DF-PN IN GO: AN APPLICATION TO THE ONE-EYE PROBLEM

A. Kishimoto, M. Müller
Department of Computing Science University of Alberta
Edmonton, Canada, T6G 2E8
{kishi,mmueller}@cs.ualberta.ca, http://www.cs.ualberta.ca/˜kishi/

Abstract Search algorithms based on the notion of proof and disproof numbers have been shown to be effective in many games. In this paper, we modify the depth-first proof-number search algorithm df-pn, in order to apply it to the game of Go. We develop a solver for *one-eye problems*, a special case of enclosed tsume-Go [life and death] problems. Our results show that this approach is very promising.

Keywords: Go, proof-number search, df-pn, one-eye problem

1. Introduction

Computer Go is one of the ultimate challenges for games researchers. Despite a lot of efforts, the best programs can still be easily beaten even by human players of moderate skill.

One weakness of current Go programs is recognizing whether groups are alive or dead. Such *tsume-Go (life and death)* problems play a critical role in deciding the outcome of many games. Currently most Go-playing programs rely on a combination of exact and heuristic rules to evaluate tsume-Go (Chen and Chen, 1999; Kraszek, 1988). However, this approach does not always guarantee the correctness of the results.

In general, search is the only way to assess correctly the life-and-death status of stones. However, the large branching factor of Go makes it hard to apply a purely search-based approach. For enclosed tsume-Go problems with a small to moderate branching factor, the state of the art is already very good. GoTools, the currently best tsume-Go solver, achieves high dan amateur level (Wolf, 2000). Search-based approaches have been very successful in other games such as chess, Othello, and shogi. In particular, in tsume-shogi (shogi checkmating problems), algorithms using proof and disproof numbers such as Seo's PN* and Nagai's df-pn (Seo, 1995; Nagai, 2002) have solved all difficult problems, including those with solution sequences of hundreds of plies. Their performance far surpasses that of human players.

In this paper, we adapt the df-pn algorithm to the game of Go, and apply it to a restricted version of tsume-Go: the problem of making *one eye* in an enclosed position. This special case can be solved with a simpler evaluation function, but retains all the search-related difficulties of tsume-Go. To our knowledge, this is the first attempt to apply df-pn to computer Go. Our results are very promising. Even with very modest game-specific enhancements, our df-pn-based solver can quickly solve enclosed positions up to about 18 empty points. This compares favourably to state of the art tsume-Go solvers, which can solve general tsume-Go problems of up to about 14 empty points in reasonable time.

The structure of this paper is as follows. Section 2 describes the one-eye problem in Go and related work on tsume-Go. Section 3 reviews the df-pn algorithm. Section 4 explains a problem of df-pn in domains with position repetition, and develops a solution. Section 5 describes the basic one-eye solver and a few problem-specific enhancements. Section 6 deals with our current implementation of ko threats. Section 7 discusses the experimental results. Section 8 concludes and outlines further research directions.

2. The One-Eye Problem in Tsume-Go

The one-eye problem in Go is the question whether a player can create an eye connected to the player's stones in a given region. Although the problem is simpler than full tsume-Go, it has many issues in common. For example, every tsume-Go problem in which the group under attack already has one eye in some region reduces to the one-eye problem on the rest of the board.

A specialized one-eye solver also promises to be useful to enhance the knowledge in a heuristic Go program. Typical current programs use elaborate heuristic rules to assign statically a number of eyes to a region of the board (Chen and Chen, 1999; Fotland, 2002). Replacing some of these heuristics by exact results can improve group strength estimation and thereby overall position evaluation.

A one-eye problem in a given Go position is defined by the following inputs.

- The two players, called the *defender* and the *attacker*. The defender tries to make an eye and the attacker tries to prevent it.
- The *region*, a subset of the board. At each turn, a player must either make a legal move within the region or pass.
- One or more blocks of *crucial stones* of the defender. The defender wins a one-eye problem by creating an eye connected to all the crucial stones inside the region. The attacker can win by either capturing at least one crucial stone, or by preventing the defender from creating an eye in the region.
- *Safe attacker stones* which surround the region together with crucial defender stones.

Figure 1 shows an example of a one-eye problem. Black is the defender and White is the attacker. Crucial stones are marked by triangles and the region is marked by crosses. Black must make an eye inside the region, while White tries to prevent that. There are unsafe stones at **C6**, **E7**, and **H6**. If these stones are captured, a player might play at such a point later, so they are part of the region.

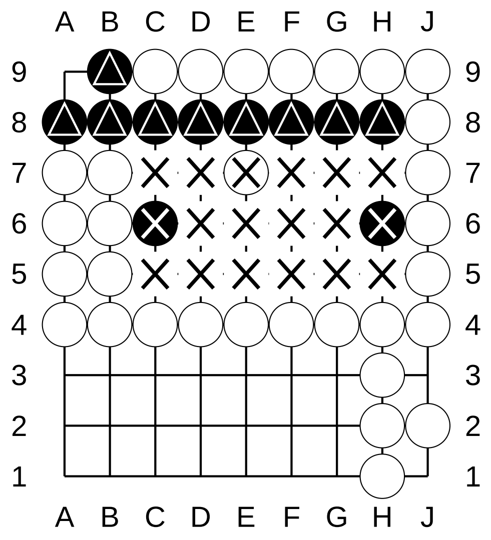


Figure 1. Example of a one-eye problem (Black to live).

