Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

Information Sciences 175 (2005) 296–314

INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL

www.elsevier.com/locate/ins

A solution to the GHI problem for depth-first proof-number search

Akihiro Kishimoto *, Martin Müller

Department of Computing Science, University of Alberta, Edmonton, Canada T6G 2E8

Accepted 26 April 2004

Abstract

The Graph–History interaction (GHI) problem is a notorious problem that causes game-playing programs to occasionally return incorrect solutions. This paper presents a practical method to cure the GHI problem for the case of the df-pn algorithm. Results in the game of Go with the situational super-ko rule show that the overhead incurred by our method is very small, while correctness is always guaranteed.
© 2004 Elsevier Inc. All rights reserved.

1. Introduction

1.1. Motivation

Developing high performance game-playing programs has been the subject of AI research for over 50 years. Game-playing programs typically employ lookahead search to improve their move decisions. Efficient search algorithms improve the strength of their programs. For example, Thompson showed that

* Corresponding author.
E-mail addresses: kishi@cs.ualberta.ca (A. Kishimoto), mmueller@cs.ualberta.ca (M. Müller).

0020-0255/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2004.04.012

there is a strong positive correlation between the explored depth of the search tree and the strength of a chess-playing system [12]. Therefore, programmers have invested a large amount of resources to enhance their search engines.

One of the most valuable search enhancements is the *transposition table*, a large cache that keeps results of previous search efforts. A program can reach the same game state via different paths—a so-called *transposition*. If the previously cached position is explored deeply enough, the search algorithm does not need to explore the position again, thus saving considerable search effort. However, if the search space includes cycles, cached results may be flawed because they ignore the path used to reach the position. This is the so-called Graph–History Interaction (GHI) problem [9]. Programmers either ignore the GHI problem, since they do not want to degrade the performance of their programs, or reduce the number of recognized transpositions in order to guarantee correctness.

1.2. Description of the GHI problem

With the help of Fig. 1 we explain the GHI problem for AND/OR trees. There are two scenarios in which the GHI problem can occur, depending on the rules of the game.

In the first scenario, which we call *first-player-loss*, a repetition is a loss for the first player (the player to play at the root node). Examples are checkmating problems in chess and shogi (tsume-shogi), since a repetition does not help the first player who is trying to checkmate. Assume *D* in the figure is a loss for the first player, and this result is stored in the transposition table. Let *G* be a win for the first player. Then searching *A* in the following order leads to the wrong result:

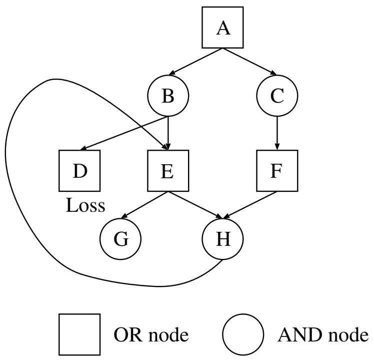


Fig. 1. The GHI problem.

1. Search $A \rightarrow B \rightarrow E \rightarrow H \rightarrow E$, then a loss is stored in the table entry for H, because the position repetition cannot be avoided.
2. Search $A \rightarrow B \rightarrow D$, then a loss is stored for the AND node B.
3. Expand $A \rightarrow C \rightarrow F \rightarrow H$, then a table look-up for H retrieves a loss which is backed up to F and C.
4. A is now incorrectly labeled as a loss because losses are stored for both successors B and C. However, A is a win by the sequence $A \rightarrow C \rightarrow F \rightarrow H \rightarrow E \rightarrow G$.

Let a *proof* be a proven win, and a *disproof* be a proven loss. In tsume-shogi, a repetition is considered to be a loss for the attacker (the first player). The GHI problem only affects disproofs. Tsume-shogi programs deal with the GHI problem by not caching disproofs caused by repetitions. That way, transpositions are available for most positions while the correctness of proofs is still guaranteed. The drawback is that the programs are unable to disprove positions involving repetitions. However, in Shogi, that is of lesser practical importance than proving a position.

The other scenario for GHI, which we call *current-player-loss* occurs when a repetition is defined to be a loss for the player who repeats a position. For instance, the situational super-ko (SSK) rule in Go declares that any move that repeats a previous board position is illegal. In this scenario, using a transposition table can lead to errors in both ways: it can change a loss into a win or a win into a loss. For example, in Fig. 1, now assume that G is a loss for the player to move at the root:

1. Searching $A \rightarrow B \rightarrow E \rightarrow H$, H is stored as a win because the opponent does not have a legal move at H.
2. Searching $A \rightarrow C \rightarrow F \rightarrow H$ backs up a win to C, based on the win stored for H.
3. A is now incorrectly regarded as a win since C's table entry shows a win. However, A is a losing position, since the sequences $A \rightarrow B \rightarrow D$, $A \rightarrow C \rightarrow F \rightarrow H \rightarrow E \rightarrow G$, and $A \rightarrow C \rightarrow F \rightarrow H \rightarrow E \rightarrow H$ all lose.

