Lambda Depth-first Proof Number Search and its Application to Go

Kazuki Yoshizoe
Dept. of Electrical, Electronic,
and Communication Engineering,
Chuo University, Japan
yoshizoe@is.s.u-tokyo.ac.jp

Akihiro Kishimoto
Department of Media Architecture,
Future University-Hakodate, Japan
kishi@fun.ac.jp

Martin Müller
Dept. of Computing Science,
University of Alberta, Canada
mmueller@cs.ualberta.ca

Abstract

Thomsen's λ search and Nagai's depth-first proof-number (DFPN) search are two powerful but very different AND/OR tree search algorithms. Lambda Depth-First Proof Number search (LDFPN) is a novel algorithm that combines ideas from both algorithms. λ search can dramatically reduce a search space by finding different levels of threat sequences. DFPN employs the notion of proof and disproof numbers to expand nodes expected to be easiest to prove or disprove. The method was shown to be effective for many games. Integrating λ order with proof and disproof numbers enables LDFPN to select moves more effectively, while preserving the efficiency of DFPN. LDFPN has been implemented for capturing problems in Go and is shown to be more efficient than DFPN and more robust than an algorithm based on classical λ search.

1 Introduction

Many search methods have been developed to solve complex AND/OR trees, as occur in games. The direction of search can be controlled in an ad hoc manner by domain-specific knowledge. Domain-independent best-first, nonuniform tree expansion techniques are a more principled approach. Research in this area has produced two families of algorithms: in the first family of algorithms based on proof and disproof numbers [Allis, 1994; Nagai, 2002; Kishimoto and Müller, 2005], search is controlled through the combined estimated difficulty of solving sets of frontier nodes that must be (dis-)proven in order to (dis-)prove the root. The second family of algorithms utilizes null (pass) moves [Donninger, 1993] for finding threats to achieve a goal [Thomsen, 2000; Cazenave, 2002]. Search proceeds in a statically determined fashion from simpler to more complex threats. Both families of algorithms can be viewed as simplest-first search paradigms, for different definitions of what constitutes a simple search. However, neither notion of simplicity matches the ideal one, which is to find a (dis-)proof as quickly as possible. For example, in the case of threat-based algorithms, a proof containing long series of simple threats is inferior to one with a small number of more complex threats. In the case of proof-number based algorithms, overestimation of proof and disproof numbers delays the search of nodes that look unpromising but can be easily solved.

This paper studies a combined approach, which utilizes proof numbers to control the effort put into search based on different threats. In this manner, the most promising threat types to use in different parts of a search tree can be selected at runtime. The contributions include:

- The lambda depth-first proof-number (LDFPN) search algorithm synthesizing depth-first proof-number search [Nagai, 2002] and λ search [Thomsen, 2000].
- Experimental results for capturing problems in the game of Go demonstrating that LDFPN outperforms DFPN with state of the art enhancements and is more robust than classical λ search.

The rest of this paper is organized as follows: Section 2 summarizes the rules of Go. Section 3 describes related work. Section 4 introduces the LDFPN algorithm. Section 5 discusses experimental results, and Section 6 concludes and outlines further research directions.

2 The Game of Go

The game of Go is a two person zero sum perfect information game. It originated in China and is most popular in East Asia. Players take turns to place stones on the intersections of a grid. The first player uses black stones and the second player uses white stones. A stone placed on the board stays in its position unless the opponent captures and removes it. The aim of the game is to surround more territory than the opponent. An example of a terminal position in Go on a 9×9 board is shown in the left of Figure 1.

One of the few rules of Go is about capturing stones. A set of directly connected stones of the same color is called a *block*. Stones connect vertically and horizontally, not diagonally. Empty intersections directly adjacent to a block are called *liberties*. A player can capture an opponent block by playing on its last liberty.

A *capturing problem* in Go is the following decision problem: Given a block b in a Go position p and a player to play first, can b be captured assuming best play by both? The player owning b is called the *defender*, the opponent is the *attacker*.

