A General Solution to the Graph History Interaction Problem

Akihiro Kishimoto and **Martin Müller**
Department of Computing Science, University of Alberta
Edmonton, Canada T6G 2E8
{kishi,mmueller}@cs.ualberta.ca

Abstract

Since the state space of most games is a directed graph, many game-playing systems detect repeated positions with a transposition table. This approach can reduce search effort by a large margin. However, it suffers from the so-called Graph History Interaction (GHI) problem, which causes errors in games containing repeated positions. This paper presents a practical solution to the GHI problem that combines and extends previous techniques. Because our scheme is general, it is applicable to different game tree search algorithms and to different domains. As demonstrated with the two algorithms $\alpha\beta$ and df-pn in the two games checkers and Go, our scheme incurs only a very small overhead, while guaranteeing the correctness of solutions.

Keywords: GHI problem, df-pn algorithm, $\alpha\beta$ algorithm, Kawano's simulation

Introduction

Heuristic search is an important topic in Artificial Intelligence. Search algorithms have many practical applications in areas such as theorem-proving, bio-informatics, and games. In particular, games have been regarded as useful test beds for search algorithms. Efficient search algorithms were shown to improve the strength of game-playing programs. Experiments in many different domains and with many different programs have shown a strong positive correlation between the depth of a search and the strength of a program. Therefore, programmers have invested a lot of effort to enhance the search performance of their programs.

One of the most valuable search enhancements is the *transposition table*, a large cache that keeps results of previous searches. A program can reach the same game state via a different path — a so called *transposition*. If the previously cached position was explored deeply enough, the search algorithm does not need to explore the position again. However, if the search space includes cycles, cached results may be unreliable because they ignore the path used to reach the position. This is the so-called GHI (Graph-History Interaction) problem (Palay 1983). In practice, so far programmers have either ignored the GHI problem, since they did not want to degrade the performance of their programs, or reduced the number of recognized transpositions in order to guarantee correctness. Our proposed solution completely solves the GHI problem with very small overhead.

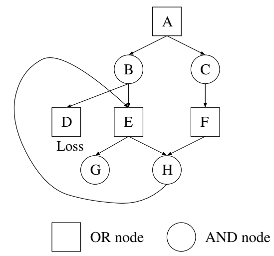

Figure 1: The GHI problem.

With the help of Figure 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 considered 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 a search starting from A in the following order leads to the wrong result:

1. Search $A \rightarrow B \rightarrow E \rightarrow H \rightarrow E$. A loss is stored in the table entry for H, because the position repetition cannot be avoided.
2. Search $A \rightarrow B \rightarrow D$. A loss is stored for AND node B.
3. Expand $A \rightarrow C \rightarrow F \rightarrow H$. 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 $A \rightarrow C \rightarrow F \rightarrow H \rightarrow E \rightarrow G$.

In the first-player-loss scenario, the GHI problem only causes invalid disproofs (first-player losses). Programs can avoid the GHI problem, accepting a loss of performance, by not storing any disproofs caused by repetitions.

The other scenario for GHI, which we call *current-player-loss*, occurs when a repetition is declared to be a loss for the player who repeats the 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 Figure 1, now assume that G is a loss for the player to move at the root:

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

This scenario does not occur in checkmating problems where only one player's king is under attack. However, (van der Werf, van den Herik, & Uiterwijk 2003) point out that when using the SSK rule in Go, this scenario can lead to invalid proofs. In their work the problem is avoided by storing a separate hash entry for each path leading to a node. Unfortunately, this resulted in over 1,000 times larger searches when solving Go on a 4×4 board.

Avoiding the GHI problem is crucial, especially when one wants to declare that games are solved by programs. Since even a single flawed transposition table entry can lead to a completely wrong solution, correct techniques must be devised.

