A Search Algorithm for Finding Multi Purpose Moves in Sub Problems of Go

Kazuki Yoshizoe†

Because of the difficulty in evaluating the board positions, the successes in chess variants does not directly strengthen Go playing programs. It is believed that most Go playing programs use subgoal oriented search to decide moves. Subgoals are local targets with well defined evaluation functions, such as capturing, connection and living. Currently, the overall strength of Go playing programs are weak. On the other hand, current state-of-the-art algorithms for solving life-and-death problems overtook professional Go players in terms of speed and accuracy.

To improve Go playing programs, it is necessary to fill in the gap between the strength of single subgoal searches and strength of overall play. It is critical to solve the dependencies between subgoals. For this purpose, it is necessary to maintain the information of each subgoal search. The information required includes not only the search result but also the possible *inversions* of the subgoals. The intersections of *inversions* are the candidates for multi purpose moves.

In this paper we focus on presenting a new algorithm for searching *inversions*. Target problems are capturing problems of Go. We have tested the algorithm on test problems and measured the number of nodes searched.

1. Introduction

Compared to the programs of chess like games, Go playing programs are weak. One of the reasons is that in the game of Go, existing evaluation functions are inaccurate and/or requires great computational effort.

Therefore, many Go playing programs use subgoal oriented search. A subgoal is a local target which is easy to evaluate, such as connecting stones, capturing stones, life and death problems, etc.

There are many sophisticated algorithms for single subgoal search. For example, life-and-death problem solvers such as GoTools[5] and Df-pn based Tsumego Solver[3] are very strong. Their speed and accuracy had overcome the level of professional players. But overall strength of Go playing programs remain at the level of novice players. (It is widely believed that the strongest Go playing program is about 10 kyu in AGA☆ rating.)

To improve Go playing programs, it is necessary to fill in the gap between the strength of subgoal searches and the strength of overall play. One critical element for this purpose is to resolve the dependencies between different subgoals. Without this ability, it is difficult to find multipurpose moves, and also difficult to evaluate such moves exactly. For example, it is difficult to find moves which aims to connect or to live locally at the same time.

The paper written by Cazenave and Helmstetter[1] and another paper by Jan Ramon and Croonenborghs[6] are two examples of searching double purpose moves using the idea of *trace*[1] or *relevancy zone*[7].

These papers assume that a pair of subgoals are given, and then search for a compound goal. But in real game play, before starting the search for compound goals, we have to detect the dependency between two (or more) subgoals. For this purpose, we have to maintain the information of the moves which will affect the results of subgoals.

We call the moves which will invert the result of a subgoal, *inversions*. The intersections of *inversions* will be the candidates for multi purpose moves. We focus on developing a good algorithm for searching *inversions*.

Section 2 presents the related work. Section 3 describe the motive for searching *inversions*. Section 4 describe the algorithm. Section 5 shows the results of our experiment. Section 6 is for analysis of the results and future work. Section 7 concludes the paper.

† University of Tokyo, Graduate School of Information Science and Technology

☆ American Go Association

2. Related Work

The definition of *relevancy zone* is described in the [Thomsen 2000] about "lambda search"[7]. This is an area which possibly inverts the result of local subgoal search, if stones are played in. Thomsen used this idea to solve local problems of Go. *Relevancy zone* is strictly defined and easy to calculate, but as Thomsen already pointed out in his paper[7], it does not guarantee 100% correctness.

[Cazenave and Helmstetter 2004] defined *trace*[1]. They used *trace* to detect the dependencies between two connections. *trace* is obtained by adding the points that involved with the tests performed during the local subgoal search.

In Cazenave's paper, unions and intersections of *traces* are used for searching transitive connections. Unsurprisingly, using union was safer but slow, and using intersection was fast but less safe. This is the first attempt to combine multiple goals into one search using the idea such as *trace*. However, the subgoals are limited to connection problems.

In [Ramon and Croonenborghs 2004][6], *relevancy zone* is used for searching compound goals. This can be said to be a generalization of Cazenave's paper[1]. In this paper, subgoals are not limited to connections but also includes capturing and living. Their algorithm builds compound goals by combining subgoals using logical AND/OR/NOT. On improving speed for searching compound goals, they use *relevancy zone* obtained from each subgoal search.

