A New Heuristic Search Algorithm for Capturing Problems in Go

Keh-Hsun Chen and Peigang Zhang

Department of Computer Science,
University of North Carolina at Charlotte, NC, USA
{chen,pzhang1}@uncc.edu

Abstract. We propose a highly selective heuristic search algorithm for capturing problems in Go. This iterative deepening search works on the crucial chain in which the prey block is located. The algorithm starts using three order liberties of the chain as the basis of the position evaluation, the value is then adjusted by the presence of few liberty-surrounding opponent blocks. The algorithm solved most capturing problems in Kano's four volumes of graded Go problems. Moreover, it is fast enough to be used by Go programs in real time.

1 Introduction

Whether a block of stones in a Go board configuration can be captured by the opposite side is a fundamentally important knowledge item for a Go program. Without any knowledge of the block capture ability, other tactical problems in Go, such as life/death and connection cannot be accurately resolved. The global positional judgement could be totally wrong due to misjudgment of the capture ability of some blocks. Move decisions could become big blunders if made without recognizing the capture ability of some key stones [4]. Most Go programs spend over 90% of their processing time in doing capturing search of blocks of stones on the board. The quality of their capturing search routine is a key factor in the strength of a Go program.

In this paper, we shall describe a highly selective heuristic capturing search algorithm based on the classical α-β game-tree-search paradigm. The algorithm, called HuPrey (from Hunter-Prey), is a significantly improved version of the capturing algorithm used in Go Intellect in the past. Experimental results show that its performance compares favorably to other Go capturing algorithms [1,2,11].

Section 2 describes ladder capturing, which is a simple form of capturing; its routine is used extensively in our heuristic search capturing algorithm. We discuss the capturing target in Sect. 3 and its evaluation in Sect. 4; the generation of candidate moves in Sect. 5. The search paradigm and enhancements are examined in Sect. 6 and 7, respectively. The special situations of seki and ko are considered in Sect. 8. The experimental results are presented in Sect. 9. The paper is concluded with suggested future work in Sect. 10.

H.J. van den Herik et al. (Eds.): CG 2006, LNCS 4630, pp. 26–36, 2007.

2 Ladder

We shall call the capturing target block the prey (block). The player who attacks the prey is the Hunter and the player who defends the prey block is called Prey. The simplest form of capturing is so-called ladder capturing. In this case, the prey block should have fewer than 3 liberties. Once the prey gets 3 or more liberties, the ladder chase fails. In a game, the Hunter will continue to play atari moves. The Prey will always extend its only liberty or capture adjacent opponent's blocks with just one liberty if they exist. The move sequence in a ladder chase usually forms a zigzag pattern like a step ladder. This is a key routine that every Go-playing program must have.

Keeping track of the prey's liberties and those of the adjacent blocks under atari is the key of an efficient ladder algorithm. We note that the average branching factor of a ladder search is near 1; so, the ladder procedure is extremely fast. A normal diagonal ladder from one side of the board to the other side takes less than one millisecond on a modern computer.

A capturing search routine frequently needs to know (1) in its move generation and (2) in its position evaluation whether a block can be captured by a ladder. Obviously, for the *general* capturing problem, we cannot just consider the prey's liberties and those of the adjacent blocks in atari as candidate moves like in a ladder search.

3 Crucial Block and Crucial Chain

To generate the candidate moves adequately and to evaluate positions appropriately, one has to consider the whole chain, called the crucial chain, to which the prey block (also called crucial block) belongs. Figure 1 shows a simple example, in which just considering liberties and second liberties of the prey block (marked by a triangle) is obviously inadequate in finding the correct capture/escape move. The key move 'a' is not a liberty or secondary liberty of the crucial block marked by the triangle. The whole crucial chain, to which the prey belongs, needs to be considered.

A chain is a set of blocks of same color, which cannot be separated by the opponent [3]. We recognize two blocks being in the same chain if they obey either one of C1 to C3.

C1 They share two or more common liberties.
C2 They share one protected common liberty. By a 'protected empty point', we mean either it is illegal for the opponent to play there or if the opponent plays there, the played stone can be captured by a ladder (and the opponent cannot reoccupy the point through ko or snapback).
C3 They are adjacent to a common opponent's dead block.