This paper describes a uniform solution to both aforementioned scenarios of the GHI problem. Our approach synthesizes and extends existing techniques. Our solution always guarantees correctness if all proven and disproven nodes are saved in the transposition table. The games of checkers, a first-player-loss scenario, and Go with the SSK rule, a current-player-loss scenario, are chosen to empirically measure the effectiveness of our approach. Since our idea does not depend on any algorithm-specific features, it can be applied to different game-tree search algorithms. We have chosen to implement our scheme for both the df-pn algorithm (Nagai 2002) and $\alpha\beta$ (Knuth & Moore 1975). Experimental results in these domains and algorithms show only a very small overhead compared to programs that ignore the GHI problem. In particular, since the only previous solution for the current-player-loss scenario was to give up all transpositions, which is very inefficient, our approach is the first attempt to handle the GHI problem with small overhead. Additionally, we empirically demonstrate that the GHI problem must be addressed since it does occur in practice.

The structure of this paper is as follows: First, the literature on the GHI problem is reviewed and the algorithms df-pn and $\alpha\beta$ are briefly explained. Then our solution to the GHI problem is described, followed by experimental results with both algorithms in Go and checkers, and some conclusions and future work.

Previous Work on the GHI Problem

The GHI problem was first pointed out in (Palay 1983). Although two possible solutions were suggested, no implementation was provided. Campbell classified the GHI problem into two cases, *draw-first* and *draw-last*, and solved the GHI problem for the draw-first case (Campbell 1985). In the draw-first case, a score involving repetition is saved in the transposition table, and is later incorrectly retrieved for a position that does not involve repetition. In the draw-last scenario, a score not involving repetition is stored in the transposition table, and is later incorrectly used for a position involving repetition.

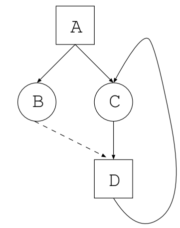

Figure 2: An example where BTA fails with the current-player-loss scenario.

(Breuker *et al.* 2001) proposed the *base-twin algorithm (BTA)* for solving the GHI problem in proof-number search. Since their implementation of BTA considered a draw to be a disproof, this model corresponds to the first-player-loss scenario in our framework. BTA uses a *possible-draw* mark combined with the depth of a node in the search graph to recognize repetitions. To find out at which depth a position causes repetitions, BTA splits repeated positions into two kinds of nodes: a *base* node to be explored and *twin* nodes which can have different values (i.e. possible-draw marks) than the base node. Possible-draw marks are propagated back to parents. When the root of the subtree that causes repetitions is detected, a real draw is stored in that root. 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:

- 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.
- The cycle detection scheme in BTA does not work with the current-player-loss scenario. Figure 2 illustrates an example. C is a node at the start of a repetition loop, but C's value can not be uniquely determined without considering the path. C via path $A \to C$ is a disproven node, since the last move in $A \to C \to D \to C$ is illegal. On the other hand, C via path $A \to B \to D \to C$ is a proven node, since after this sequence the move to D is a repetition for the opponent.
- All possible-draw marks are removed for each iteration of proof-number search. This is necessary in BTA since marks are path-dependent information. As long as a real draw is not stored, nodes causing repetitions must be explored again and again to mark possible-draws, resulting in a large overhead from tree reexpansion.

(Nagai 2002) proposes a solution to the GHI problem for df-pn. This modified df-pn is applied to tsume-shogi problems, a first-player-loss scenario. In Nagai's algorithm, df-pn first sets large thresholds of proof and disproof numbers at the root. 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 be repetition-free. However, if df-pn returns to the root by exceeding one of the large thresholds of proof and disproof numbers, df-pn re-searches by assuming that a move reaching the root is disproven. A similar repetition detection scheme is used at all interior nodes. One drawback of Nagai's approach is that it may take a long time for the proof or disproof numbers to exceed the preset threshold. If there is a large number of branches this approach is impractical for detecting disproofs with repetitions. Furthermore, Nagai's approach does not work with the current-player-loss scenario. Since this approach does not use any path information, it also cannot store two different path-dependent results for one node.

