GAME-TREE SEARCH USING PROOF NUMBERS: THE FIRST TWENTY YEARS

Akihiro Kishimoto[1] *Mark H.M. Winands*[2] *Martin Müller*[3] *Jahn-Takeshi Saito*[4]

ABSTRACT

Solving games is a challenging and attractive task in the domain of Artificial Intelligence. Despite enormous progress, solving increasingly difficult games or game positions continues to pose hard technical challenges. Over the last twenty years, algorithms based on the concept of proof and disproof numbers have become dominating techniques for game solving. Prominent examples include solving the game of checkers to be a draw, and developing checkmate solvers for shogi, which can find mates that take over a thousand moves. This article provides an overview of the research on Proof-Number Search and its many variants and enhancements.

1. INTRODUCTION

Search plays a fundamental role for problem solving in Artificial Intelligence, with a wide range of applications including database systems, web mining, theorem proving and game-playing. In particular, much research has gone into inventing new search algorithms for game-playing programs, which are capable of defeating the best human players. These efforts have resulted in computers successfully outperforming the best humans in popular games such as Othello (Buro, 1997), checkers (Schaeffer, 1997), and chess (Campbell, Hoane Jr., and Hsu, 2002). These games are categorized as *two-player zero-sum games with perfect information* or *two-player games* in short in this article.

The $\alpha\beta$ algorithm (Knuth and Moore, 1975) has been the dominating search method in the domain of two-player games, until the more recent advent of Monte Carlo Tree Search (MCTS) (Coulom, 2007; Browne *et al.*, 2012) which has achieved remarkable success in games such as Go. $\alpha\beta$ relies on the minimax framework where the first player tries to maximize his or her advantage, while the second player tries to minimize it. Basic $\alpha\beta$ performs depth-first search with fixed depth, and calls an evaluation function at each leaf node which returns a numeric score that approximates the advantage of the first player. The algorithm backpropagates the best score to determine the next move to play in a given position P, and prunes subtrees that are irrelevant for computing the score of P by updating a lower bound α and an upper bound β on the score. In conjunction with good move ordering, $\alpha\beta$ pruning allows the algorithm to search much more deeply, resulting in significantly improved performance. Many variants and enhancements of $\alpha\beta$ have been developed over decades (see Marsland, 1986; Junghanns, 1998).

Developing super-human-strength programs has been of great interest to game researchers. Another big research challenge is to create solvers that solve games - determine the final result of the game with best play on both sides - or at least solve difficult positions within a game (van den Herik, Uiterwijk, and van Rijswijck, 2002). This task is often more challenging than "just" playing well. Given a position

[1]IBM Research, Ireland. Email: AKIHIROK@ie.ibm.com. This research was performed while the author was affiliated with Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology.

[2]Games and AI Group, Department of Knowledge Engineering, Faculty of Humanities and Sciences, Maastricht University, Maastricht, The Netherlands. Email: m.winands@maastrichtuniversity.nl

[3]Department of Computing Science, University of Alberta, Edmonton, Canada. Email: mmueller@ualberta.ca

[4]Berlin, Germany. Email: jahn.saito@gmail.com

P to solve, a solver must select a move that is *provably* best, while a tournament program only has to select a move that is *probably* best.

In principle, given an unlimited amount of time, $\alpha\beta$ can determine whether a position P of any finite game is a win or not, even on moderate hardware. For a given game position, assign a score of ∞ for a first player win, $-\infty$ for a terminal position that is not a win, and possibly other, heuristic values which approximate the winning probability of the first player in the undecided positions. Then exploring a game tree rooted at P deeply enough with $\alpha\beta$ will determine the score of P as either ∞ or $-\infty$. In practice, due to its property of being essentially a *fixed-depth search*, $\alpha\beta$ inherently suffers from the search tree growing exponentially with the search depth. This exponential growth limits the performance of $\alpha\beta$ search, even with all the effective enhancements such as iterative deepening (Slate and Atkin, 1977), move ordering (Akl and Newborn, 1977; Schaeffer, 1983), quiescence search (Beal, 1990), forward pruning (Donninger, 1993; Björnsson and Marsland, 2001), fractional-depth search based on realization probability (Tsuruoka, Yokoyama, and Chikayama, 2002), and search extensions (Anantharaman, Campbell, and Hsu, 1988; Campbell *et al.*, 2002; Tsuruoka *et al.*, 2002). Such techniques can greatly improve the effective branching factor and the search depth reached, but do not change the basic fact of exponential growth.

An implicit assumption in solvers based on $\alpha\beta$ search is that proofs at shallow search depths are easier to find. However, many popular games such as Go-Moku or checkmating puzzles are characterized by solutions containing narrow but very deep lines of play. It is difficult to adjust $\alpha\beta$ search to perform well in these cases. *Proof-Number Search (PNS)* (Allis, van der Meulen, and van den Herik, 1994) performs *variable-depth search* that has no explicit bounds on the search depth. The notion of proof and disproof numbers in PNS originates from McAllester's conspiracy numbers, which measure the reliability of the score in the minimax framework (McAllester, 1985; McAllester, 1988). Like PNS, McAllester's conspiracy number search (CNS) performs variable-depth search by trying to expand an unreliable leaf node, as estimated by conspiracy numbers, in order to make the score of that node more reliable. Unlike CNS, PNS specializes conspiracy numbers to AND/OR tree search with binary (win/loss) outcomes. This specialization leads to a significantly reduced memory requirement of PNS compared to CNS. Additionally, in contrast to conspiracy numbers, proof and disproof numbers also estimate the difficulty of solving a node. As a result, PNS implements a "simplest-first" search paradigm, which can find small but potentially deep proofs efficiently.

While the original formulation of PNS was already quite powerful, it was still plagued by some severe problems such as its hunger for memory, problems in applying the method to state spaces that are not trees, and excessive depth-first behavior leading to overly long solutions (Allis, 1994). To address these shortcomings, many variations of the algorithm have been developed (cf. van den Herik and Winands, 2008), and these variants have been successfully applied to a large number of domains including chess (Breuker, 1998), Othello (Nagai, 2002), shogi (Seo, Iida, and Uiterwijk, 2001; Nagai, 2002), Lines of Action (LOA) (Winands, Uiterwijk, and van den Herik, 2004), Go (Kishimoto and Müller, 2005b), checkers (Schaeffer *et al.*, 2007), Connect6 (Xu *et al.*, 2009; Wu *et al.*, 2011), the multi-player game Rolit (Saito and Winands, 2010), and even chemical synthesis (Heifets and Jurisica, 2012). All *PNS variants* share two features: (1) they are algorithms for solving binary goals, such as proving a win or a loss in a game position, and (2) they rely on the concept of proof and disproof numbers. Extensions which relax the binary goal assumption are reviewed in Section 9.

This article gives an overview of popular PNS variants and their enhancements developed over the last twenty years. It also describes several successful applications of PNS variants. The material is organized as follows: Section 2 introduces the terminology for AND/OR trees and their relation to solving games or game positions. Next, Section 3 presents the basic PNS algorithm. Subsequently, Section 4 discusses depth-first proof-number search, and Section 5 explains how PNS variants deal with limited memory. Section 6 discusses extensions of PNS from trees to more general graphs. Section 7 describes several techniques which further enhance the performance of PNS variants. Section 8 is dedicated to parallel PNS methods. Section 9 briefly reviews PNS extensions for multi-valued outcomes, Section 10 gives an overview of application domains for PNS variants, and the final Section 11 concludes and discusses future research directions.

2. AND/OR TREES, SOLVING GAME POSITIONS AND SOLVING GAMES

This section introduces the terminology for AND/OR trees and explains the relation to solving game positions or whole games.

An AND/OR tree is a rooted, finite tree consisting of *OR* and *AND* nodes. OR nodes correspond to positions with the first player to play, while in AND nodes, the second player is to play. All nodes except the *root* node have a parent. A directed edge between a parent and a child is drawn for each legal move. Each edge is labeled by a description of the move, for example by move coordinates. In this article the root is always an OR node, but can be in practice an AND node as well.

From the viewpoint of the first player, each node in an AND/OR tree can have a value of *win*, *loss*, or *unknown*. Nodes of value *win* and *loss* indicate positions that are known wins and losses for the first player, respectively, while nodes with value *unknown* must be examined further by *expanding a subtree*, in order to determine whether the node is a win or loss with best play. An AND/OR tree is called *solved* when the value of the root node has been determined to be either a win or loss[5]. A *terminal leaf* (or short *terminal*) node has no children. Its value is either a win or loss, as determined by the rules of the game. A node with at least one child is an *internal* node. A *non-terminal leaf* (or short *leaf*) node is a node which has not been expanded yet. It is unknown whether a leaf node is terminal or internal. The phrase "a node is x" is short for "a node has value x".

The values of internal nodes can be computed from the values of its children. If at least one child of an internal OR node n is a win, then n is also a win since the first player can move to that child. If all children of n are losses, then n is also a loss. Similarly, an internal AND node n is a loss if at least one of its children is a loss, and a win if all its children are wins. A node that has been determined to be a win is also called a *proven* node, while a node that is known to be a loss is called a *disproven* node. A *proof* is a computed win, while a *disproof* is a computed loss.

A *proof tree*[6] is a subtree of an AND/OR tree that contains a winning strategy for the first player and therefore guarantees that a node is proven. A proof tree T with root node r has the following properties:

1. The root node r is in T.
2. For each internal OR node of T, at least one child is contained in T.
3. For each internal AND node of T, all children are contained in T.
4. All terminal nodes in T are wins.

A *disproof tree*, which contains a winning strategy for the second player, is defined in a dual way. All its terminal nodes are losses, it contains one child of internal AND nodes, and all children of internal OR nodes (including the root).

Figure 1 shows an example of an AND/OR tree. OR nodes are represented by square boxes and AND nodes are represented by circles. Node A is the root node. D, F and I are terminal nodes and G, H and J are leaf nodes that have value unknown. A, B, C and E are internal nodes. The values of the internal nodes are calculated by propagating back the values of the leaf and terminal nodes. E is a win because I is a win. C is a loss because F is a loss. Because D and E are wins, B is a win, which leads A to be a win. A proof tree of A is shown with dashed lines.

3. PROOF-NUMBER SEARCH

Proof-Number Search (PNS) by Allis *et al.* (1994) is the original search algorithm using proof numbers. All other variants of proof-number search are descendants of PNS. Its conceptually closest forerunner

[5] Although many games have draws, AND/OR tree search can return only the binary outcome. Therefore, for the sake of simplicity, draws are considered to be losses for the first player, if the paper does not explicitly describe the value of draws. The techniques for proving draws with AND/OR tree search will be described in Section 9.

[6] So-called *solution trees* are defined in a similar way (de Bruin, Pijls, and Plaat, 1994; Pijls and de Bruin, 1999).

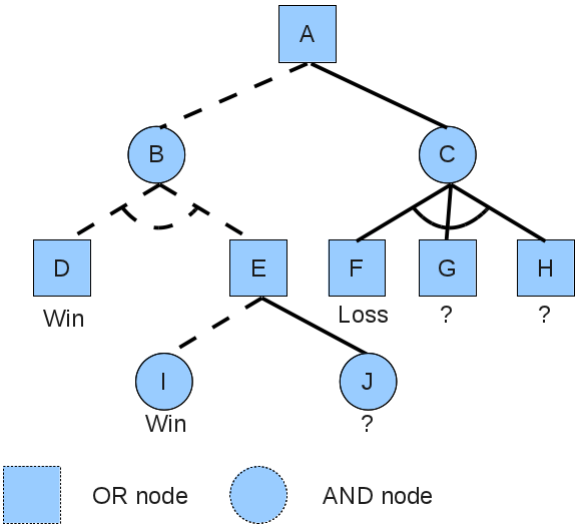

Figure 1: Example of an AND/OR tree. OR nodes shown as squares, AND nodes as circles.

is Conspiracy-Number Search (CNS) by McAllester (1985, 1988). CNS is a best-first search algorithm, which determines the cardinality of the smallest set of leaf nodes which have to "conspire" to change their values in order to change the minimax value of the root. One of the ideas underlying Conspiracy-Number Search is that the distribution of the values over the leaf nodes of the tree, and the shape of the tree, should influence the selection of the next node to be investigated. This last aspect has been singled out in PNS (Allis, 1994).