A block with only one liberty is said to be in *atari*. A *ladder* is a special capturing problem where the attacker uses a sequence of *ataris*. A successful *ladder* is shown in the right of Figure 1. For more information about Go, please refer to `http://www.usgo.org/`

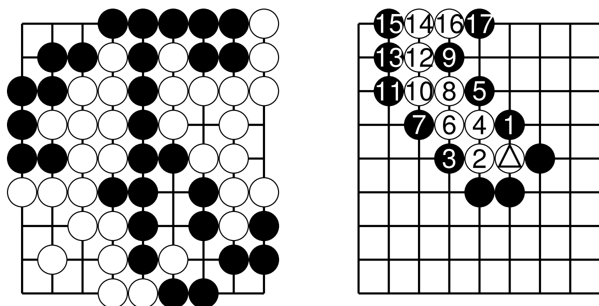


Figure 1: Example of a terminal position in Go, and a *ladder*

3 Related Work

3.1 Threats, Threat-based Search and λ Search

Threats can be used to direct and focusing search. A position with many threats is usually good for the attacker, while an absence of threats indicates that no quick success can be expected. If a player has a threat to win, it is worth investigating it with high priority. Threats severely restrict the opponent's choice of replies to moves that can avert the threat, and often lead to further follow-up threats.

Threats can be found by using null moves. Intuitively, a threat is a move that, if ignored, leads to a quick win. In the ladder example in Figure 1, each black move is a threat to capture, and white has only one possible move to avert that threat.

The threat-based approach of [Allis *et al.*, 1996] was generalized to λ search [Thomsen, 2000] and further by Cazenave [2002]. While this paper concentrates on a comparison with λ search, its ideas should apply to threat-based methods in general.

The basic concept of λ search is the λ order, a measure of how fast a player can achieve a goal, or in other words, the directness of a threat. λ search creates λ^n*-trees* consisting of λ^n*-moves*, which are recursively defined as follows: A successful λ^0*-tree* for the attacker, denoted by $\lambda^0 = 1$, consists of a single attacker move that achieves the goal directly. A λ^0*-move* is such a winning attacker move.

An attacker win by an order n threat is denoted by $\lambda^n = 1$, while $\lambda^n = 0$ indicates that the attacker cannot achieve the goal by order n threats. A λ^n-tree is a search tree which consists of λ^n-moves. If there is no λ^n-move for a node, that node is terminal and a loss for the player to move. The definition of λ^n-moves is as follows.

Definition 1 *[Thomsen, 2000] A λ_a^n-move is an attacker move such that if the defender passes in reply, there exists a λ^i-tree with $\lambda^i = 1$ $(0 \leq i \leq n-1)$. The attacker threatens to win within a λ order of less than n.*

Definition 2 *[Thomsen, 2000] A λ_d^n-move is a defender move such that after the move, there is no subsequent λ^i-tree with $\lambda^i = 1$ $(0 \leq i \leq n-1)$. The defender move averts all lower order attacker threats.*

For example, in the capturing problem, a λ^0-move occupies the last liberty of the target block and captures it. Attacker's λ_a^1 moves are moves for capturing the target stones in a ladder or directly, while defender's λ_d^1 moves are moves preventing a direct capture. Attacker's λ_a^2 moves threaten to capture the target in a ladder or directly.

The underlying concept of λ search is that λ^n moves are defined by λ^{n-1} searches, and the size of a λ^{n-1} tree is expected to be far smaller than a λ^n tree. Therefore λ search can prune moves according to the λ order with little effort.

λ search is especially efficient for problems for which λ order is meaningful. Capturing problems in Go are an ideal case, because lower bounds on the λ order can be derived from game specific knowledge, the number of liberties of a block [Thomsen, 2000].