2.1 Related Work on Tsume-Go

Wolf's (1994) GoTools is the currently best tsume-Go solver that specializes in solving completely enclosed positions. GoTools contains a sophisticated evaluation function that includes dynamic aspects, powerful rules for life-and-death recognition, and learning dynamic move ordering from the search (Wolf, 2000). Most competitive Go programs also contain a tsume-Go module. The commercial database Tsume-Go Goliath uses a proof-number search engine to check the user's inputs.

3. Df-pn: Depth-First Proof-Number Search

In this section we give an overview of the standard df-pn algorithm. Nagai's (2002) thesis is available for a detailed explanation.

3.1 Proof and Disproof Numbers

Proof and disproof numbers and Allis' proof-number search (PNS) (Allis, Van der Meulen, and Van den Herik, 1994) are the basis of this algorithm. The proof number of a node in an AND/OR tree is defined as the minimum number of leaf nodes that must be proven to prove the node for the first player, while the disproof number is the minimal number of leaf nodes that must be disproven (proven a win for the second player) in order to disprove the node. Proof and disproof numbers can be viewed as an estimate of how easy it is to prove or disprove a tree.

Proof-number search (PNS) maintains a proof number and a disproof number for each node. The leaf node to expand next is chosen in a best-first manner. Starting from the root, PNS traverses the tree by continuously selecting a child whose (dis)proof number is minimum at OR (AND) nodes, until it reaches a leaf node called a *most-proving node*. PNS expands that node and recomputes the proof and disproof numbers on the path to the root. This process continues until the root is either proven or disproven.

3.2 The Df-pn Algorithm

Df-pn (Nagai, 2002) turns PNS into a depth-first search algorithm by generalizing ideas behind Seo's (1995) PN* algorithm. As a depth-first search, df-pn can expand less interior nodes and use a smaller amount of memory than PNS. Like PNS, it always expands a most-proving node.

Figure 2, adapted from Nagai (2002) , presents pseudocode of the df-pn algorithm. Df-pn utilizes two thresholds, one for proof numbers and one for disproof numbers. For the sake of simplicity, the code is written in the negamax form, because disproof numbers are a dual notion of proof numbers. For each node n, two variables ϕ and δ are defined as follows:

$$n.\phi = \begin{cases} \mathbf{pn}(n) & (n \text{ is an OR node}) \\ \mathbf{dn}(n) & (n \text{ is an AND node}) \end{cases}$$

$$n.\delta = \begin{cases} \mathbf{dn}(n) & (n \text{ is an OR node}) \\ \mathbf{pn}(n) & (n \text{ is an AND node}) \end{cases}$$

While the iterative deepening method usually has a global threshold, df-pn's thresholds work as local thresholds at each recursive call. This approach is similar to recursive best-first search (Korf, 1993). The main function *Df-pn* initializes both thresholds to infinity, and then calls the recursive function *MID* that iterates over nodes. When returning from *MID*, the root node is either proven or disproven. *MID* traverses the subtree below node n in a depth-first manner. It explores nodes while proof or disproof numbers do not exceed the threshold, or until it finds a terminal node that determines a winner. In the code, *IsTerminal* checks if n is a terminal node, while *WinforCurrentNode* checks whether a terminal node is a win or a loss. When a node n is expanded, the best child n_c in terms of proof and disproof numbers is selected by *SelectChild* for a recursive call to *MID* with the following new thresholds: $n_c.\delta$ is set to the minimum of the current threshold for n and the value when n's child with the second smallest δ becomes the most-proving node during the exploration of n_c's subtree. Note that $n.\phi$ corresponds to $n_c.\delta$ because of the negamax formulation. $n_c.\phi$ works like the cost function of the IDA* algorithm (Korf, 1985).