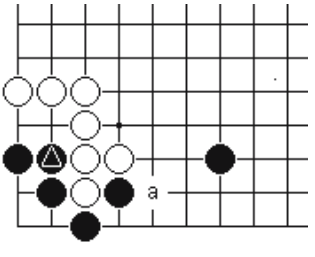

Fig. 1. Why the crucial chain should be connected

In practice, we can compute the transitive closure of the crucial block under C1, C2, and C3 to obtain the crucial chain. We should note that C1 to C3 do not completely define connectivity; they are just simple heuristics used by the capturing search routine in recognizing chains.

4 Position Evaluation

We define the empty points vertically or horizontally adjacent to a block as its liberties. A block is captured when it loses all its liberties. The first-order liberties are the ordinary liberties as defined above. We define a second-order liberty as a liberty of the augmented block when one additional stone of the block's color is added to the block (this stone may join additional blocks), which is not a first-order liberty. Similarly, we define a third-order liberty as a liberty of the augmented block when two additional stones of the block's color are added to the block (these stones may join additional blocks), which is not a first-order or second-order liberty. The first-order, second-order, and third-order liberties of the crucial chain can be computed via a breadth-first search. Let n_1, n_2, and n_3 be the total numbers of the first-order, second-order, and third-order liberties respectively. Assume Prey is to play. The basic evaluation is

$$V = n_1 \times 16 + n_2 \times 8 + n_3 \times 4\,. \tag{1}$$

We consider a first-order liberty twice as valuable as a second-order liberty and four times as valuable as a third-order liberty. If there is an adjacent Hunter block which can be captured by a ladder, we add 50 to V. If Prey has a block that can be captured by a ladder, we subtract 25 (since Prey plays next, the block could be saved).

When the crucial chain has two eyes, then it does not matter how many liberties it has, it is completely safe. Hence, in addition to liberties we should also check for solid eyes of the crucial chain. We go through each of the first-order liberties of the crucial chain to determine whether it is a solid eye. An eye is solid if (1) a liberty is surrounded only by stones of Prey and borders plus at least 3 of the 4 diagonal board points are secure for liberties away from edges, or (2) both diagonals are secure for liberties on a board edge, or (3) the diagonal is secure for liberties at board corners.

By a diagonal being secure, we mean it is occupied by a Prey's stone or it is a protected empty point. When we find a chain having two solid eyes, the evaluation value V will be reassigned to a large positive constant representing *definite safe*. Of course, when the prey is captured, the evaluation will be a negative constant with a large absolute value representing capture. We use the numbers of liberties of surrounding opponent chains to adjust the evaluation. We add 15 to the evaluation for each surrounding opponent chain with 2 or fewer effective liberties.

When it is Hunter to play, then we reverse the sign of evaluation V except when the prey block can be captured by a ladder, a very large positive constant representing successful capture will be assigned to V. This evaluation is used in

the node evaluation as well as in the candidate move generating and ordering in the search tree.

5 Candidate Move Generation

Below we shall discuss the generation of candidate moves. The relevant blocks include:

(A) the adjacent blocks of blocks in the crucial chain with:
  (a1) max(3, #liberties of crucial block) or fewer liberties if Hunter is to play;
  (a2) max(3, #liberties of crucial block) + 1 or fewer liberties if Prey is to play;
(B) the adjacent blocks of the relevant hunter blocks, not in the crucial chain and with 3 or fewer liberties.

The blocks listed under (A) will be called the relevant hunter blocks, the blocks listed under (B) will be called the relevant prey blocks.

We first take up to 4 first-order and second-order liberties at which the move would produce the best evaluations (see Section 4). Almost all Prey's moves that can be captured by a ladder or that fill its own solid eyes are eliminated. Hunter's sacrifice moves are allowed. We then first add 2 or 3 selected liberties of the relevant hunter blocks and subsequently add 2 selected liberties of the relevant prey blocks. The algorithm selects the best liberties by checking the surroundings of the liberties. It favors the liberties of the blocks with fewer liberties; liberties could further increase liberties or could connect to other blocks.

If a move can be captured by a ladder then we add the first move of the ladder capturing sequence into the candidate move set. This procedure introduces moves with an indirect approach. All the capturing moves, which can capture a block in a crucial chain or a relevant block or a block adjacent to a relevant prey block, are promoted or inserted to the front of the move list during the move ordering. When the number of candidate moves is two or one, Pass move is added as a candidate.