These papers solve problems where two subgoals are given. *Trace* and *relevancy zone* are not so strict, and they might detect false dependencies. But that would not be a problem if two subgoals are already given, because false dependency could be correctly resolved by the multi purpose search.

Instead, in this paper we focus on how to find fairly strict *inversion*. Our aim is to find dependencies during real game play. If we want to go on to the safe side, we can use an area which is large than true *inversion*. In that case, false dependencies would be detected. But finding strict dependencies might get too time consuming in many cases. We have to consider the trade off between accuracy and speed.

3. The Purpose for Searching *Inversion*

Compared to Chess like games, evaluating a board position in the game of Go is difficult for programs. One reason which is making evaluation difficult is that in Go most position are not quiet.

Quiescence search is a strong tool to evaluate positions in Chess like games. But in Go, quiescence search is not so effective as it is in Chess.

The main reason for this difference is, unlike in Chess, the moves played in quiescence search will not be actually played in Go. In Go, when we reach an terminal node in quiescence search, we have to draw back to the original node where quiescence search had started, and then evaluate the board using the result of quiescence search.

Therefore, during the middlegame of Go, there are unstable situations remaining all over the board.

When strong human players play Go, they do not just focus on direct subgoals. To detect and use dependencies between subgoals, they maintain information of the *inversions* of subgoals. Without information of *inversions*, players will easily overlook double purpose moves.

Maintaining information of *inversions* does not just benefit in using dependencies of subgoals. Human players do not investigate board position from beginning on every turn. They keep what they thought in the past turns, and use the information to investigate the current position. In other words, they analyze the board position incrementally.

For incremental recognition of the board positions, the ability to distinguish independent subgoals and dependent subgoals are necessary. An easy example is shown in figure 1.

The white stone marked with an "A" is captured. It can not escape even if white plays first. The white stone marked with an "B" is also captured. This is confirmed by subgoal search, and the board condition will be evaluated assuming that "A" and "B" are both captured. But this is not true.

If white plays at the point marked with an triangle, white can save one of "A" or "B". To evaluate the board condition right, we have take *inversion* into account.

4. Algorithm for Searching Inversion

Definition 1 Inversion : An inversion of a subgoal is a set of points in which stones were played, the result of the subgoal search will get inverted.

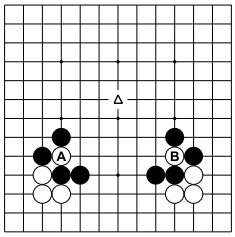


Fig. 1 Two ladders

Our target is to develop an algorithm which is suitable for finding *inversions*. In this section, we describe our algorithm for searching *inversion*.

In this section, first, we introduce an examination about the difficulty of this target. And then we describe the definition of *Relevancy Zone* and also the outline of df-pn algorithm. Finally, overall algorithm will be shown.

Please note that, in this paper, our target is limited to capturing of stones.

4.1 Preliminary Examination

An example is shown in figure 2. Let's consider capturing the white stone marked with a triangle. For Go players it is easy to see that the white stone can't escape from being captured even if white plays first.

Our aim is to find an area in which if white plays and black passes, the white stone can escape. The area is shown in the right of figure 2. The points marked with crosses are the *inversion* about capturing the white stone.

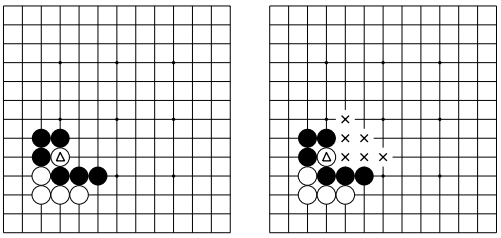
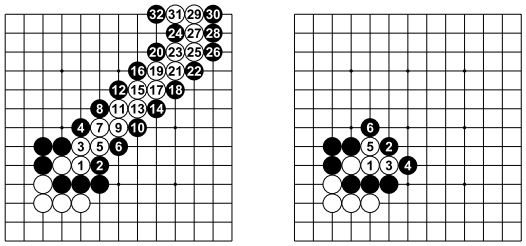


Fig. 2 Inversion on capturing the white stone.

