depend are disjoint, then the transitive connection is winnable. Indeed, in order to win it suffices for max player to play the winning move in connection C_{BC}.

We now consider the case where the player min (*i.e.* the player to disconnect) plays first. There are again two cases where we can be sure of the status of the transitive connection:

- If C_{AB} or C_{BC} is lost, assuming min player plays first, then the transitive connection is lost.
- If both connections C_{AB} and C_{BC} are won, assuming min player plays first, and if the traces on which those two results depend are disjoint, then the transitive connection is won.

4.2 Choose of min moves

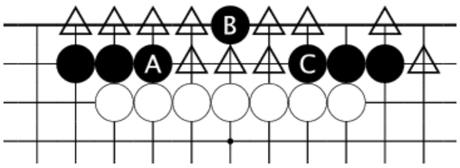


Fig. 5. Min moves

In case the evaluation function can't decide the status of the transitive connection we have to continue the main Alpha-Beta search which deals with the transitive connection as a whole. Hopefully the searches that have been made on the connections C_{AB} and C_{BC} can give valuable information to find a relevant set of moves.

We take as set of min moves (moves for min player) the moves that threaten to break either connection C_{AB} or C_{BC}. This is the union of the traces of the two searches that have shown that the connections C_{AB} and C_{BC} are won when min player plays first. An example of a set of relevant min moves found by our program is given in figure 5.

It is possible to use the intersection of the traces rather than the union. This is less safe as can be seen for instance in problem 13 of our test suite (figure 10), where the only move that disconnects may not be in the intersection. In practice, using the intersection of the traces solves more problems and takes less time as can be seen in the experimental results section.

In fact, even taking the union of the traces as the set of min moves is not perfectly safe. We have found that it is safe for the problems of our test suite, but we have built a transitivity problem (figure 6) where it misses a move. The move white 1 does not threaten either connection C_{AB} or C_{BC}, but it does break the transitive connection because black cannot defend against both white 2 and white 3. One can note that white 3 would have directly worked too, so even in this problem we would find at least one disconnecting move.

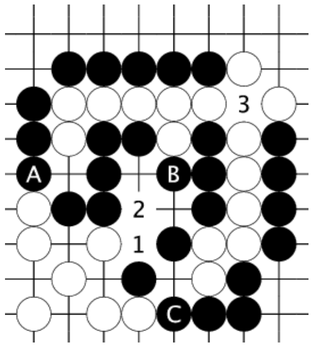


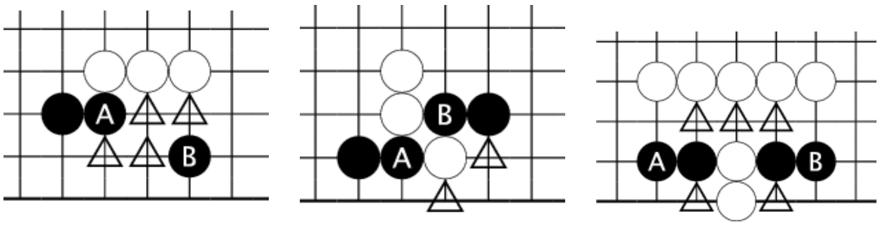
Fig. 6. Pathological position

4.3 Choose of max moves

In order to find a set of max moves, it is not suitable to use traces as for min moves; instead it is better to use the notion of *order*. The order of a connection is the number of moves in a row that are needed to join the two strings in the same string, if the opponent does not play [2]. Figure 7 (a) shows max moves to connect of order 2. There is a path of length 2 composed of empty intersections between strings A and B. Connecting may also involve capturing opponent strings adjacent to both strings A and B. Figure 7 (b) gives an example of the order 2 connection moves related to capturing a common adjacent string. Figure 7 (c) details the moves of order 3 to connect strings A and B.

Our algorithm to find order n moves between the strings A and B is shown in figure 8. In case this algorithm is applied to a connection of order less than n, it will not return all the moves of order n (which would be all the legal moves!), only those close to the connection, which is usually an advantage.