Two simple dominance relations hold for the value of λ trees.

$$\text{if} \quad \lambda^n = 1 \quad \text{then} \quad \lambda^i = 1 \quad (n \leq i) \tag{1}$$

$$\text{if} \quad \lambda^n = 0 \quad \text{then} \quad \lambda^j = 0 \quad (0 \leq j \leq n) \tag{2}$$

A positive result of a λ search, $\lambda^n = 1$, is correct provided that pass is allowed or zugzwang is not a motive in a game. A negative search result, $\lambda^n = 0$, means that either there is no solution, or a solution will be found at a higher order $n' > n$.

3.2 Proof-Number Search Variants

In an AND/OR tree, let a *proof* be a win for the first player (corresponding to an OR node) and a *disproof* a win for the opponent (represented by an AND node). [Allis *et al.*, 1994] introduced proof and disproof numbers as an estimate of the difficulty to find proofs and disproofs in a partially expanded AND/OR tree. The proof number of node n, $pn(n)$, is defined as the minimum number of leaf nodes that must be proven in order to find a proof for n, while the disproof number $dn(n)$ is the minimum number of leaf nodes to disprove for a disproof for n. $pn(n) = 0$ and $dn(n) = \infty$ for a proven terminal node n, and $pn(n) = \infty$ and $dn(n) = 0$ for a disproven terminal node. $pn(n) = dn(n) = 1$ is assigned to any unproven leaf. Let $n_1, \cdots, n_k$ be children of interior node n. Proof and disproof numbers of an OR node n are:

$$pn(n) = \min_{i=1,\cdots,k} pn(n_i), \qquad dn(n) = \sum_{i=1}^{k} dn(n_i).$$

For an AND node n proof and disproof numbers are:

$$pn(n) = \sum_{i=1}^{k} pn(n_i), \qquad dn(n) = \min_{i=1,\cdots,k} dn(n_i).$$

Figure 2 shows the calculation.

Proof-number search (PNS) is a best-first search algorithm that maintains proof and disproof numbers for each node. PNS finds a leaf node from the root by selecting a child with smallest proof number at each OR node and one with smallest

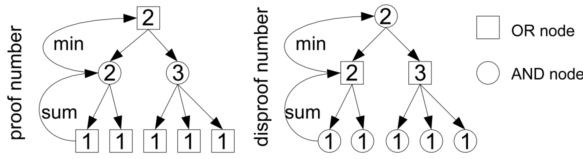


Figure 2: Calculation of proof/disproof numbers

disproof number at each AND node. It then expands that leaf and updates all affected proof and disproof numbers along the path back to the root. This process continues until it finds either a proof or disproof for the root.

Depth-first proof-number (DFPN) search [Nagai, 2002] is a depth-first reformulation of PNS which re-expands fewer interior nodes and can run in space limited by the size of the transposition table. Thresholds for proof and disproof numbers are gradually incremented and used to limit a depth-first search, similar to Recursive Best-First Search [Korf, 1993].

DFPN has been a very successful search method, and is used in the best tsume-shogi solver [Nagai, 2002], the best tsume-Go solver [Kishimoto and Müller, 2005], and a back-end prover for solving checkers [Schaeffer *et al.*, 2005]. Yoshizoe [2005] presented a DFPN-based solver for the capturing problem in Go to find an *inversion*, a set of points on which a move possibly changes the outcome of a search. The dual lambda search algorithm [Soeda *et al.*, 2005] tracks threats by both players in the mutual king attack typical of shogi endgames. [Soeda, 2006] develops a family of algorithms closely related to our work and applies them to the Japanese game of shogi.

4 The LDFPN Search Algorithm

4.1 Details of LDFPN

LDFPN is a proof number based λ search. Proof/disproof numbers for AND/OR nodes are initialized and propagated as in DFPN. However, each attacker node is split into several pseudo nodes corresponding to different λ orders $0, \cdots, l$, and search dynamically selects the most promising λ order to pursue at each node.

An attacker node n is shown at the top of Figure 3. It is divided into pseudo nodes of different λ orders up to a limit of $l = 3$ in the example. Each pseudo node corresponds to a subtree with different λ order. Let n be part of a λ^l tree, and $c_0, \cdots, c_l$ the pseudo nodes of n. The attacker wants to prove that $\lambda^l = 1$. By the dominance relation of equation (1), the attacker need only prove any one of the pseudo nodes. However, disproving any lower order λ tree is no help for disproving the λ^l tree as in equation (2). Therefore, the disproof number of attacker's node n is the same as for c_l.