According to (Breuker *et al.* 2001), Thompson noticed that his tactical chess analyzer suffered from the GHI problem. He cured it for interior nodes by using a DCG (directed cyclic graph) representation and considering the history.

(Baum & Smith 1995) suggest a solution to the GHI problem for their best-first search algorithm. Their algorithm stores the whole DCG in memory and recognizes the case when a node reached through different paths must be split into two nodes to save different results. However, their idea was not implemented, and they concluded that a low storage algorithm would probably be too costly.

(Schijf, Allis, & Uiterwijk 1994) investigated proof-number search in domains where the search graph is a DAG (directed acyclic graph) or DCG. Schijf implemented three algorithms to deal with the GHI problem. However, two of these methods are inefficient, since they give up on using transpositions, while the third approach sometimes results in wrong disproofs.

Overview of df-pn and $\alpha\beta$

Depth-First Proof-Number Search

The depth-first proof-number algorithm df-pn (Nagai 2002) turns the best-first proof-number search (PNS) method (Allis, van der Meulen, & van den Herik 1994) into a depth-first search algorithm. 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. Df-pn utilizes local thresholds for both proof and disproof numbers, selects a most promising node, and performs iterative deepening until exceeding either one of the thresholds. Because df-pn is an iterative deepening method that re-expands interior nodes, the heart of the algorithm is its use of the transposition table. Whenever a node is explored, the transposition table is used to cache previous search efforts (i.e., proof and disproof numbers).

The $\alpha\beta$ Algorithm

The $\alpha\beta$ algorithm (Knuth & Moore 1975) has been the most popular algorithm among game programmers. The algorithm utilizes a search window defined by two bounds, α and β, which represent lower and upper bounds on the minimax score of a game tree. The search window is narrowed during minimax search, and used for pruning subtrees if the score of a node is proven to be outside the window. Many variants and enhancements have been developed over the years, but a transposition table is almost always used.

A New General Solution to the GHI Problem

Our solution utilizes two techniques: We encode path information using methods from (Zobrist 1970) and use Kawano's simulation technique (Kawano 1996) to search efficiently. 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 it. 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 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.

Duplicating Transposition Table Entries

In order to reuse the results of previous search efforts, unproven identical positions reached via different paths are considered to be transpositions. We reuse the values stored in the transposition table: proof and disproof numbers for df-pn, and minimax values for $\alpha\beta$. 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. If A is proven via a different path q, another twin table entry for q is created and the new proof is stored there. When reaching A via a path other than p, the proofs of the twin table entries are simulated (see later). If at least one verifies then that proof is used; otherwise the information from the unproven base table entry is used in the search. Disproofs are handled in the same way.

Encoding Paths

Identical positions reached via different paths can be differentiated by computing a signature of a path. A variant of the Zobrist function, which is used to hash a position into its corresponding transposition table key (Zobrist 1970), can 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 precomputed random table R with *MaxMove*$\times$*MaxDepth* 64 bit integers is prepared to encode a path. The sequence of moves to reach that position is encoded by a technique inspired by Zobrist's method. Let the path p be $(m_1, m_2, \cdots, m_k)$, where m_i are moves. Then p is encoded as follows:

$$\text{code}(p) = R[m_1][1] \oplus\ R[m_2][2] \oplus \cdots \oplus\ R[m_k][k]$$

An important property of this path-encoding scheme is that the order of moves is not commutative, since the random table entries for the same move played at different depths are different. For example, the codes of the two paths $p_1 = (m_1, m_2, m_3)$ and $p_2 = (m_3, m_2, m_1)$ are not the same, since code$(p_1) = R[m_1][1] \oplus R[m_2][2] \oplus R[m_3][3]$ is different from code$(p_2) = R[m_1][3] \oplus R[m_2][2] \oplus R[m_3][1]$.

We note that the size of the random table is small enough for current hardware. For example, in our experiments on 19×19 Go, setting *MaxMove* = 362 and *MaxDepth* = 50 the size is about 140KB. In games with a large number of different possible 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.

