Game Solvers

1

Akihiro Kishimoto and Martin Mueller

Contents

A. Kishimoto (✉)
IBM Research, Ireland Research Lab, Dublin, Ireland
e-mail: akihirok@ie.ibm.com

M. Mueller
University of Alberta, Edmonton, AB, Canada
e-mail: mmueller@ualberta.ca

R. Nakatsu et al. (eds.), *Handbook of Digital Games and Entertainment Technologies*,
DOI 10.1007/978-981-4560-50-4_35

Abstract

Games have simple, fixed rules as well as clear results such as win, draw, or loss. However, developing algorithms for solving games has been a difficult challenge in Artificial Intelligence, because of the combinatorial complexity that the algorithms must tackle.

This chapter presents an overview of successful approaches and results accomplished thus far on game solving. Conducting tree search is a standard way to solve games and game positions. Remarkable progress has been made in developing efficient search algorithms over the last few decades. The chapter describes several standard techniques including $\alpha\beta$ search, proof-number search, and endgame databases.

Keywords

AND/OR tree • Search • $\alpha\beta$ algorithm • Proof-number search • df-pn • Endgame databases

Introduction

Researchers have invested significant resources on research in *two-player zero-sum games with perfect information*. Many popular board games such as chess, checkers, and Go fall into this category, and these games have been used as test beds for testing algorithms in artificial intelligence (AI) research. In this type of zero-sum game, the two players' goals are strictly opposite: when one player wins, the opponent loses. Perfect information means that all information is available to both players. Game positions are typically represented by a board state and the turn to play. Depending on the game, extra information such as the history of the moves played so far may be needed to play and score the result according to the rules.

In a two-player zero-sum game with perfect information, if both players continue playing optimal moves from a position, the final outcome for that position, called the *game-theoretic value* or *value*, is either a win for the first player (i.e., a loss for the second player), or a loss for the first player (i.e., a win for the second player), or a draw (if allowed by the rules of the game). Any finite game can be *solved* in principle since the value of the starting position can be determined by following optimal moves of both players.

Allis (1994) defines three levels of solving a game:

1. *Ultra-weakly solved*. The game-theoretic value of the start position has been determined.
2. *Weakly solved*. A strategy from the start position has been determined to obtain the game-theoretic value of the start position under reasonable computing resources.
3. *Strongly solved*. The game-theoretic value and a strategy have been determined for all legal positions under reasonable computing resources.

There are often significant differences among these three levels in terms of difficulties of achieving the levels of solving. For example, the game of $n \times n$ Hex can be proven to be a win for the first player (Nash 1952). However, winning strategy is only known for $n \leq 10$ (Pawlewicz and Hayward 2014).

For weakly or strongly solving games, the availability of computing resources is restricted to only reasonable ones. In principle, as remarked in Allis (1994), given a large enough amount of time, CPU, and memory resources, games such as chess or Go could be weakly or strongly solved by performing $\alpha\beta$ search or retrograde analysis described later in this chapter. In practice, many games are far too large for a brute force approach, and therefore the development of game solvers that work efficiently under the available resources has been an ongoing challenge.

This chapter gives an overview of the most popular computational approaches for finding strategies for game positions of interest, that is, for at least weakly solving them. Search algorithms are the core of these approaches. In practice, high-performance game solvers combine game-independent search algorithms with game-specific knowledge. While both game-independent and game-specific approaches are necessary to significantly improve the performance of the solvers, the chapter mainly deals with game-independent search algorithms due to their applicability to many games and even to other domains.

Terminology and Definitions on AND/OR Tree and Minimax Tree

Assume that a player p tries to prove a win for a position where p is to play. Then, p must prove that *at least one* of the legal moves leads to a win. However, if it is the opponent's turn to play, p must be able to win against *all* the opponent's moves. This check can be performed recursively, leading to the concept of an AND/OR tree search. The definitions and terminology for AND/OR tree search introduced in this section follow (Kishimoto et al. 2012).

An AND/OR tree is a rooted, finite tree consisting of two types of nodes: *OR* and *AND* nodes. OR nodes correspond to positions where the first player is to play and AND nodes to positions where the second player moves next. A directed edge representing a legal move is drawn from node n to node m if that move played at position n leads to position m.

All nodes except the *root* node have a parent. In this chapter, players are assumed to move alternately. Therefore, each child of an OR node is an AND node, and each child of an AND node is an OR node. In addition, the root is assumed to be an OR node with no loss of generality, but it can be an AND node as well in practice.

Each node in an AND/OR tree has three possible types of values: *win*, *loss*, or *unknown*. As in Kishimoto et al. (2012), the phrase "a node is x" is short for "a node has value x." A node of value win/loss indicates that its corresponding position is a sure win/loss for the first player, respectively. For the sake of simplicity, the value of draw is defined to be the value of loss if possible game outcomes are not explicitly defined. Several techniques for dealing with draws are surveyed in

Kishimoto et al. (2012). A node of value *unknown* indicates that the game-theoretic value of its corresponding position has not yet been proven. To determine its game-theoretic value, such a node must be examined further. *Expanding* a node is the procedure of generating all children of the node, which represent legal moves, and connecting the node to these children by directed edges.

A node with no children is called a *terminal* node. A terminal node is either a win or a loss, as determined by the rules of the game. An *internal* node is a node that has at least one child. A *leaf* node is an unexpanded node with unknown value. A leaf node must be expanded to determine whether it is internal or terminal.

AND/OR tree search aims to *solve* an AND/OR tree, i.e., determine whether the root is a win or a loss. The value of an internal node is calculated 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. At position n, the first player can play a move that leads to that child and win against the second player. If all children of n are losses, n is a loss since all the legal moves of the first player at position n lead to losing positions. Otherwise, n is unknown. Similarly, an internal AND node n is a loss if at least one of its children is a loss, a win if all its children are wins, and unknown otherwise.

A node that is a win is also called a *proven* node, while a node that has been determined to be a loss is a *disproven* node. A *proof* is a computed win, while a *disproof* is a computed loss.