Because df-pn is an iterative-deepening method that expands interior nodes again and again, the heart of the algorithm is the transposition table, a large

```
// Set up for the root node
int Df-pn(node $r$) {
    $r.\phi = \infty$;  $r.\delta = \infty$;
    MID($r$);
    if ($r.\delta = \infty$)
        return win_for_root;
    else
        return loss_for_root;
}

// Iterative deepening at each node
void MID(node $n$) {
    TTlookup($n$,$\phi$,$\delta$);
    if ($n.\phi \leq \phi$ || $n.\delta \leq \delta$) {
        // Exceed thresholds
        $n.\phi = \phi$; $n.\delta = \delta$;
        return;
    }
    // Terminal node
    if (IsTerminal($n$)) {
        if (WinforCurrentNode($n$)) {
            $n.\phi = 0$; $n.\delta = \infty$;
            return;
        } else {
            $n.\phi = \infty$; $n.\delta = 0$;
            return;
        }
    }
    GenerateMoves($n$);
    // Store larger proof and disproof
    // numbers to detect repetitions
    TTstore($n$,$n.\phi$,$n.\delta$);
    // Iterative deepening
    while ($n.\phi > \Delta$Min($n$)  &&
            $n.\delta > \Phi$Sum($n$))  {
        $n_c$ = SelectChild($n$,$\phi_c$,$\delta_c$,$\delta_2$);
        // Update thresholds
        $n_c.\phi = n.\delta + \phi_c - \Phi$Sum($n$);
        $n_c.\delta$ = min($n.\phi$,$\delta_2 + 1$);
        MID($n_c$);
    }
    // Store search results
    $n.\phi = \Delta$Min($n$); $n.\delta = \Phi$Sum($n$);
    TTstore($n$,$n.\phi$,$n.\delta$);
}
```

```
// Select the most promising child
node SelectChild(node $n$, int &$\phi_c$,
                 int &$\delta_c$, int &$\delta_2$)  {
    node $n_{best}$;
    $\delta_c = \phi_c = \infty$;
    for (each child $n_{child}$) {
        TTlookup($n_{child}$,$\phi$,$\delta$);
        // Store the smallest and second
        // smallest $\delta$ in $\delta_c$ and $\delta_2$
        if ($\delta < \delta_c$) {
            $n_{best} = n_{child}$;
            $\delta_2 = \delta_c$; $\phi_c = \phi$; $\delta_c = \delta$;
        }
        else if ($\delta < \delta_2$)
            $\delta_2 = \delta$;
        if ($\phi = \infty$)
            return $n_{best}$;
    }
    return $n_{best}$;
}

// Compute the smallest $\delta$ of
// $n$'s children
int $\Delta$Min(node $n$) {
    int $min = \infty$;
    for (each child $n_{child}$) {
        TTlookup($n_{child}$,$\phi$,$\delta$);
        $min$ = min($min$,$\delta$);
    }
    return $min$;
}

// Compute sum of $\phi$ of $n$'s children
int $\Phi$Sum(node $n$) {
    int $sum = 0$;
    for (each child $n_{child}$) {
        TTlookup($n_{child}$,$\phi$,$\delta$);
        $sum = sum + \phi$;
    }
    return $sum$;
}
```

Figure 2. Pseudocode of the df-pn algorithm.

cache storing previous search efforts, i.e., proof and disproof numbers for visited nodes. *TTstore* stores proof and disproof numbers of a node in the table. *TTlookup* checks the table for information on proof and disproof numbers of a node. If no result is found, both numbers are initialized to 1.

4. Computing Proof and Disproof Numbers in Domains with Repetitions

When we tried to apply df-pn to the one-eye problem in Go, df-pn could not solve some easy problems. The standard df-pn algorithm has a fundamental problem when applied to a domain with repetitions. Figure 3 shows an example. Assume F is unknown, then the df-pn algorithm computes $\mathbf{pn}(E) = \mathbf{pn}(A) + \mathbf{pn}(F)$. Hence, $\mathbf{pn}(E)$ is larger than $\mathbf{pn}(A)$. Df-pn's termination condition is (see Figure 2):

$$n.\phi \leq \Delta\text{Min}(n) \;||\; n.\delta \leq \Phi\text{Sum}(n)$$

Usually the threshold of the proof number is only a little bit larger than $\mathbf{pn}(A)$ when exploring A's subtree in df-pn. Therefore, assuming that df-pn reaches E, df-pn exceeds the proof number threshold, stops expanding and updates A's proof number to $\mathbf{pn}(E) = \mathbf{pn}(A) + \mathbf{pn}(F)$. Even if E is chosen in a later iteration, this phenomenon continues and F is never explored. These repetitions often happen in Go, because passes are allowed. Two consecutive passes lead back the same position in a short loop.

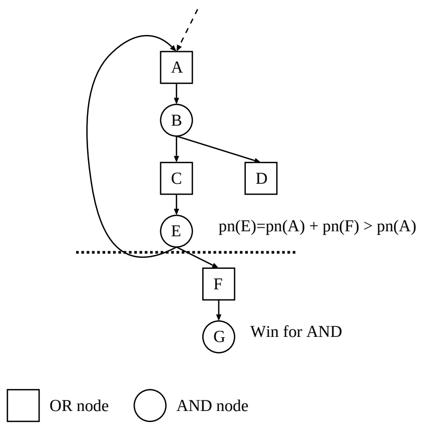


Figure 3. A problem with repetitions in df-pn.

Adding proof numbers from an ancestor to a node seems intuitively bad, since it leads to double-counting of the leaf nodes below. In our solution to this problem, we classify the children of a node into two types. A field *minimal distance (md)* of a node n is initially set to the length of the shortest path from the root to n, the depth of n in the search tree. We call a child n_i *normal* if $n_i.md > n.md$, and *old* if $n_i.md \leq n.md$. Among the children $n_1 \cdots, n_k$ of n, let $n_1 \cdots, n_l$ $(1 \leq l \leq k)$ be the normal and $n_{l+1}, \cdots, n_k$ the old children. We modify the computation of proof and disproof numbers in the following way:

$$n.\phi = \min_{1 \leq i \leq k} n_i.\delta$$

$$n.\delta = \begin{cases} \sum_{i=1}^{l} n_i.\phi & (\text{if } \sum_{i=1}^{l} n_i.\phi \neq 0) \\ \max_{l+1 \leq i \leq k} n_i.\phi & (\text{if } \sum_{i=1}^{l} n_i.\phi = 0) \end{cases}$$

Figure 4 illustrates an example of computing proof numbers. If F is neither proven nor disproven, then F's proof number cannot be 0. Therefore we ignore A to compute E's proof number, since A is an old child.

When a node has only old children, since all normal (and possibly some old) children have been solved, that node itself must be considered old, since now there is no way to prove or disprove it without exploring old nodes. Therefore, the md of that node must be updated. We set it to the minimum of the md fields of the currently unsolved old children.