This scenario does not occur in tsume-shogi. However, van der Werf et al. [13] pointed out that when using the SSK rule in Go, this scenario can lead to invalid proofs. In [13] the problem was avoided by storing a separate hash entry for each path leading to a node. Unfortunately, this resulted in over 1000 times larger searches when solving 4×4 Go.

1.3. A new approach to the GHI problem

This paper presents our solution to the GHI problem for the case of df-pn search [8]. We believe that our approach is general, and applicable to other game tree search algorithms. We are currently working on an implementation for $\alpha\beta$ search. The df-pn algorithm combines properties of depth-first and best-first search. Since GHI occurs more frequently in best-first search than in depth-first search [4], we need to address this problem.

The GHI problem is caused by ignoring paths if a node is either proven or disproven. To solve the problem, we add path information to the transposition table whenever a node n is (dis)proven via a path p. If n is later reached via a different path q, the (dis)proof for n is not used blindly. Instead, an efficient additional search is performed to check if the (dis)proof of n via p also works for n via q. If n is neither proven nor disproven yet, the table entry for n is always used as a transposition.

This approach requires information on paths as well as extra memory and additional searches. However, we will show that the overhead can be minimized. Our experiments in Go with the SSK rule, a current-player-loss-scenario, indicate that the overhead incurred by our solution is a very small price to pay for guaranteeing correct solutions.

1.4. Outline

The structure of this paper is as follows:

Section 2 reviews the literature on the df-pn algorithm and the GHI problem. Section 3 describes our solution to the GHI problem. Section 4 discusses the experimental results on Go. Section 5 concludes this paper and gives future work on this research topic.

2. Literature review

2.1. The depth-first proof-number search algorithm

Nagai's df-pn (depth-first proof-number) algorithm [8] turns Allis' proof-number search (PNS) [1] into a depth-first search 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 uses proof and disproof numbers and always expands a most-proving node.

Fig. 2, adapted from [8], presents pseudo-code of the df-pn algorithm. The code is written using the ϕ and δ notation for proof and disproof numbers. Let **pn**(n) be the proof number and **dn**(n) be the disproof number of node n. Then

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

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

```
// 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 ($n.\phi \le \phi$ || $n.\delta \le \delta$) {
    // Exceed thresholds
    $n.\phi = \phi$; $n.\delta = \delta$;
    return;
  }
  // Terminal node
  if (IsTerminal($n$)) {
    if (WinForCurrentNode($n$)) {
      $n.\phi = 0$; $n.\delta = \infty$;
    } else{
      $n.\phi = \infty$; $n.\delta = 0$;
    }
    TTstore($n$,$n.\phi$,$n.\delta$);
    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 $sum$;
}
```

Fig. 2. Pseudo-code of the df-pn algorithm.

Df-pn utilizes iterative deepening with local thresholds for both proof and disproof numbers. This approach is similar to recursive best-first search in single-agent search [7]. For details, see [8].

Because df-pn is an iterative deepening method that expands interior nodes again and again, the heart of the algorithm is its use of the transposition table. In Fig. 2, *TTstore* stores proof and disproof numbers of a node in the table.

TTlookup tries to retrieve proof and disproof numbers of a node from the table. If no entry is found, both numbers are initialized to 1.

2.2. Previous work on the GHI problem

2.2.1. Palay's suggestions

Palay first pointed out the GHI problem and suggested two solutions [9]. The first solution is to refrain from using transpositions. Van der Werf used this approach to solve 4×4 Go with the SSK rule [13]. The drawbacks are a large number of expansions of duplicated nodes and large space requirements. Palay's second solution is to continue using a graph representation but attempt to recognize the GHI problem. When GHI is recognized, the nodes that are on the path from a repeated node to the nodes having more than one parent are duplicated to be able to store different results for these duplicated nodes. However, Palay did not implement this strategy, since GHI did not occur so frequently in his tests. He conjectured that the second solution would take additional time since the graph must be revised occasionally.