To find an *inversion*, ideally we need to search for all possible capturing ways. If a white player's move can make the stone escape if followed by black player's pass, then such a move is an *inversion*.

But searching for other winning moves means less pruning. As shown in figure 3, there are many possible ways to capture the white stone, but it is quite time consuming if the search will not stop by finding one winning way.



Ladder Net

Fig. 3 Ladder and Net

We have implemented an algorithm which search for all possible winning moves, and then find the *inversion*. But the speed of the algorithm was close to mini-max algorithm. To find the *inversion* shown in figure 2, it needed to search more than 2 million nodes[8]. We had to implement an algorithm which can avoid this problem.

4.2 Definition of Relevancy Zone

The idea of *Relevancy Zone* is first described by Thomsen[7].

Original definition of *Relevancy Zone* requires the idea of lambda order. It is not exact and also not preferable to define Relevancy Zone without using lambda order, but for this paper we do not go further into the definition of lambda order.

We define our definition of Relevancy Zone. Please not that this definition is different from (and not better than) Thomsen's original definition.

Definition 2 Shadow Stone : Shadow stones are stones being played in a search.

Definition 3 n-Surrounding Block : 0-Surrounding Block is the target block for capturing. n-surrounding block is a group of stones which are adjacent to $n-1$-surrounding blocks, and the number of liberties are less than a threshold.
The threshold used in this paper is, if in attackers turn $l-n+2$, and if in defenders turn $l-n+3$, where l is the number of liberties of 0-surrounding block. ☆

Definition 4 Relevancy Zone : The union of shadow stones and the liberties of surrounding blocks.

☆ This is different from the definition given in Thomsen's paper. To be strict, l should be lambda order. Instead of lambda order, we used number of liberties and added 1 for safety.

4.3 df-pn

This is just an outline of the df-pn algorithm. Please refer to the original papers[2),4)] for precise explanation.

We have used df-pn (depth-first proof number search[4)]) algorithm for our capture search. For finding Relevancy Zone, we should use lambda search (or some other threat based search) but we have not yet implemented threat based search for our Go program so we used df-pn.

Figure 4 shows the idea of df-pn algorithm. Df-pn is an search algorithm for searching AND-OR trees. It uses proof number and disproof number. In capturing problem of Go, intuitively, proof number is the number of ways in which the defender can escape, and disproof number is the number of ways the attacker can chase the stone.

Proof number and disproof number have a duality. Figure 4 shows that the proof number of a node is the MINimum of disproof numbers of the children. And also, the disproof number of a node is the SUM of the proof numbers of the children.

The child node with the smallest disproof number is searched first.

For the terminal nodes, the value of the proof number and disproof number are defined as follows.

If the node is	win	: pn = 0, dn = ∞
	lose	: pn = ∞, dn = 0
	unknown	: pn=1, dn=1

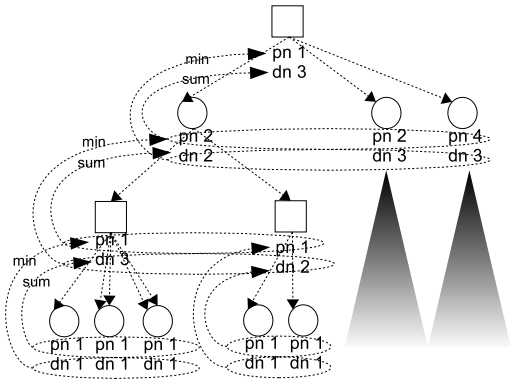


Fig. 4 df-pn algorithm

4.4 df-pn with Simulation

Simulation is a technique often used in mate search. It uses the search result of similar nodes. It is based on a speculation that in similar board positions, the same move would be good also.

What we expect is that while checking the candidates for *inversion*, in many cases the moves in the past searches are also good moves. Intuitively, it seems better if we check the moves in the past searches first, and if it fails then start searching other possibilities.