This section describes PNS by explaining the basic idea in Subsection 3.1, introducing proof and disproof numbers in Subsection 3.2, describing the PNS algorithm in Subsection 3.3, and finally giving the associated pseudo-code in Subsection 3.4.

3.1 The Basic Idea of PNS

PNS is a best-first search algorithm. Its heuristic determines the most promising leaf by selecting a *most-proving node (MPN)*[7], which is a leaf that, if solved, can contribute to either a proof or a disproof of the root. A MPN is found by exploiting two characteristics of the search tree: (1) its shape (determined by the branching factor of every internal node), and (2) the values of the leaves. Basic, unenhanced PNS is an uninformed search method that does not require any game-specific knowledge beyond its rules.

3.2 Proof and Disproof Numbers

In order to find a MPN, PNS maintains two numbers for each node n. (1) The proof number $\mathrm{pn}(n)$ is the smallest number of leaf nodes in the subtree starting with n that have to be proven in order to prove that n is a win. (2) The disproof number $\mathrm{dn}(n)$ is the minimum number of leaf nodes that have to be disproven in order to prove that n is a loss.

When the node n is clear from context, we sometimes just write pn short for $\mathrm{pn}(n)$ and dn for $\mathrm{dn}(n)$. The values of pn and dn can be calculated for each node in a tree in a bottom-up manner. Proof and disproof numbers for a node are set by initialization and can be modified later by backpropagation from its children. In a terminal node t, the game-theoretic value is known. If t is a win, then $\mathrm{pn}(t) = 0$ and $\mathrm{dn}(t) = \infty$. If t is a loss, then $\mathrm{pn}(t) = \infty$ and $\mathrm{dn}(t) = 0$.

A non-terminal leaf node n is initialized according to an *initialization rule* measuring the estimated cost of (dis)proving n. The simplest and most optimistic initialization rule only counts the node itself:

[7] Also called a most-promising node by some authors.

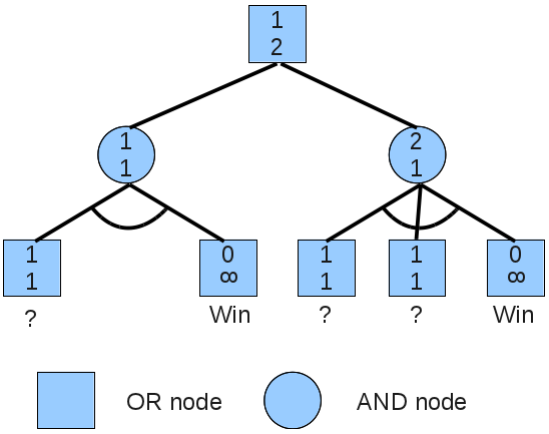

Figure 2: Example of a game tree. OR nodes shown as squares, AND nodes as circles. pn shown above dn.

$\mathrm{pn}(n) = \mathrm{dn}(n) = 1$. Proof and disproof numbers for an internal node n are calculated from its children $\mathrm{succ}(n)$ as follows: For OR nodes, $\mathrm{pn}(n) = \min_{s \in \mathrm{succ}(n)} \mathrm{pn}(s)$ and $\mathrm{dn}(n) = \sum_{s \in \mathrm{succ}(n)} \mathrm{dn}(s)$. For AND nodes, $\mathrm{pn}(n) = \sum_{s \in \mathrm{succ}(n)} \mathrm{pn}(s)$ and $\mathrm{dn}(n) = \min_{s \in \mathrm{succ}(n)} \mathrm{dn}(s)$.

The rules for handling infinity in computations are: $\infty + x = \infty$ and $\min(\infty, x) = x$ for all $x \geq 0$. However, for bounds in the df-pn algorithm of Section 4, $\infty - x < \infty - y$ if $0 \leq y < x$ for finite x, y. In practice, a large finite value is used to represent ∞.

In the example of Figure 2, there are two terminal nodes, both wins with $\mathrm{pn} = 0$ and $\mathrm{dn} = \infty$. The non-terminal leaves have been initialized with $\mathrm{pn} = \mathrm{dn} = 1$. The values of the leaves' parents are computed according to the backpropagation rule for AND nodes, while the root is an OR node.

The only MPN in this example is the leftmost leaf. It is found by descending in the tree from the root. At each internal OR node n, a child with smallest proof number among n's children is chosen, while at each internal AND node, the method selects a child with smallest disproof number. When a leaf is reached, it is chosen as the MPN.

Ties can be broken in an arbitrary way. This article simply assumes that the first encountered child with smallest (dis)proof number (matching that of the parent) is chosen.

3.3 The PNS Algorithm

At the start of PNS, pn and dn of the root are initialized. The search iteratively expands an AND/OR tree until the value of the root is decided - either pn or dn becomes 0, indicating a proof or disproof. In practice, a search can also be aborted when time or memory are exhausted. At each iteration, pn and dn are kept up to date and consistent for all nodes in the tree. Each iteration consists of four phases:

(1) *Selection.* A MPN is chosen as described above.

(2) *Expansion.* The MPN is expanded by adding a new child node for each legal successor position.

(3) *Evaluation.* The new leaves are evaluated and their pn and dn are initialized as described above.

(4) *Backpropagation.* The values of pn and dn of nodes along the path from the MPN back to the root are recomputed.

We note that this four-phase process of PNS is the same as that of MCTS (cf. Chaslot *et al.*, 2008b) with different instances for the various procedures.

Allis (1994) observed that as soon as the proof and disproof number of some ancestor node a do not change, the update is complete, and selection of the next MPN can be started by descending from a

instead of the root. As an additional optimization, Nagai (1998) suggested to stop as soon as both proof and disproof number of a node are not larger than the proof and disproof number of its parent. This breaks consistency of the values with nodes higher up in the tree, but does not break MPN selection since the node already had a minimum proof or disproof number amongst its siblings, and that number only became smaller.

3.4 Pseudo-Code for PNS

Figure 3 gives pseudo-code for PNS. *PNS* is the main procedure of the algorithm (lines 1-12). The procedure *Evaluate* evaluates a position, and assigns one of the following three values to a node: *disproven*, *proven*, or *unknown*. The proof and disproof numbers of a node are initialized by *SetProofAndDisproofNumbers* (lines 13-46). The function *SelectMostProvingNode* finds a MPN (lines 47-67). Expanding the MPN is done by *ExpandNode* (lines 68-80). After the expansion of the MPN, the new information is backed up by *UpdateAncestors* (lines 81-95).

4. DEPTH-FIRST PROOF-NUMBER SEARCH

One inefficiency of PNS, even with the optimizations from the previous section, is that proof and disproof numbers are recomputed even if the decision about which child is the one with minimal (dis)proof number does not change. For example, if the proof numbers of the children of an OR node are $\{15, 6, 5, \mathbf{2}, 8\}$, then the fourth child remains a MPN until its proof number exceeds 5, the proof number of the second-smallest child. Expressing the search termination condition for subtrees in terms of bounds on the (dis)proof numbers leads to Nagai's Depth-first Proof-number Search (df-pn) algorithm (Nagai, 1999a; Nagai, 2002). It has been proven that the algorithm always selects a MPN (cf. Nagai, 2002).

Df-pn uses two thresholds to limit the current iteration below a node n: $\mathrm{pt}(n)$ for the proof number and $\mathrm{dt}(n)$ for the disproof number. As with (dis)proof numbers, we sometimes omit the node when it is clear from the context and simply write pt and dt. The thresholds of n are used to check whether a MPN exists in n's subtree. The search in the subtree of a node n continues until the *termination condition* is reached: $\mathrm{pn}(n) \geq \mathrm{pt}(n)$ or $\mathrm{dn}(n) \geq \mathrm{dt}(n)$. When df-pn examines a node n, it recalculates $\mathrm{pn}(n)$ and $\mathrm{dn}(n)$ from n's children. If the termination condition holds at n after this recalculation, then df-pn backtracks to n's parent. In this case, n's subtree no longer contains a MPN. Otherwise, the algorithm selects the MPN as usual in the subtree of n.

Thresholds are updated when the best child is selected. Assume that df-pn examines OR node p, and child c_1 has a smallest proof number among p's children. Let pn_2 be the second smallest proof number among the list of proof numbers of all children. Then, df-pn examines c_1 with the following thresholds:

$$\mathrm{pt}(c_1) = \min(\mathrm{pt}(p), pn_2 + 1),\ \mathrm{dt}(c_1) = \mathrm{dt}(p) - \mathrm{dn}(p) + \mathrm{dn}(c_1).$$

The condition of $\mathrm{pt}(c_1)$ indicates that df-pn must backtrack to p in one of two cases: (1) when the MPN switches to the child with the proof number of pn_2 and (2) when p's subtree no longer contains a MPN. $\mathrm{dt}(c_1)$ is set so that df-pn can examine c_1's subtree as long as the total disproof number of p's children does not exceed $\mathrm{dt}(p)$. This guarantees the existence of a MPN in c_1's subtree.

Analogously, for an AND node p let dn_2 be the second smallest disproof number among the list of disproof numbers of p's children. Then pt and dt are assigned as follows:

$$\mathrm{pt}(c_1) = \mathrm{pt}(p) - \mathrm{pn}(p) + \mathrm{pn}(c_1),\ \mathrm{dt}(c_1) = \min(\mathrm{dt}(p), dn_2 + 1).$$

Figure 4 shows an example of how df-pn works. Like PNS, df-pn selects the MPN D via path $\mathsf{A} \rightarrow \mathsf{B} \rightarrow \mathsf{D}$. Df-pn initially sets $\mathrm{pt}(\mathsf{A}) = \mathrm{dt}(\mathsf{A}) = \infty$ to indicate that A is either proven or disproven if the updated proof or disproof number of A is at least as large as $\mathrm{pt}(\mathsf{A})$ or $\mathrm{dt}(\mathsf{A})$. Df-pn selects B because $\mathrm{pn}(\mathsf{B}) < \mathrm{pn}(\mathsf{C})$. When B is not on the path to a MPN, $\mathrm{pn}(\mathsf{B}) > \mathrm{pn}(\mathsf{C})$ holds. Df-pn therefore sets $\mathrm{pt}(\mathsf{B}) = \mathrm{pn}(\mathsf{C}) + 1 = 4$ to indicate that C must be examined if $\mathrm{pn}(\mathsf{B}) \geq 4 > \mathrm{pn}(\mathsf{C})$. In terms of the disproof number, df-pn examines B as long as $\mathrm{dn}(\mathsf{B}) + \mathrm{dn}(\mathsf{C}) < \mathrm{dt}(\mathsf{A})$. It therefore assigns

```
1: // Set up for the root node
2: void PNS(node Root) {
3:    Evaluate(Root);
4:    SetProofAndDisproofNumbers(Root);
5:    node current = Root;
6:    node mostProving;
7:    while (Root.proof ≠ 0 && Root.disproof ≠ 0) {
8:       mostProving = SelectMostProvingNode(current);
9:       ExpandNode(mostProving);
10:      current = UpdateAncestors(mostProving, Root);
11:   }
12: }

13: // Calculating proof and disproof numbers
14: void SetProofAndDisproofNumbers(node N) {
15:    if (N.expanded) // Internal node;
16:       if (N.type == AND) {
17:          N.proof = 0;
18:          N.disproof = ∞;
19:          for (each child C of N) {
20:             N.proof = N.proof + C.proof;
21:             if (C.disproof < N.disproof)
22:                N.disproof = C.disproof;
23:          }
24:       }
25:       else { // OR node
26:          N.proof = ∞;
27:          N.disproof = 0;
28:          for (each child C of N) {
29:             N.disproof = N.disproof + C.disproof;
30:             if (C.proof < N.proof)
31:                N.proof = C.proof;
32:          }
33:       }
34:    else // Terminal or non-terminal leaf
35:       switch (N.value) {
36:          case proven:
37:             N.proof = ∞;
38:             N.disproof = 0;
39:          case disproven:
40:             N.proof = 0;
41:             N.disproof = ∞;
42:          case unknown:
43:             N.proof = 1;
44:             N.disproof = 1;
45:    }
46: }

47: // Select a MPN
48: node SelectMostProvingNode(node N) {
49:    int value = ∞;
50:    node best;
51:    while (N.expanded) {
52:       if (N.type == OR) // OR Node
53:          for (each child C of N)
54:             if (value > C.proof) {
55:                best = C;
56:                value = C.proof;
57:             }
58:       else // AND Node
59:          for (each child C of N)
60:             if (value > C.disproof) {
61:                best = C;
62:                value = C.disproof;
63:             }
64:       N = best;
65:    }
66:    return N;
67: }

68: // Expand node
69: void ExpandNode(node N) {
70:    GenerateChildren(N);
71:    for (each child C of N) {
72:       Evaluate(C);
73:       SetProofAndDisproofNumbers(C);
74:       // Addition to original code
75:       if ((N.type == OR && C.proof == 0) ||
76:          (N.type == AND && C.disproof == 0))
77:          break;
78:    }
79:    N.expanded = true;
80: }

81: // Update ancestors
82: void UpdateAncestors(node N, node Root) {
83:    loop {
84:       int oldProof = N.proof;
85:       int oldDisProof = N.disproof;
86:       SetProofAndDisproofNumbers(N);
87:       // No change on the path
88:       if (N.proof == oldProof &&
89:          N.disproof == oldDisProof)
90:          return N;
91:       if(N == Root)
92:          return N;
93:       N = N.parent;
94:    }
95: }
```