2.2.2. Campbell's analysis

Campbell partially solved the GHI problem for $\alpha\beta$ search [4]. In his algorithm, each transposition table entry contains a field that stores the depth searched below a node. If a transposition is recognized and the depth stored in the table entry is at least as deep as the depth that must be explored, the table information is retrieved and no further search for that node is performed. Campbell classified the GHI problem into two cases, *draw-first* and *draw-last*. Fig. 3, adapted from [4], illustrates an example. In the *draw-first* case, path (1) is explored first and a score is stored in the table entry for X. Then if Y is

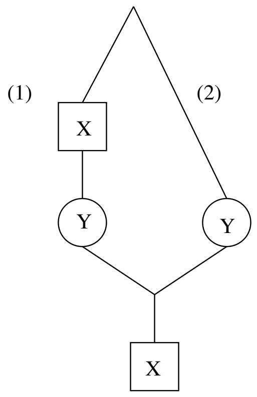


Fig. 3. Prototypical case of GHI.

searched via path (2), Y might be incorrectly computed because of the table entry for X. In the *draw-last* case, (2) is explored first and Y's incorrect score is used when reaching Y via (1). This happens because of the implementation of the transposition table. When reaching X via (1), the search depth for X via (2) is shallower than via (1). Therefore, X via (1) must be explored. On the other hand, if Y is reached via (2) first, Y is already explored deeply enough via (1) to reuse the table information on Y.

Campbell noted that draw-first GHI is curable by not storing scores that might cause the GHI problem, while draw-last is incurable. However, in practice most GHI problems can be avoided by combining the $\alpha\beta$ algorithm with iterative deepening. Experimentally, the GHI problem appears much less frequently in iterative deepening $\alpha\beta$ search than in best-first search.

2.2.3. Breuker's base-twin algorithm

Breuker et al. proposed the *base-twin algorithm (BTA)* for solving the GHI problem in proof-number search [3]. BTA is described for a 3-valued evaluation model with values *win*, *loss*, and *draw*. If a draw is considered a disproof as in their experiments, this model is the same as the first-player-loss scenario in our framework.

BTA uses a *possible-draw* mark combined with the depth of a node to recognize repetitions. To find out which level of the node causes repetitions, BTA utilizes two kinds of nodes: a *base* node to be explored and *twin* nodes that have different parents, link to their base node, but are not explored. When more than one path reaches identical positions, these positions are not represented by a single node, but split into one base node and one or more twin nodes, which can have different values (i.e. possible-draw marks) than the base node. Whenever a most-proving node is selected, BTA also checks if possible-draw marks can be stored by recognizing repetitions. Possible-draw marks are passed back to parents and when the root of the subtree that causes repetitions is detected, then a real draw is stored in that root. See [3] for details of the algorithm.

Although Breuker et al. claim that BTA is a general solution to the GHI problem for best-first search, there are three issues that must be addressed:

1. Since BTA was implemented for a best-first search algorithm that keeps an explicit graph in memory, it is an open question if BTA is applicable to depth-first search algorithms with limited memory. They concluded in [3] "What remains is solving the GHI problem for depth-first search. This will need a different approach, storing additional information in transposition tables rather than in the search tree/graph in memory. However, Campbell already noted that in depth-first search the frequency of GHI problems is considerably smaller than in best-first search [4]. The solution of the GHI problem for depth-first search remains a nearly theoretical exercise".

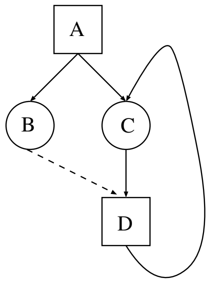


Fig. 4. An example where BTA fails with the current-player-loss scenario.

However, curing the GHI problem for depth-first search algorithms using proof and disproof numbers is necessary, because these algorithms combine properties of best-first search and depth-first search, and GHI occurs more frequently than in iterative deepening $\alpha\beta$.

2. The cycle detection scheme in BTA does not work with the current-player-loss scenario. Fig. 4 illustrates an example. Assume that B has not been expanded, and it is currently unknown whether B has child D. Then assume that BTA explores $A \rightarrow C \rightarrow D \rightarrow C$. Since this repetition occurs at C, a score (disproof in this case) is saved in C's transposition table entry without any condition. Then, assume that path $A \rightarrow B \rightarrow D$ is searched, and recognizes D as a child of B. Next, BTA reaches C by expanding D. In BTA, C's entry has a disproof and a disproof is retrieved from the entry. However, this is incorrect, since the *second* player cannot make a move at C if it is reached via $A \rightarrow B \rightarrow D \rightarrow C$.
3. All the possible-draw marks are removed for each iteration of proof-number search. The deletion of possible-draw marks is necessary in BTA since it is path-dependent information (see Appendix A for details). Therefore, as long as real draws are not stored, the nodes causing repetitions must be explored again and again to mark possible-draws, resulting in much tree expansion overhead.

