Search versus Knowledge for Solving Life and Death Problems in Go

Akihiro Kishimoto
Department of Media Architecture,
Future University-Hakodate
116-2, Kamedanakano-cho, Hakodate,
Hokkaido, 041-8655, Japan
kishi@fun.ac.jp

Martin Müller
Department of Computing Science,
University of Alberta
Edmonton, Canada T6G 2E8
mmueller@cs.ualberta.ca

Abstract

In games research, Go is considered the classical board game that is most resistant to current AI techniques. Large-scale knowledge engineering has been considered indispensable for building state of the art programs, even for subproblems such as Life and Death, or tsume-Go. This paper describes the technologies behind TSUMEGO EXPLORER, a high-performance tsume-Go search engine for enclosed problems. In empirical testing, this engine outperforms *GoTools*, which has been the undisputedly best tsume-Go program for 15 years.

Introduction

Progress in AI can be achieved in many different ways, through new algorithms, combinations of existing approaches, knowledge transfer from other disciplines, and many more. Progress can also be demonstrated through leaps in practical performance on problems that are considered hard for AI. This paper falls into the latter category. Its contributions are:

- The design and implementation of a high-performance search engine for the difficult AI domain of Life and Death problems, or *tsume-Go*.
- A synthesis and extension of several recent improvements of the depth-first proof-number search algorithm (df-pn) (Nagai 2002).
- A small but effective and efficient set of domain-specific search enhancements.
- Experimental results that demonstrate that TSUMEGO EXPLORER improves upon the current state of the art in solving tsume-Go.

Proficiency in solving tsume-Go is one of the most important skills for AI programs that play the ancient Asian game of Go. For 15 years, Thomas Wolf's program *GoTools* (Wolf 1994; 2000) has been the undisputedly strongest program for solving tsume-Go. Wolf's groundbreaking work led to the first program that could play an interesting, highly nontrivial part of the game of Go on a level equivalent to strong human masters. One distinctive feature of GoTools is that it contains a large amount of Go-specific knowledge. Such knowledge is used for move ordering heuristics that speed up the search, and for static position evaluation that recognizes wins and losses early.

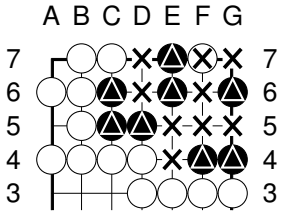

Figure 1: A typical tsume-Go problem: White to play and kill all black stones.

This paper presents TSUMEGO EXPLORER, a different approach to solving tsume-Go problems, with a focus on efficient search techniques rather than extensive domain knowledge. The core of the algorithm is an enhanced version of the depth-first proof-number search algorithm (df-pn) (Nagai 2002). The enhancements allow the search to effectively deal with the complications of Go such as position repetitions, called *ko*. Even with relatively simple domain knowledge, TSUMEGO EXPLORER is shown to outperform GoTools, and scale better to larger problems. The experimental results demonstrate the potential of efficient search-based approaches to Go, at least for the restricted domain of tsume-Go. This success may have implications on the design of future generations of Go programs.

As in GoTools, this research focuses on *enclosed* problems which are separated from the rest of a Go board by a wall of safe, invulnerable stones. Figure 1, adapted from (Wolf 2000), shows a typical example. The long unmarked chain of white stones forms the outside boundary of the problem. In games, both fully enclosed and loosely surrounded open boundary positions occur frequently.

The Tsume-Go Problem

An enclosed tsume-Go problem is defined by the following parameters:

- Two players, called the *defender* and the *attacker*. The defender tries to live and the attacker tries to kill. Either player can be specified as moving first.

- The *region*, a subset of the board. At each turn, a player must either make a legal move within the region or pass.
- A wall of *safe attacker stones* surrounding the region.
- A set of *crucial* defender stones within the region.

In Figure 1, Black is the defender and White is the attacker. Crucial stones are marked by triangles and the rest of the region is marked by crosses.

The outcome of a tsume-Go problem is binary, win or loss. The defender wins by saving at least one crucial stone from capture, typically by creating two eyes connected to the stone(s). The attacker wins by capturing *all* crucial stones, which can be achieved by preventing the defender from creating two eyes in the region. Coexistence in *seki* is considered a win for the defender, since the stones become safe from capture. The *situational super-ko (SSK) rule* is used, under which any move that repeats a previous board position, with the same color to play, is illegal. For details, see the section on treatment of ko below.