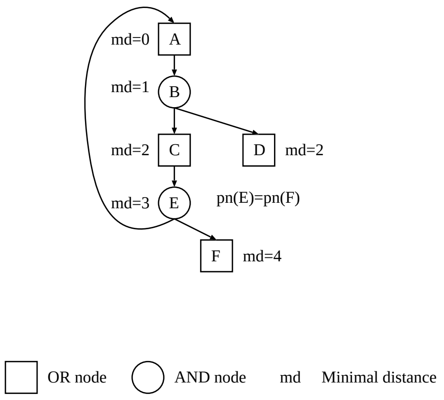
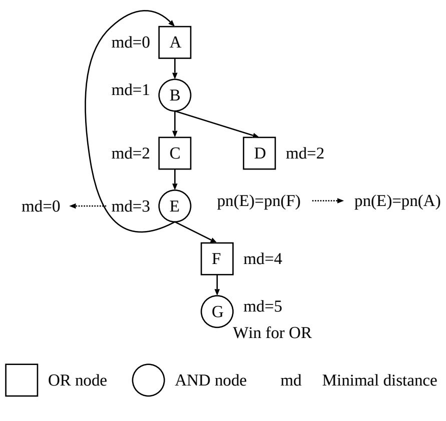


Figure 4. Df-pn with minimal distance md.

Figures 5 and 6 depict an example of updating md. In this figure, assuming that G is proven, E now has only an old child to explore, because F is also proven. In that case E's minimal distance is updated to A's distance, and $\mathbf{pn}(E)$ becomes $\mathbf{pn}(A)$. Further, $C.md$ is set to $E.md$ (see Figure 6). As a result, $\mathbf{pn}(C)$ is now ignored in the computation of $\mathbf{pn}(B)$, since C has become an old child.

Dealing with overcounting proof numbers caused by repetitions was essential to make df-pn work in Go. We note that Nagai (2002) achieves impressive



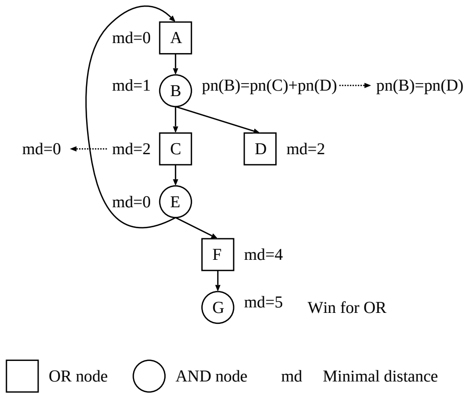


Figure 5. Updating E's minimal distance. *Figure 6.* Computing C's minimal distance.

results with his tsume-shogi solver, and described the GHI problem, which returns incorrect results involving cycles . However, this problem was not described in his papers. One possibility is that although the same problem could happen in shogi, it might happen much less often than in Go. Search in Go can easily return to identical states, for example by consecutive pass moves. Another possibility is that this problem tends to happen less frequently with additional search enhancements. Because Nagai's tsume-shogi solver is enhanced with a great deal of domain-dependent knowledge, it might not occur in his case in practice. However, in a personal communication the existence of this problem in shogi was confirmed by Tsuruoka and Maruyama of team GEKISASHI. As well, Sakuta found that df-pn did not work better than PDS (Nagai, 1999) in his tsume-shogi solver, and gave as possible explanation the occurrence of DCGs (Sakuta, 2001).

5. Application of Df-pn to the One-eye Problem

Below we apply the df-pn algorithm to the one-eye problem. We start with the basic one-eye algorithm (5.1). Then we provide several game-specific search enhancements (5.2). The section is concluded by a simulation (5.3).

5.1 The Basic One-eye Algorithm

The basic algorithm, due to Anders Kierulf, is quite simple, and has been used as part of the tsume-Go search in the program EXPLORER for many years. It detects single-point eyes and false eyes.

The algorithm checks for all points in the region whether they are a potential eye point for the defender. Eyes are created by either surrounding empty points or by capturing attacker stones. If a safe eye connected to the crucial stones can be created in the region, the defender wins. If there is no potential eye space in the region, the attacker wins.

Whether a point E is a *potential eye point* is computed as follows:

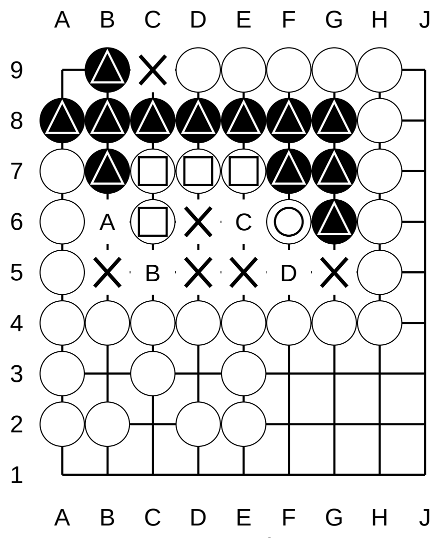
- E occupied by unsafe attacker stone: *yes*.
- E occupied by safe attacker stone: *no*.
- E occupied by defender stone: *no*.
- E is empty: check the neighbours and the diagonal neighbours of E.
    - Some direct neighbour is occupied by the attacker: *no*.
    - E is at the edge of the board and at least one diagonal neighbour contains a safe attacker: *no*.
    - At least two diagonal neighbours contain a safe attacker: *no*.
    - Otherwise: *yes*.