Invoking Simulation for Correctness

Tree *simulation* was invented by Kawano to effectively deal with useless interposing piece drops in tsume-shogi (Kawano 1996). Tanase applied this idea extensively in shogi, to reduce the overhead of calling the tsume-shogi solver within the normal $\alpha\beta$ search (Tanase 2000).

In AND/OR trees, which are the common concept on which both df-pn and $\alpha\beta$ are based, a *proof tree* T provides a proof that a node n is proven. Such a proof tree contains n, at least one child of each interior OR node of T, and all children of interior AND nodes of T. All terminal nodes of T must be proven. A *disproof tree* T which provides a disproof is defined in an analogous way.

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 attempt a quick proof of Q. The winning move for each OR node in the proof tree is obtained from the transposition table of the proof tree of P. Likewise, *dual simulation* attempts to find a disproof.

Compared to a normal search, simulation requires much less effort to confirm whether a position is proven or not. Even with good move ordering, a newly created search tree is typically much larger than an existing proof tree. Also, since moves are borrowed from the transposition table at OR nodes, there is no need to invoke the move generator there.

Assume that position A was proven via path p. If A is reached via a different path q, we can check if A via q can be proven quickly 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 the proof of A via q is created. If more than one twin table entry is available, they are tried one after another. However, since proof trees often have the same shape, it is rare that more than one tree simulation is needed. The analogous verification by dual simulation is tried to find disproofs.

Reducing Simulation Calls

Since simulation incurs an 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 taken by the search. 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, this (dis)proof can be reused.

Correctness of Our Solution

Assume that all proven and disproven nodes are stored in the transposition table. The following theorem guarantees correctness of the solutions:

Theorem 1 *Our solution suffers neither from the draw-first nor from the draw-last case.*

For unproven nodes, our proposed solution might compute incorrect proof and disproof numbers for df-pn, and incorrect heuristic values for $\alpha\beta$ search. However, Theorem 1 guarantees that (dis)proofs returned by our approach are always correct. This theorem is proven for the case of df-pn in (Kishimoto & Müller 2004), and can be analogously proven for $\alpha\beta$ with some modifications (see the next section).

Algorithm-Specific Implementation Details

Df-pn We made the following modifications to the original df-pn algorithm:

- Proof and disproof numbers in a base table entry are re-initialized to 1 whenever a (dis)proof is saved in a twin table entry. This is because df-pn tends to create large proof and disproof numbers before a (dis)proof is found, which made df-pn unable to solve some positions.
- As in (Nagai 2002), we initialize the thresholds of proof and disproof numbers at the root to $\infty - 1$, not ∞ as in the original df-pn algorithm. This is necessary to avoid the GHI problem at the root, since df-pn saves thresholds in the transposition table before expanding a node. If df-pn with our modification returns a proof number of 0 and a disproof number of ∞, or vice versa, it is a correct (dis)proof. Otherwise, df-pn returns the value *unknown*.

$\alpha\beta$ The following modifications are made:

- We modified the scheme for transposition table lookups. A normal transposition table entry contains a field that stores the depth searched below a node. If a transposition is recognized, the depth stored in the table entry is at least as deep as the depth that must be explored, and the table entry has a tight $\alpha\beta$ bound, then the table information is retrieved and no further search for that node is performed. We use this strategy only for unproven nodes. (Dis)proofs saved in the transposition table are always retrieved without checking the explored depth, since they are correct. This modification not only makes more use of the transposition table but also solves Campbell's draw-last case.
- Our current $\alpha\beta$ search uses only the three values (*win*, *unknown*, or *loss*). However, our solution works for the general case of more values in between *win* and *loss*. In our implementation of checkers, a draw is considered as a loss for the first player. To prove a draw, a second search must be performed in which a draw is regarded as a win for the first player. We note that determining a draw within a single search is not a trivial problem for the $\alpha\beta$ algorithm, since the values *draw* and *unknown* are incomparable. A

correct heuristic value can be obtained by performing a sequence of null window searches as in MTD(f) (Plaat *et al.* 1996).