Related Work

Previous Work on Tsume-Go

A tsume-Go solver consists of two main parts: evaluation and search. Both exact solvers and inexact heuristic approaches are popular in practice.

The simplest solvers use only static evaluation and no search. Algorithms include Benson's method for detecting unconditional life (Benson 1976), Müller's safety by alternating play (Müller 1997), and Vilà and Cazenave's method for classifying large eye shapes (Vilà & Cazenave 2003).

All strong computer Go programs contain a module for analyzing life and death, often using search with a combination of exact and heuristic rules (Chen & Chen 1999; Fotland 2002). The downside of the use of heuristics are possibly incorrect answers, which might lose a game.

Among exact solvers, for 15 years, Wolf's *GoTools* (Wolf 1994) has been the best. GoTools uses a special-purpose depth-first $\alpha\beta$ search algorithm. A transposition table reduces search effort by storing won and lost positions. GoTools contains a sophisticated evaluation function that includes look-ahead aspects, powerful rules for static life and death recognition, and learning of dynamic move ordering from the search (Wolf 2000). One of the most important enhancements in GoTools is dynamic move ordering using the subtrees explored so far. If a move m_1 at position P is refuted by the opponent playing m_2, then m_2 is tried next at P, since it is a likely "killer" move. Successful moves from subsequent positions in the search also get some credit, which achieves better move ordering.

Depth-First Proof-Number Search

Df-pn (Nagai 2002) is an efficient depth-first version of proof-number search (Allis, van der Meulen, & van den Herik 1994). Nagai used df-pn to develop the currently best solver for tsume-shogi, checkmating problems in Japanese chess. Df-pn(r) is an enhancement of df-pn (Kishimoto & Müller 2003; Kishimoto 2005) that is able to deal with position repetitions, which are very common in Go. (Kishimoto & Müller 2003) applied df-pn(r) to the one-eye problem, a special case of tsume-Go. Despite a relatively small amount of Go-specific knowledge, the method could solve harder problems than the best general tsume-Go solvers.

The question addressed in the current paper is whether an approach along the lines of (Kishimoto & Müller 2003) can be effective for full tsume-Go. Evaluation in tsume-Go is much more complicated than in the one-eye problem. It requires checking for two eyes, dynamic detection of *seki*, and testing connections between the stones surrounding the eyes. In strong previous solvers such as GoTools, years of hard work have gone into the development of game-specific knowledge for static position evaluation.

The TSUMEGO EXPLORER Algorithm

This section discusses evaluation by static life and death detection, dynamic detection of seki, addition of basic game-specific knowledge, and standard df-pn enhancements such as Kawano's simulation (Kawano 1996) and heuristic initialization of proof and disproof numbers.

Evaluation of Terminal Positions

The defender can win by creating two complete eyes connected to at least one crucial stone in the region. The attacker aims to eliminate *potential eye points*, where an eye can possibly be created. The attacker wins by creating a *dead shape*, where no two nonadjacent potential eye points remain in the region. Seki is considered to be a defender win. It is detected dynamically by search, when the defender passes and the attacker still cannot win. Only basic one and two point eyes are recognized statically.

Game-Specific Knowledge

The following game-specific knowledge is incorporated into TSUMEGO EXPLORER: Connections to safe stones, forced moves, Kawano's simulation (Kawano 1996), and heuristic initialization of proof and disproof numbers.

Safety by connections to safe stones Connections by a *miai* strategy (Müller 1997) are used to promote unsafe attacker stones to safe. Promoted safe attacker stones help to reduce the number of potential eye points. This reduces the search depth by detecting attacker wins earlier.

Move Generation All moves in the given region plus a pass are generated, except when *forced moves* exist. Forced moves are a safe form of pruning, which can decrease the branching factor. A *forced attacker move* prevents the defender from making two eyes immediately, for example **A** in Figure 2(a). A *forced defender move* is a point that the defender must occupy immediately. It is defined as follows:

- There is only one unsafe attacker block b which has a single-move connection to safe stones.
- If the defender plays any other move and the attacker connects b to safety, the defender is left with a dead shape.

An example of a forced defender move is **B** in Figure 2(b).