The power of HuPrey comes from the compact yet generally adequate candidate move set. The following three examples (see Figs. 2 to 4) illustrate the point.

Figure 2 shows candidate moves of Hunter (Black) in an attempt to capture the marked block. Candidate moves 1 to 4 are all from first-order and second-order liberties of the crucial block which is marked by a triangle. The relevant hunter blocks contain only the adjacent single stone block. It is Hunter's turn to play, the adjacent hunter blocks with more liberties than the prey are not considered. But the only relevant hunter block did not contribute any new candidate moves in this case. The relevant prey blocks are empty. Move 1 was promoted to become first candidate, since it is a ladder capture move (of the single stone adjacent hunter block). Candidate move 3 is the correct first move to capture the prey (followed by W2 andB4).

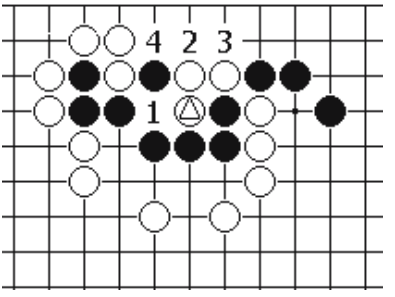

Fig. 2. Candidate moves for Black for capturing the marked white block. The correct black first move is at 3.

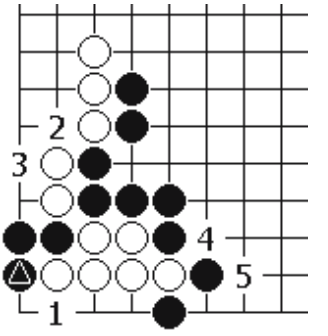

Fig. 3. A compact set of candidate moves from the crucial chain, relevant hunter blocks, and relevant prey blocks contains the winning first move at location 2

The next example (Fig. 3) is for Black to play to save the marked prey. Move 1 is the only move from the first-order liberties and second-order liberties of the prey, the other liberty moves can be captured by ladders, and are thus suppressed. Candidate moves 2 and 3 are liberties of the relevant hunter blocks and candidates 4 and 5 are liberties from the relevant prey blocks. Candidate move 2 is the key first move to save the prey.

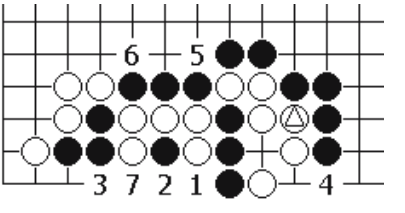

Fig. 4. Candidate moves for Black to capture the marked white block. The correct black first move is at 7 (followed by W2, B4).

In Fig. 4, Black to play, the crucial chain has two blocks, the marked one and the single white stone at the edge. They produce only one candidate move 4, a secondary liberty of the prey. Candidates 1, 2, and 3 are all ladder capturing moves, they have been promoted to the front. The two three-in-a-line black blocks form the relevant hunter blocks. They produce candidates 5 and 6. The relevant prey block is the n shape white five-stone block in the middle, which produces candidate move 7 - the winning first move.

6 Iterative Deepening with Hash Table

We use iterative deepening with a minimum of depth 2, a maximum of depth 20, and a depth increment of 2. The search terminates when the definitive result is obtained, or the maximum depth is reached, or the time limit is over 50% consumed. When Hunter is the first player, a definitive result means the crucial block (1) is removed from the board or (2) has two eyes or (3) is in seki. When

Prey is the first player, a definitive result means that the absolute value of the evaluation reaches a value above the search cut value (150).

A hash table is used to store the search results from transposition and from previous iterations. Zobrist's hashing method [13] is used to produce hash codes. Subtree search results with sufficient search depth are stored. At each node of the search tree, the hash table is searched to find the previously found best next move, which will be tried first.

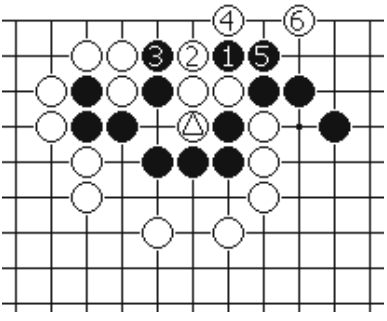

Fig. 5. Search outcome of the problem in Fig. 2. Note that at this point the algorithm knows the prey can be captured by a ladder, it returns success.