When a node is computed to be a win, a subtree of an AND/OR tree contains a winning strategy for the first player. Such a subtree is called a *proof* tree and guarantees that node is a win. A proof tree T with root node r is constructed as follows:

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

A *disproof tree*, which contains a winning strategy for the second player, is defined in an analogous way, by swapping AND and OR in the definition above, and requiring all terminal nodes to be losses.

Figure 1 illustrates an example of an AND/OR tree. OR nodes are shown by squares and AND nodes are shown by circles. Values win, loss, and unknown are shown by W, L, and U, respectively. Nodes D, F, H, and K are terminal nodes and nodes G, J, and L are leaf nodes. The other nodes are internal nodes for which values are calculated by propagating back the values of the leaf and terminal nodes. For example, node C is a loss because node F is a loss. Irrespective of the value of G, the second player can win against the first player by selecting a move that leads to F. Node I is unknown because node K is a win and node L is unknown. The second player still has a chance to win against the first player by examining L. By following this back-propagation procedure, the value of the root node A is determined to be a win. A proof tree of A is shown with bold lines. Note that for weakly solving the root, AND/OR tree search can ignore nodes that are not part of the

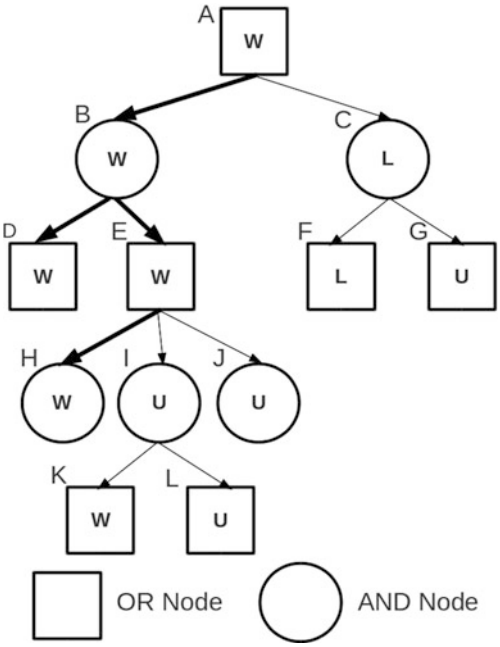

Fig. 1 Example of AND/OR tree

constructed proof tree. For example, in Fig. 1, there is no need to examine the nodes that are not along the bold lines. A high-performance AND/OR tree search algorithm focuses on finding a proof tree as quickly as possible. Assume the standard search space for trees with depth d and branching factor (the number of legal moves at each internal node) b. Also, assume there is only one proof tree in this search space and all terminal nodes are located at depth d. In the worst case, search examines all b^d terminal nodes to find a proof. In contrast, the proof tree contains only $\left\lfloor b^{\frac{d}{2}} \right\rfloor$ terminal nodes.

There may be many proof trees, but finding one is sufficient to solve the root.

In many games, more than one sequence of moves lead to the same position (e.g., Hex and Othello) and a move sequence may lead to a repeated position (e.g., chess and checkers). In other words, the search space of such games can be represented by a directed acyclic graph (DAG) or a directed cyclic graph (DCG). The notion of AND/OR trees, proof, and disproof trees can be generalized for such graphs.

Minimax trees are a generalization of AND/OR trees. Instead of Boolean values, numerical scores are assigned to leaf and terminal nodes. An OR node in such a minimax tree is called a *Max node,* and an AND node is called a *Min node*. The scores are assigned by calling an *evaluation function* that approximates the chance of the first player winning. A larger score indicates that a position is more favorable for the first player. As in AND/OR trees, the score at each internal node of a minimax tree is calculated from the leaf nodes in a bottom-up manner. At an

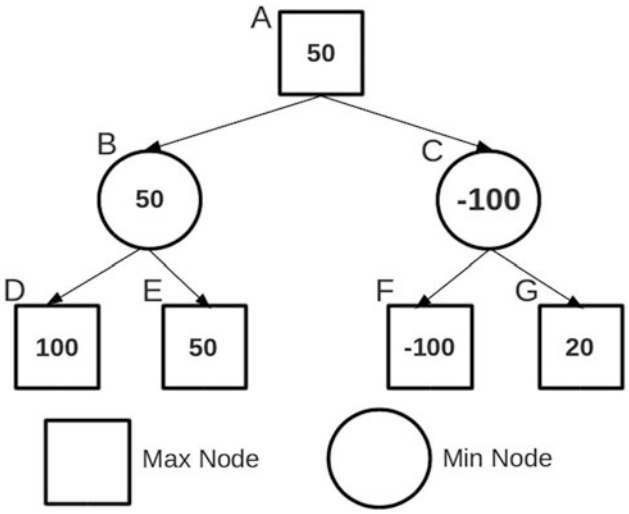

Fig. 2 Example of minimax tree

internal Max node, the first player aims to maximize its advantage by taking the maximum score of all children of that node. At an internal Min node, the second player aims to minimize the advantage of the first player by calculating the minimum score of the children.

Figure 2 illustrates an example of a minimax tree where Max and Min nodes are represented by squares and circles, respectively, and evaluation scores are written inside the squares and circles. The score of the root node *A* becomes 50 by propagating back the scores of leaf nodes *D*, *E*, *F,* and *G*.

The *solution tree* which contains a strategy in the minimax framework is defined in a similar way to the proof tree. For details, see e.g., (de Bruin et al. 1994).

Algorithms for Game Solvers

This section describes general approaches to solve games or game positions. *Forward search* explores a tree from the root until it reaches terminal nodes. *Depth-first search (DFS)* and *best-first search (BFS)* are standard search methods commonly used in many applications including game solvers. While DFS requires only a small amount of memory, it suffers from a combinatorial explosion of its time complexity when the search space is large. In contrast, while BFS tends to explore much smaller search space than DFS, BFS suffers from a combinatorial explosion of its space complexity caused by storing the explored search space in memory. However, game research has revealed that the issue on BFS large memory requirement can be resolved by preserving only important portions of the search

space. In addition, BFS can often be enhanced further by incorporating ideas behind DFS.

Backward search is the other approach to search for a solution and deals with the scenario where all terminal nodes can be enumerated with the available computing resources. Backward search starts with terminal nodes and determines the values of positions toward the root. It can be combined with forward search (e.g., Gasser 1996; Schaeffer et al. 2007). The following sections introduce standard forward and backward search algorithms that can be used for game solvers.