The set of max moves depends on the order at which we want to



(a) (b) (c)

Fig. 7. Moves of order 2 (a and b) and 3 (c)

$S \leftarrow \emptyset$
For each move m at a liberty of string A, or at a liberty of an opponent string adjacent to A that has at most n liberties and which is either adjacent to B, or that has a common liberty with B :
    Play(m)
    If the connection is of order $n-1$:
        $S \leftarrow S \cup \{m\} \cup \{$*moves of order* $n-1\}$
    Undo(m)
return S

Fig. 8. Algorithm to find order n moves between strings A and B

search the transitive connection. We have chosen to take as set of max moves the union of the sets of moves of order up to *ordermax* in each connection C_{AB} and C_{BC}. The variable *ordermax* equals at most the order of the maximum threat for the connection search plus one.

An example of a set of max moves found by our program is given in figure 9. In this case it is obviously far from perfect, because our program selects moves of order 3 although the two connections are of order 2.

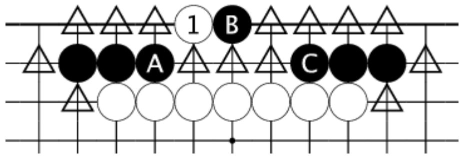


Fig. 9. Set of moves when Black is to play

5 Experimental results

In our experiments, the maximum threat used for the Generalized Threats connection search has been set to (8, 5, 2, 0). We only choose moves of order less or equal to three for the connections C_{AB} and C_{BC} in order to find the max moves in the transitive search. The maximum number of moves in each of the two Generalized Threats searches is limited to 100,000 unless stated otherwise.

We have built a test suite of 22 problems depicted in figure 10. The disconnecting moves, if any, are marked. Only problem 17 involves a ko. Two thirds of the problems are taken from Golois games, and one third are classical problems. The problems from Golois are taken from games where it failed to analyze transitivity correctly. This test suite is available for download on the first author's web page.

In order to compare transitive search with another algorithm, we have tested an optimized Alpha-Beta algorithm that uses Golois moves generators for connections and disconnections. The evaluation function of this Alpha-Beta returns Lost as soon as one of the two connections is Lost. A connection is Lost if the length of any path between the two strings is strictly greater than three. A connection is Won if the two stones to connect are in the same string.

In table 1, for each problem, the number of moves played in the search and the elapsed time used for the search are given. The experiments were run on a 3.0 GHz Pentium with 2 GB of RAM. The maximum number of nodes is set to 10,000,000 in the transitive search and in the Alpha-Beta. The left part of the table details the performance of the Alpha-Beta algorithm, and the right part the performance of the Transitive search algorithm.

Concerning the results of the Transitive search algorithm with union of the traces, problem 3 is not solved because in one of the forced lines a black cutting string gains 4 liberties and is therefore considered stable. In problem 17, our algorithm finds the good move but fails to see that it depends on a ko. In problem 20, the program fails

Table 1
Nodes and time for the transitivity problems.

	Alpha-Beta			Transitive Search (with union of the traces)		
Problem	$moves$	$time(ms)$	$solved$	$moves$	$time(ms)$	$solved$
1	61,188	190	yes	115,065	180	yes
2	348,950	1,390	yes	56,130	90	yes
3	85,582	330	no	397,880	560	no
4	147	10	yes	1,857	0	yes
5	2,687,471	6,850	yes	384,933	620	yes
6	8,655	40	yes	11,670	20	yes
7	234,806	1,910	yes	86,656	160	yes
8	43,537	180	yes	5,920	10	yes
9	72,800	320	yes	6,231	0	yes
10	101	0	yes	6,503	10	yes
11	8,307	100	yes	31,742	50	yes
12	88,152	400	yes	93,959	160	yes
13	10,000,108	39,510	no	1,906,627	3,330	no
14	21,266	40	yes	27,442	40	yes
15	198,569	610	yes	372,134	490	yes
16	10,000,256	33,300	no	10,437,826	14,320	no
17	10,000,183	29,530	no	783,545	1160	no
18	3605	10	yes	269,258	430	yes
19	3,828,385	16,360	yes	109,155	220	yes
20	22,871	50	yes	10,186,430	14,900	no
21	988	0	yes	6,436	260	yes
22	13,703	20	yes	18,846	40	yes
Total			18			17

because it does not verify that connected strings are not captured in a ladder, and therefore the trace does not contain the capturing move for the attacker. In problem 16, it fails because it is short of nodes in the main search, and in problem 13 it fails because it is short of nodes in one of the connection searches. The problem 13 involves two opponent strings that cannot be captured, this is why our algorithm fails to see the disconnection: for each move in the first capture search, it searches the other capture. In order to solve it faster, we should decompose it into two independent capture problems. This is not currently done by our connection algorithm.