A potential eye point is a *safe eye* if all direct neighbours and all but one diagonal neighbour are occupied by defender stones. All diagonal neighbours are needed at the edge of the board. A safe eye is a defender win if the surrounding block is connected to crucial stones, and all crucial stones are connected. The search generates all moves in the region, unless there are forced moves (see below).

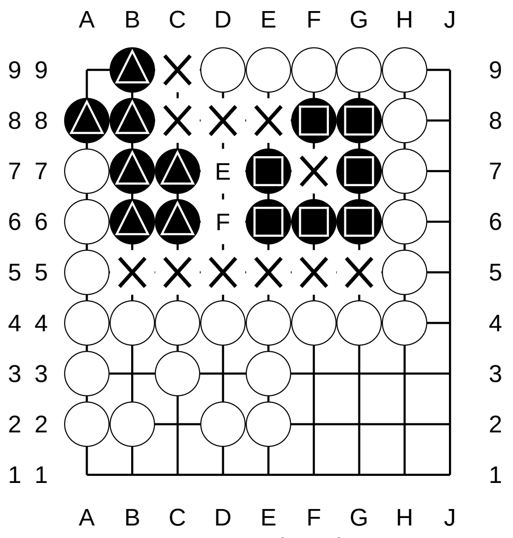
5.2 Game-specific Search Enhancements

Safety by Connections to Safe Stones. Connectivity is a fundamental aspect of the game of Go. Most Go programs recognize connected blocks. We use connections to promote unsafe attacker stones to safe, and to prove that a defender eye is connected to crucial stones. Both types of connections help to reduce the search depth.

Our current implementation recognizes simple *miai* strategies (Müller, 1997) and some protected liberties for connections. Figure 7 gives examples of the strategy. In the left diagram, White has two ways (**A** and **B**) to connect. Even if Black plays first, the white block marked with squares can connect to safe stones. The stone at **F6** is also safe now, because it has a connection either at C or at D. Since there is no eye space, this position can be statically evaluated as a loss for Black. Similarly, in the right diagram in Figure 7, the connection at **E** or **F** guarantees a win for Black. The algorithm to compute these connections is straightforward. It checks if safe blocks S have two liberties to connect to a block b. If this is the case, b is included in S and the two liberties are marked to not be used for other connections. The process continues until no further blocks can be added to S.



Connection for attacker



Connection for defender

Figure 7. Connections to safe stones.

We find more safe stones by recognizing some forms of protected liberties. Figure 8 shows an example. The stone marked with a square has only one connection point at **B** to a safe white block. However, this connection is safe since the stone has another liberty and the opponent cannot play at **B**.

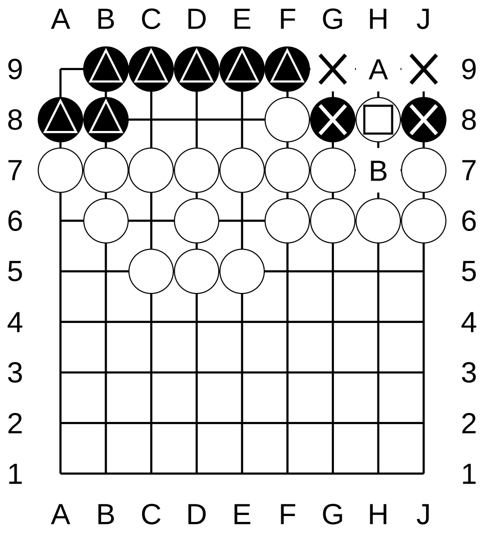


Figure 8. Connection to safe stones on protected liberty.

Forced Moves. Forced moves are a safe form of pruning when one player threatens to win immediately. We defined two kinds of forced moves, *forced attacker moves*, and *forced defender moves,* which correspond to **ip1** or **gi1** threats in Abstract Proof Search (APS) (Cazenave, 2002).

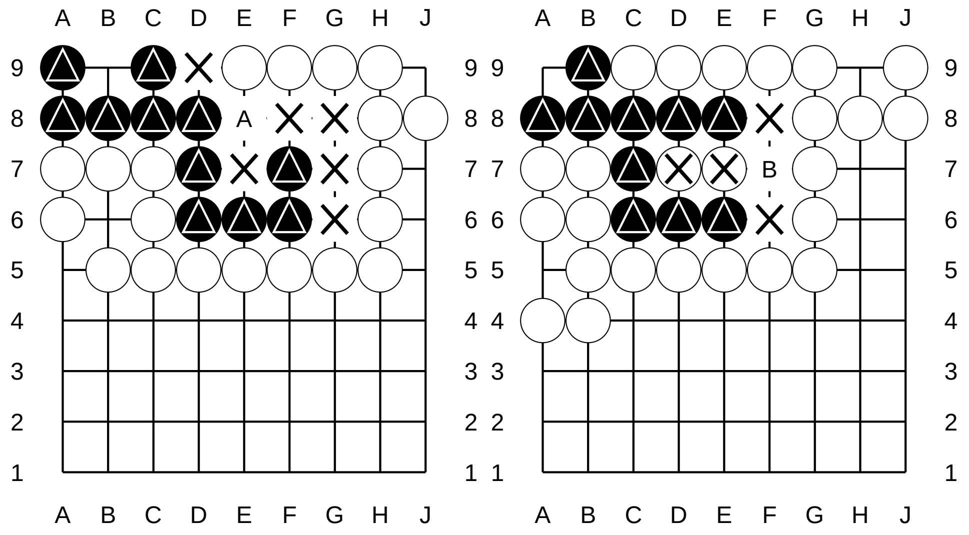


Figure 9. Forced Moves.