The $\alpha\beta$ Algorithm

The $\alpha\beta$ algorithm (Knuth and Moore 1975) is a depth-first forward search algorithm commonly used in many two-player game-playing programs. $\alpha\beta$ is used to determine the next move to play but can be applied to solve games, for example, by assigning a score of ∞ to terminal positions that are wins for the first player, $-\infty$ to terminal positions that are not wins, and other heuristic values to undecided positions. Basic $\alpha\beta$ examines a minimax tree in a depth-first manner with a fixed depth d to compute the best score of the root node. If d is set deep enough, $\alpha\beta$ returns the winner at the root by returning the score of either ∞ or -∞.

$\alpha\beta$ preserves a lower bound α and a upper bound β on the score of a minimax tree. During performing search, the scores of α and β are updated and used for pruning subtrees that are irrelevant for calculating the score at the root. By incorporating good move ordering such as (Schaeffer 1989), $\alpha\beta$ can reduce the search space to examine by increasing the frequency of pruning subtrees. This enables $\alpha\beta$ to search much deeper and contributes to significantly improving the performance of $\alpha\beta$-based game solvers.

Many variants and enhancements to $\alpha\beta$ have been developed over decades (see the literature review such as (Marsland 1986). *Iterative deepening (ID)* (Slate and Atkin 1977) is a standard enhancement to $\alpha\beta$. ID carries out a series of shallower depth-first search before performing direct search to depth d. That is, ID first performs depth-first search from the root with $d = 1$. Then, if ID finds no solution, it performs depth-first search again from the root with $d = 2$, then $d = 3, 4, \cdots$, and so on. This procedure is repeated until ID either finds a solution, proves that there is no solution, or exhausts resources. Intuitively, because of extra overhead of reexamining previously examined nodes, $\alpha\beta$ combined with ID looks less efficient than basic $\alpha\beta$ that performs direct search to depth d. However, $\alpha\beta$ with ID is empirically more efficient, because previous search results can be used to improve move ordering. As a result, the cost paid for shallower search becomes a small price in order to significantly increase the chance of pruning subtrees for deeper search. For example, the best move calculated in previous shallower search has a high probability that is also the best in deeper search. Examining this move first therefore reduces a large amount of work.

ID is enhanced further by using a *transposition table (TT)* (Greenblatt et al. 1967; Slate and Atkin 1977), a large cache which stores search results of

previously examined nodes such as scores, flags indicating whether these scores are exact, lower bounds or upper bounds, search depths, best moves for shallower search, etc. The transposition table is usually constructed as a hash table and takes an advantage of the fact that the search space of many games are a graph where more than one path can lead to the same node, a so-called *transposition*. The transposition table prevents ID from examining the subtree again by merely retrieving and returning a score saved in the TT, when ID encounters a transposition and verifies that the cached score can be reused. In addition, even if the cached score does not result in eliminating the examination of subtrees, such as the case where a cached node has not been explored deep enough, the best move information in the TT can be used to improve move ordering of $\alpha\beta$, significantly reducing search effort.

Proof-Number Search Variants

An essential drawback of $\alpha\beta$ is that search is limited by fixed depth that causes the minimax tree to grow exponentially with the search depth. The drawback can be alleviated by introducing enhancements such as fractional depth and search extensions (e.g., Campbell et al. 2002; Tsuruoka et al. 2002). However, because they cure the problem of exponential tree growth only partially, $\alpha\beta$-based solvers are still unable to solve positions that require deep search. For example, it is difficult to adjust $\alpha\beta$ to solve positions that depend on narrow but deep lines of play, as occur in Go-Moku and checkmating puzzles in chess-like games (Kishimoto et al. 2012). This section describes proof-number search variants that address this problem.

Basic Proof-Number Search

Proof-Number Search (PNS) (Allis et al. 1994) is a best-first forward search algorithm. PNS calculates the proof and disproof numbers that estimate the difficulty of solving nodes. Based on the proof and disproof numbers, PNS aims to examine nodes in simplest-first order. As long as a node is considered to be easy because of a low proof or disproof number, PNS keeps exploring its subtree without any bound on the search depth. This characteristic enables PNS to find narrow but deep proofs or disproofs efficiently.

Formally, the proof number of a node is defined as the minimum number of leaf nodes in its subtree that must be proven to prove that the node is a win, while the disproof number is the minimum number of such leaf nodes that must be disproven to prove that the node is a loss. The smaller the proof/disproof number is, the easier PNS assumes that it is to prove that a node is a win/loss.

Let n be a node with children $n_1, \cdots, n_k$. One proven child suffices to prove a win at an OR node, while all children must be proven to show a win at an AND node

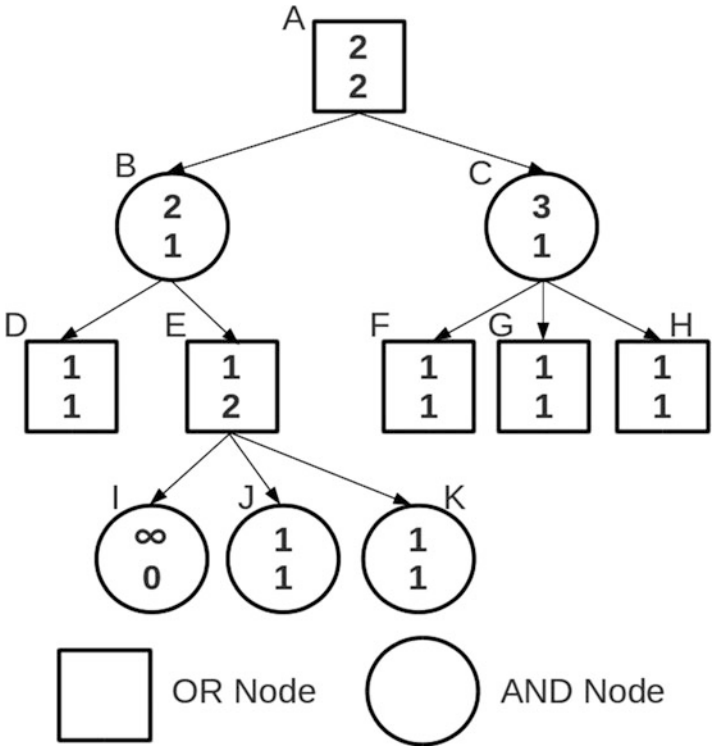

Fig. 3 Example of proof and disproof numbers

(and vice versa for disproof). The proof number $\mathbf{pn}(n)$ and the disproof number $\mathbf{dn}(n)$ of node n are therefore calculated as follows:

1. For a proven terminal node n, $\mathbf{pn}(n) = 0$ and $\mathbf{dn}(n) = \infty$.
2. For a disproven terminal node n, $\mathbf{pn}(n) = \infty$ and $\mathbf{dn}(n) = 0$.
3. For an unknown leaf node n, $\mathbf{pn}(n) = \mathbf{dn}(n) = 1$.
4. For an internal OR node n that has children $c_1, \cdots, c_k$,
   $\mathbf{pn}(n) = \min(\mathbf{pn}(c_1), \cdots, \mathbf{pn}(c_k))$, $\mathbf{dn}(n) = \mathbf{dn}(c_1) + \cdots + \mathbf{dn}(c_k)$.
5. For an internal AND node n that has children $c_1, \cdots, c_k$,
   $\mathbf{pn}(n) = \mathbf{pn}(c_1) + \cdots + \mathbf{pn}(c_k)$, $\mathbf{dn}(n) = \min(\mathbf{dn}(c_1) + \cdots + \mathbf{dn}(c_k))$.

Figure 3 shows an example. The proof and disproof numbers of a node are shown inside that node. The proof number is shown above the disproof number. In this Figure, node I is a terminal node, a loss. Nodes D, F, G, H, J, and K are leaf nodes with proof and disproof numbers initialized to 1. The proof and disproof

numbers of internal nodes are calculated by the rule described above. For example, $\mathbf{pn}(E) = \min(\mathbf{pn}(I), \mathbf{pn}(J), \mathbf{pn}(K)) = \min(\infty, 1, 1) = 1$ and $\mathbf{dn}(E) = \mathbf{dn}(I) + \mathbf{dn}(J) + \mathbf{dn}(K) = 0 + 1 + 1 = 2$.

PNS maintains a proof and a disproof number for each node. In the beginning, the AND/OR tree of PNS consists only of the root node and its proof/disproof numbers are initialized to 1. Then, until either the value of the root is determined, or resources are exhausted, PNS repeats the following four steps:

1. Starting from the root, one path in the tree is traversed until PNS finds a leaf node called a *most-promising node (MPN)* (aka *most-proving node*). To find a MPN, PNS selects an AND child with the smallest proof number among all children at internal OR nodes, and an OR child with the smallest disproof number at internal AND nodes. Ties are broken arbitrarily. In practice, game dependent knowledge can sometimes be used here.
2. The MPN is expanded by generating all its children and adding new edges from the MPN to them. The MPN becomes an internal node and the children are new leaf nodes.
3. If the MPN turns out to be a terminal node, the proof and disproof numbers of the MPN are set according to the rules of the game. Otherwise, the proof and disproof numbers of the new leaf nodes are initialized to 1.
4. The proof and disproof numbers of the affected nodes are recomputed along the path from the MPN back to the root.

Figure 4 illustrates an example of the procedure of PNS. Starting from the root node A, PNS traverses path $A \rightarrow B \rightarrow D$ and finds MPN D. Then, PNS expands D and generates three children of which proof and disproof numbers are initialized to 1. Next, it updates the proof and disproof numbers of D, B, and then A.

Depth-First Proof-Number Search

One inefficiency of basic PNS is that it always propagates back updated proof and disproof numbers from a MPN to the root even if the child with the smallest (dis) proof number remains the same. For example, assume that a MPN is located 100 levels down in the tree from the root. To expand only one leaf node (i.e., MPN), basic PNS must traverse back and forth 100 nodes along the path from the root to the MPN. The depth-first proof-number (df-pn) search algorithm (Nagai 2002) reduces the frequency of reexamining internal nodes. If the search space is a tree, Nagai proves that df-pn is equivalent to PNS in the sense that both algorithms can always select a MPN. This section briefly describes the idea behind df-pn. See (Kishimoto et al. 2012) as well as (Nagai 2002) for precise, detailed descriptions.

The node selection scheme of df-pn is still identical to PNS, which makes df-pn explore the search space in a best-first manner. However, df-pn uses two thresholds to explore the search space in a depth-first manner as well: one for the proof number and the other for the disproof number. If both proof and disproof numbers of a node

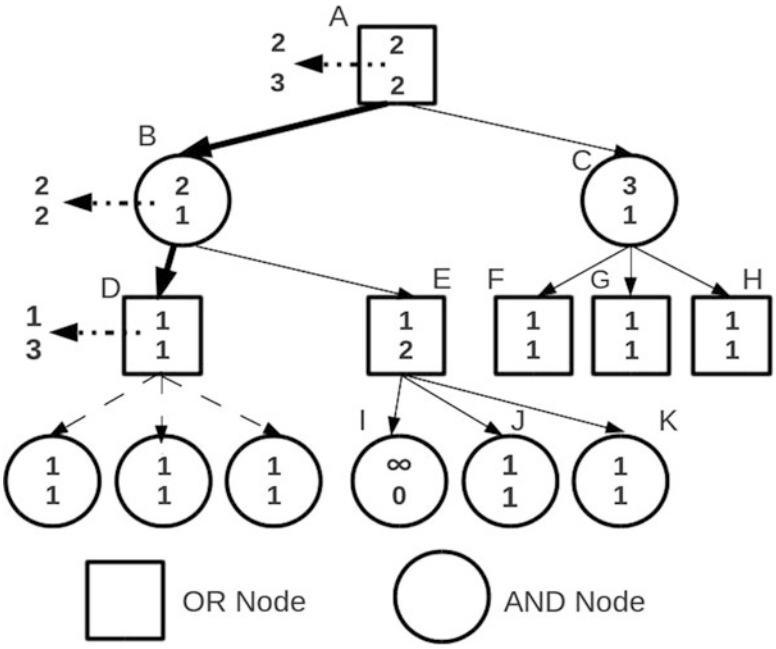

Fig. 4 Example of PNS procedure

n are smaller than the thresholds of proof and disproof numbers, respectively, df-pn continues examining n's subtree without backtracking to n's parent.