2.2.4. Nagai's approach

Nagai proposed a solution to the GHI problem for df-pn [8]. He applied this modified df-pn to tsume-shogi problems (a first-player-loss scenario). In his algorithm, df-pn first sets large thresholds of proof and disproof numbers at the root. These thresholds are not ∞ as in the original df-pn, but $\infty - 1$. In case of a repetition, df-pn simply returns to the parent node without storing a disproof. If a proof for the root is found, the proof tree is guaranteed to not contain repetitions. However, if df-pn returns to the root by exceeding one of the large thresholds of proof and disproof numbers, df-pn re-searches with a threshold of ∞. If the root position is reached by a move, this move is now considered disproven. A similar process is performed for all interior nodes. If df-pn exceeds one of the large thresholds, it re-searches with a threshold of ∞, assuming that that node is a disproven position.

There are two drawbacks of Nagai's approach:

1. It may take a long time for the proof or disproof numbers to exceed the preset threshold. For example, if there is a large number of branches, the thresholds for expanding a node are much smaller than $\infty - 1$, since df-pn locally sets the thresholds based on the proof and disproof numbers of the children. Hence, because this approach has to wait for proof or disproof numbers of all the children to reach $\infty - 1$, it is impractical for detecting disproofs with repetitions. Nagai measured the ability of his tsume-shogi solver only with positions that can be proven. He did not measure the overhead incurred by this approach, or the performance in positions where a node must be disproven.
2. Nagai's approach also does not work with the current-player-loss scenario. Since this approach does not use any path information, it cannot store two different path-dependent results for one node. Again, Fig. 4 serves as an example. In this figure, D via $A \to B \to D$ is a proven node, since $A \to B \to D \to C \to D$ is not allowed. On the other hand, D via $A \to C \to D$ is a disproven node, since $A \to C \to D \to C$ is illegal. D's value cannot be determined without considering the path used.

2.2.5. Other related work

According to [3], Thompson noticed that his tactical chess analyzer suffered from the GHI problem. He cured it by using a DCG (directed cyclic graph) representation. When a node was expanded, his analyzer took the history into account to avoid returning an incorrect result. However, when leaf nodes were evaluated, the history was not considered, possibly resulting in incorrect evaluation values.

Baum and Smith suggested a solution to the GHI problem for their best-first search algorithm [2]. Their algorithm stores the whole DCG in memory to be able to check all ancestors and descendants of a node. Then, if node P spawns a child Q and Q has another parent P', their method checks if the ancestors of P and the descendants of Q contain P'. If this is the case, Q is split into two nodes to be able to store different results. However, this idea was not implemented, and they only concluded that a low storage algorithm would probably be too costly.

Schijf et al. investigated proof-number search in domains where the search graphs are DAGs (directed acyclic graphs) and DCGs [10]. They observe that in practice it is not necessary to compute proof and disproof numbers correctly for DAGs, as long as correct results are returned. Three algorithms for DCGs are presented:

1. The *tree method* does not use transpositions, which has the disadvantage of not reusing results, while correctness is always guaranteed.
2. In the *DAG method*, two classes of moves are defined: *conversion* moves are irreversible and *non-conversion* moves that may be reversible. The DAG method maps identical positions to a single node for conversion moves, while identical positions reached by non-conversion moves are treated as different nodes. This approach is also applied in the solution of 5×5 Go with Japanese and Chinese rules by van der Werf et al. [13]. This approach can cure the GHI problem, but a disadvantage is that duplicated searches are performed for all nodes with non-conversion moves.
3. The *DCG method* maps identical nodes to a single node unless a cycle is created. If a repetition is detected, a node creating a cycle is mapped to a second node and recognized as a disproven leaf node. Identical positions are mapped to at most two nodes. Although the DCG method is shown to be effective in their experiments in chess, as pointed out in [10] this approach sometimes results in wrong disproofs.

3. A new solution to the GHI problem

The outline of our solution to the GHI problem is as follows: When a proven or disproven position stored in the transposition table is reached via a new path, instead of blindly retrieving the result, a search is performed to verify the result. If the proof/disproof verifies, the result can be safely reused; otherwise the transposition table entry is treated as a different position. Kawano's simulation technique [5] is used to reduce the search overhead. For efficiency, this approach requires a good scheme for storing and comparing paths, and a technique for minimizing the number of simulation calls. We provide details below.

3.1. Duplicating transposition table entries

Since we want to reuse the results of previous search efforts, unproven identical positions reached via different paths are considered to be transpositions, and we reuse the proof and disproof numbers from the transposition table. When position A is proven via path p, the transposition table entry for A is split into a *base* and a first *twin* table entry. A proof is stored in the twin table entry to indicate that A is proven when reaching A via p. The proof and disproof numbers in the base table entry are re-initialized to 1. If A is proven via path q, another twin table entry for q is created and a proof is stored in this twin table entry. When reaching A via a path other than p, the proofs of the twin table entries are simulated (see Section 3.3). If at least one verifies that proof is used; otherwise the proof and disproof numbers from the unproven base table entry are used in the search. Disproofs are handled in the same way.