The first type of forced move is on a point where the defender could complete an eye that is connected to the crucial stones. The left position in Figure 9 presents an example. Black can make an eye at **A**. White must play at **A** to stop an immediate win for Black.

The second type of forced move is defined as follows:

1 There is no empty eye space for the defender in the region.

2 There is exactly one unsafe attacker's block b.

3 b has a single-move connection to safe stones. If the defender plays any other move, the attacker can connect b to safety, leaving the defender with no potential eye points.

For instance, in the right position of Figure 9 the move at **B** is forced.

Forced moves give a large reduction of the search space by decreasing the branching factor.

5.3 Simulation

Simulation was invented by Kawano (1996) to solve effectively positions with useless interposing piece drops in tsume-shogi problems . Later, Tanase (2000) extensively applied this idea to his $\alpha\beta$-search engine to reduce the overhead of calling the tsume-shogi solver inside the normal search . Assume that P is a proven position and Q is a "similar" one we want to prove. Simulation borrows moves from P's proof tree to try to find a quick proof of Q. A dual notion called *dual simulation* can be used to disprove a position.

In our solver, we apply simulation and dual simulation as follows:

- At an AND node n, if one of n's children, n_c, is proven at some point in the search, apply simulation to all unsolved children of n.
- Similarly, at an OR node n, apply dual simulation if one of n's children is disproven.

This use of simulation is much more extensive than in tsume-shogi. See the experimental section for a discussion.

6. Ko and Ko Threats

Sometimes the outcome of a one-eye problem depends on ko. It is therefore important to model ko threats and ko recaptures in the search algorithm.

The approach taken in GoTools can require several searches (Wolf, 1994). The parameter to each search is how many ko recaptures are allowed for a specified kowinner.

Our current implementation allows only two options: one is to disallow any immediate ko recaptures; the other is to always allow ko recaptures for the designated kowinner. We search in one or two phases. The first search of a position, phase 1, disallows immediate ko recaptures, but marks nodes where such moves exist. If the search result depends on marked nodes, in phase 2 a re-search is performed. The loser of the phase 1 search is the designated kowinner for phase 2.

Phase 2 reuses the contents of the transposition table from phase 1. The following implementation of the transposition table aims to reduce the amount of re-search:

1 The Zobrist (1970) hash function is modified to account for a stone captured in the previous ko capture, to differentiate identical positions with different histories.

2 Two flags, one for each colour, in each transposition table entry keep track of any possible ko captures in the subtree below that node. If there is a ko capture for a player, the flag for the other player is set to indicate that we will allow a ko recapture after that node in a re-search. When a node n is proven (similarly for disproven), flags are set as follows:

- If n is an OR node and n_c is n's proven child, n's flags are set the same as n_c's flags.
- If n is an AND node and the flag of one of the children is set, n's flag is set. Otherwise, n's flag is cleared.

In the phase 2 re-search, many phase 1 (dis)proofs can be reused. For example, assume that a node is proven and the flag for the kowinner is not set. Then we can use the proof from the transposition table. Similarly, we can also reuse disproofs. Even for nodes that are not proven or disproven, the proof and disproof numbers from phase 1 are valuable information for directing the re-search.

Re-searches usually have a low overhead, since we keep the previous results in the transposition table and reuse the table entries in most cases. However, if the solution changes dramatically by ko compared to the solution from the first search, a higher overhead results.

7. Empirical Results

This section consists of: test data (7.1), setup of experiments (7.2), test runs (7.3), and further comments on the experiments (7.4).

7.1 Test Data

In contrast to full tsume-Go, for which many large collections of test problems are available, we could not find any specialized collection of one-eye problems. Our current set of 70 test positions was created mainly by the authors. The problems can be played for both colours going first, resulting in a total of 140 problems. All problems are of the following form: a black group already has one safe eye, and is completely surrounded at a distance by safe white stones. The area in between forms the region, and the fate of the black group depends on whether it can form a second eye in the region. Problems of this kind are also suitable for solution by a general tsume-Go solver, since making one eye is equivalent to solving the tsume-Go problem.

The test set is available at `http://www.cs.ualberta.ca/~games/go/oneeye`. The problems include a mix of easy and hard problems. Some prob-

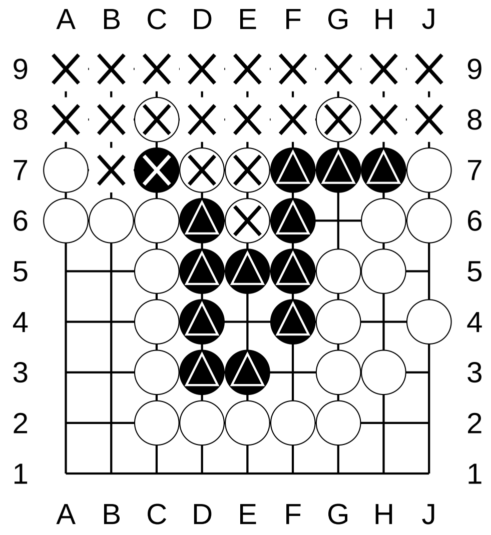


Figure 10. Example of a hard problem (Black to live).

lems are challenging only for one colour playing first, and are very easy if the other colour plays first. Some of the positions are hard to solve for current tsume-Go programs. For an example, see Figure 10.