Let **thpn**(n) and **thdn**(n) be the thresholds of the proof and disproof numbers at node n, respectively. For example, in Fig. 5, the root OR node A has three children B, C, and D where $\mathbf{pn}(B) = 4$, $\mathbf{pn}(C) = 8$ and $\mathbf{pn}(D) = 10$. B remains on a path to a MPN until $\mathbf{pn}(B)$ exceeds $\mathbf{pn}(C) = 8$, the second smallest proof number among all children. Therefore, the proof number threshold for B, $\mathbf{thpn}(B) = 9$, and search can stay in this subtree without updating exact proof numbers until the threshold is reached. When $\mathbf{pn}(B) \geq 9 = \mathbf{pn}(C) + 1$, the MPN switches to a node below C in the tree.

Thresholds of the disproof number at AND nodes are handled analogously with a disproof threshold. Df-pn can remain in the subtree of the child c_1 with smallest disproof number as long as $\mathbf{dn}(c_1)$ is better than the disproof number of the second best child. For example, in Fig. 5, df-pn sets $\mathbf{thdn}(E) = \mathbf{dn}(F) + 1 = 4 + 1 = 5$ to be able to switch as soon as the disproof number of F becomes strictly smaller than E.

Thresholds for proof numbers of children at AND nodes and disproof numbers of children at OR nodes are set as illustrated below for AND node B in Fig. 5. Assume that $\mathbf{thpn}(B) = 9$, and B has two children E and F with $\mathbf{pn}(E) = 3$ and $\mathbf{pn}(F) = 1$. E is selected for expansion since $\mathbf{dn}(E) < \mathbf{dn}(F)$, and search can stay in its subtree until $\mathbf{pn}(E) + \mathbf{pn}(F)$ reaches the threshold for the parent,

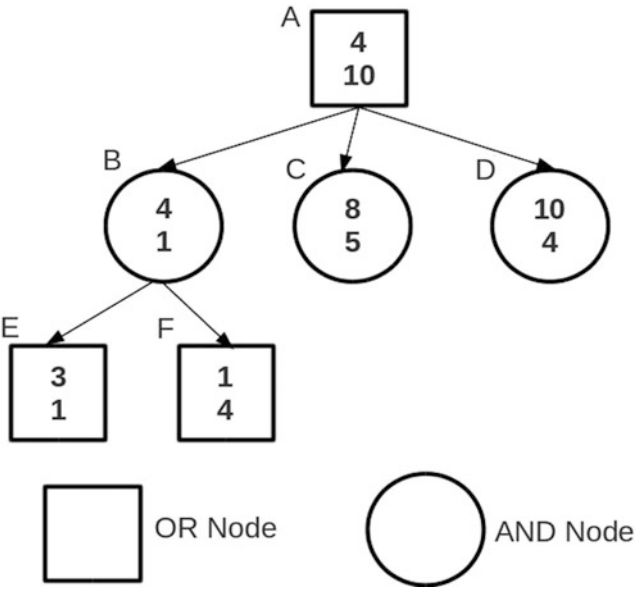

Fig. 5 Illustration of df-pn behavior

thpn$(B) = 9$. Therefore, for searching E, its threshold **thpn**(E) is set to **thpn**(B) – **pn**$(F) = 9–1 = 8$.

Reduction of Memory Requirement

The baseline PNS and df-pn algorithms store all nodes that they expand in main memory. For difficult problems, they quickly run out of space. More practical algorithms use a fixed-size transposition table (TT) with a replacement strategy (Breuker 1998; Seo et al. 2001; Nagai 2002). This solves the memory problem but adds overhead from re-expanding nodes that have been overwritten in the table.

The *SmallTreeGC* algorithm (Nagai 1999) is an effective replacement strategy used in many high-performance solvers (Nagai 2002; Kishimoto and Müller 2005; Kishimoto 2010; Kaneko 2010). The TT entry for each node stores the size of the subtree rooted at that node. SmallTreeGC assumes that a small subtree can be reconstructed easily with a small amount of effort even if that subtree is not available in the TT. 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.

According to Kishimoto et al. (2012), Pawlewicz uses the multiple-probe replacement strategy of Beal and Smith (1996) in their Hex df-pn implementation. Their multiple-probing TT replacement strategy implementation 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.

PN^2 and its variants are another approach which reduces the memory requirement of PNS (Allis 1994; Breuker 1998; Winands et al. 2004). The main idea is to use two levels of PNS. The first level, PN_1, stores a tree starting from the root node. At a leaf node of PN_1, PN^2 invokes the second level of PNS, PN_2, with limited number of nodes expanded and a separate memory allocation. After PN_2 completes search, this PN_2 tree is discarded and only the proof and disproof numbers of the root, corresponding to a PN_1 leaf node, are passed back to PN_1.

A hybrid approach was used to solve the game of checkers (Schaeffer et al. 2007). As in PN^2, a disk-based first-level search called the *front-end manager* was based on PNS. It invoked a memory-only second level search called the *back-end prover*. This back-end used df-pn with a TT and SmallTreeGC.

PNS Variants in Directed Acyclic and Cyclic Graphs

There are three problems to address when PNS and df-pn searches in DAG or DCG. The overestimation problem results from double-counting proof and disproof numbers of nodes in a DAG that can be reached along multiple paths (Allis et al. 1994). The Graph-History Interaction (GHI) problem (Palay 1983; Campbell 1985), which can occur both in $\alpha\beta$ and PNS variants, originates from incorrect handling of cycles in DCG. The infinite loop problem (Kishimoto and Müller 2003; 2008) refers to the phenomenon that df-pn may loop forever without expanding any new nodes in DCG. Current solutions to these problems are summarized in Kishimoto et al. (2012).

Search Enhancements

Although PNS variants are already powerful without domain knowledge, high-performance game solvers incorporate many search enhancements to be able to solve difficult game positions. This section describes some of the well-known methods. See Kishimoto et al. (2012) for comprehensive survey.