3.2. Encoding paths

Since we must differentiate between identical positions reached via different paths, we need an effective method to compute a signature of a path. A variant of the Zobrist function, which is used to hash a position into its corresponding transposition table key [14], can also be used to encode a path. In our implementation, each transposition table entry contains an additional 64-bit field to encode a signature of the path from the root to a position. Let *MaxMove* be the number of different moves in a game, and *MaxDepth* be the maximum search depth. A random table with *MaxMove* × *MaxDepth* 64 bit integers is prepared to encode a path. The sequence of moves to reach that position is encoded by a similar technique to Zobrist's method using that random table. Let R be the random table, and the path p be $(m_1, m_2, \ldots, m_k)$, where m_i are moves. p is encoded as follows:

$$R[m_1][1] \oplus R[m_2][2] \oplus \cdots \oplus R[m_k][k]$$

An important property of this path-encoding scheme is that the order of moves is not commutative, since the random table entries for identical moves with different depths contain different values. For example, two paths $p_1 = (m_1, m_2, m_3)$ and $p_2 = (m_3, m_2, m_1)$ are encoded to different 64 bit integers, because $R[m_1][1] \oplus R[m_2][2] \oplus R[m_3][3]$ (for p_1) is different from $R[m_1][3] \oplus R[m_2][2] \oplus R[m_3][1]$ (for p_2).

We note that the size of the random table is small enough for state-of-the-art machines. In our experiments on 19 × 19 Go, we set *MaxMove* = 362 and *MaxDepth* = 50. The memory required for this random table was about 140 KB. In games with a large number of different moves, such as Shogi or Amazons, a move can be split into two or three partial moves, for example by separating the from-square information from the to-square information. This way *MaxMove* can be greatly reduced, while *MaxDepth* increases by a factor of 2 or 3.

3.3. Invoking simulation for correctness

Tree *simulation* was invented by Kawano to effectively deal with useless interposing piece drops in tsume-shogi [5]. Later, Tanase extensively applied this idea in his $\alpha\beta$ search engine to reduce the overhead of calling the tsume-shogi solver within the normal search [11].

In AND/OR trees, a *proof tree* T provides a proof that a node n is proven. Such a proof tree has the following properties:

1. n is contained in T.
2. For each interior OR node of T, at least one child is contained in T.
3. For each interior AND node of T, all children are contained in T.
4. All terminal nodes of T are proven.

Assume that P is a proven node and Q is a "similar" one that we want to prove. Simulation borrows moves from P's proof tree to try to find a quick proof of Q. A proof is obtained from the transposition table by retrieving the winning move for each OR node in the proof tree of P. A dual approach, called *dual simulation*, is used to find a disproof.

Compared to a normal search, simulation requires much less effort to confirm whether a position is proven or not. An existing proof tree is typically much smaller than a new search tree would be. Also, since moves are borrowed from the transposition table at OR nodes, there is no need to invoke the move generator. We must only check the legality of the moves, which is a much faster operation. Furthermore, while df-pn must look up all children in the transposition table at every node, simulation does not require transposition table lookups for children.

Assume that A is a proven position with path p. If we reach A by a different path q, we can check if A via q can be proven by invoking simulation. A proof is borrowed from the twin table entry (with path p). If a proof for A via q is verified, an additional twin table entry for A via q is created and the proof is saved. If more than one twin table entry is available, they are tried one after another. However, since most of the proof trees have the same shape, more than one tree simulation happens only rarely. An analogous verification is tried for a disproven position by invoking dual simulation.

3.4. Reducing simulation calls

Since simulation incurs extra overhead to assess the correctness of a transposition table entry, we devised a method to reduce the number of simulation calls. If a node is (dis)proven without detecting a repetition, that node can always be used as a transposition, since it is independent of the path that df-pn takes. In this case, the (dis)proof is stored directly in the base table entry, without creating a twin node. If another path leads to that position, a (dis)proof can be retrieved directly. Therefore, our scheme first looks at the path-independent base table entry for a position, and if the (dis)proof is found, that position is considered to be (dis)proven. Otherwise, the twin table entry is retrieved if available.

3.5. Other implementation details

In the pseudo-code of the df-pn algorithm (see Fig. 2 again), the thresholds of proof and disproof numbers are set to ∞ at the root. This causes the GHI problem, since df-pn saves the thresholds in the transposition table before expanding a node. For example, if a node explored with the threshold of the proof number (**pn**) of ∞ has only one child A, A is explored with the threshold **pn** = ∞. Before expanding A, df-pn stores **pn** = ∞ in A's table entry. This is wrong, since A is not always disproven. There may be another path leading to A without involving repetitions, resulting in a proof. Following Nagai's approach [8], we initialize both thresholds at the root to $\infty - 1$ to avoid the above case. If our df-pn algorithm returns a proof number of 0 and a disproof number of ∞, or vice versa, it has found a correct (dis)proof. Otherwise, df-pn returns the value *unknown*.