7.2 Setup of Experiments

All experiments were performed on a Pentium III/700 Mhz with a 100 MB transposition table. The time limit was 5 minutes per problem.

The following abbreviations are used for the methods and enhancements described above.

- **Df-pn**: The basic df-pn algorithm.
- **MIN**: Minimal distance modification for computing proof and disproof numbers.
- **AC**: Connections to safe stones for attacker
- **DC**: Connections to crucial stones for defender
- **FAM**: Forced attacker's moves
- **FDM**: Forced defender's moves
- **SIM**: Simulation and dual simulation

7.3 Test Runs

Adding Enhancements. Table 1 shows the results on the test set, starting with basic df-pn and switching on enhancements one by one. The total execution time and number of nodes expanded were computed using the subset of 126 problems that are all solved by methods (2) - (7) in the table.

Enhancements used		Number of problems solved	Total time (sec) 126 Problems	Total nodes expanded 126 Problems	Nodes expanded per second
(1):	**Df-pn**	20	-	-	-
(2):	(1) + **MIN**	126	806	11,933,976	14,806
(3):	(2) + **AC**	132	424	5,431,557	12,810
(4):	(3) + **DC**	132	444	5,377,408	12,116
(5):	(4) + **FDM**	132	436	5,142,100	11,802
(6):	(5) + **FAM**	133	113	1,354,506	11,970
(7):	(6) + **SIM**	134	81	1,168,683	14,347

Table 1. Performance for successively switching on enhancements.

The table shows the importance of the **MIN** modification. The only problems solved by basic df-pn were very easy ones that needed at most 400 nodes.

Search speed decreases a little with more enhancements, but improves again with simulation. Simulation provides a fast way to generate moves, faster than our current normal move generator, which has some overhead such as checking connections.

Leave-One-Out Experiments. The results for switching off a single enhancement at a time are shown in Table 2.

Enhancement Turned Off	Number of Problems Solved	Total Time (s) (129 Problems)	Total Nodes Expanded (129 Problems)	Nodes Expanded per Second
MIN	74	-	-	-
AC	129	393	7,096,603	18,058
DC	134	138	2,081,344	15,070
FDM	134	264	3,705,778	14,052
FAM	133	402	5,590,511	13,907
SIM	133	175	2,123,969	12,137

Table 2. Performance for turning off single enhancements.

Performance of Simulation. Table 3 shows the performance data for simulation in phase 1 searches. Since the method is applied in a very basic way, 45.2 % success seems to be a good initial result, with plenty of room for further refinements.

Total Nodes	Nodes by **SIM**	**SIM** calls	successful calls
6,265,984	1,116,386 (17.8 %)	262,628	118,706 (45.2 %)

Table 3. Performance data on simulation for all 134 solved problems. All enhancements on. Phase 1 searches only.

Re-searches for Ko. Table 4 shows a summary of the overhead incurred by re-searches for ko. In phase 1, immediate ko recaptures are not allowed. Phase 2 are the researches with a designated kowinner. The results in this table are also with all enhancements.

Total Nodes (134 Problems)	
Phase 1	Phase 2
6,265,984 (95.4 %)	304,107 (4.6 %)

Table 4. Overheads for ko re-searches

The overhead is quite small, but of course this is mainly a property of the test set used, which contains only a few cases with complex ko fights. In the worst case encountered, problem oneeyeb.10.sgf with Black to play, phase 1 took 7,340 nodes and phase 2 took 11,728 nodes.

7.4 Further Comments on the Experiments

Reexpansion of Interior Nodes. One concern in df-pn is the overhead of reexpansion of interior nodes. In our experiments, the ratio of interior nodes expanded to total nodes is about 30 %. In Seo's experiments in shogi, this ratio was about 20 %. Since information achieved dynamically is usually more reliable than static evaluations, we think that our 30 % is still a very small price to pay to achieve more cut-offs.

Currently Unsolved Problems. Our solver currently cannot solve 6 problems in our test suite. Figure 11 shows an example. All unsolved problems feature large regions with many possible moves. Besides, some problems such as in Figure 10 and 11 stretch the limits of the one-eye problem, such as semeai, and tsume-Go. Figure 11, for example, can be seen as a problem whether white stones adjacent to black crucial stones can make two-eyes or not, having no split between the one-eye and tsume-Go problems. As well, the practical limit of our current solver seems to be at around 18 empty points, which compares favourably with about 14 empty reported for GoTools. However, we need further investigations to assess this limit and improve the ability of our solver for difficult problems.