The proof and disproof numbers of the leaf node are always set to 1 in their original definition. However, in practice, some leaf nodes are easier to (dis)prove than others. With *heuristic initialization*, leaf nodes are heuristically initialized with proof and disproof numbers (Allis 1994). One example for this is using the number of legal moves at a leaf node, which can be calculated with little overhead in some games (Allis et al. 1994). Another popular approach is using domain-specific heuristic functions (Nagai 2002). One important remark is that the thresholds of df-pn must be increased to reduce the overhead of internal node re-expansions when heuristic initialization is combined with df-pn, as in df-pn^+ (Nagai 2002; Kishimoto and Müller 2005).

If more than one promising sibling exist and the search space does not fit into the TT, df-pn sometimes suffers from thrashing the TT by fast switches between subtrees. The $1 + \varepsilon$ trick (Pawlewicz and Lew 2007) increases the threshold when

df-pn selects the best child. This enables df-pn to stay longer in the subtree of one child, without frequently switching to other promising siblings.

Given a proven node n, tree *simulation* (Kawano 1996) performs a quick proof check for an unknown node m that looks similar to n. The similarity of positions is usually defined in a game-specific way. Unlike normal search such as df-pn that generates all legal moves, simulation is restricted to generate moves borrowed from n's proof tree at each OR node and checks if a proof tree of m can be constructed in a similar way to n's proof tree. If simulation succeeds, m is proven as well. Otherwise, m's value remains unknown and normal search is performed. Considering that any search must examine all nodes in m's proof tree even in an ideal case, simulation makes almost the smallest effort in proving m, therefore, requires much less effort than normal search.

Schaeffer et al. (2007)'s PNS-based solver estimates the game-theoretic value of a node with high confidence by using scores from their $\alpha\beta$-search-based game-playing program. Let s be such a score of node n, and th be a threshold for an iterative PNS algorithm. If $s \geq th$, n is regarded as a terminal node with value *likely win*. Similarly, if $s \leq -th$, n is considered to be a *likely loss*. As in iterative deepening, the algorithm starts with a small value of th and gradually increases th after constructing each heuristic proof tree. When the root is solved with $th = \infty$, a true proof tree is constructed, where all leaf nodes are true values, and the proof is complete.

Endgame Databases

In many *converging* games (Allis 1994) such as chess and checkers, the number of pieces on the board decreases as the game progresses. This indicates that it is often feasible to enumerate all *endgame positions* that occur close to the end of the game. In this case, the game-theoretic values of these endgame positions can be precomputed and saved as *endgame databases* that map a position to its corresponding game-theoretic value. If the endgame databases contain the game-theoretic values of all the legal positions for a game, that game is strongly solved. As an example, the database for the game of Awari calculated by Romein and Bal (2003) contains the exact scores of all legal Awari positions.

Retrograde analysis (Bellman 1965; Thompson 1986) is a standard backward search algorithm to systematically construct endgame databases, starting from terminal nodes and progressing toward the root. Retrograde analysis has been successfully used in many domains (Lake et al. 1994; Schaeffer et al. 2003; Romein and Bal 2003). Additionally, endgame databases contribute to significantly improving the ability of game solvers that employ forward search (Schaeffer et al. 2007), because forward search can obtain the game-theoretic value of a node without deeper search.

Assume that the game-theoretic value of a game is either a win, a loss, or a draw, and a repeated position is defined to be a draw by the rule of that game. Then, one way to implement retrograde analysis is summarized as follows:

1. Let S be a set containing all positions, that may include unreachable ones from the root.
2. Initialize: assign the game-theoretic value determined by the rules of the game to all terminal nodes and a value of *unknown* to all other positions.
3. For each node $n \in S$ with value unknown, check if n's value can be determined from its children. For example, OR node n is a win if n has at least one child in S that is a win.
4. Repeat step 3 until the number of nodes with value unknown no longer decreases.
5. Mark the nodes of value unknown as draws since these nodes are either unreachable from the root, or no win or loss can be forced by either player.

When constructing a database for a difficult game in practice, retrograde analysis requires large CPU and storage resources. Implementations typically use parallel machines as well as large amounts of both main memory and hard disk space. There are several approaches that achieve efficient parallelism and reduce disk I/O. For example, as in paging of the operating system, retrograde analysis determines which part of the databases should be preserved in the main memory to alleviate the overhead of disk I/O (Lake et al. 1994; Romein and Bal 2003). Schaeffer et al. (2003) not only compress databases space-efficiently but also decompress the databases in real-time so that forward search can use them both time- and memory-efficiently. Techniques to reduce the main memory requirement at the cost of a small computational overhead are presented in Lincke (2002); Romein and Bal (2003).

To initiate parallelism, Romein and Bal (2003) present an asynchronous distributed-memory parallel algorithm that overlaps database computation and processor communication via network. In contrast, Lake et al. (1994) split the search space into a set of small slices that can be solved easily and independently. The correctness of databases must be verified, since both software and hardware errors including disk and network errors may occur during a large-scale computation (Schaeffer et al. 2003; Romein and Bal 2003).

Other Approaches

This section briefly describes other approaches related to forward search.

Threat-Based Approaches

Threats are moves to which a player must reply directly to win, or to avoid an immediate loss. If threats exist in a position, threat-space search (Allis 1994; Allis et al. 1996) considers only threats and safely ignores other legal moves. Threat-space search can significantly reduce the search space to explore, because the number of threats is usually much smaller than that of the legal moves. The idea

behind threat-space search is used in solvers such as Henderson et al. (2009) and Kishimoto and Müller (2005).

Different levels of threat sequences, which are a generalization of direct threats, can be detected by using *pass* moves. Intuitively, moves are limited by using the information on how many moves a player needs to make in a row so that that player has a forced win. Examples of such methods are λ-search (Thomsen 2000) and generalized threats search (Cazenave 2002). These techniques can be applied to both $\alpha\beta$ and df-pn (Nagai 2002; Yoshizoe et al. 2007).

Early Win/Loss Detection

Game-specific features can sometimes be used to statically detect whether a player is winning or losing. Such static win/loss detection can significantly reduce the size of a proof tree, as well as the search space explored by forward search. Examples include virtual connections in the game of Hex (Anshelevich 2002; Henderson et al. 2009) and the detection of eye space and potential eye space in the life and death problem in Go (Wolf 1994; Kishimoto and Müller 2005).