To implement this technique we have changed the terminal value of proof number and disproof number as follows.

If the node is	
win	: pn = 0, dn = ∞
lose	: pn = ∞, dn = 0
unknown(win in past)	: pn=1, dn=10000
unknown(lose in past)	: pn=10000, dn=1
unknown(unknown)	: pn=100, dn=100

Df-pn uses a hash table for storing the search result. We select one hash table used in a search starting from a similar board position, and use it as a reference. During search for *inversion*, we use this modified method for calculating proof number and disproof number. In this way the winning moves in the past search will be checked first. This would improve the speed of search if these the winning moves are really similar.

In this paper we used the value 10000, 100 and 1. However, there was no great difference in search speed by using 100, 10, and 1 respectively.

We are currently using only 1 past result as a reference. We select the past search result with the smallest size of *relevancy zone*, but this is only a heuristic. As a future work, we are planning to use 2 or more past search result as references. In that case, the value of the proof/disproof number of unknown nodes would be calculated from the win/lose/unknown ratio of the past searches. We expect this would result in a more flexible use of past searches. For example, if a move leads to win 2 times and lose 1 time, we put a higher priority for searching the move than totally unknown move. The problem is the trade-off between the number of memory accesses and pruning.

4.5 An Example

The basic idea is very simple.

First, we do a normal search. As the result of the normal search, we get a set of points which is relevant to this search. These points are temporal candidates for *inversion*. For each point we do a search to see if that point is a part of the *inversion*.

Figure 5 shows the example of the first search.

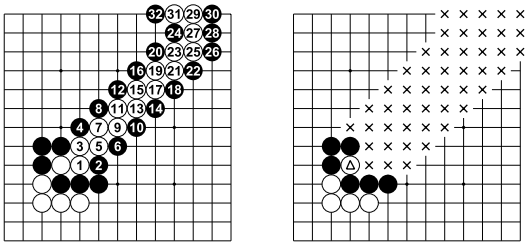


Fig. 5 First candidates for *inversion*

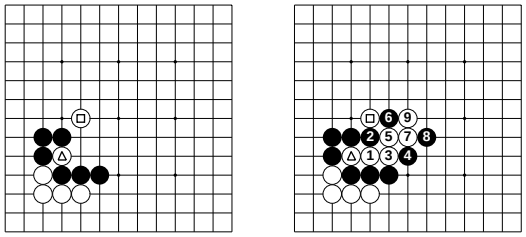


Fig. 7 An *inversion*

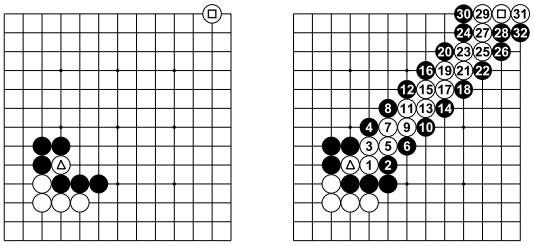


Fig. 6 Not an *inversion*

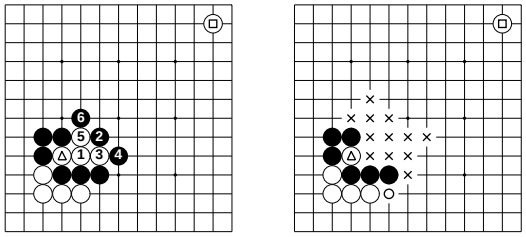


Fig. 8 *Inversion* with different RZone

The left figure shows white failing to save the stone. The candidates for *inversion* obtained from this search is shown in the right.

This set of points are retrieved from the search tree of the first search. Df-pn uses a hash table. We crawl through the searched hashtable and from all nodes within a winning path, retrieve *relevancy zone* which is defined in section 4.2.

Then, check the points in the candidates one by one, to see if it is a part of the *inversion*.

There will be two cases. One is that the point turned out that it is not an *inversion* and the other is that the point turned out to be an *inversion*.