Forward pruning is used in a conservative way: only when the evaluation of the current node is outside the α-β window by the search cut, Δ value (50) or more, a forward pruning is performed and no further node expansion will be done under the node.

Figure 5 shows the solution sequence found by the heuristic capturing search HuPrey on the problem in Fig. 2. It took 0.422 second searching to 6 plies. When Hunter finds that the prey block can be captured by a ladder, the search terminates. The moves in the ladder capturing sequence are not counted in the search depth.

7 Try Opponent's Best Refute Move

GoTools [12] tries the opponent's best refute move as the next candidate move in life/death search with great success. So, we borrowed this idea in HuPrey. It explores the opponent's refute move next regardless whether the move is in the current candidate list or not. This technique speeds up the search significantly. Since when a refute move does not help, it slows the search by a small fraction; when a refute move helps identifying a key sequence, it usually cuts the search time drastically. On the capturing problems in Kano [7], this technique speeds up the search by about 20% on average (see Table 1).

There is an additional merit of this technique. When the set of selected candidate moves does not include the winning key move, the technique may introduce the missing move in the search tree. In Fig. 6, the left-hand-side diagram shows the initial candidate moves for White. The right-hand-side diagram shows the successful capturing sequence found by the augmented HuPrey. Looking closely on the left-hand-side diagram, we notice the "best" four first-order and second-order liberties of the crucial chain are candidates 3, 4, 5, and 6 — the winning first move was not even there!

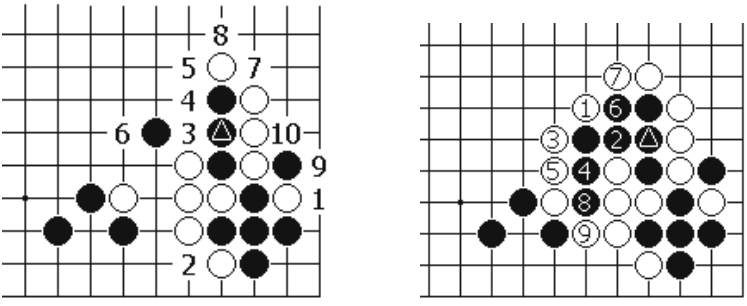

Fig. 6. The left diagram shows the initial candidate moves of Hunter (White). The right diagram shows the winning capturing sequence produced by HuPrey. The winning first move was not generated by the move generator. The technique of trying the opponent's best refute move introduced the winning move to the search tree.

8 Seki and Ko

When there are two consecutive passes, the prey is considered safe — either it has made two eyes or it has created a seki situation. In this case a large value will be returned by the evaluation function instead of first counting the three order liberties.

We use a simple approach to deal with ko. We assume the first player for the capturing problem can win kos up to three times and the other player cannot win any ko. If the capturing relied on winning kos, the value returned would not be as high as capturing without requiring to win a ko. This ko treatment is modified from Kierulf [9].

9 Experimental Results

We tested HuPrey on capturing problems in Kano's *Graded Go Problems for Beginners* four-book series [5,6,7,8]. Book 1 contains trivial problems. Book 2 requires some Go knowledge. Book 3 contains problems interesting to average players. Book 4 contains problems challenging to advanced amateur players. The books contain all types of Go problems: life/death, connection, capturing, and opening. We test our algorithm exclusively on capturing problems. There are a total of 180 capturing problems in the four books.

Book 1 (34 problems) – problems 1→22, 61→64, 179→182, 193→196.
Book 2 (34 problems) – problems 31→36, 130→141, 219→222, 316→327.
Book 3 (61 problems) – problems 2, 3, 7, 8, 11→13,15, 16, 20, 21, 24→26, 30, 31, 36, 38→43, 45, 95, 98, 105→108, 112→115, 118, 123, 124, 127, 128, 132, 157→160, 163, 278→293.
Book 4 (51 problems) – problems 1, 2, 11, 18, 19, 23, 27→41, 103→115, 135→138, 168, 184, 257→263, 387→390.