Detecting a dominance relationship between positions can also contribute to recognizing wins or losses early. An important example is the checkmating problem in shogi (Japanese chess) (Seo 1999). For example, assume that forward search proves that node n is a win and then encounters unproven node m that dominates n. Then, m is also a win because the first player can copy the winning strategy from n.

Monte Carlo Tree Search Solver

Monte Carlo Tree Search (MCTS) is a forward search algorithm that has achieved remarkable success in playing the game of Go and many other games where accurate evaluation functions are difficult to develop (see the chapter on computer Go, written by Yoshizoe and Müller). MCTS combines tree search with Monte Carlo simulation that is used to evaluate a leaf node. In addition to Monte Carlo simulation results, if game-theoretic values of wins or losses are available, the MCTS-Solver propagates back these values (Winands et al. 2008). This modification enables MCTS-Solver to solve positions.

Probability Propagation

Instead of using proof and disproof numbers as the criteria to explore an AND/OR tree, probability propagation (PP) performs best-first forward search based on a probability of a first player win for each node in an AND/OR tree (Pearl 1984; Stern et al. 2007). Enhancements similar to the ones for PNS can be incorporated into PP,

including heuristic initialization (Stern et al. 2007), transposition tables, and two-level search (Saffidine and Cazenave 2013).

Results Accomplished on Solving Games

The techniques described thus far contributed to solving many nontrivial, popular games that people actually play, such as Awari (Romein and Bal 2003), checkers (Schaeffer et al. 2007), Connect-Four (see http://tromp.github.io/c4/c4.html and (Allis 1988), and Go-Moku (Allis et al. 1996).

Researchers use games with smaller boards as a research test bed to develop new algorithms. Examples of games solved in this way are 5 × 5 Go (van der Werf et al. 2003), 6 × 6 Othello (see http://www.tothello.com/), 10 × 10 Hex (Pawlewicz and Hayward 2014), and 5 × 6 Amazons (Song and Müller 2014).

In chess, endgame databases are constructed for most positions with 3–7 pieces (see http://chessok.com/?page_id=27966). Puzzle or endgame solvers based on PNS variants have been developed to achieve state-of-the-art performance in many games, such as chess (Breuker 1998), checkers (Schaeffer et al. 2007), Hex (Arneson et al. 2011), life and death in Go (Kishimoto and Muller 2005), and tsume-shogi (Seo et al. 2001; Nagai 2002; Kishimoto 2010). In particular, research on tsume-shogi has achieved remarkable success by testing with many difficult tsume-shogi problems created by human experts. State-of-the-art solvers based on df-pn can solve all the existing hard problems with solution sequences longer than 300 steps, including Microcosmos with 1525 steps.

Sophisticated game solvers combine many of the techniques discussed. For example, the checkers solver of Schaeffer et al. (2007) incorporates PNS, df-pn, $\alpha\beta$, endgame databases, and many other game-independent and game-specific enhancements.

Conclusions

This chapter presented an overview of the techniques for weakly or strongly solving games or game positions. Game positions can be solved by forward search, backward search, or a combination. As standard algorithms to perform forward search, the chapter introduced $\alpha\beta$ and PNS variants. Backward search for building endgame databases uses retrograde analysis. Additionally, the chapter gave an overview of other approaches including threat-space search, MCTS and PP, followed by a summary of the results accomplished thus far on solving games.

Recommended Reading

L.V. Allis, *A Knowledge-Based Approach of Connect Four: The Game is Over, White to Move Wins*. Master's thesis, Vrije Universiteit Amsterdam, Amsterdam, 1988. Report No. IR-163

L.V. Allis, *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, 1994

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

L.V. Allis, M.P.H. Huntjes, H.J. van den Herik, Go-moku solved by new search techniques. Comput. Intell. **12**(1), 7–23 (1996)

V. Anshelevich, A hierarchical approach to computer Hex. Artif. Intell. **134**(1–2), 101–120 (2002)

B. Arneson, R.B. Hayward, P. Henderson, Solving Hex: beyond humans, in *Computers and Games 2010*, ed. by H.J. van den Herik, H. Iida, A. Plaat. Lecture Notes in Computer Science (LNCS), vol. 6515 (Springer, Berlin, 2011), pp. 1–10

D. Beal, M.C. Smith, Multiple probes of transposition tables. ICCA J. **19**(4), 227–233 (1996)

R. Bellman, On the application of dynamic programming to the determination of optimal play in chess and checkers. Proc. Natl. Acad. Sci. U. S. A. **53**, 244247 (1965)

D.M. Breuker, *Memory Versus Search in Games*. PhD thesis, Universiteit Maastricht, Maastricht, 1998

M. Campbell, The graph-history interaction: on ignoring position history, in *Proceedings of the 1985 ACM Annual Conference on the Range of Computing: Mid-80's Perspective*, (ACM, New York, 1985). pp. 278–280

M. Campbell, A.J. Hoane Jr., F. Hsu, Deep Blue. Artif. Intell. **134**(1–2), 57–83 (2002)

T. Cazenave, A generalized threats search algorithm, in *Computers and Games 2002*, ed. by J. Schaeffer, M. Müller, Y. Björnsson. Lecture Notes in Computer Science (LNCS) (Springer, Heidelberg, 2002), pp. 75–87

A. de Bruin, W. Pijls, A. Plaat, Solution trees as a basis for game tree search. ICCA J. **17**(4), 207–219 (1994)

R. Gasser, Solving Nine Men's Morris, in *Games of No Chance*, ed. by R.J. Nowakowski. MSRI Publications, vol. 29 (Cambridge University Press, Cambridge, 1996), pp. 101–113

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

P. Henderson, B. Arneson, R.B. Hayward, Solving 8×8 Hex, in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, ed. by C. Boutilier (AAAI Press, Pasadena, 2009), pp. 505–510

T. Kaneko, Parallel depth first proof number search, in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, (AAAI'10)*, ed. by M. Fox, D. Poole (AAAI Press, Menlo Park, 2010), pp. 95–100