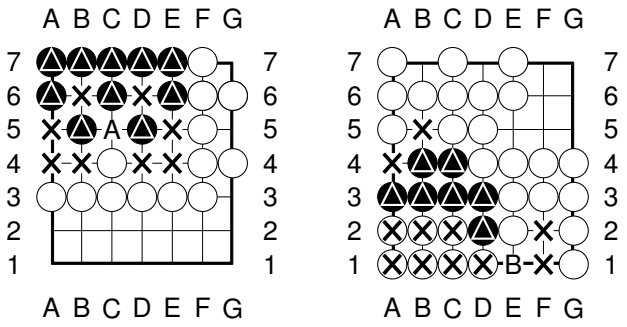

Figure 2: Forced Moves.

Simulation Kawano's *simulation* (Kawano 1996) borrows moves from the proof tree of a proven position P in order to find a quick proof of a "similar" position Q. The winning move for each OR node in the proof tree below P is tried for the analogous position below Q. If simulation is successfully applied to Q, it returns a correct proof for Q. If simulation fails, the normal df-pn search is performed. A successful simulation requires much less effort than a normal search, since even with good move ordering, a newly created search tree is typically much larger than an existing proof tree. In TSUMEGO EXPLORER, similar positions are defined as follows:

- Let n be an AND node with a proven child n_c, and let m be a winning move from n_c. Based on n_c's proof tree, apply simulation to all unsolved children of n, except for the child n_e that results from the opponent playing m from n (if such a move is legal).
- A dual procedure is used at OR nodes.

The handling of n_e is different from previous approaches. (Kishimoto & Müller 2003) treat n_e as a similar position. However, n_e does not seem to be similar since one important point on the board has been occupied by the *other* player. In GoTools, n_e is tried next if one of n's children is (dis)proven (Wolf 2000). On the other hand, TSUMEGO EXPLORER first tries to simulate all child nodes *except* for n_e. This choice is motivated by the behavior of df-pn. A successful simulation allows df-pn to immediately explore further children of n at the current threshold. This use of simulation is much more extensive than in tsume-shogi (Kawano 1996). Motivations are that a position changes more gradually in Go, and that many bad moves can be refuted in the same way.

Heuristic Initialization Df-pn initializes the proof and disproof numbers of a leaf node to 1. The standard df-pn enhancement df-pn$^+$ (Nagai & Imai 1999) uses heuristic initialization of proof and disproof numbers, as proposed for proof-number search in (Allis 1994). In TSUMEGO EXPLORER, proof or disproof numbers for the defender are initialized by an approximation of the method in (Kierulf 1990), which computes the minimum number of successive defender moves required to create two eyes. A similar heuristic for the number of moves to create a dead shape is computed to initialize (dis)proof numbers for the attacker.

Nonuniform Heuristic Threshold Increments Heuristic proof and disproof numbers are typically larger than the default value of 1. This increases the reexpansion overhead at interior nodes, since thresholds are increased only by the minimum possible amount: If n is an OR node, n_c is n's child selected by df-pn, and pn_2 the second largest proof number among n's children, then df-pn sets a threshold of $\mathbf{th_{pn}}(n_c) = \mathbf{min}(\mathbf{th_{pn}}(n), pn_2 + \delta)$ with $\delta = 1$. To reduce reexpansions, at the cost of possibly making the direction of search less precise, a larger δ is chosen, namely the average value of the heuristic initialization function of all moves. For an AND node n, the disproof threshold of n_c is set analogously. The standard df-pn threshold computation is used in the other two cases, for proof thresholds of AND nodes and disproof thresholds of OR nodes. For example, if n is an OR node, n_i are n's children and $\mathbf{dn}(n)$ is n's disproof number, $\mathbf{th_{dn}}(n_c) = \mathbf{th_{dn}}(n) - \sum \mathbf{dn}(n_i) + \mathbf{dn}(n_c)$.

Experimentally, this technique reduced the ratio of reexpanded nodes to total nodes from 45% to 33%, and achieved about a 21% node reduction for harder problems. Investigating the trade-off between the ratio of reexpansions and decreasing the total execution time remains as future work.