An example of the first case is shown in figure 6. The white stone marked with an square is being checked if this is an *inversion*. As the right figure shows, this does not help the white stone marked with an triangle.

An example of the other case is shown in figure 7. This case, the white stone with a square is an *inversion*.

If all candidates obtained from the first search were *inversions*, then it is very easy to get *inversion*.

But we have to consider cases similar to the case in figure 8. In this case, the white move with a square is not an *inversion*. From this search, we can get a new set of points for the candidates for *inversion*, which is shown in the right of figure 8. This is totally different from the first candidates.

Before checking other candidates, we take the intersection of the past candidates and this these new candidates. In general, while checking the candidates, we will encounter points which are not an *inversion*. In such cases, we take the intersection of the candidates and continue checking for *inversion*. In this way, we can recursively limit the candidates for *inversion*.

Please note that the empty point marked with a circle in the right of figure 8 was not included in the first example. This means we can not just switch to the new smaller set of candidates for *inversion*. We have to take the intersection of candidates point set.

4.6 Overall Algorithm

Overall algorithm is shown in figure 9. The algorithm is described in a ML like pseudo code.

This version uses simulation. We have also implemented an algorithm without simulation, and compared the performance.

The outcome of this algorithm is not only the *inversion*. For detecting candidates for multi purpose, maintaining *inversion* would be enough. But for incremental analysis of board positions, we have to know when we have to do subgoal search again.

If there are no moves with in the *inversion*, the result of the subgoal remains unchanged. But, the *inversion* would change by a move outside the *inversion*.

For example, figure 2 shows the *inversion* for

capturing the white stone. If there is a move played in the stone marked with an square in the figure 8, the *inversion* would be changed to the area shown in figure 8.

Therefore, we have to detect the change in *inversion* and search again. To detect this we are planning to use the union of *relevancy zone* denoted rzunion in the pseudo-code.

5. Results and Analysis

We have measured the number of nodes searched for problems in figure 10,11,12.

The problems multi5, multi5b and multi5c are provided by Tristan Cazenave.

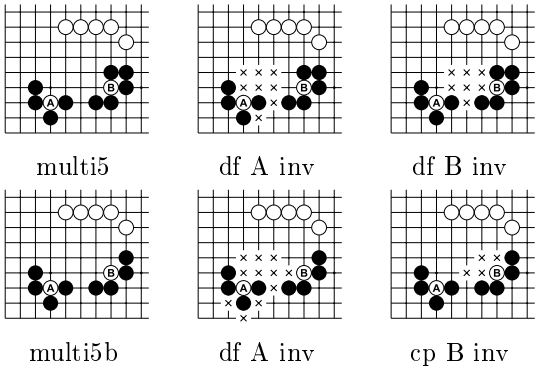


Fig. 10 problems1

The *inversions* are marked with crosses. "df" stands for defending the stone from capturing and "cp" stands for capturing. "df inv" means, the stone cannot be saved and searched for the *inversion* which make it possible to escape.

For some problems, the *inversion* is wrong. For prob1 and prob2, there are some missing points, and in multi5 and multi5b, there are some false points. These errors are because of the faults of our capturing algorithm. We expect that if we improve our search algorithm, we can avoid these errors.

The table 1 shows the number of nodes searched. The row "1st search" shows the nodes searched in the normal search. In other words, for problem "multi5", white tried to save the stone marked with an "A" and failed. The number of nodes search in the search is 94.

The row "inv" shows the number of nodes searched in the search for *inversion* of the 1st search, and the row "inv sim" shows the *inversion* search with simulation enabled.