Table 2 details the runs of the Transitive search algorithm using the intersection of the traces for min moves instead of the union as in table 1. The algorithm using the intersection is faster, especially for problem 16 that is solved relatively quickly compared to Alpha-Beta and Transitive Search that do not solve it because of a lack of nodes. In problems 5 and 8, connections are transitive. These two problems are representative of the usual problems that a transitive search algorithm has to solve (i.e. connections are usually transitive). If we compare the transitive search algorithm using intersection of the traces with the Alpha-Beta algorithm, we see that for these problems, the transitive search algorithm is much faster.

The transitive search can be tuned with two parameters: the maximum transitive search time and the maximum number of nodes allowed to each connection search. Connections searches are also named secondary searches. The Transitive search algorithm with intersection of the traces is used for the experiments. In order to choose the best algorithm for a given maximum response time, we have tested Transitive search for different secondary nodes and different maximum response time. Table 3 gives the number of problems solved depending on the two parameters. When choosing the appropriate number of secondary nodes for a given maximum time, we see that Transitive search is better than Alpha-Beta for response time inferior or equal to one second.

Some problems that human find relatively easy such as the double keima on the second line in problem 16 are difficult for our program, while some problems that humans find relatively difficult such as problem 21 are easy for our program.

Table 2
Transitive search with intersection of the traces.

Problem	$moves$	$time(ms)$	$solved$
1	104,025	150	yes
2	2,558	10	yes
3	359,609	490	no
4	1,502	0	yes
5	5,962	10	yes
6	10,463	10	yes
7	23,268	50	yes
8	1,230	0	yes
9	6,231	10	yes
10	5,471	0	yes
11	14,262	20	yes
12	93,959	130	yes
13	1,913,120	2,720	no
14	19,165	20	yes
15	194,050	250	yes
16	1,214,948	1,660	yes
17	382,636	490	no
18	67,642	90	yes
19	99,858	150	yes
20	145,989	250	no
21	4,298	10	yes
22	7,723	10	yes
Total			18

6 Conclusion

We have described an algorithm to detect non transitive connections in the game of Go. An optimized Alpha-Beta search is used on top of two Generalized Threats searches. Our program is able to solve problems such as the double monkey jump or the double keima on the second line. It deals with full board situations such as the ones

Table 3
Number of problems solved depending on max. time, algorithm and max. secondary nodes for Transitive Search

	Transitive Search					Alpha-Beta
$time(ms)$	1000	5000	10000	30000	100000	
10	7	6	9	9	8	4
30	7	10	10	11	11	5
100	7	13	14	13	13	9
300	7	13	15	15	17	11
1000	7	13	15	15	17	15

encountered in real games.

The program can be used with the intersection of the traces for choosing moves at min nodes. It then solves transitive problems much faster than Alpha-Beta and Transitive Search with the union of the traces, as can be seen for problems 2, 5, 8 and 16.

It can also be used in the safe mode, taking the union of the traces: even if the results are not theoretically perfect as can be seen on a pathological position, they are pretty reliable given the results on our test suite.

The program could be used in a Go program in fast mode, and with the intersection of the traces, to detect relatively simple non transitivities. The good point of the transitive search algorithm is that it can solve problems that cannot be solved by some of the strongest Go programs, the drawback is that it is much slower than a pattern based approach that only solves the common cases. The utility of this approach is dependent on the architecture of the program. Since the algorithm is still slow, it would be difficult to integrate it in a Go program based on global search, because the search for transitivities would have to be done at each call of the global evaluation, unless perhaps the results are cached. However, it could be used in programs that are not based on global search, and that spend more time on the evaluation of the position.