Figure 3: Pseudo-code of the PNS algorithm with Allis' enhanced backpropagation scheme

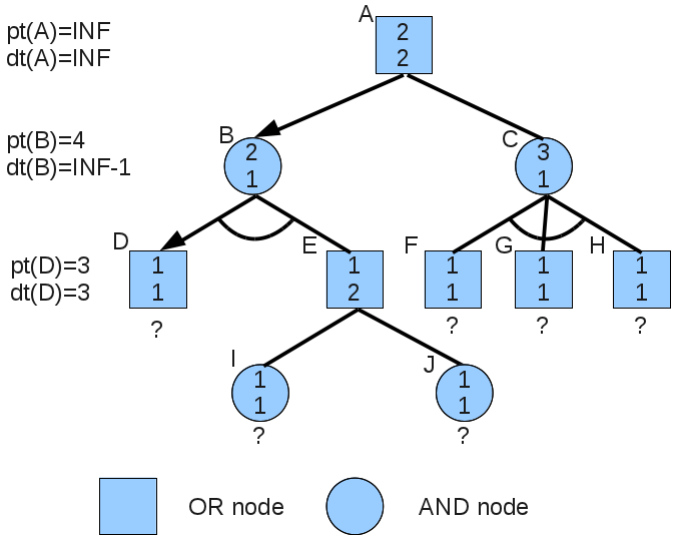

Figure 4: Example of the behavior of df-pn. Squares are OR nodes and circles are AND nodes. Each node's pn is given by the upper number, its dn by the lower number.

$\mathrm{dt}(\mathsf{B}) = \mathrm{dt}(\mathsf{A}) - \mathrm{dn}(\mathsf{C}) = \infty - 1$. At B, since $\mathrm{dn}(\mathsf{D}) < \mathrm{dn}(\mathsf{E})$, df-pn selects D and examines D as long as both B and D are on the path to a MPN. This termination condition holds if $\mathrm{dn}(\mathsf{D})$ is at least as large as $\mathrm{dt}(\mathsf{B})$ or $\mathrm{dn}(\mathsf{E}) + 1$. Df-pn therefore sets $\mathrm{dt}(\mathsf{D}) = \min(\mathrm{dt}(\mathsf{B}), \mathrm{dn}(\mathsf{E}) + 1) = 3$. Analogously, if $\mathrm{pn}(\mathsf{D}) + \mathrm{pn}(\mathsf{E}) \geq \mathrm{pt}(\mathsf{B})$, B is not on the path to a MPN. Hence, $\mathrm{pt}(\mathsf{D}) = \mathrm{pt}(\mathsf{B}) - \mathrm{pn}(\mathsf{E}) = 4 - 1 = 3$. As long as $\mathrm{pn}(\mathsf{D}) < 3$ and $\mathrm{dn}(\mathsf{D}) < 3$, df-pn examines the subtree rooted at D without backpropagating proof and disproof numbers to A and B because D's subtree is guaranteed to contain a MPN.

Figure 5, adapted and modified from Kishimoto and Müller (2008), shows pseudo-code of df-pn[8]. The code is written in negamax fashion to avoid two dual cases. For node n, $n.\phi$ and $n.\delta$ are defined as follows:

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

With this notation, $n.\phi$ and $n.\delta$ for an internal node n simply become:

$$n.\phi = \min_{i=1,2,\cdots,k} c_i.\delta, \quad n.\delta = \sum_{i=1}^{k} c_i.\phi,$$

where c_i are the children of n.

In the pseudo-code, df-pn returns the value of the root. The fields $n.\phi$ and $n.\delta$ are used for two purposes: initially they hold the thresholds for ϕ and δ. After n is examined, they hold the proof and disproof numbers of n. The main function *Df-pn* (lines 1-10) initializes both thresholds to infinity, and then calls the recursive function *MID* that performs multiple iterative deepening at a node. When returning from *MID*, the root node is either proven or disproven (see lines 6-9). *MID* (lines 11-34) traverses the subtree below node n in a depth-first manner. It performs no backtracks while proof or disproof numbers do not exceed the threshold and while it does not encounter a terminal leaf (see lines 21-30). *IsTerminal* checks whether n is a terminal leaf, while *Evaluate* stores proof and disproof numbers for an (either proven or disproven) terminal leaf in $n.\phi$ and $n.\delta$, respectively.

[8]The presented code is slightly different from the original df-pn pseudo-code of Nagai (2002) in order to fix an essential performance problem.

```
1: // Set up for the root node
2: // Note that the root is an OR node
3: int Df-pn(node R) {
4:    R.φ = ∞;  R.δ = ∞;
5:    MID(R);
6:    if (R.δ = ∞)
7:       return proven;
8:    else
9:       return disproven;
10: }
11: // Perform search with thresholds
12: void MID(node N) {
13:    // Terminal leaf
14:    if (IsTerminal(N)) {
15:       Evaluate(N);
16:       // Store (dis)proven node
17:       SaveProofandDisproofNumbers(N,N.φ,N.δ);
18:       return;
19:    }
20:    GenerateMoves(N);
21:    // Continue search until satisfying
22:    // the termination condition
23:    while (N.φ > ΔMin(N)  &&
24:            N.δ > ΦSum(N))  {
25:       C_best = SelectChild(N,φ_c,δ_2);
26:       // Update thresholds
27:       C_best.φ = N.δ + φ_c - ΦSum(N);
28:       C_best.δ = min(N.φ,δ_2 + 1);
29:       MID(C_best);
30:    }
31:    // Store search results
32:    N.φ = ΔMin(N); N.δ = ΦSum(N);
33:    SaveProofandDisproofNumbers(N,N.φ,N.δ);
34: }
35: // Select the best child
36: node SelectChild(node N, int &φ_c, int &δ_2)  {
37:    node C_best;
38:    δ_c = φ_c = ∞;
39:    for (each child C of N) {
40:       RetrieveProofandDisproofNumbers(C,φ,δ);
41:       // Store the smallest and second
42:       // smallest δ in δ_c and δ_2
43:       if (δ < δ_c) {
44:          C_best = C;
45:          δ_2 = δ_c; φ_c = φ; δ_c = δ;
46:       }
47:       else if (δ < δ_2)
48:          δ_2 = δ;
49:       if (φ = ∞)
50:          return C_best;
51:    }
52:    return C_best;
53: }
54: // Compute the smallest δ of N's children
55: int ΔMin(node N) {
56:    int min = ∞;
57:    for (each child C of N) {
58:       RetrieveProofandDisproofNumbers(C,φ,δ);
59:       min = min(min,δ);
60:    }
61:    return min;
62: }
63: // Compute sum of φ of N's children
64: int ΦSum(node N) {
65:    int sum = 0;
66:    for (each child C of N) {
67:       RetrieveProofandDisproofNumbers(C,φ,δ);
68:       sum = sum + φ;
69:    }
70:    return sum;
71: }
```

Figure 5: Pseudo-code of the df-pn algorithm

When a node n is expanded, a best child c_{best} in terms of proof and disproof numbers is selected by *SelectChild* (lines 35-53) for a recursive call to *MID* with the aforementioned new thresholds (see lines 25-29). $n.\phi$ corresponds to $c_{best}.\delta$ in the negamax formulation.

PNS variants including df-pn require a data structure such as a transposition table (see Subsection 5.1 for more details) that preserves the proof and disproof numbers previously computed at internal nodes. In the code, *RetrieveProofandDisproofNumbers* and *SaveProofandDisproofNumbers* are used to read and write table entries.

5. PNS VARIANTS WITH LIMITED MEMORY

The memory requirements of PNS are huge since the whole search tree is stored in memory. This section describes the two main approaches for dealing with limited memory for PNS. First, transposition tables and their associated replacement strategies are explained in Subsection 5.1. Next, PN^2 variants are explained in Subsection 5.2.

5.1 Transposition Table and Replacement Strategies

Because all PNS variants repeatedly traverse paths from the root to a MPN, they must preserve proof and disproof numbers at internal nodes in some data structure. A transposition table (TT) (Greenblatt, Eastlake, and Croker, 1967) is a hash table which can be used to store previously computed proof and disproof numbers by mapping a game position to its corresponding transposition table entry. TTs are used in many PNS implementations, such as (Breuker, 1998; Seo *et al.*, 2001; Nagai, 2002). A TT can quickly retrieve proof and disproof numbers for each revisited node and reuse them for positions that can be reached via multiple paths. Techniques for ensuring correctness of the search when using a TT are described in Subsection 6.2.

No matter which data structure is used to store proof and disproof numbers, it is indispensable to control the memory use during search, in case the explored search space does not fit completely into the available memory. One way to reduce the memory footprint is to use the PN^2 algorithm, which will be described in the next subsection. Another way is to use a TT with replacement strategies. While strategies based on two-level TT are commonly used in $\alpha\beta$ search (Breuker, Uiterwijk, and van den Herik, 1996), the consensus in the computer shogi community working on tsume-shogi solvers is that they are ineffective for PNS variants in that domain.

The df-pn Hex solver of Pawlewicz (2012) uses the multiple-probe scheme of Beal and Smith (1996), which is popular with $\alpha\beta$ chess programs such as FRUIT and STOCKFISH. The technique probes four consecutive entries in a single hash table, and overwrites a TT entry with smallest subtree size among the four. Similarly, Seo's tsume-shogi solver examines about 100 TT entries to find a candidate to replace (Seo, 1998). His solver uses the PNS variant PN* (Seo, 1995; Seo *et al.*, 2001).

Allis *et al.* (1994) introduce two pruning techniques that remove useless nodes from the search tree. (1) *DeleteSolvedSubtree* deletes all subtrees with proof or disproof number equal to 0 when an ancestor is proven or disproven by backpropagation. (2) *DeleteLeastProving* removes the child of an OR node with the maximum proof number and the child of an AND node with the maximum disproof number. Gnodde (1993) describes a detailed implementation of *DeleteLeastProving*, and suggests to use the highest ratio of proof and disproof numbers for selecting a least promising node.