3.6. Correctness of our solution

Assume that all proven and disproven nodes are stored in the transposition table. Although our proposed solution might compute incorrect proof and disproof numbers for unproven nodes, the following theorems guarantee correctness of the solutions:

Theorem 3.1. *Our solution does not suffer from the draw-first case.*

Proof. This theorem is proven with the help of Fig. 3. Although X is an OR node and Y is X's child in Fig. 3, the only assumption is that X is Y's ancestor and also Y's descendant. In the draw-first scenario, if proving Y via (1) involves repetitions related to X, X and Y are stored in the transposition table with "via path (1)". Hence, if Y is reached via (2), a search is performed below Y and X in our algorithm. The case of disproofs is similar. Thus the draw-first scenario does not happen in our solution. □

Theorem 3.2. *Our solution does not suffer from the draw-last case.*

Proof. Again, in Fig. 3 with the same assumption in Theorem 3.1, assume that path (2) is explored and X's proof is saved in the transposition table. We have to consider the following cases:

1. If X via (2) is proven without repetitions, Y is not included in X's proof tree. When X is reached via (1), a proof is immediately retrieved from X's table entry and this is a correct proof, since it does not contain any repetitions. Y is never explored via (1), which would cause the draw-last case.
2. If X via (2) is proven with repetitions:
   (a) If Y via (2) is proven without repetitions, X's proof tree must not be a part of Y's proof tree. When reaching X via (1), X's proof via (2) is not retrieved because it is stored in the twin table entry via (2). Then, when reaching Y via (1), Y's table entry is retrieved. Because Y's proof

tree does not contain any repetitions such as $Y \rightarrow X \rightarrow Y$, Y's proof tree can be reused.

(b) If Y via (2) is proven with repetitions, neither X's nor Y's proof via (2) is retrieved at X and Y via (1) in our algorithm. Hence, X and Y are explored, guaranteeing a correct result.

The case of disproofs is similar. Thus we guarantee that the draw-last case never happens in our solution. □

4. Experimental results

4.1. Setup of experiments

We implemented a solver for the one-eye problem in Go, a restricted version of the tsume-Go problem. We use the situational super-ko (SSK) rule, a current-player-loss scenario. The task of the one-eye problem is to (dis)prove that one player can make an eye in an enclosed region. Our current set of 70 test positions was created mainly by the authors. The problems can be played for both colors going first, resulting in a total of 140 problems. The test set is available at http://www.cs.ualberta.ca/~games/go/oneeye.

To measure the effectiveness of our new method, four versions of the solver were implemented: the first version ignores the GHI problem. The other versions employ our new scheme with different numbers of optimizations. The experiments were measured on an Athlon 2400MP with a 200 MB transposition table. All proven and disproven table entries are kept in memory in our experiments. The game-specific and game-independent search enhancements described in [6] are used by both solvers. The time limit was set to 5 min per problem.

The following abbreviations for the methods are used:

- **DUP** Duplicate transposition table entries. This guarantees correctness of the solutions.
- **SIM** Invoke simulation and dual simulation to reduce the overhead of checking correctness of (dis)proven nodes.
- **NOCYCLE** Reduce simulation calls by detecting (dis)proofs that do not depend on cycles.

4.2. Results

Table 1 summarizes the results in terms of the number of problems solved, total node expansions, total node expansions by simulation, the number of simulation calls, and the number of failed simulation calls. These statistics are col-

Table 1
Performance comparison between ignoring and dealing with the GHI problem

	Method used	Number of problems solved	Total nodes	Nodes by **SIM**	**SIM** calls	Failed **SIM** calls	Total time (s)
IGNORE-GHI		134 + 2	3,614,539	–	–	–	81
(a)	**DUP**	134	9,411,063	–	–	–	256
(b)	(a) + **SIM**	134	7,015,856	2,699,556	239,820	2956	175
(c)	(b) + **NOCYCLE**	136	3,689,363	2813	356	156	82

All statistics are computed for the subset of 134 problems solved by all program versions.

lected for the subset of 134 problems for which all program versions gave solutions.

Both IGNORE-GHI and version (c) solved 136 problems. However, IGNORE-GHI gave incorrect proof trees for two of the test positions. All the problems solved by **DUP** were also solved by the other three versions. Furthermore, both test positions in which IGNORE-GHI returned incorrect proofs were solved correctly by the other three versions.