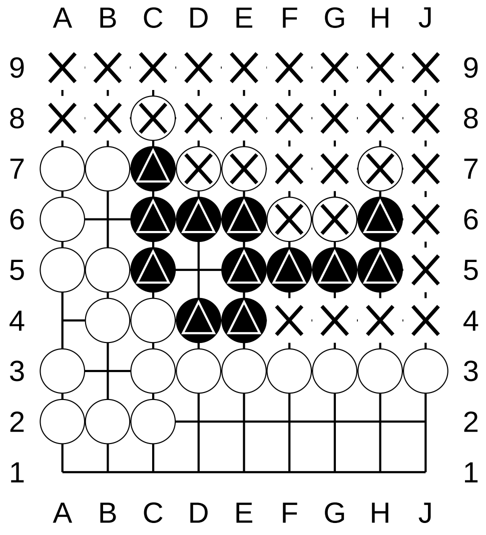


Figure 11. Black to play and live: A currently unsolved problem.

8. Conclusions and Future Work

The early results of our work on applying df-pn to Go and specifically to the one-eye problem are very encouraging. There are numerous possible enhancements, both for improving the search algorithm and for adding Go-specific knowledge. Examples are recognizing larger eyes, refining the knowledge about connections, generalizing forced moves similar to Cazenave's APS, heuristic initialization of proof and disproof numbers, and search in open-ended areas.

To apply these ideas to other problems in Go is also an interesting research topic. Examples include full tsume-Go (two-eye problems), tactical capture search and connection search.

8.1 Comparison with related Programs

We would like to compare our program with general tsume-Go solvers to assess its performance. However, it is hard to make a fair comparison since our algorithm solves only a restricted problem. Evaluation for two eyes is much harder than for one eye, and many years of hard work have gone into the development of the Go knowledge in programs such as GoTools. However, we believe that as a search algorithm our modified df-pn works very well for Go. In informal experiments it seems that our algorithm can already solve harder problems in our test set than other programs. One possible advantage of the df-pn algorithm is that it uses the transposition table more extensively. Only solved positions are saved in the transposition table in GoTools (Wolf, 2000), while in df-pn proof and disproof numbers of previous iterations are stored in the transposition table to improve the order of tree expansion (Nagai, 2002).

8.2 The GHI Problem in Df-pn

So far in this paper, we have not addressed the graph history interaction (GHI) problem (Palay, 1985). This problem occurred in our experiments, for example in double or triple ko situations. If GHI is ignored, incorrect results are stored in the transposition table. We developed a new approach that differs from the one described in Breuker et al. (2001) for the case of proof-number search. The method will be described in a forthcoming publication (Kishimoto and Müller, 2003).

Acknowledgments

Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Alberta Informatics Circle of Research Excellence (iCORE).

References

Allis, L. V., van der Meulen, M., and van den Herik, H. J. (1994). Proof-number search. *Artificial Intelligence*, 66(1):91–124.

Breuker, D. M., van den Herik, H. J., Uiterwijk, J. W. H. M., and Allis, L. V. (2001). A solution to the GHI problem for best-first search. *Theoretical Computer Science*, 252(1-2):121–149.

Cazenave, T. (2002). Abstract proof search. In Marsland, T. A. and Frank, I., editors, *Computers and Games (CG 2000)*, volume 2063 of *Lecture Notes in Computer Science*, pages 39–54. Springer.

Chen, K. and Chen, Z. (1999). Static analysis of life and death in the game of Go. *Information Sciences*, 121:113–134.

Fotland, D. (2002). Static eye analysis in "The Many Faces of Go". *ICGA Journal*, 25(4):203–210.

Kawano, Y. (1996). Using similar positions to search game trees. In Nowakowski, R. J., editor, *Games of No Chance*, volume 29 of *MSRI Publications*, pages 193–202. Cambridge University Press.

Kishimoto, A. and Müller, M. (2003). A solution to the GHI problem for depth-first proof-number search. Manuscript in preparation.

Korf, R. E. (1985). Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109.

Korf, R. E. (1993). Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78.

Kraszek, J. (1988). Heuristics in the life and death algorithm of a Go-playing program. In *Computer Go*, volume 9, pages 13–24.

Müller, M. (1997). Playing it safe: Recognizing secure territories in computer Go by using static rules and search. In Matsubara, H., editor, *Game Programming Workshop in Japan '97*, pages 80–86, Tokyo, Japan. Computer Shogi Association.

Nagai, A. (1999). A new depth-first search algorithm for AND/OR trees. Master's thesis, Department of Information Science, University of Tokyo.

Nagai, A. (2002). *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, Department of Information Science, University of Tokyo.

Palay, A. J. (1985). *Searching with Probabilities*. PhD thesis, Boston University.

Sakuta, M. (2001). *Deterministic Solving of Problems with Uncertainty*. PhD thesis, Department of Science and Engineering, Shizuoka University.

Seo, M. (1995). The C^* algorithm for AND/OR tree search and its application to a tsume-shogi program. Master's thesis, Department of Information Science, University of Tokyo.

Tanase, Y. (2000). Algorithms in ISshogi. In Matsubara, H., editor, *Advances in Computer Shogi 3*, pages 1–14. Kyouritsu Shuppan Press. In Japanese.

Wolf, T. (1994). The program GoTools and its computer-generated tsume Go database. In Matsubara, H., editor, *Game Programming Workshop in Japan '94*, pages 84–96, Tokyo, Japan. Computer Shogi Association.

Wolf, T. (2000). Forward pruning and other heuristic search techniques in tsume Go. *Information Sciences*, 122(1):59–76.

Zobrist, A. L. (1970). A new hashing method with applications for game playing. Technical report, Department of Computer Science, University of Wisconsin, Madison, WI. Republished (1990) in *ICCA Journal*, 13(2): 69-73.