Nagai (1999b) presents several replacement and garbage collection strategies. The *SmallTreeGC* algorithm is used in high-performance tsume-shogi solvers and in the world's best life-and-death solver in Go (Nagai, 2002; Okabe, 2005; Kishimoto and Müller, 2005b; Kishimoto, 2010; Kaneko, 2010). When the TT becomes full, SmallTreeGC discards a fixed fraction R of the TT entries, starting with those of smallest subtree size. The value of R is determined empirically. A hybrid approach using both a replacement strategy and SmallTreeGC has been proposed by Nagai (2002).

5.2 PN^2 and PDS–PN

PN^2 is proposed by Allis (1994) as an algorithm to reduce the memory requirements of PNS. It has been elaborated upon in Breuker (1998). Its implementation and testing for chess positions is extensively described in Breuker, Uiterwijk, and van den Herik (2001). PN^2 consists of two levels of PNS. The first level consists of a PNS called PN$_1$, which calls a PNS at the second level (PN$_2$) for an evaluation of the MPN of the PN$_1$ search tree. This PN$_2$ search is limited by a given maximum number of nodes that can be stored in memory. Setting this maximum carefully is crucial because it determines the trade-off between the memory consumption and the speed of PN^2.

There are two main approaches for setting this limit for PN$_2$ search. (1) Allis (1994) suggests using the size (i.e., number of nodes) of the PN$_1$ search tree. The disadvantage of this approach is that small problems take much longer to be solved than with standard PNS. To counter this disadvantage, (2) Breuker (1998) suggests a limit that is a fraction of the size of the PN$_1$ search tree. This fraction should start small, and grow larger as the size of the first-level search tree grows. Breuker denotes this fraction as $f(x)$, where x is the size of the PN$_1$ search tree. In his experiments on chess positions, Breuker (1998) proposes a logistic growth function $f(x) = \frac{1}{1+e^{\frac{a-x}{b}}}$. The parameters a and b are strictly positive and require tuning for optimal performance (Breuker, 1998). The size of PN$_2$ is further limited by the amount of physical memory. If N is the physical limit of nodes that can be stored, the maximum size of PN$_2$ is $y = \min(x \times f(x), N - x)$. The PN$_2$ search is stopped when the number of nodes stored in memory exceeds y or the subtree is (dis)proven. After completing a PN$_2$ search, the children of the root of the PN$_2$ search tree are preserved, but their subtrees are removed from memory. The children of the MPN (the root of the PN$_2$ search tree) are not immediately evaluated by a second-level search; such child nodes are evaluated only after they become the MPN. This is called *delayed evaluation*. We remark that for PN$_2$-search trees, *immediate evaluation*, where each generated node is immediately evaluated, is used (Allis, 1994). This two-level search is schematically sketched in Figure 6.

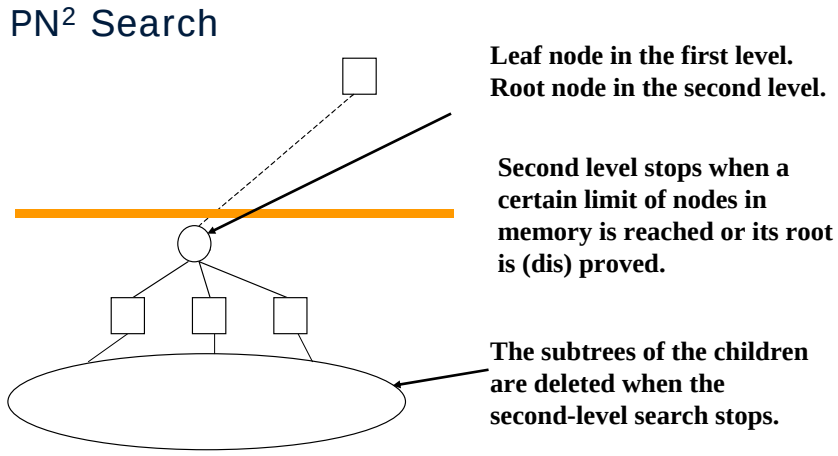

Figure 6: Schematic sketch of PN^2.

PDS–PN by Winands *et al.* (2004) combines PN^2 with a predecessor algorithm of df-pn called PDS (Nagai, 1998; Nagai, 1999b). PDS–PN is a PN^2 variant that uses PDS for its PN$_1$ search and normal PNS for its PN$_2$ search. Since the first level search uses the TT, PDS–PN is not restricted by memory in practice. At the second level, it profits from the speed given by the immediate evaluation of PNS[9]. While it has not yet been seen in practice, a DFPN-PN algorithm should probably replace PDS-PN in domains where PDS-PN is preferable to df-pn and PN^2.

[9]Nagai's Othello implementation (Nagai, 1999b) improves the node expansion rate of PDS with the TT by preserving in each TT entry the pointers to the TT entries of generated children. With the help of this data structure, proof and disproof numbers are computed efficiently by only traversing these pointers. This technique improves speed at the price of using extra memory.

6. PNS TECHNIQUES FOR DIRECTED ACYCLIC AND CYCLIC GRAPHS

For most popular games, representing the search space as a tree is quite inefficient, since equivalent positions can be reached via different move sequences. In the tree model, all these positions have to be solved independently, while in a *directed acyclic graph* (DAG) model, a single node reached through different paths can represent all these equivalent positions. A big complication is caused by *repetition rules*: the value of a game state may depend not only on the position on the board, but also on the history. For example, a position in chess may be a sure win for one player, but if that player carelessly allows the same position to repeat three times, then the opponent can claim a draw by repetition. A game with repetitions can be modeled by a *directed cyclic graph* (DCG). Care must be taken to prevent infinite loops in algorithms on such graphs. Also, the outcome of a game no longer depends only on the position reached, but also on the *move history*, the specific sequence of moves taken to reach the position. Ignoring history can lead to the *graph history interaction* (GHI) problem (Campbell, 1985).

Trying to run a tree-based PNS algorithm on a DAG or DCG runs into several problems: (1) overestimation of proof/disproof numbers, (2) the Graph-History Interaction problem, and (3) infinite loops. These problems together with their published solutions are discussed in detail below.

6.1 The Overestimation Problem of Proof and Disproof Numbers

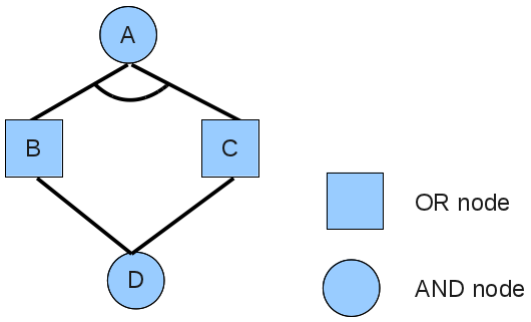

Figure 7: Example of overestimating pn(A)

If the search space is a directed acyclic graph (DAG), then PNS variants applying the standard back-propagation rule from trees can count the same node more than once when computing proof and disproof numbers. We call this the overestimation problem. In Figure 7, pn(A) is overestimated by counting pn(D) twice:

$$\mathrm{pn}(\mathsf{A}) = \mathrm{pn}(\mathsf{B}) + \mathrm{pn}(\mathsf{C}) = \mathrm{pn}(\mathsf{D}) + \mathrm{pn}(\mathsf{D}) = 2\,\mathrm{pn}(\mathsf{D}).$$

However, the correct result would be pn(A) = pn(D) since proving D proves B, C and A.

Simply ignoring the overestimation problem can be a practical choice in some applications (Allis *et al.*, 1994; Schadd *et al.*, 2008). However, in difficult domains such as tsume-shogi the overestimation problem can cause severe performance degradation (Seo, 1995; Seo *et al.*, 2001; Kishimoto, 2010). Nodes that would be easy to prove but have many transpositions in their sub-graph can be assigned huge proof numbers and therefore will not be expanded for a long time.

Schijf (1993) proposes an algorithm that always accurately computes proof and disproof numbers for a DAG. However, this algorithm was impractical even for solving 3×3 Tic-Tac-Toe due to both immense computational overhead and a huge memory requirement. Considerable work has gone into approximation algorithms, which may sometimes over- or underestimate proof and disproof numbers. Schijf *et al.* (1993, 1994) present a PNS-specific algorithm that is successfully used to solve Connect-Four and Qubic (Uiterwijk, van den Herik, and Allis, 1990; Allis and Schoo, 1992; Allis, 1994). When generating a new node, the algorithm checks whether that node is a transposition. If so, then it is merged with the existing identical node. Additionally, in the update procedure of PNS, the algorithm recursively updates all parents instead of just one.

Proof-set search (Müller, 2003) backs up *proof sets* containing sets of leaf nodes, which are sufficient to prove or disprove positions, instead of proof numbers which count the size of such sets. While the algorithm is of theoretical interest and allows to trade off between memory requirement and solution quality, it has not been a success in practice.

Weak proof-number search (WPNS) (Ueda *et al.*, 2008), a small modification of work by Okabe (2005), sets $\mathrm{pn}(n) = \max_{1 \le i \le k} \mathrm{pn}(c_i) + k - 1$, where $c_1, \cdots, c_k$ are the children of AND node n. The disproof numbers of OR nodes are handled analogously, while proof numbers at OR nodes and disproof numbers at AND nodes are computed in the regular way. WPNS avoids overestimation when computing $\mathrm{pn}(n)$ by lowering the contribution of all children except the one with largest proof number to 1. Therefore a WPNS-style proof number consists a term for the "heaviest" child plus the count of the number of other children.

Nagai (2002) presents a method to detect an AND node n that suffers from overestimation of the proof number and takes the maximum of proof numbers of n's children instead of summing them up. This approach is incorporated into df-pn. The disproof number at an OR node is handled analogously. Nagai's algorithm stores a pointer to one parent p in each TT entry. When n is reached via p, p is saved in n's TT entry. Then, if n is reached via another parent q, there may exist a node m that counts $\mathrm{pn}(n)$ or $\mathrm{dn}(n)$ more than once. Such m is detected by checking if one of n's ancestors obtained by keeping traversing pointers in the TT is merged into m on the path of the current df-pn search. For example, assume that df-pn first reaches D via path $\mathsf{A} \to \mathsf{B} \to D$ in Figure 7. The TT entry for D keeps the pointer to B to indicate that D has one parent B. Analogously, the TT entry for B contains a pointer to A. Next, if df-pn reaches D via $\mathsf{A} \to \mathsf{C} \to \mathsf{D}$, it recognizes that D has more than one parent, indicating a possible overestimation. By comparing the path of the current df-pn search with the path obtained from traversing parent pointers in the TT, df-pn detects the possible overestimation at the shared ancestor node A. In this case, Nagai's approach calculates $\mathrm{pn}(\mathsf{A}) = \max(\mathrm{pn}(\mathsf{B}), \mathrm{pn}(\mathsf{C})) = \mathrm{pn}(\mathsf{D})$.

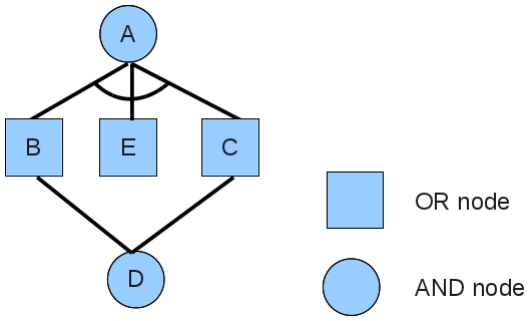

Figure 8: Example in which SNDA accurately calculates $\mathrm{pn}(\mathsf{A})$

The Source Node Detection Algorithm (SNDA) generalizes Nagai's idea in order to compute proof and disproof numbers more accurately (Kishimoto, 2010). As in Nagai's approach, SNDA preserves the pointer to one parent in each TT entry. Additionally, SNDA saves the moves in the TT entry if by using Nagai's pointer-based approach it detects that via children generated by these moves df-pn can reach the same descendant causing overestimation. SNDA then computes the sum of the maximum of (dis)proof numbers of such children and the (dis)proof numbers of the other children that are irrelevant to overestimating (dis)proof numbers. For example, in Figure 8, Nagai's approach calculates $\mathrm{pn}(\mathsf{A}) = \max(\mathrm{pn}(\mathsf{B}), \mathrm{pn}(\mathsf{C}), \mathrm{pn}(\mathsf{E})) = \max(\mathrm{pn}(\mathsf{D}), \mathrm{pn}(\mathsf{E}))$, because it detects that $\mathrm{pn}(\mathsf{A})$ may be overestimated and then takes the maximum of the proof numbers of *all of* A's children. In contrast, SNDA not only detects that overestimation may occur at A but also saves branches $A \to B$ and $A \to C$ in the TT to indicate that these branches may be a cause of overestimation. As a result, SNDA computes $\mathrm{pn}(\mathsf{A}) = \max(\mathrm{pn}(\mathsf{B}), \mathrm{pn}(\mathsf{C})) + \mathrm{pn}(\mathsf{E}) = \mathrm{pn}(\mathsf{D}) + \mathrm{pn}(\mathsf{E})$, which is an accurate proof number in this example.