Treatment of Ko

Sometimes the outcome of a tsume-Go problem depends on position repetition, called ko. A move may be illegal locally, within the searched region, but become legal in the larger context of a full board game after a nonlocal *ko threat* has been played. It is therefore important to model nonlocal ko threats followed by local ko recaptures within the search. As in (Kishimoto & Müller 2003), if ko is involved in a proof or disproof in the first search phase, a re-search is performed by assuming that the loser can immediately re-capture ko as often as needed. Within a search, more complicated repetitions such as double ko and triple ko are handled correctly. The solver also includes the techniques for solving the *Graph History Interaction* problem (Kishimoto & Müller 2004a).

GoTools uses a more sophisticated approach, with re-searches in order to make a finer distinction between how many external ko threats must be played to win a ko.

Experimental Results

This section compares the performance of TSUMEGO EXPLORER against GoTools experimentally, on an Athlon XP 2800 with a time limit of 5 minutes per problem instance. TSUMEGO EXPLORER used a 300 MB transposition table. GoTools used a 2MB table.[1]

The two test suites used for the experiments were:

1. LV6.14 contains 283 positions in the hardest category from the database of 40,000 tsume-Go problems automatically generated by GoTools (Wolf 1996b). Figure 3 shows

[1]The version of GoTools used in our experiments was provided by Thomas Wolf. A different version of GoTools is used in the SmartGo program by Anders Kierulf. It is about 3.4 times faster. However, it still cannot solve most of the problems in our test suite that are unsolved by the original GoTools.

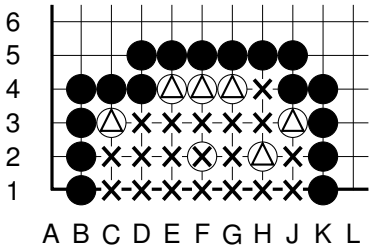

Figure 3: A position from LV6.14 (White lives with **D2**).

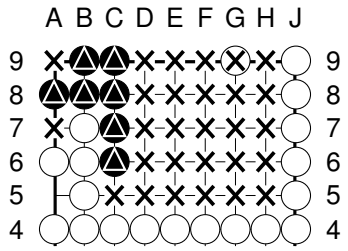

Figure 4: A hard problem from ONEEYE (Black lives with **E8**).

a typical example. All problems are solved for either color playing first, resulting in a total of 566 instances. The results shown are for the subset of 418 problems whose solution does not involve ko. For the remaining 148 problems involving ko, overall results are similar but excluded here, since GoTools spends more resources on computing a more fine-grained result type for ko.

2. ONEEYE (Kishimoto & Müller 2004b) is an extended version of the test set used by (Kishimoto & Müller 2003) containing 162 instances, of which 148 can be solved without ko. Hard problems in ONEEYE usually contain a large empty area, as in Figure 4.

Results

Tables 1 and 2 summarize the performance of the two solvers on LV6.14 and ONEEYE. Both programs solve all problems in LV6.14. TSUMEGO EXPLORER is about 2.8 times faster in total. In ONEEYE, TSUMEGO EXPLORER solves all 119 problems solved by GoTools plus 23 additional problems. TSUMEGO EXPLORER solves the 119 common problems more than 20 times faster.

Table 1: Performance comparison between TSUMEGO EXPLORER and GoTools in LV6.14.

	Problems solved	Total time (s) (418 Problems)
GoTools	418	1,235
TSUMEGO EXPLORER	418	448
Total Problems	418	-

Table 2: Performance comparison between TSUMEGO EXPLORER and GoTools in ONEEYE.

	Problems solved	Total time (s) (119 Problems)
GoTools	119	957
TSUMEGO EXPLORER	142	47
Total Problems	148	-

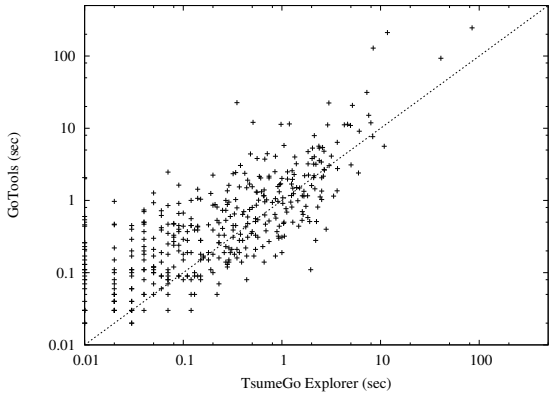

Figure 5: Comparison of solution time for individual instances in LV6.14.