Invoking simulation and reducing simulation calls in case of transpositions greatly improves the performance of the checking process. In particular, reducing simulation calls reduced the total node count by the factor of two, resulting in two more problems solved within the time limit. We conclude that it is worth adding our method to df-pn search with the following arguments:

- GHI can be checked with negligible overhead, both in terms of extra nodes and execution time. It is worth paying this price to always guarantee correctness.
- Even if GHI does not appear in the final proof tree, it appears in the search occasionally. In Table 1, simulation caught 156 flawed transposition table entries. This number is conservative, because some incorrect proofs or disproofs may be stored but never retrieved.

4.3. A real example in Go with the SSK rule

We found a position that suffered from the GHI problem with the SSK rule when we implemented the solver that ignores GHI (see Figs. 5 and 6(a)). First the solver explored the position by tracing the move sequence in Fig. 6(b). Path **C**7 → (1) → (3) → **A**8 → (4) in Fig. 5 corresponds to this move sequence. Then, Black cannot play at **A8**, because this move leads back to the position after move 4, shown in Fig. 6(c). Based on this result, a win for White is saved in the table entry for position (c). However, after the sequence in Fig. 6(d), position (c) is no longer a win for White. When the solver traced move sequence (d), it first reached (e) and then encountered (c). However, after the sequence

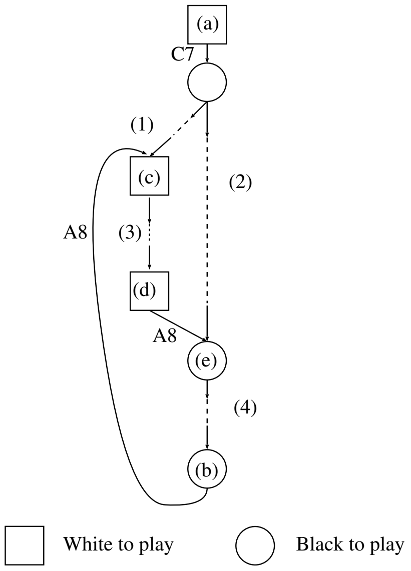


Fig. 5. Repetitions in the move sequences in Fig. 6.

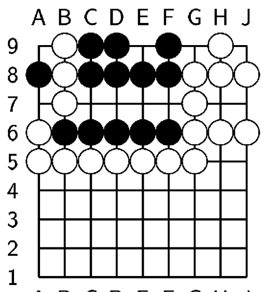
(d), White cannot play a move at **A8**, since it leads back to position (e). Path **C**7 → (2) → (4) → **A**8 → (3) in Fig. 5 stands for this sequence of moves. The correct result for position (c) via Fig. 6(d) is a win for Black.

We remark that White **15 at 7** in Fig. 6(b) would be better than **15 at 1**, and White **15 at 5** would be better in Fig. 6(d), because then White can win without repetition in both cases. However, even if adding more game-specific knowledge to the one-eye solver could reduce the number of such cases, there is no general way to always find a non-repetition proof first.

5. Conclusions

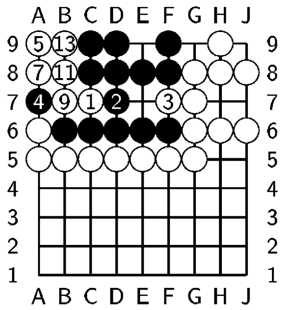
Since the overhead incurred by our new scheme is small, we conclude that it is a practical solution for curing the GHI problem in the df-pn algorithm. Furthermore, our solution is more general than previous approaches. The only previous solution for the current-player-loss scenario, giving up all transpositions, is very inefficient.

There are numerous topics to be investigated for future work. Since our approach can be applied to the first-player-loss scenario and also seems to be suited to proof-number search, it can be empirically compared with both Nagai's approach and Breuker's BTA. Besides, since we believe that our approach is so general that it can be applicable to other search algorithms such as $\alpha\beta$ search, implementing our approach in these algorithms will be a further



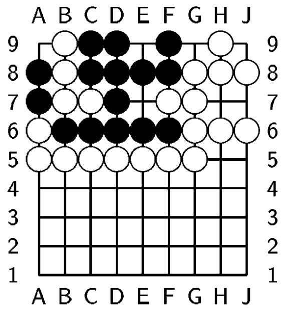
The original position: White to play.

(a)



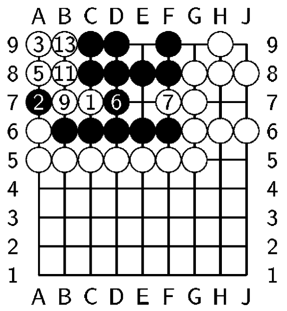
6 at 4, 8 at 4, 10 at 7, 12 at 5, 14 at 4, 15 at 1.