Game-Specific Implementation Details

Go Domain-specific enhancements from (Kishimoto & Müller 2003) are incorporated in our df-pn and $\alpha\beta$ implementations. $\alpha\beta$ performs iterative deepening, extends the search for forced moves, and searches the best move from a previous iteration first.

Checkers 8-piece endgame databases are incorporated to our df-pn and $\alpha\beta$ implementations. Scores obtained by database lookups are considered to be correct, because these scores are path independent. Simulation is not invoked for trees involving only database scores. The $\alpha\beta$ implementation performs a variable depth-first search with state-of-the-art enhancements.

Experiments

Setup

We applied the df-pn and $\alpha\beta$ algorithms to Go and checkers. We focused on the one-eye problem with situational super-ko in Go, a current-player-loss scenario. Checkers is a first-player loss scenario.

The experiments for programs ignoring and dealing with the GHI problem for both games were performed on an Athlon 2400MP with a 300 MB transposition table. All proven and disproven nodes are saved in the transposition table in both programs. 140 positions in Go and 200 positions in checkers were prepared. The time limit was set to 5 minutes per position in Go. In checkers the execution time was unstable because it is dominated by I/O to access the databases. Therefore, we did not limit the execution time here. Instead, the checkers searches were limited to 10 million node expansions per position.

Results in Go

Tables 1 and 2 summarize the results for df-pn and $\alpha\beta$ in Go in terms of the number of problems solved and total node expansions. These statistics are collected from the two programs ignoring (IGNORE-GHI) and handling (HANDLE-GHI) the GHI problem. We could not test other approaches such as Nagai's in Go, since they do not handle the current-player-loss scenario. Both IGNORE-GHI and HANDLE-GHI solve the same subset of problems. However, IGNORE-GHI gave incorrect proof trees for two positions in df-pn and for three positions in $\alpha\beta$. Although the scores returned by IGNORE-GHI were correct, we conclude that it is important to have a scheme to handle the GHI problem, since GHI happens both in df-pn and $\alpha\beta$. Even if GHI does not appear in the final proof tree, it occasionally appears in the search. In the 136 problems solved, HANDLE-GHI in df-pn invoked simulation 648 times, explored 13,505 nodes by simulation, and discovered 190 flawed transposition table entries. In the 132 problems solved by $\alpha\beta$, HANDLE-GHI invoked simulation 12,005 times, explored 147,946 nodes by simulation, and detected 4,174 flawed entries. These numbers are conservative, because some incorrect proofs or disproofs may have been stored but never retrieved. The results show that HANDLE-GHI can avoid the GHI problem with negligible overhead in terms of expanded nodes and execution time. In our tests, HANDLE-GHI explored 2.5% extra nodes in $\alpha\beta$, and 1.5 % less nodes in df-pn. This is a very small price to pay for guaranteeing correctness.

Table 1: Performance comparison between ignoring and dealing with the GHI problem for df-pn in Go. All statistics are computed for 136 problems solved by both program versions.

Method Used	Number of problems solved	Total nodes	Total time (sec)
IGNORE-GHI	$134 + 2$	22,294,119	589
HANDLE-GHI	136	21,938,585	587

Table 2: Performance comparison for $\alpha\beta$ in Go. All the statistics are computed for 132 problems solved by both versions.

Method Used	Number of problems solved	Total nodes	Total time (sec)
IGNORE-GHI	129 + 3	102,077,944	1,078
HANDLE-GHI	132	104,679,229	1,101