6.2 The Graph-History Interaction Problem

Despite the existence of an exact algorithm (Schijf, 1993), computing true proof and disproof numbers in directed cyclic graphs (DCG) becomes even more difficult than in DAG. Often, practical game programs

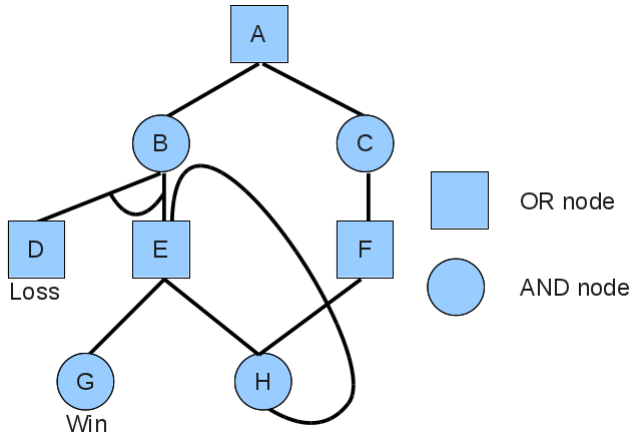

Figure 9: The GHI problem

represent history information incompletely, or not at all, leading to DCG structures even if the game itself is loop-free. Many implementations of PNS variants *ignore the history of the position* in a DCG when retrieving previously calculated proof and disproof numbers from data structures such as the TT. Such programs may mistakenly regard proven positions as disproven and vice versa. This is the so-called Graph-History Interaction (GHI) problem (Palay, 1983; Campbell, 1985).

Figure 9, adapted from Kishimoto and Müller (2004), shows an example of the GHI problem. Assume that a move leading to a repeated position is a loss for the first player and a PNS variant uses the TT to store search results. In this example, node A is a winning position for the first player, because A → C → F → H → E → G leads to a win. However, if the algorithm searches nodes in the order below it leads to the wrong outcome:

1. Search A → B → E → H → E. A loss is stored in the TT entry for H, because the position repetition cannot be avoided.
2. Search A → B → D. A loss is stored for AND node B.
3. Search A → C → F → H. A loss is retrieved from the TT for H and is backpropagated to F and C.
4. Although A is a win, it is now incorrectly labeled as a loss because losses are stored for both children B and C.

In their work on the GHI problem in PNS variants, Schijf *et al.* (1993, 1994) note that it is unnecessary to compute correct proof and disproof numbers for unproven nodes and propose three algorithms. The *tree method* uses no transpositions and has the disadvantage of not reusing results. The *DAG method* defines two classes of moves: *conversion* moves that 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 at least one non-conversion move are treated as different nodes. Despite its effectiveness in practice, their *DCG method* sometimes results in incorrect disproofs, because the internal nodes of the DCG method might contain repetition-dependent outcomes as if they were repetition-independent outcomes. (Schijf, 1993).

Moldenhauer (2009) addresses GHI in the Cops and Robber domain and provides a specific solution to this scenario. In this approach, df-pn stores the minimum cost from the root to n (called the g-value) in the TT entry for n.

Breuker *et al.* present the Base-Twin Algorithm (BTA), a PNS-specific solution to GHI (Breuker, 1998; Breuker *et al.*, 2001). A df-pn-specific approach is presented by Nagai (2002). These approaches deal with a special case of GHI, where a move leading to a repetition leads to a single fixed outcome such as draw.

Kishimoto and Müller (2004) propose the first general and practical GHI solution that can correctly and efficiently handle complicated GHI scenarios. As in (Breuker *et al.*, 2001), a TT entry is split into

a base entry and additional twin entries. When a result that involves repetitions is saved in the TT at a node via path p, it is saved in a twin entry by indicating that the result is valid via path p. If the result is either repetition-free or unproven, it is saved in the base entry. In the case when proven or disproven nodes are replaced in the TT, Kishimoto and Müller's (2004) GHI solution still guarantees returning the correct binary value, but may construct an incorrect proof tree or disproof tree. Kishimoto (2005) bypasses this issue by efficiently reconstructing the (dis)proof tree starting from the root after df-pn solves the game. The algorithm of Kishimoto (2005) works correctly even when arbitrary TT entries are replaced.

6.3 The Infinite Loop Problem

Given unlimited memory, df-pn is proven to be a complete algorithm on finite DAG (Kishimoto and Müller, 2008). However, it is incomplete on finite DCG even when incorporating a correct GHI solution, due to the infinite loop problem (Kishimoto and Müller, 2008). When node n is part of a cycle in the graph, and $\mathrm{pn}(n)$ or $\mathrm{dn}(n)$ is computed, the standard computation of proof and disproof numbers may define $\mathrm{pn}(n)$ or $\mathrm{dn}(n)$ in terms of itself, resulting in an overestimated proof or disproof number. As a result, it is possible that $\mathrm{pn}(n) \geq \mathrm{pt}(n)$ or $\mathrm{dn}(n) \geq \mathrm{dt}(n)$ always holds at n, and this node is never expanded.

So far, two methods are published that seem to avoid the infinite loop problem in practice. The theoretical completeness of df-pn with these solutions on finite DCG remains an open question. The df-pn(r) algorithm (Kishimoto and Müller, 2003; Kishimoto, 2005) computes the minimum distance from the root to each node in the TT. When calculating $\mathrm{pn}(n)$ at an AND node or $\mathrm{dn}(n)$ at an OR node, df-pn(r) ignores n's *old children*, for which the minimum distance is not larger than that of n, until the remaining "normal" children are solved. This approach is based on the observation that the children with shorter distance than n may cause the overestimation.

The threshold controlling algorithm (TCA) (Kishimoto, 2010) uses the standard way of computing proof and disproof numbers. It increases pt or dt when possible infinite loops are detected by using the minimum distance information. A simple practical approach used in Nagai's tsume-shogi solver is to increase the thresholds of proof and disproof numbers when no new leaf nodes have been expanded for a while (Nagai, 2010).

7. SEARCH ENHANCEMENTS FOR PNS VARIANTS

This section presents several techniques for enhancing the performance of PNS variants, including heuristic initialization, modified update rules for (dis)proof numbers, threshold control and more.

7.1 Initialization of Proof and Disproof Numbers

The default initialization in Subsection 3.2 sets $\mathrm{pn} = \mathrm{dn} = 1$ for non-terminal leaf nodes. If a heuristic measure of the effort to (dis)prove such a node is available, it can be used instead (Allis *et al.*, 1994).

Let $\mathrm{bf}(n)$ be the branching factor at leaf node n. A popular initialization rule (Allis, 1988; Allis *et al.*, 1994; Breuker, Allis, and van den Herik, 1994; Winands, 2004) sets $\mathrm{dn}(n) = \mathrm{bf}(n), \mathrm{pn}(n) = 1$ for OR nodes and $\mathrm{pn}(n) = \mathrm{bf}(n), \mathrm{dn}(n) = 1$ for AND nodes. Effectively, this initialization rule adds an additional ply of search tree information without changing the characteristics of PNS variants, and it results in improved performance.

Another simple domain-independent way of initializing nodes is to run a bounded search. An example is $\mathrm{PN}^2$where a second-level PNS is performed to initialize a leaf node.

Sound "admissible" effort measures can be constructed from single-player relaxations of the game, such as counting the number of moves to complete n-in-a-row in games such as Go-Moku, or until two eyes are formed in Go (Kishimoto and Müller, 2005b).

Initialization can use domain-dependent heuristic functions, which estimate the number of further expansions required to prove or disprove the goal. While this approach might result in substantially different search behavior than without heuristic initialization, it usually can be tuned to achieve significant improvements. Domain-dependent knowledge can be coded by hand (Allis, 1994; Nagai, 2002; Kishimoto and Müller, 2005b; Schaeffer *et al.*, 2005; Winands and Schadd, 2011; Kishimoto, 2010; Wu *et al.*, 2011), computed using machine learning techniques (Nagai, 1999a; Kaneko *et al.*, 2004; Miwa, Yokoyama, and Chikayama, 2004), or created by Monte Carlo sampling (Saito *et al.*, 2007).

7.2 Modifying the Calculation of Proof and Disproof Numbers

A number of techniques modify PNS by considering only a subset of the children of a node. Threat-space search (Allis, 1994; Allis, Huntjes, and van den Herik, 1996) uses the notion of *threats*, a set of candidate moves a player is forced to choose from in order to win, or avoid an immediate loss. Instead of considering all legal moves, threat-space search considers only threats when calculating proof and disproof numbers.

Dual lambda search (Soeda, Kaneko, and Tanaka, 2006) and Lambda depth-first proof-number search (Yoshizoe, Kishimoto, and Müller, 2007; Yoshizoe, 2009) combine df-pn with λ-search (Thomsen, 2000), a more general threat-based search technique. By using pass moves, these algorithms find different levels of threat sequences that use df-pn with limited moves, and can find proofs more quickly. Nagai (2002) uses an idea similar to λ-search with pass moves for solving the brinkmate problem in shogi.

Yoshizoe (2008) observes that proof and disproof numbers often do not reflect the actual difficulty of solving a position in cases where the number of legal moves is large and does not dramatically change between the current and child nodes. In such cases, PNS degenerates to breadth-first search. His Dynamic Widening algorithm computes proof and disproof numbers from a small number of best children only and still guarantees the correctness of solutions. A similar approach is used in the Hex solver briefly described by Arneson, Hayward, and Henderson (2011).

In practice, the values of siblings in a game tree are often highly correlated. Once a child c_1 is proven, other children $c_2, \cdots, c_k$ representing similar positions are likely to be proven as well. An example of highly correlated moves are useless interposing piece drops in tsume-shogi. Whether moves are highly correlated or not is typically determined by domain-dependent heuristics. Let $c_1, \cdots, c_k$ be such highly correlated children. Many df-pn-based tsume-shogi solvers such as (Seo, 1995) take only one of the unproven c_j $(1 \leq j \leq k)$ and the remaining unrelated children $c_{k+1}, \cdots$ into account when calculating proof numbers at AND nodes and disproof numbers at OR nodes.

In the one-eye problem in Go, Kishimoto and Müller (2005a) combine df-pn with a divide-and-conquer approach that splits a position into subproblems to be solved independently. Moves are limited to one subproblem, which reduces proof and disproof numbers. The method combines the results of these subproblems for solving the original position.

7.3 Threshold Control of Df-pn

Heuristic initialization of proof and disproof numbers[10] increases the overhead of re-expanding internal nodes in df-pn (Kishimoto and Müller, 2005b). This problem occurs when heuristic proof and disproof numbers are typically larger than 1, since thresholds are incremented by the minimum possible amount, which tends to be smaller than the average heuristic initialization value.