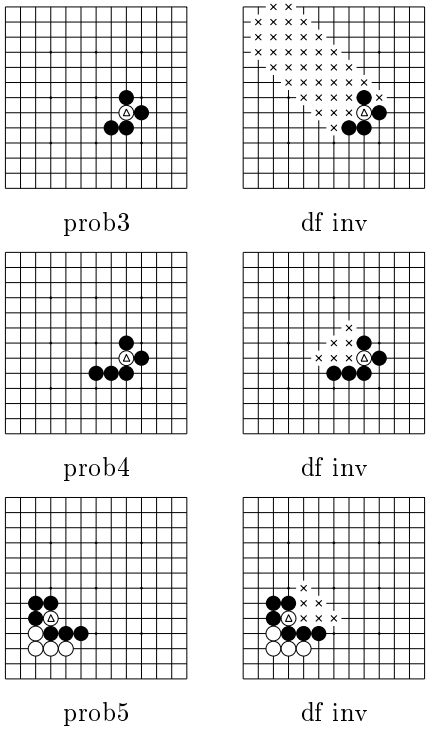


Fig. 12 problems3

problem	1st search	inv	inv sim
multi5 df A	94	16978	15198
multi5 df B	782	19721	13729
multi5b df A	910	36914	36399
multi5b cp B	246	3074	2486
multi5c df A	96	1861	3370
multi5c cp B	135	1474	1162
prob1 cp	603	16430	9875
prob2 cp	755	15852	11411
prob3 df	199	419732	378116
prob4 df	185	21012	19641
prob5 df	216	10748	10594

Table 1 Number of nodes searched

6. Analysis and Future Work

First point is that simulation is not so effective in many cases. One important thing about simulation is that the order for choosing the candidates of *inversions* in the algorithm. First we expected it will be better if we choose candidates further from the target block first. But it turned out that there are not difference between the method choosing from the furthest candidates and the method choosing according to the coordinate order.

The effectiveness of simulation relies on the similarity of the positions. We expect that it

```
let capture_inversion node =
  (* do normal search *)
  let (orig_winner, orig_rzone, orig_hashtable) = catpure_search node in

  let rec sub inversion rzinter rzunion checked rest reference =
    if PointSet.is_empty rest then
      (inversion, rzinter, rzunion)
    else
      let candidate = PointSet.choose rest in
      let inv_node = play_move candidate node in                          10

      let (inv_winner, inv_rzone, inv_hashtable) =
        capture_search_with_simulation inv_node reference
      in

        if orig_winner = inv_winner then
          (* candidate is not inversion *)
          let newrzinter = PointSet.inter rzinter inv_rzone in
          let newrzunion = PointSet.union rzunion inv_rzone in
          let newchecked = PointSet.add candidate checked in              20
          let newrest = PointSet.diff newrzinter newchecked in

          let newref =
            if (inv_rzone is better than reference) then inv_hashtable
            else reference
          in
            sub inversion newrzinter newrzunion newchecked newrest newref
        else
          (* candidate is inversion *)
          let newinversion = PointSet.add candidate inversion in          30
          let newchecked = PointSet.add candidate checked in
          let newrest = PointSet.remove candidate rest in
            sub newinversion rzinter rzunion newchecked newrest reference
  in
    sub emptyset orig_rzone orig_rzone emptyset orig_rzone orig_hashtable
;;
```

Fig. 9 Pseudo code of the algorithm

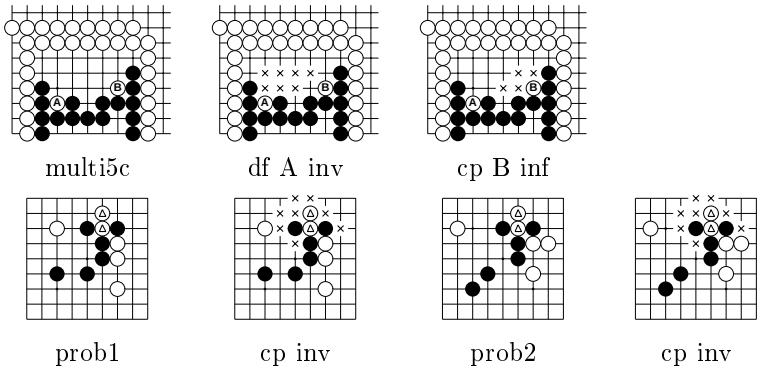


Fig. 11 problems2

could be improved more if we use more than 2 references, and/or use some heuristics for detecting the similarity of nodes and candidates ordering.