Table 1. Performance of HuPrey on Kano's graded Go capturing problems. The columns contain the following item for each book (from left to right column): the average time in seconds per problem, the average number of nodes per problem, the average depth per problem, the number of solved problems, the number of unsolved problems.

	ave. time	ave. nodes	ave. depth	num. solved	num. unsolved
Book 1	0.007	27	2.00	34	0
Book 2	0.830	4,912	6.76	33	1
Book 3	0.750	4,304	6.29	55	6
Book 4	8.100	35,933	5.97	40	11

We found that HuPrey can solve all the capturing problems in Books 1 and 2, 90% of the capturing problems in Book 3, and 80% of the capturing problems in Book 4, which outperforms all known testing results so far [1,2].

HuPrey is implemented in the latest version of Go Intellect. We run our tests on a 2.8 GHz Pentium 4. Table 1 summarizes the average testing results per solved problem.

For HuPrey, Book 2 capturing problems are slightly more time consuming than Book 3 capturing problems. Problem 28 of Book 2 can be solved by HuPrey in its stated setting in 165 seconds generating over 300,000 nodes. In order not to obscure the average statistics, we count it as unsolved in Table 1. This problem is not really a hard problem. After widening the candidate move set by one, our algorithm solved it in less than 0.01 second.

Every capturing problem in Book 1 was solved by HuPrey in less than 0.1 second. Figs. 7, 8, and 9 show the time allowance vs. percentage of problems solved for capturing problems in Kano Books 2, 3, and 4, respectively [6,7,8]. In each figure, the X-axis represents log(time allowance in seconds), the Y-axis represents the percentage of problems solved under the time constraint.

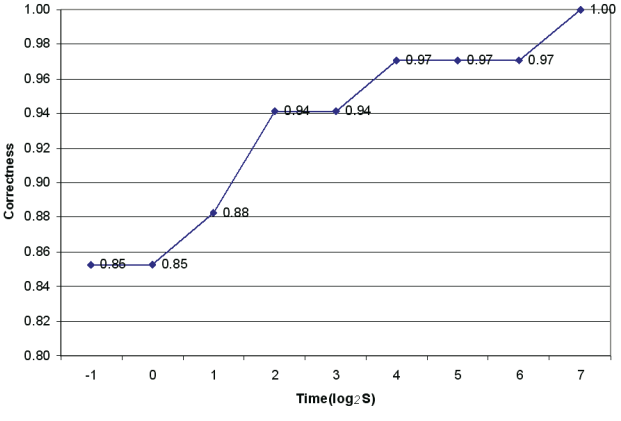

Fig. 7. The algorithm's time and solvability trade-off on the capturing problems in Kano Book 2

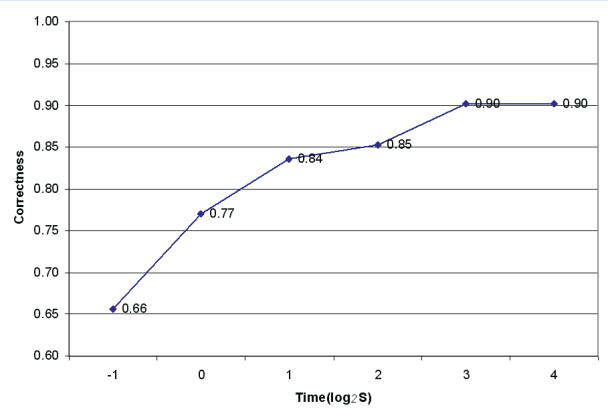

Fig. 8. The algorithm's time and solvability trade-off on the capturing problems in Kano Book 3

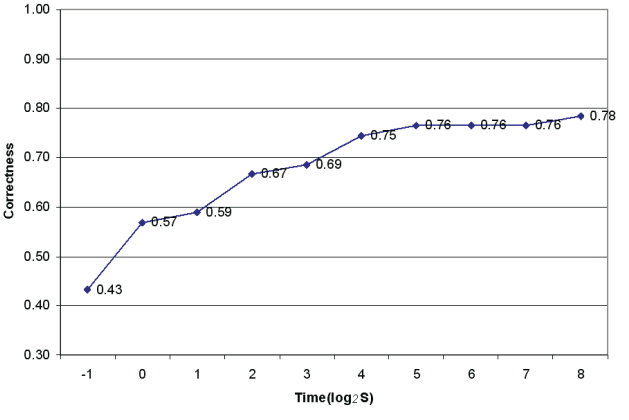

Fig. 9. The algorithm's time and solvability trade-off on the capturing problems in Kano Book 4