One way to reduce re-expansions is to increase the threshold increments over those of original df-pn. Assume that an OR node n with threshold $\mathrm{pt}(n)$ is searched and that df-pn selects n's child c. Let pn_2 be the second smallest proof number among the list of proof numbers of n's children. Df-pn sets $\mathrm{pt}(c) = \min(\mathrm{pt}(n), \mathrm{pn}_2 + \delta)$ with $\delta = 1$. Nagai (1999a, 2002) uses a constant value of $\delta > 1$. In (Kishimoto, 2005; Kishimoto and Müller, 2005b), δ is set dynamically, to the average value of

[10] Nagai calls a closely related algorithm considering edge costs df-pn$^+$. We still call it df-pn when the heuristic initialization is done in nodes.

heuristically initialized proof numbers of n's children. Disproof thresholds are handled in an analogous way.

Pawlewicz and Lew (2007) observe that normal df-pn suffers from thrashing the TT in their target domains if more than one promising sibling exists and if the search space does not fit into the TT. Their $1+\epsilon$ trick performs threshold increments in a more radical way even without heuristic initialization to avoid the thrashing issue. It sets $\mathrm{pt}(c) = \min(\mathrm{pt}(n), \lceil \mathrm{pn}_2 \times (1+\epsilon) \rceil)$ where ϵ is a small constant. Thus, df-pn or PDS can stick to searching the subtree rooted at a selected child for a longer time.

7.4 Other Enhancements

In this section four additional enhancements are discussed: (1) *refining heuristic proofs*, (2) *tree simulation*, (3) *adding shallow searches to leaf nodes*, and (4) *detecting terminal positions early.*

7.4.1 Refining Heuristic Proofs

If the game-theoretic value of node n can be estimated with high confidence by a heuristic function, then calculating full proofs or disproofs can profitably be postponed until it later. The iterative algorithm of Schaeffer *et al.* (2005) introduces a heuristic threshold th to define likely wins and losses. Let h be a heuristic score. If $h \geq th$ at node n, then the algorithm considers n to be a "likely win" and does not expand it. Similarly, if $h \leq -th$, then n is treated as a "likely loss". PNS with the heuristic threshold is run until it generates a heuristic proof, a proof tree for which the heuristic evaluation of each leaf node is above the threshold. The algorithm starts with a small value ($th = 125$ in Schaeffer *et al.*, 2005), which is gradually increased after each heuristic proof. This method is used to provide additional guidance for growing the tree. When the root is solved with $th = \infty$, the proof is complete since all leaf nodes in the proof tree are true wins.

7.4.2 Kawano's Simulation

Tree *simulation* (Kawano, 1996) is originally developed to effectively deal with interposing piece drops in tsume-shogi. The technique is later combined with df-pn in applications to Go (Kishimoto, 2005; Yoshizoe *et al.*, 2007). Assume that n is a proven node and m is a "similar" unproven node. Simulation borrows moves from n's proof tree at each OR node to try to find a quick proof of m. If simulation returns a proof for m, m is proven as well. Otherwise, m's value remains unknown and normal df-pn search is performed. A successful simulation requires much less effort than discovering a proof "from scratch" by search. A related technique originally presented in (Tanase, 2000) and later applied to LOA in (Sakuta *et al.*, 2003) is the so-called Killer-Tree Heuristic. Applying this enhancement in the game of LOA, the number of nodes investigated and the solving time are reduced by 20% for PDS.

7.4.3 Adding Shallow Searches To Leaf Nodes

PNS variants often delay finding an easier proof or disproof if the proof and disproof number do not accurately reflect the difficulty of (dis)proving a position. Kaneko *et al.* (2005) perform either one-ply or three-ply depth-first search at each non-terminal OR leaf to try to find a shallow proof quickly. Hashimoto's brinkmate solver (Hashimoto, 2002) calls PDS with a time limit at each leaf of a depth-first iterative deepening search.

7.4.4 Early Win/Loss Detection

Many generic game-playing techniques, which are originally developed for other algorithms such as $\alpha\beta$, also work with PNS variants. Examples include earlier detection of terminal positions and reuse of proofs and disproofs for different positions. For example, the standard technique of retrograde analysis can be combined with PNS variants for detecting terminal positions earlier (Schaeffer *et al.*, 2007; Schadd *et al.*,

2008). Kishimoto and Müller (2003, 2005b) propose a number of game-specific techniques to statically detect winning and losing shapes in the one-eye and life and death problems in Go. Saffidine, Jouandeau, and Cazenave (2012) present domain-specific static patterns that detect wins several moves ahead in the game of Breakthrough. Seo (1999) develops a technique for detecting a dominance relationship of shogi positions, which can reuse proofs mainly related to positions involving interposing piece drops in tsume-shogi.

8. PARALLEL PROOF NUMBER ALGORITHMS

Due to the wide availability of multi-core CPUs in commodity PCs as well as commodity PC clusters, parallelization has become the *de facto* standard approach to improve both the running time and solving abilities of PNS variants. However, achieving reasonable parallel performance is notoriously difficult because of the different inter-dependent types of overhead in parallel search: *search overhead* is extra work performed by parallel search but not by sequential search, *synchronization overhead* is idle time wasted at the synchronization points of parallel computation, and *communication overhead* is caused by the need to exchange messages over a network. Additionally, parallel PNS variants must share data structures such as a TT that preserve proof and disproof numbers of previously expanded nodes. Effective data sharing is especially difficult in distributed memory environments because of communication delays.

In shared memory environments, current approaches share the search tree (or graph) among threads by using mutual exclusion locks. This causes synchronization overhead[11] but avoids search overhead (Saito *et al.*, 2010; Kaneko, 2010).

RP–PNS by Saito *et al.* (2010) uses a randomized work distribution strategy proposed by Shoham and Toledo (2002) to initiate parallelism. There are two kinds of threads in RP–PNS: (1) principal-variation (PV), and (2) alternative threads. RP–PNS maintains one PV thread. All other threads operating on the search tree are alternative threads. In the PV thread, the most-proving node is always selected (similar to PNS). In the alternative threads, either the MPN or one of its siblings is selected probabilistically.

Kaneko (2010) presents a parallel version of df-pn based on asynchronous cooperation of df-pn threads on a shared TT. To initiate parallelism, he uses a *virtual proof number* at each OR node: the sum of the proof number and the number of threads entering that node. A *virtual disproof number* is defined analogously for AND nodes. Each thread computes a MPN based on the virtual proof and disproof numbers, resulting in distributing work to different parts of the search space.

To solve the problem of sharing the TT in distributed-memory environments, current approaches control search by using one master process and a series of slave processes (Kishimoto and Kotani, 1999; Schaeffer *et al.*, 2007; Wu *et al.*, 2011; Saffidine *et al.*, 2012). The master process manages a subtree of the root node, which contains the most important search results, and it coordinates the work of the slave processes. Each slave independently examines its assigned work until a condition determined by the master is satisfied. The search result of the slave is sent back to the master, which then updates its search tree.

In ParaPDS, the master manages a search tree up to a fixed depth d (Kishimoto and Kotani, 1999). The master and slaves run PDS, a predecessor of df-pn. The master traverses the tree in a depth-first manner to depth d and the leaf nodes of the master are passed to the slaves that perform independent search. Finding a set of such leaf nodes is controlled by the thresholds that might be larger than those of the original PDS algorithm.

Unlike ParaPDS, the master process in the following approaches performs variable-depth search. Schaeffer *et al.* (2007) semi-automatically select a set of nodes (at most about 200 nodes) assigned to the slaves (Burch, 2012). It is first determined whether to try to prove or disprove the root by choosing the smaller of proof and disproof numbers of the root node. Assume that the solver tries to prove the root.

[11] The amount of synchronization overhead depends on the choice of sequential and parallel algorithms. While Kaneko (2010) describes that synchronization overhead caused by locks is only about 1% to the total computation, Saito, Winands, and van den Herik (2010) suffer from relatively high synchronization overhead.

Then, the master called the proof tree manager assigns the leaf nodes in the master's subtree to the slaves. Starting from the root, it keeps selecting all the children with smallest proof number at each OR node and all the children at each AND node. An analogous selection scheme is used when the master tries to disprove the root.

In Job-Level Proof-Number Search (JLPNS) (Wu *et al.*, 2011; Wu *et al.*, 2012), the master using PNS distributes heavy-weight work to the slaves. Each job takes about one minute. The master spawns a number of tasks by using *virtual losses*[12], as originally suggested for Monte Carlo Tree Search by Coulom according to Chaslot, Winands, and van den Herik (2008a). The slaves perform heuristic search with domain-specific knowledge.

Parallel PN^2 is an extension of JLPNS and parallelizes PN^2 (Saffidine *et al.*, 2012). The master performs the first level of PNS while the slaves run the second level of PNS. Instead of using the virtual-loss mechanism, PN^2 uses flags to indicate which leaves are already assigned to a slave. Slaves regularly send the pn and dn of their root position, which helps to better guide the first-level search.

9. PROOF-NUMBER SEARCH FOR MULTI-VALUED OUTCOME

Since PNS returns only a binary value, it must perform a series of searches to compute a multi-valued outcome. For example, if the game-theoretic value of the root is a draw, PNS must prove that the root is neither a win nor a loss. To prove that the root is a draw, PNS can assume that a draw is a loss in the first search, and if that search ends with a disproof then redefine draws as wins for the second search. Examples for this technique include solving the games of checkers (Schaeffer *et al.*, 2007) and Fanorona (Schadd *et al.*, 2008).

In general, PNS can determine either a lower bound or an upper bound on the game-theoretic value of a game in a single boolean-valued search. Allis *et al.* (1994) describe two techniques to determine the exact value: examining the root values linearly and using binary search. Kloetzer, Iida, and Bouzy (2008) use the former to solve Amazons endgames.

To determine a three-valued outcome, Kishimoto and Müller's two-phase df-pn search reuses the contents of the transposition table from the first phase, which reduces the amount of re-search in the second phase (Kishimoto and Müller, 2003). The same technique is used to prove seki in Go (Niu, Kishimoto, and Müller, 2006).

Iterative Proof-Number Search determines the exact outcome among more than three values efficiently by caching and updating the lower and upper bounds of each node in the TT to prune the search (Moldenhauer, 2009). For search spaces that are trees, Multiple-Outcome Proof Number Search can determine the multi-valued outcome without performing multiple calls of PNS (Saffidine and Cazenave, 2012).

10. APPLICATIONS OF PNS

To the best of our knowledge, Allis was the first to use the notion of proof and disproof numbers in his analysis of 7×6 Connect Four, although proof and disproof numbers are called *conspiracy numbers* in this early work (Allis, 1988). Elkan (1989) applied proof numbers, which were also still called conspiracy numbers, to theorem proving.

Solving positions in two-player games with perfect information has been the major application of PNS variants. Games solved by using PNS variants for the first time include Go-Moku (Allis *et al.*, 1996), many small board Hex openings (Arneson *et al.*, 2011), Fanorona (Schadd *et al.*, 2008) and checkers (Schaeffer *et al.*, 2007). Examples of two-player games used to investigate new ideas in PNS research, and for which state-of-the-art performance has been achieved, include Awari (Allis *et al.*, 1994), chess (Breuker, 1998), checkers (Schaeffer *et al.*, 2007), Hex (Arneson *et al.*, 2011), Lines of Action (Winands *et al.*, 2004; Pawlewicz and Lew, 2007), Connect6 (Wu *et al.*, 2011), life and death in Go

[12] Wu *et al.* (2011) also use virtual wins, as well as a greedy policy that decides whether to use virtual wins or losses.

(Kishimoto and Müller, 2005b) and tsume-shogi (Seo *et al.*, 2001; Nagai, 2002; Kishimoto, 2010).

PNS variants are also applied to games in other categories such as multi-player games (Saito and Winands, 2010), two-player games with imperfect information (Sakuta, 2001) and moving target search (Moldenhauer, 2009).

A fascinating recent paper by Heifets and Jurisica (2012) formulates the problem of chemical synthesis from given simpler molecules as an AND/OR graph search problem, which is solved by PNS. Finally, ideas from PNS and df-pn$^+$ are incoporated in Minimal Proof Search (MPS), an algorithm for solving model checking problems (Saffidine, 2012).