Results in Checkers

Table 3 gives the results for df-pn in checkers. We additionally implemented Nagai's solution (NAGAI) to the GHI problem (Nagai 2002). Of the 200 problems in the data set, NAGAI solved 138. IGNORE-GHI solved 144, including the 138 which NAGAI solved. However, IGNORE-GHI had incorrect disproofs in 18 cases, whereas NAGAI solves all problems correctly. HANDLE-GHI solves 143 problems correctly, including all 138 problems solved by NAGAI, some additional problems solved by IGNORE-GHI, plus some problems not solved by either of the other two methods. HANDLE-GHI solved 2 problems which IGNORE-GHI did not solve. IGNORE-GHI 'solved' 3 problems which our scheme did not solve, but only one of those problems was solved correctly. In these experiments, our solution has no overhead. It even expands slightly fewer nodes than IGNORE-GHI. Simulation again detects flawed transposition table entries. In the 138 problems solved by all programs, HANDLE-GHI invoked simulation 243,885 times with 970,373 node expansions and discovered 87,181 flawed transposition table entries. These numbers confirm that the GHI problem occurs in search. Some incorrect results were contained in the final disproof trees.

Table 4 shows the results for $\alpha\beta$. HANDLE-GHI solved all positions solved by IGNORE-GHI, plus one more. IGNORE-GHI returned incorrect disproofs for 8 positions.

Table 3: Performance comparison for df-pn in checkers. All statistics are computed for the subset of 138 problems solved by all program versions.

Method Used	Problems solved	Total nodes
IGNORE-GHI	126 + 18	129,006,133
HANDLE-GHI	143	128,082,295
NAGAI	138	137,888,627

Table 4: Performance comparison for $\alpha\beta$ in checkers. All statistics are computed for the subset of 111 problems solved by both program versions.

Method Used	Problems solved	Total nodes
IGNORE-GHI	103 + 8	119,289,609
HANDLE-GHI	112	116,673,224

In the 111 problems solved by both programs, HANDLE-GHI invoked simulation 22,536 times with 31,170 node expansions and detected 14,418 flawed entries. These numbers indicate that the GHI problem occurs in $\alpha\beta$ in checkers, but a bit less frequently than in df-pn.

In conclusion, since the GHI problem happens both in df-pn and $\alpha\beta$ in checkers, it is dangerous to ignore. Because our method not only incurs low overhead but also always returns correct answers, it is a worthwhile addition to any search engine susceptible to GHI.

Our method could be compared with Breuker's BTA. However, we note that BTA needs an explicit graph representation, and complicated operations to deal with repetitions. This causes a problem when BTA uses up available memory. On the other hand, our approach does not need any explicit graph representation. Unproven nodes can be replaced when the table becomes full. Breuker's scheme to detect real draws is specific to the first-player-loss scenario, and could be added to our framework.

Conclusions and Future Work

In this paper, we presented a framework to solve an important open problem raised by (Palay 1983) 20 years ago. Our approach incurs very small overhead and is applicable to two algorithms df-pn and $\alpha\beta$. Therefore, we conclude that our solution to the GHI problem is practical and general.

An interesting topic for further consideration is the relation between the GHI problem with replacement and garbage collection schemes with limited memory. Since our algorithm currently needs to keep all proven and disproven nodes in memory, there is still an open question as to which nodes can be replaced or garbage-collected.

Acknowledgments

We would like to thank Jonathan Schaeffer and Yngvi Björnsson for beneficial discussions and making their checkers code available. Nathan Sturtevant and Markian Hlynka read drafts of the paper and gave valuable comments. 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.

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

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.

Campbell, M. 1985. The graph-history interaction: On ignoring position history. In *1985 Association for Computing Machinery Annual Conference*, 278–280.

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

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

Kishimoto, A., and Müller, M. 2004. A solution to the GHI problem for depth-first proof-number search. *Information Sciences*. To appear.

Knuth, D. E., and Moore, R. W. 1975. An analysis of alpha-beta pruning. *Artificial Intelligence* 6:293–326.

Nagai, A. 2002. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. Ph.D. Dissertation, Department of Information Science, University of Tokyo.

Palay, A. J. 1983. *Searching with Probabilities*. Ph.D. Dissertation, Carnegie Mellon University.

Plaat, A.; Schaeffer, J.; Pijls, W.; and de Bruin, A. 1996. Best-First Fixed-Depth Minimax Algorithms. *Artificial Intelligence* 87(1-2):255–293.

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

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

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

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