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

A. Kishimoto, Dealing with infinite loops, underestimation, and overestimation of depth-first proof-number search, in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, (AAAI'10)*, ed. by M. Fox, D. Poole (AAAI Press, 2010)

A. Kishimoto, M. Müller, About the completeness of depth-first proof-number search, in *Computers and Games 2008*, ed. by H.J. van den Herik, X. Xu, Z. Ma, M.H.M. Winands. Lecture Notes in Computer Science, vol. 5131 (Springer, Heidelberg, 2008), pp. 146–156

A. Kishimoto, M. Müller, Df-pn in Go: an application to the one-eye problem, in *Advances in Computer Games 10 (ACG'03): Many Games, Many Challenges*, ed. by H.J. van den Herik, H. Iida, E.A. Heinz (Kluwer, Boston, 2003), pp. 125–141

A. Kishimoto, M. Müller, Search versus knowledge for solving life and death problems in Go, in *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'05)*, ed. by M.M. Veloso, S. Kambhampati (AAAI Press/MIT Press, Menlo Park, 2005), pp. 1374–1379

A. Kishimoto, M. Winands, M. Müller, J.-T. Saito, Game-tree search using proof numbers: the first twenty years. ICGA J. **35**(3), 131–156 (2012)

D.E. Knuth, R.W. Moore, An analysis of alpha-beta pruning. Artif. Intell. **6**(4), 293–326 (1975)
R. Lake, J. Schaeffer, P. Lu, Solving large retrograde analysis problems using a network of workstations. Advances in Computer Games **7**, 135–162 (1994)
T. Lincke, *Exploring the Computational Limits of Large Exhaustive Search Problems*. PhD thesis, ETH Zürich, 2002
T. Marsland, A review of game-tree pruning. ICCA J. **9**(1), 3–19 (1986)
A. Nagai, A new depth-first-search algorithm for AND/OR trees. Master's thesis, The University of Tokyo, Tokyo, 1999
A. Nagai, *Df-pn Algorithm for Searching AND/OR trees and its Applications*. PhD thesis, The University of Tokyo, 2002
J. Nash, Some games and machines for playing them. *Technical Report D-1164*, Rand Corp., 1952
A.J. Palay, *Searching with Probabilities*. PhD thesis, Carnagie Mellon University, 1983
J. Pawlewicz, R. Hayward, Scalable parallel depth first proof number search, in *Computers and Games (CG 2013)*, vol. 8427. *Lecture Notes in Computer Science* (Springer, 2014), pp. 138–150
J. Pawlewicz, L. Lew, Improving depth-first pn-search: 1+ε trick, in *Proceedings of the 5th Computers and Games Conference (CG'06)*, ed. by H.J. van den Herik, P. Ciancarini, H.H.L.M. Donkers. Lecture Notes in Computer Science, vol. 4630 (Springer, Heidelberg, 2007), pp. 160–170
J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (Addison-Wesley, Reading, 1984)
J.W. Romein, H. Bal, Solving the game of Awari using parallel retrograde analysis. IEEE Comput. **36**(10), 26–33 (2003)
A. Saffidine, T. Cazenave, Developments on product propagation. in *Computer Games 2013*, vol. 8427. *Lecture Notes in Computer Science*, 2013, pp. 100–109
J. Schaeffer, The history heuristic and alpha-beta search enhancements in practice. IEEE Trans. Pattern Anal. Mach. Intell. **11**(1), 1203–1212 (1989)
J. Schaeffer, Y. Björnsson, N. Burch, R. Lake, P. Lu, S. Sutphen, Building the checkers 10-piece endgame databases, in *Advances in Computer Games 10*: Many Games, ed. by H.J. van den Herik, H. Iida, E.A. Heinz. (Kluwer, Boston, 2003), pp. 193–210
J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, S. Sutphen, Checkers is solved. Science **317**(5844), 1518–1522 (2007)
M. Seo, On effective utilization of dominance relations in tsume-shogi solving algorithms, in *Proceedings of the 8th Game Programming Workshop*, 1999, pp. 137–144 (in Japanese)
M. Seo, H. Iida, J.W.H.M. Uiterwijk, The PN*-search algorithm: Application to tsume shogi. Artif. Intell. **129**(1–2), 253–277 (2001)
D.J. Slate, L.R. Atkin, Chapter 4. Chess 4.5 – Northwestern University chess program, in *Chess Skill in Man and Machine*, ed. by P.W. Frey (Springer, New York, 1977), pp. 82–118
J. Song, M. Müller, An enhanced solver for the game of Amazons, 2014, in *Accepted for IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG),* 12 pp
D. Stern, R. Herbrich, T. Graepel, Learning to solve game trees, in *Proceedings of the 24th International Conference of Machine Learning (ICML)*, 2007, pp. 839–846
K. Thompson, Retrograde analysis of certain endgames. ICCA J. **9**(3), 131–139 (1986)
T. Thomsen, Lambda-search in game trees – with application to Go. ICGA J. **23**(4), 203–217 (2000)
Y. Tsuruoka, D. Yokoyama, T. Chikayama, Game-tree search algorithm based on realization probability. ICGA J. **25**(3), 132–144 (2002)
E.C.D. van der Werf, H.J. van den Herik, J.W.H.M. Uiterwijk, Solving Go on small boards. ICGA J. **26**(2), 92–107 (2003)
M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, An effective two-level proof-number search algorithm. Theor. Comput. Sci **313**(3), 511–525 (2004)
M.H.M. Winands, Y. Björnsson, J.-T. Saito, Monte-Carlo tree search solver, in *Proceedings of the 6th Computers and Games Conference (CG'08)*, ed. by H.J. van den Herik, X. Xu, Z. Ma,

M.H.M. Winands. Lecture Notes in Computer Science, vol. 5131 (Springer, Berlin, 2008), pp. 25–36

T. Wolf, The program GoTools and its computer-generated tsume Go database, in *1st Game Programming Workshop in Japan (Hakone)*, 1994

K. Yoshizoe, A. Kishimoto, M. Müller, Lambda depth-first proof number search and its application to Go, in *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, ed. by M.M. Veloso (2007), Morgan Kaafmann, San Francisco, pp. 2404–2409