11. CONCLUSION AND OUTLOOK

In this article we gave a detailed account of PNS variants. Proof-Number Search was designed as an algorithm for solving games, and that remains its main application. Over the past twenty years, a family of PNS variants and a large number of technical improvements have evolved. Based on the large number of successes surveyed in this article, we may conclude that currently PNS variants constitute the state of the art for solving a wide range of games. The choice of a particular PNS variant depends on the problem or game to be solved. If sufficient memory is available, PNS or PN^2 may already be quite efficient in performing the job. For larger problems df-pn may be the best choice.

Based on our experience with PNS variants, we propose the following future research topics:

(1) Perform a detailed empirical study of different PNS variants and enhancements on a representative set of test domains. While many isolated data points exist (cf. van den Herik and Winands, 2008), such a comprehensive study is sorely missing. A more ambitious study would also compare PNS with other solvers based on $\alpha\beta$ (van der Werf, van den Herik, and Uiterwijk, 2003) and MCTS (Winands, Björnsson, and Saito, 2008; Cazenave and Saffidine, 2011; Ewalds, 2012).

(2) Compare existing approaches to parallel PNS, and work on scaling to massively parallel PNS. In Section 8 we surveyed the existing approaches for parallelizing PNS variants. We remark that a direct comparison of ParaPDS, RP–PNS, and parallel df-pn is not possible, as each of them is a parallel version of a different proof-number algorithm. Moreover, ParaPDS was designed for distributed memory systems whereas RP–PNS and Parallel df-pn were designed for shared memory systems. A look at the published raw scaling factors leads us to speculate that none of the three algorithms will scale well to systems with many more processors.

(3) Increase the number and variety of application domains. Until now, most research for PNS variants has been conducted on games. Just like MCTS has been successfully applied to non-game optimization problems, we believe that PNS variants can advance the state of the art in other domains with AND/OR graph structure, such as the chemistry work mentioned above (Heifets and Jurisica, 2012). At this moment this research direction is in its infancy, and it has to be investigated to which extent PNS variants and their enhancements can be transferred to such domains.

(4) Better understand the foundations of search using PNS variants on DCG (cf. Hashimoto, 2011). Either show that current algorithms are complete, or develop new ones that have this property.

Acknowledgements

The authors gratefully acknowledge support from the NWO Project Go for Go, grant number 612.066.409, KAKENHI, and NSERC. The authors are very grateful to Professor Jaap van den Herik for all improvements of the current version.

12. REFERENCES

Akl, S. G. and Newborn, M. M. (1977). The Principal Continuation and the Killer Heuristic. *1977 ACM Annual Conference Proceedings*, pp. 466–473, ACM Press, New York, NY, USA.

Allis, L. V. (1988). A Knowledge-Based Approach of Connect Four: The Game is Over, White to Move Wins. M.Sc. thesis, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. Report No. IR-163.

Allis, L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands.

Allis, L. V., Huntjes, M. P. H., and Herik, H. J. van den (1996). Go-Moku Solved by New Search Techniques. *Computational Intelligence*, Vol. 12, No. 1, pp. 7–23.

Allis, L. V., Meulen, M. van der, and Herik, H. J. van den (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91–124.

Allis, L. V. and Schoo, P. N. A. (1992). Qubic Solved Again. *Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad* (eds. H. J. van den Herik and L. V. Allis), pp. 192–204, Ellis Horwood Ltd., Chichester, England.

Anantharaman, T., Campbell, M., and Hsu, F. (1988). Singular Extensions: Adding Selectivity to Brute-Force Searching. *ICCA Journal*, Vol. 11, No. 4, pp. 135–143. Also published (1990) in *Artificial Intelligence*, Vol. 43, No. 1, pp. 99–109.

Arneson, B., Hayward, R. B., and Henderson, P. (2011). Solving Hex: Beyond Humans. *Computers and Games 2010* (eds. H. J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–10, Springer, Berlin, Germany.

Beal, D. (1990). A Generalized Quiescence Search Algorithm. *Artificial Intelligence*, Vol. 43, No. 1, pp. 85–98.

Beal, D. and Smith, M. C. (1996). Multiple Probes of Transposition Tables. *ICCA Journal*, Vol. 19, No. 4, pp. 227–233.

Björnsson, Y. and Marsland, T. (2001). Multi-Cut Alpha-Beta Pruning in Game-Tree Search. *Theoretical Computer Science*, Vol. 252, Nos. 1–2, pp. 177–196.

Breuker, D. M. (1998). *Memory versus Search in Games*. Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands.

Breuker, D. M., Allis, L. V., and Herik, H. J. van den (1994). How to Mate: Applying Proof-Number Search. *Advances in Computer Chess 7* (eds. H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk), pp. 251–272. University of Limburg, Maastricht, The Netherlands.

Breuker, D. M., Herik, H. J. van den, Uiterwijk, J. W. H. M., and Allis, L. V. (2001). A Solution to the GHI Problem for Best-First Search. *Theoretical Computer Science*, Vol. 252, Nos. 1–2, pp. 121–149.

Breuker, D. M., Uiterwijk, J. W. H. M., and Herik, H. J. van den (1996). Replacement Schemes and Two-level Tables. *ICCA Journal*, Vol. 19, No. 3, pp. 175–180.

Breuker, D. M., Uiterwijk, J. W. H. M., and Herik, H. J. van den (2001). The PN^2-Search Algorithm. *Advances in Computer Games 9* (eds. H. J. van den Herik and B. Monien), pp. 115–132, IKAT, Universiteit Maastricht, Maastricht, The Netherlands.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 1, pp. 1–49.

Bruin, A. de, Pijls, W., and Plaat, A. (1994). Solution Trees as a Basis for Game Tree Search. *ICCA Journal*, Vol. 17, No. 4, pp. 207–219.

Burch, N. (2012). Private Communication.

Buro, M. (1997). The Othello Match of the Year: Takeshi Murakami vs. Logistello. *ICCA Journal*, Vol. 20, No. 3, pp. 189–193.

Campbell, M., Hoane Jr., A. J., and Hsu, F. (2002). Deep Blue. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 57–83.

Campbell, M. (1985). The Graph-History Interaction: On Ignoring Position History. *Proceedings of the 1985 ACM Annual Conference on the Range of Computing: Mid-80's Perspective*, pp. 278–280, ACM, New York, NY, USA.

Cazenave, T. and Saffidine, A. (2011). Score Bounded Monte-Carlo Tree Search. *Computers and Games (CG'10)* (eds. H. J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *Lecture Notes in Computer Science*, pp. 93–104, Springer-Verlag, Berlin, Germany.

Chaslot, G. M. J.-B., Winands, M. H. M., and Herik, H. J. van den (2008a). Parallel Monte-Carlo Tree Search. *Proceedings of the 6th Conference on Computers and Games (CG'08)* (eds. H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 60–71, Springer, Berlin, Germany.

Chaslot, G. M. J.-B., Winands, M. H. M., Uiterwijk, J. W. H. M., Herik, H. J. van den, and Bouzy, B. (2008b). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357.

Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Proceedings of the 5th Computers and Games Conference (CG'06)* (eds. H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 72–83, Springer, Heidelberg, Germany.

Donninger, C. (1993). Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137–143.

Elkan, C. (1989). Conspiracy Numbers and Caching for Searching And/Or Trees and Theorem-Proving. *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89)* (ed. N. S. Sridharan), pp. 341–348.

Ewalds, T. (2012). Playing and Solving Havannah. M.Sc. thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

Gnodde, J. (1993). Aïda: New Search Techniques Applied to Othello. M.Sc. thesis, University of Leiden, Leiden, The Netherlands.

Greenblatt, R., Eastlake, D., and Croker, S. (1967). The Greenblatt Chess Program. *Proceedings of the Fall Joint Computer Conference*, pp. 801–810. Reprinted (1988) in Computer Chess Compendium (ed. D.N.L. Levy), pp. 56-66. Batsford, London, UK.

Hashimoto, J. (2011). *A Study on Game-Independent Heuristics in Game-Tree Search*. Ph.D. thesis, School of Information Science, Japan Advanced Institute of Science and Technology, Nomi, Ishikawa, Japan.

Hashimoto, T. (2002). *Searching for Solutions in Complex Games: Shogi Endgame and Amazons*. Ph.D. thesis, Shizuoka University, Hamamatsu, Japan.

Heifets, A. and Jurisica, I. (2012). Construction of New Medicines via Game Proof Search. *Proceedings of the 26th AAAI Conference on Artificial Intelligence* (eds. J. Hoffmann and B. Selman), pp. 1564–1570.

Herik, H. J. van den, Uiterwijk, J. W. H. M., and Rijswijck, J. van (2002). Games Solved: Now and in the Future. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 277–311.

Herik, H. J. van den and Winands, M. H. M. (2008). Proof-Number search and its variants. *Oppositional Concepts in Computational Intelligence* (eds. H. R. Tizhoosh and M. Ventresca), Vol. 155 of *Studies in Computational Intelligence*, pp. 91–118. Springer, Heidelberg, Germany.

Junghanns, A. (1998). Are there Practical Alternatives to Alpha-Beta? *ICCA Journal*, Vol. 21, No. 1, pp. 14–32.

Kaneko, T. (2010). Parallel Depth First Proof Number Search. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, (AAAI'10)* (eds. M. Fox and D. Poole), pp. 95–100, AAAI Press, Menlo Park, CA, USA.

Kaneko, T., Tanaka, T., Yamaguchi, K., and Kawai, S. (2004). Evaluation Functions for df-pn+ in Shogi based on Prediction of Proof and Disproof Numbers after Expansion. *Proceedings of the 9th Game Programming Workshop (GPW'04)*, pp. 14–21. In Japanese.

Kaneko, T., Tanaka, T., Yamaguchi, K., and Kawai, S. (2005). Df-pn with Fixed-Depth Search at Frontier Nodes. *Proceedings of the 10th Game Programming Workshop (GPW'05)*, pp. 1–8. In Japanese.

Kawano, Y. (1996). Using Similar Positions to Search Game Trees. *Games of No Chance* (ed. R. J. Nowakowski), Vol. 29 of *MSRI Publications*, pp. 193–202, Cambridge University Press.

Kishimoto, A. (2005). *Correct and Efficient Search Agorithms in the Presence of Repetitions*. Ph.D. thesis, University of Alberta, Edmonton, Alberta, Canada.

Kishimoto, A. (2010). Dealing with Infinite Loops, Underestimation, and Overestimation of Depth-First Proof-Number Search. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, (AAAI '10)* (eds. M. Fox and D. Poole), AAAI Press.

Kishimoto, A. and Kotani, Y. (1999). Parallel AND/OR Tree Search Based on Proof and Disproof Numbers. *Proceedings of the 5th Game Programming Workshop*, Vol. 99(14) of *IPSJ Symposium Series*, pp. 24–30, Hakone, Japan.

Kishimoto, A. and Müller, M. (2003). Df-pn in Go: An Application to the One-Eye Problem. *Advances in Computer Games 10 (ACG'03): Many Games, Many Challenges* (eds. H. J. van den Herik, H. Iida, and E. A. Heinz), pp. 125–141, Kluwer Academic Publishers, Boston, MA, USA.

Kishimoto, A. and Müller, M. (2004). A General Solution to the Graph History Interaction Problem. *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04)* (eds. D. L. McGuinness and G. Ferguson), pp. 644–649.

Kishimoto, A. and Müller, M. (2005a). Dynamic Decomposition Search: A Divide and Conquer Approach and its Application to the One-Eye Problem in Go. *2005 IEEE Symposium on Computational Intelligence and Games (CIG'05)*, pp. 164–170, IEEE Press.

Kishimoto, A. and Müller, M. (2005b). Search versus Knowledge for Solving Life and Death Problems in Go. *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'05)* (eds. M. M. Veloso and S. Kambhampati), pp. 1374–1379, AAAI Press / MIT Press, Menlo Park, CA, USA.