Second point is, for problems3, this algorithm is very time consuming. For the capturing problem, if only possible capturing is by a ladder, there are great number of candidates for *inversion*. The difference in problem3 and problem4 clearly shows the difference. We are planning to

develop some special heuristic for ladder problems.

Third point is that enclosed problems could be solved faster. This is common for all Go subproblems. In the results, mutli5c is solved faster than multi5b.

For searching *inversions*, often the 1st search is fast, but the search for *inversion* takes much time. Searching for a successful ladder takes very short time but for unsuccessful ladders, it often takes 10 times or more time to search. For this problem, an idea we call *weak inversion* might be useful.

Normal *inversion* inverts the result from win to lose (or lose to win), but *weak inversion* changes the result from win(or lose) to unknown. If the search for *inversion* takes too long time, we cut off the search at a threshold and mark it as a *weak inversion*.

We will list some other future work. We already mentioned in this paper, that *inversion* is useful for analyzing the board position incrementally. We will develop such a Go playing program. Our search algorithm should be improved using threat based search. We have only shown how to find *inversions*. We will use this to solve compound subgoal problems of Go, and see if we can find some improvements.

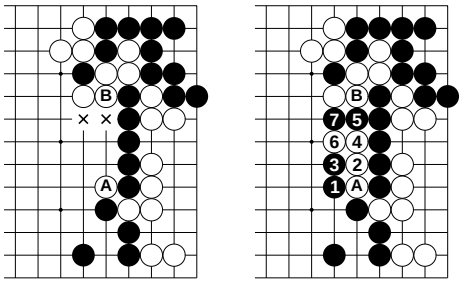


Fig. 13 How to handle order 2 *inversion*?

Figure 13 shows a board position. If black plays first, white stone A or B will be captured. There are points marked with crosses in the left figure. The *inversion* mentioned in this paper was 1st order. In other words, with only 1 move, the search result will get inverted. But this example shows that there are higher order *inversions* in real games. This problem is strongly related to sequence of forcing moves and higher order *inversions*. However, this is difficult to detect because in the Game of Go, 1st order *inversion* could be represented by a set of points, but 2nd order *inversion* is requires a set of pairs of points. Solving this kind of dependencies is remaining as an open point. ☆

7. Conclusion

We have implemented an algorithm for searching *inversions* of subproblems of Go. This information could be used for detecting candidates of multi purpose moves.

The experimental results shows that our algorithm can find *inversions* in a practical time. There are certain kinds of problems which takes long time and we are planning to improve it using the idea described in section 6.

Acknowledgments Thank you very much to Tristan Cazenave for providing his problem sets.

References

1) Tristan Cazenave and Bernard Helmstetter. Search for transitive connections. *Information Sciences*, December 2004.
2) Akihiro Kishimoto and Martin Muller. Df-pn in go: An application to the one-eye problem. In *Advances in Computer Games 10*, pages 125–141. Kluwer Academic Publishers, 2003.
3) Akihiro Kishimoto and Martin Muller. Dynamic decomposition search: A divide and conquer approach and its application to the one-eye problem in go. In *IEEE Symposium on Computational Intelligence and Games (CIG'05)*, pages 164–170, 2005.
4) Ayumu Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, Department of Information Science, University of Tokyo, Tokyo, 2002.
5) M. Pratola and T. Wolf. Optimizing gotools' search heuristics using genetic algorithms. *ICGA Journal*, 26(1):28–49, March 2003.
6) J.Ramon and T.Croonenborghs. Searching for compound goals using relevancy zones in the game of go. In J.vanden Herik, Y.Bjornsson, and N. Netanyahu, editors, *Fourth International Conference on Computers and Games*, Ramat-Gan, Israel, 2004. ICGA.
7) Thomas Thomsen. Lambda-search in game trees - with application to go. *ICGA Journal*, 23(4):203–217, December 2000.
8) Kazuki Yoshizoe and Hiroshi Imai. Searching for double threats in subproblems of the game of go. In *IPSJ SIG GI (2005-GI-14)*, pages 63–70, 2005.

☆ One excuse for this problem is that, this example is taken from a real game, and in the game both players (one higher than AGA 3dan) missed.