(b)



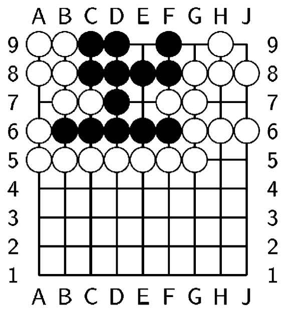
White to play after move 4.

(c)



4 at 2, 8 at 2, 10 at 5, 12 at 3, 14 at 2,
15 at 1, 16 at 5, 17 at 3, 18 at 2.

(d)



Black to play after move 7.

(e)

Fig. 6. An instance of the GHI problem in Go.

extension of this research. Finally, another interesting research direction is the relation between the GHI problem and replacement and garbage collection schemes in limited memory situations. Our algorithm as described in this paper keeps all proven and disproven nodes in memory. Which nodes can be replaced or garbage collected?

Acknowledgments

We would like to express our deepest appreciation to Jonathan Schaeffer and Yngvi Björnsson for valuable feedback. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Alberta Informatics Circle of Research Excellence (iCORE).

Appendix A. Removing possible-draw marks in BTA

This appendix describes the reason why BTA has to clear possible-draw marks every time it explores a most-proving node. Fig. 7 illustrates an example. Assume that the first-player-loss scenario is used in Fig. 7. If path $A \rightarrow B \rightarrow C \rightarrow D \rightarrow c$ is traced down, a possible-draw mark is stored at c. Since BTA keeps C's depth combined with the possible-draw mark, the depth of 2 is stored in c to indicate that C is the node involving a repetition. Then, if the possible-draw mark at c is not deleted and c is reached via $A \rightarrow B \rightarrow F \rightarrow D$ via $d \rightarrow c$, the mark at c is passed back to d (and F). As a result, a real draw is stored at F. However, this is incorrect, since F is a win via path $F \rightarrow D$ via $d \rightarrow C$ via $c \rightarrow E$.

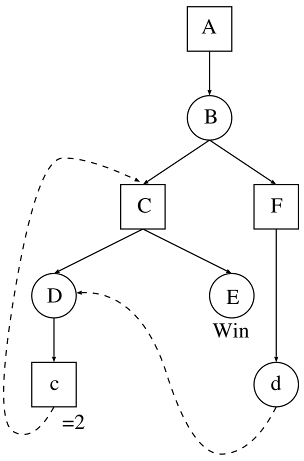


Fig. 7. An example showing why BTA must remove possible-draw marks.

References

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

[2] E.B. Baum, W.D. Smith, Best play for imperfect players and game tree search; part I—theory. Technical report, NEC Research Institute, 1995. Available at <http://citeseer.nj.nec.com/baum95best.html>.

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

[4] M. Campbell, The graph-history interaction: on ignoring position history, in: 1985 Association for Computing Machinery Annual Conference, 1985, pp. 278–280.

[5] Y. Kawano, Using similar positions to search game trees, in: R.J. Nowakowski (Ed.), Games of No Chance, MSRI Publications, vol. 29, Cambridge University Press, 1996, pp. 193–202.

[6] A. Kishimoto, M. Müller, Df-pn in Go: Application to the one-eye problem, in: Advances in Computer Games Many Games, Many Challenges, Kluwer Academic Publishers, 2003, pp. 125–141.

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

[8] A. Nagai, Df-pn Algorithm for Searching AND/OR Trees and its Applications, Ph.D. Thesis, Department of Information Science, University of Tokyo, 2002.

[9] A.J. Palay, Searching with Probabilities, Ph.D. Thesis, Carnegie Mellon University, 1983, Also published by Pitman, 1985.

[10] M. Schijf, L.V. Allis, J.W.H.M. Uiterwijk, Proof-number search and transpositions, International Computer Chess Association Journal 17 (2) (1994) 63–74.

[11] Y. Tanase, Algorithms in ISshogi, in: Hitoshi Matsubara (Ed.), Advances in Computer Shogi 3, Kyouritsu Shuppan Press, 2000, pp. 1–14 (in Japanese).

[12] K. Thompson, Computer chess strength, in: M. Clarke (Ed.), Advances in Computer Chess 3, Pergamon Press, 1982, pp. 55–56.

[13] E.C.D. van der Werf, H.J. van den Herik, J.W.H.M. Uiterwijk, Solving Go on small boards, International Computer Games Association Journal 26 (2) (2003) 92–107.

[14] A.L. Zobrist, A new hashing method with applications for game playing. Technical report, Department of Computer Science, University of Wisconsin, Madison, 1970.