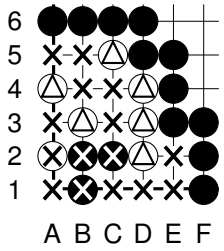

Figure 6: Knowledge wins: A position that GoTools solves faster (White to live with **D1**).

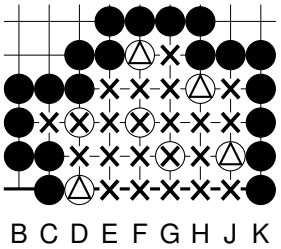

Figure 7: Search wins: A position that TSUMEGO EXPLORER solves faster (Black to kill with **E2**).

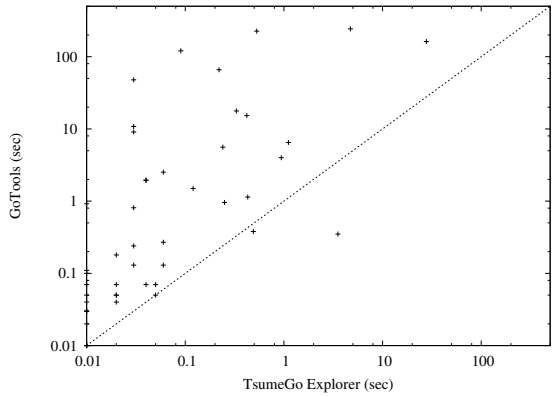

Figure 8: Solution time for problems solved by both programs in ONEEYE.

Detailed Results for LV6.14 Figure 5 compares the execution time for individual problems in a doubly logarithmic plot. 46 problems solved within 0.01 seconds by TSUMEGO EXPLORER are hardly visible on the left edge of the graph. In problem instances above the diagonal, TSUMEGO EXPLORER was faster. No program completely dominates the other. TSUMEGO EXPLORER, with its efficient search, is faster in 291 cases, GoTools, with its large amount of Go-specific knowledge, in 127 cases. For hard problems, where at least one program needs more than 5 seconds, TSUMEGO EXPLORER is faster in 25 out of 29 instances.

In Figure 6, GoTools' knowledge and move ordering work perfectly. It takes only 0.08 seconds, with 167 leaf nodes expanded to a maximum depth of 19. In contrast, TSUMEGO EXPLORER needs 0.44 seconds, with 22,773 node expansions and maximum depth 23. Figure 7 is hard for GoTools. It needed 211 seconds compared to 11.7 seconds for TSUMEGO EXPLORER.

Detailed Results for ONEEYE Figure 8 plots the execution time for ONEEYE for the subset of 119 problems solved by both programs. The superiority of TSUMEGO EXPLORER on most problems in this test suite is clearly visible. TSUMEGO EXPLORER outperforms GoTools by a large margin, and is faster in 93 out of the 119 instances solved by both. In Figure 9, GoTools needed 121 seconds against 0.14 seconds for TSUMEGO EXPLORER. However, in some cases the Go knowledge of GoTools is very valuable. The position in Figure 10 with White to play is solved by the static evaluation of GoTools, while TSUMEGO EXPLORER searches 3,159 nodes.

For the 23 problems solved only by TSUMEGO EXPLORER, the difficulty ranges from very easy to hard. As an extreme example, Figure 11 was solved in just 0.73 seconds.

Limitations of TSUMEGO EXPLORER

The current TSUMEGO EXPLORER can solve enclosed positions with around 20 empty points in a few seconds. The practical limit of our solver seems to be 22-29 empty points.

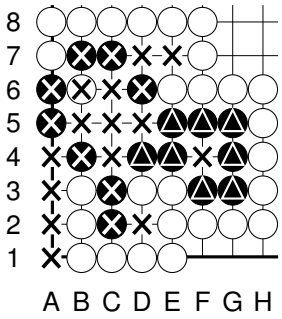

Figure 9: Search wins: White to kill with **C5**.

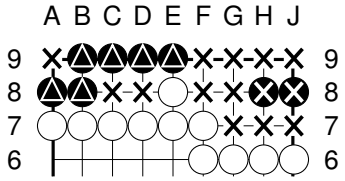

Figure 10: A position that GoTools solves statically: White to kill, for example with **F9**.