Figure 9 shows that about 43% of the capturing problems in Book 4 can be solved in less than 0.5 seconds. If we double the time limit to one second, about 57% of the problems can be solved. Double the time again to 2 seconds, 59% the problems can be solved. With a 4-second time limit, 67% of the problems can be solved. After passing the 75% result by this procedure, doubling the time produces very little improvements on the number of problems that can be solved.

Typically, a Go program will allocate 1 second or less for a capturing tactic search. Longer than one second time allocation for a capturing problem can be made only during the opponent time in a tournament setting. In practice, Hunter is set to be the first player. When the search finds that the prey can be captured, a second search with Prey going first is done to determine whether the prey can escape. Solved problems are stored in a solution hash table to be retrieved many

times until the surrounding changes, which will be reflected in the hash code. This solution hash table is not the hash table used by the capturing search, which is cleared for each new problem before the start of the capturing search.

10 Future Work

Since the candidate move generation of HuPrey is highly selective, inevitably it will miss some key moves once in a while and if these key moves are not produced by the opponent's best refute then the algorithm may spend unnecessary long time in searching for a solution - it may even claim a wrong conclusion. Incorporating some suitable widening scheme into the iterative deepening framework is likely to fix the problem and to produce a more powerful capturing search algorithm. We shall investigate such a mixed widening/deepening algorithm in the near future.

Proof number search has been used very successfully on the tactical problems with fully enclosed boundaries [10]. Capturing problems are in general quite open, requiring dynamic determination of the candidate move region. Whether PN+ search can work nearly as well for open-region capturing problems is worth investigating.

Acknowledgments

We would like to thank Martin Müller for providing SGF files of the graded Go problems in Kano Books 1, 2, and 3 [5,6,7]. As a result, we only needed to enter the capturing problems of Book 4 by hand [8]. This saved us much time.

References

1. Cazenave, T.: Abstract Proof Search. In: Marsland, T., Frank, I. (eds.) CG 2001. LNCS, vol. 2063, pp. 39–54. Springer, Heidelberg (2002)
2. Cazenave, T.: Iterative Widening. In: Nebel, B. (ed.) Proceedings of IJCAI-01, vol. 1, pp. 523–528 (2001)
3. Chen, K.: Group Identification in Computer Go. In: Levy, D., Beal, D. (eds.) Heuristic Programming in Artificial Intelligence, pp. 195–210. Ellis Horwood, Chichester (1989)
4. Chen, K.: Computer Go: Knowledge, Search, and Move Decision. ICGA Journal 24(4), 203–215 (2001)
5. Kano, Y.: Graded Go Problems For Beginners. In: Introductory Problems, vol. 1. Kiseido Publishing Company (1985)
6. Kano, Y.: Graded Go Problems For Beginners. In: Elementary Problems, vol. 2. Kiseido Publishing Company. ISBN 1985
7. Kano, Y.: Graded Go Problems For Beginners. In: Intermediate Problems, vol. 3. Kiseido Publishing Company (1987) ISBN 4-906574-48-3
8. Kano, Y.: Graded Go Problems For Beginners. In: Advanced Problems, vol. 4. Kiseido Publishing Company (1990)

9. Kierulf, A.: Smart Game Board: A Workbench for Game Playing Programs, with Go and Othello as Case Studies. PhD thesis, ETH Zurich (1990)
10. Kishimoto, A.: Correct and Efficient Search Algorithms in the Presence of Repetitions, PhD thesis, University of Alberta (2005)
11. Thomsen, T.: Lambda-Search in Game Trees - with Application to Go. ICGA Journal 23(4), 203–217 (2000)
12. Wolf, T.: Forward Pruning and Other Heuristic Search Techniques in Tsume Go. Information Sciences 122(1), 59–76 (2000)
13. Zobrist, A.L.: A New Hashing Method with Application for Game Playing, Techn. Rep. No. 88, Univ. of Wisconsin, Madison, 1970. Republished in 1990, ICGA Journal, 13(2):69–73 (1970)