Kishimoto, A. and Müller, M. (2008). About the Completeness of Depth-First Proof-Number Search. *Computers and Games 2008* (eds. H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 146–156, Springer, Heidelberg, Germany.

Kloetzer, J., Iida, H., and Bouzy, B. (2008). A Comparative Study of Solvers in Amazons Endgames. *2008 IEEE Symposium on Computational Intelligence and Games (CIG'08)* (eds. P. Hingston and L. Barone), pp. 378–384.

Knuth, D. E. and Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326.

Marsland, T. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3–19.

McAllester, D. A. (1985). A New Procedure for Growing Min-Max Trees. Technical report, Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA.

McAllester, D. A. (1988). Conspiracy Numbers for Min-Max Search. *Artificial Intelligence*, Vol. 35, No. 3, pp. 287–310.

Miwa, M., Yokoyama, D., and Chikayama, T. (2004). Prediction of Mates in Shogi Using SVM and its Application. *Proceedings of the 9th Game Programming Workshop (GPW'04)*, pp. 143–150. In Japanese.

Moldenhauer, C. (2009). Game Tree Search Algorithms for the Game of Cops and Robber. M.Sc. thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

Müller, M. (2003). Proof-Set Search. *Computers and Games 2002* (eds. J. Schaeffer, M. Müller, and Y. Björnsson), Vol. 2883 of *Lecture Notes in Computer Science (LNCS)*, pp. 88–107, Springer, Heidelberg, Germany.

Nagai, A. (1998). A New AND/OR Tree Search Algorithm using Proof Number and Disproof Number. *Proceedings of Complex Games Lab Workshop*, pp. 40–45, ETL, Tsukuba, Japan.

Nagai, A. (1999a). Application of df-pn+ to Othello Endgames. *Proceedings of the 5th Game Programming Workshop*, Vol. 99(14) of *IPSJ Symposium Series*, pp. 16–23.

Nagai, A. (1999b). A New Depth-First-Search Algorithm for AND/OR Trees. M.Sc. thesis, The University of Tokyo, Tokyo, Japan.

Nagai, A. (2002). *Df-pn Algorithm for Searching AND/OR trees and its Applications*. Ph.D. thesis, The University of Tokyo, Japan.

Nagai, A. (2010). Private Communication.

Niu, X., Kishimoto, A., and Müller, M. (2006). Recognizing Seki in Computer Go. *Advances in Computer Games 11* (eds. H. J. van den Herik, S. Hsu, T. Hsu, and H. H. L. M. Donkers), Vol. 4250 of *Lecture Notes in Computer Science (LNCS)*, pp. 88–103, Springer, Berlin, Germany.

Okabe, F. (2005). Application of the Route Branch Number for Solving Tsume Shogi Problems. *Proceedings of the 10th Game Programming Workshop*, pp. 9–16. In Japanese.

Palay, A. J. (1983). *Searching with probabilities*. Ph.D. thesis, Carnagie Mellon University, PA, USA.

Pawlewicz, J. (2012). Private Communication.

Pawlewicz, J. and Lew, L. (2007). Improving depth-first pn-search: $1+\epsilon$ trick. *Proceedings of the 5th Computers and Games Conference (CG'06)* (eds. H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 160–170, Springer, Heidelberg, Germany.

Pijls, W. and Bruin, A. de (1999). Game Tree Algorithms and Solution Trees. *Computers and Games (CG'98)* (eds. H. van den Herik and H. Iida), Vol. 1558 of *Lecture Notes in Computer Science*, pp. 195–204, Springer-Verlag, Berlin, Germany.

Saffidine, A. (2012). Minimal Proof Search for Modal Logic K Model Checking. *JELIA 2012* (eds. L. Fariñas del Cerro, A. Herzig, and J. Mengint), Vol. 7519 of *Lecture Notes in Computer Science (LNCS)*, pp. 346–358, Springer, Berlin, Germany.

Saffidine, A. and Cazenave, T. (2012). Multiple-Outcome Proof Number Search. *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)* (eds. L. De Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas), pp. 708–713, IOS Press.

Saffidine, A., Jouandeau, N., and Cazenave, T. (2012). Solving Breakthrough with Race Patterns and Job-Level Proof Number Search. *Advances in Computer Games 13* (eds. H. J. van den Herik and A. Plaat), Vol. 7168 of *Lecture Notes in Computer Science*, pp. 196–207, Springer, Berlin, Germany.

Saito, J.-T., Chaslot, G. M. J. B., Uiterwijk, J. W. H. M., and Herik, H. J. van den (2007). Monte-Carlo Proof-Number Search. *Proceedings of the 5th Computers and Games Conference (CG'06)* (eds. H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 50–61, Springer, Berlin, Germany.

Saito, J.-T. and Winands, M. H. M. (2010). Paranoid Proof-Number Search. *In Proceedings of the Computational Intelligence and Games Conference (CIG'10)* (eds. G. N. Yannakakis and J. Togelius), pp. 203–210, IEEE Press.

Saito, J.-T., Winands, M. H. M., and Herik, H. J. van den (2010). Randomized Parallel Proof-Number Search. *Proceedings of the 13th Advances in Computers Games Conference (ACG'09)* (eds. H. J. van den Herik and P. Spronck), Vol. 6048 of *Lecture Notes in Computer Science*, pp. 75–87, Springer-Verlag, Heidelberg, Germany.

Sakuta, M. (2001). *Deterministic Solving of Problems with Uncertainty*. Ph.D. thesis, Shizuoka University, Hamamatsu, Japan.

Sakuta, M., Hashimoto, T., Nagashima, J., Uiterwijk, J., and Iida, H. (2003). Application of the Killer-tree Heuristic and the Lamba-Search Method to Lines of Action. *Information Sciences*, Vol. 154, Nos. 3–4, pp. 141–155.

Schadd, M. P. D., Winands, M. H. M., Uiterwijk, J. W. H. M., Herik, H. J. van den, and Bergsma, M. H. J. (2008). Best Play in Fanorona Leads to Draw. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 369–387.

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag, New York, NY, USA.

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers is Solved. *Science*, Vol. 317, No. 5844, pp. 1518–1522.

Schaeffer, J., Björnsson, Y., Burch, N., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2005). Solving Checkers. *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)* (eds. L. P. Kaelbling and A. Saffiotti), pp. 292–297.

Schijf, M. (1993). Proof-Number Search and Transpositions. M.Sc. thesis, University of Leiden, Leiden, The Netherlands.

Schijf, M., Allis, L. V., and Uiterwijk, J. W. H. M. (1994). Proof-Number Search and Transpositions. *ICCA Journal*, Vol. 17, pp. 63–74.

Seo, M. (1995). The C* Algorithm for AND/OR Tree Search and its Application to a Tsume-Shogi Program. M.Sc. thesis, Department of Information Science, The University of Tokyo, Tokyo, Japan.

Seo, M. (1998). Solving Tsume-Shogi by Using Conspiracy Numbers. *Advances in Computer Shogi 2* (ed. H. Matsubara), pp. 1–21. Kyoritsu Shuppan Press. In Japanese.

Seo, M. (1999). On Effective Utilization of Dominance Relations in Tsume-Shogi Solving Algorithms. *Proceedings of the 8th Game Programming Workshop*, pp. 137–144. In Japanese.

Seo, M., Iida, H., and Uiterwijk, J. W. H. M. (2001). The PN*-Search Algorithm: Application to Tsume Shogi. *Artificial Intelligence*, Vol. 129, Nos. 1–2, pp. 253–277.

Shoham, Y. and Toledo, S. (2002). Parallel Randomized Best-First Minimax Search. *Artificial Intelligence*, Vol. 137, Nos. 1–2, pp. 165–196.

Slate, D. J. and Atkin, L. R. (1977). Chapter 4. Chess 4.5 – Northwestern University Chess Program. *Chess Skill in Man and Machine* (ed. P. W. Frey), pp. 82–118, Springer-Verlag, New York, NY, USA.

Soeda, S., Kaneko, T., and Tanaka, T. (2006). Dual Lambda Search and Shogi Endgames. *Advances in Computer Games 11* (eds. H. J. van den Herik, S. Hsu, T. Hsu, and H. H. L. M. Donkers), Vol. 4250 of *Lecture Notes in Computer Science (LNCS)*, pp. 126–139, Springer, Berlin, Germany.

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

Thomsen, T. (2000). Lambda-Search in Game Trees - with Application to Go. *ICGA Journal*, Vol. 23, No. 4, pp. 203–217.

Tsuruoka, Y., Yokoyama, D., and Chikayama, T. (2002). Game-tree Search Algorithm based on Realization Probability. *ICGA Journal*, Vol. 25, No. 3, pp. 132–144.

Ueda, T., Hashimoto, T., Hashimoto, J., and Iida, H. (2008). Weak Proof-Number Search. *CG '08: Proceedings of the 6th International Conference on Computers and Games* (eds. H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands), pp. 157–168, Springer-Verlag, Heidelberg, Germany. ISBN 978–3–540–87607–6.

Uiterwijk, J. W. H. M., Herik, H. J. van den, and Allis, L. V. (1990). A Knowledge-Based Approach to Connect Four: The Game is Over, White to Move Wins. *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad* (eds. D. N. L. Levy and D. F. Beal), pp. 113–133, Ellis Horwood Ltd., Chichester, England.

Werf, E. C. D. van der, Herik, H. J. van den, and Uiterwijk, J. W. H. M. (2003). Solving Go on Small Boards. *ICGA Journal*, Vol. 26, No. 2, pp. 92–107.

Winands, M. H. M. (2004). *Informed Search in Complex Games*. Ph.D. thesis, Maastricht University, The Netherlands.

Winands, M. H. M., Björnsson, Y., and Saito, J.-T. (2008). Monte-Carlo Tree Search Solver. *Proceedings of the 6th Computers and Games Conference (CG'08)* (eds. H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 25–36, Springer, Berlin, Germany.

Winands, M. H. M. and Schadd, M. P. D. (2011). Evaluation-Function Based Proof-Number Search. *Computers and Games (CG'10)* (eds. H. J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *Lecture Notes in Computer Science*, pp. 23–35, Springer-Verlag, Berlin, Germany.

Winands, M. H. M., Uiterwijk, J. W. H. M., and Herik, H. J. van den (2004). An Effective Two-Level Proof-Number Search algorithm. *Theoretical Computer Science*, Vol. 313, No. 3, pp. 511–525.

Wu, I.-C., Lin, H.-H., Lin, P.-H., Sun, D.-J., Chan, Y.-C., and Chen, B.-T. (2011). Job-Level Proof-Number Search for Connect6. *Computers and Games (CG'10)* (eds. H. J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *Lecture Notes in Computer Science*, pp. 11–22, Springer-Verlag, Berlin, Germany.

Wu, I.-C., Lin, H.-H., Sun, D.-J., Kao, K.-Y., Lin, P.-H., Chan, Y.-C., and Chen, P.-T. (2012). Job-Level Proof-Number Search. *IEEE Transactions on Computational Intelligence and AI in Games*. In press.

Xu, C., Ma, Z. M., Tao, J., and Xu, X. (2009). Enhancements of Proof Number Search in Connect6. *Control and Decision Conference*, pp. 4525–4529, IEEE.

Yoshizoe, K. (2009). *AND-OR Tree Search Algorithms for Domains with Uniform Branching Factors*. Ph.D. thesis, The University of Tokyo, Tokyo, Japan.

Yoshizoe, K., Kishimoto, A., and Müller, M. (2007). Lambda Depth-First Proof Number Search and Its Application to Go. *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)* (ed. M. M. Veloso), pp. 2404–2409.

Yoshizoe, K. (2008). A New Proof-Number Calculation Technique for Proof-Number Search. *CG '08: Proceedings of the 6th International Conference on Computers and Games* (eds. H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands), pp. 135–145, Springer-Verlag, Heidelberg, Germany. ISBN 978–3–540–87607–6.