There is still room for improvements in solving more quickly prob-

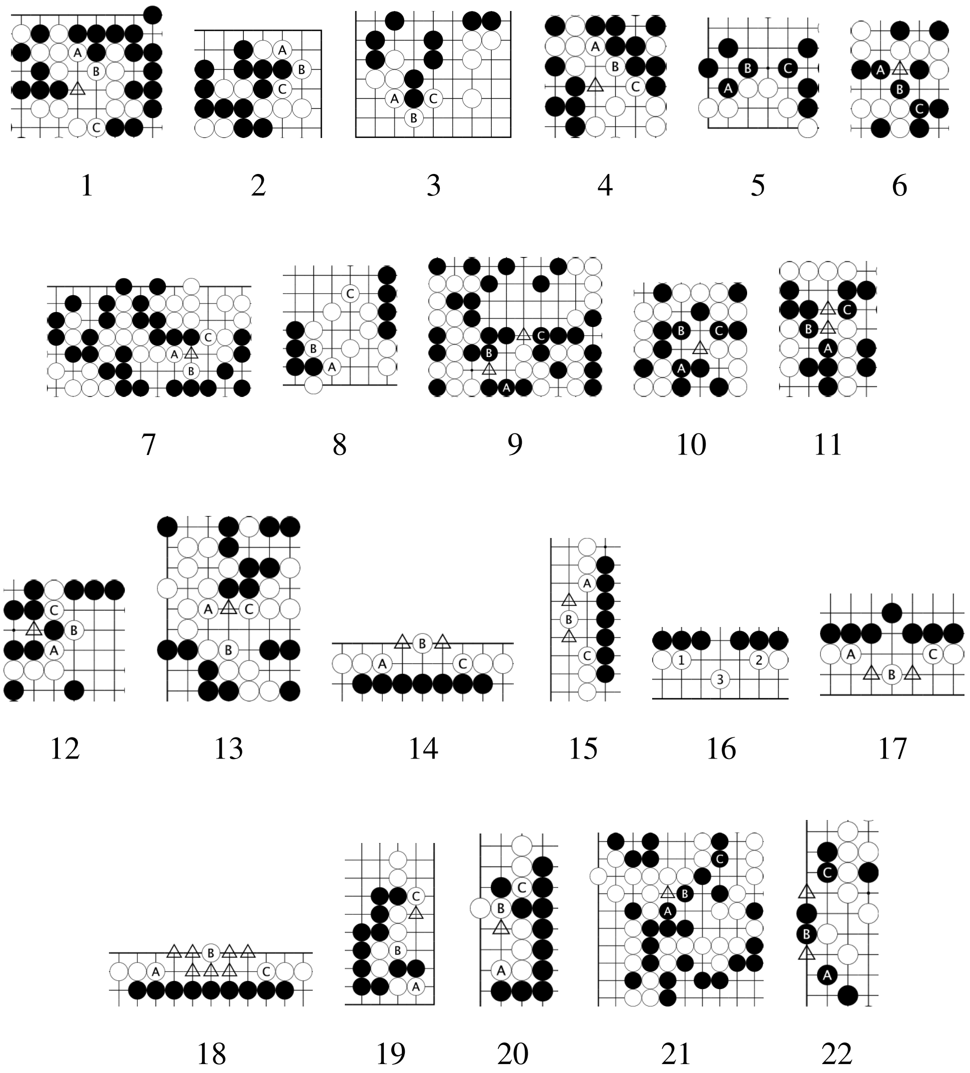


Fig. 10. Problems of the test suite

lems such as problem 13. It involves improving the connection search algorithm. Future work also includes extending it toward a more general search program for combinations of sub-games.

References

[1] N. Wedd, Goemate wins go tournament, ICGA Journal 23 (3) (2000) 175–178.

[2] T. Cazenave, A Generalized Threats Search Algorithm, in: Computers and Games 2002, Lecture Notes in Computer Science, Springer, Edmonton, Alberta, Canada, 2002, pp. 75–87.