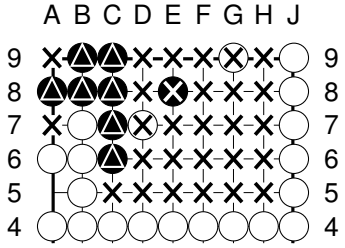

Figure 11: A position solved only by TSUMEGO EXPLORER: Black to live with **D8**.

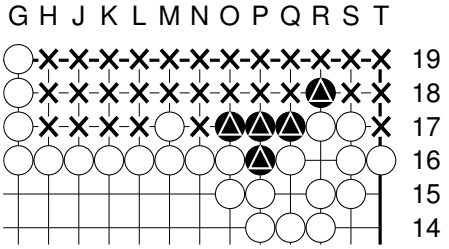

Figure 12: A hard tsume-Go problem for TSUMEGO EXPLORER (White kills with **S18**).

As a borderline case, Figure 12, with 29 empty points, was solved in 750 seconds with more than 16 million expanded nodes. These numbers compare favorably to GoTools, which scales up to about 14 empty points.

Conclusions and Future Work

In computer games research, there is an ongoing competition between the proponents of search-intensive and knowledge-intensive methods. So far, computer Go researchers have been mainly in the knowledge camp. TSUMEGO EXPLORER shows the potential of search methods in Go, at least for restricted problems such as tsume-Go.

One advantage of df-pn is that it uses the transposition table more extensively in the search. Only solved (won or lost) positions are stored in GoTools' transposition table (Wolf 2000), while df-pn utilizes proof and disproof numbers from previous search iterations to choose a promising direction for tree expansion (Nagai 2002).

Future work includes the integration of more knowledge into the solver, in order to study the trade-offs between speed and knowledge in this domain more closely, and create a solver that combines the best aspects of both GoTools and TSUMEGO EXPLORER. The next practical step will be an extension to open boundary tsume-Go problems. (Wolf 1996a) describes some difficulties of open-boundary problems. Unlike in enclosed problems, the set of moves to be considered is not well-defined, leading to heuristic pruning or threat-based approaches such as (Cazenave 2001). Finally, integration with a full playing program will be an important topic to improve the strength of computer Go programs.

Acknowledgments

We would like to thank Thomas Wolf for providing a copy of GoTools, and for valuable comments about this research. Adi Botea, Markus Enzenberger, Xiaozhen Niu, Jonathan Schaeffer, and Ling Zhao read drafts of the paper and gave beneficial feedback. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Alberta Informatics Circle of Research Excellence (iCORE).

References

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

Allis, L. V. 1994. *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Dissertation, Department of Computer Science, University of Limburg.

Benson, D. B. 1976. Life in the game of Go. *Information Sciences* 10:17–29.

Cazenave, T. 2001. Abstract proof search. In Marsland, T. A., and Frank, I., eds., *Computers and Games (CG 2000)*, volume 2063 of *Lecture Notes in Computer Science*, 39–54. Springer.

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

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

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

Kierulf, A. 1990. *Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies*. Ph.D. Dissertation, Swiss Federal Institute of Technology Zürich.

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

Kishimoto, A., and Müller, M. 2004a. A general solution to the graph history interaction problem. In *19th National Conference on Artificial Intelligence (AAAI'04)*, 644–649. AAAI Press.

Kishimoto, A., and Müller, M. 2004b. One-eye problems. http://www.cs.ualberta.ca/~games/go/oneeye/ .

Kishimoto, A. 2005. *Correct and Efficient Search Algorithms in the Presence of Repetitions*. Ph.D. Dissertation, Department of Computing Science, University of Alberta.

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

Nagai, A., and Imai, H. 1999. Application of df-pn$^+$ to Othello endgames. In *Game Programming Workshop in Japan '99*, 16–23.

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

Vilà, R., and Cazenave, T. 2003. When one eye is sufficient: A static approach classification. In *Advances in Computer Games. Many Games, Many Challenges*, 109–124. Kluwer Academic Publishers.

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

Wolf, T. 1996a. About problems in generalizing a tsumego program to open positions. In Matsubara, H., ed., *Game Programming Workshop in Japan '96*, 20–26.

Wolf, T. 1996b. Gotools: 40,000 problems database. http://www.qmw.ac.uk/~ugah006/gotools/t.wolf.gotools.problems.html .

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