Formally the proof and disproof numbers of attacker's node n are calculated as follows:

$$pn(n) = \min_{i=0,\cdots,l} pn(c_i), \qquad dn(n) = dn(c_l).$$

A defender node n is shown in the bottom of Figure 3. Defender's aim is to disprove this node by showing $\lambda^l = 0$. By equation (2), proofs of lower order λ subtrees are irrelevant

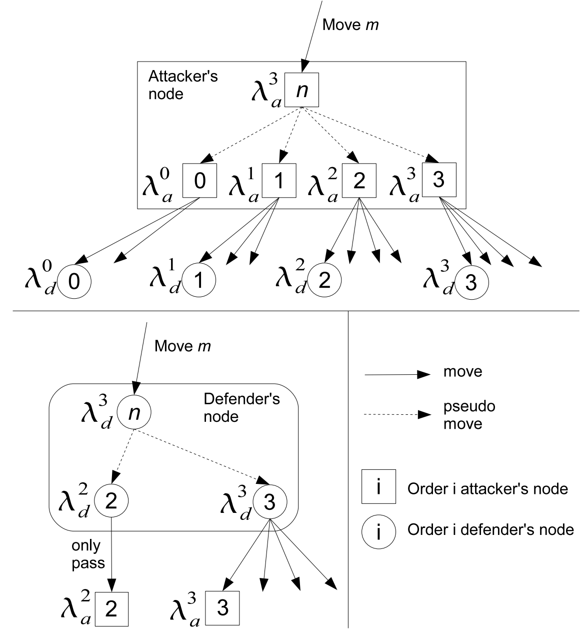


Figure 3: Attacker's node and Defender's node

for the proof of n. The proof of the highest λ order subtree is the only valid proof. However, a λ^{l-1} subtree is searched for the exceptional case of a pass move. In the bottom of this Figure, if the attacker cannot win by λ order 2 after a defender pass, then the attacker was not threatening to win by λ order 2 and according to the definition cannot win by λ order 3. From the attacker's point of view, the purpose for searching this pseudo node c_{l-1} is to check if the attacker's previous move m, which lead to node n, was a λ_a^l move or not. If $\lambda^{l-1} = 0$ is proven, the search can immediately return to the parent of n. Since a λ^{l-1} subtree is typically smaller than a λ^l subtree, the search for c_{l-1} will often finish quickly.

The defender, looking for a disproof at AND node n, can choose between the most promising move in the λ^l tree and a pass move that searches a λ^{l-1}-tree. Therefore, the proof and disproof numbers of a defender node n are calculated by:

$$pn(n) = pn(c_l), \qquad dn(n) = \min(dn(c_{l-1}), dn(c_l)).$$

Like DFPN, LDFPN uses two thresholds for proof and disproof numbers. LDFPN selects the leaf node with the smallest proof/disproof number, regardless of the λ order.

The main advantage of LDFPN over classical λ search is the ability of LDFPN to seamlessly switch between different λ order child nodes. There is no need to wait for a proof or disproof of all lower λ order subtrees. Since disproofs are often more difficult than proofs, this ability to skip a difficult disproof of a λ^{n-1} subtree and start search of a λ^n subtree improves the search behavior.

4.2 Search Enhancements

Heuristic Initialization of Proof and Disproof Numbers

The basic DFPN algorithm initializes proof and disproof numbers of an unproven leaf node to 1. As in [Allis, 1994;

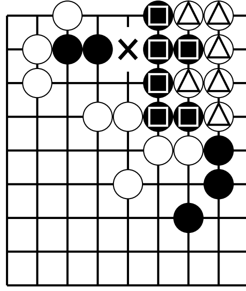


Figure 4: Sample test problem (Black to play)

Nagai, 2002], one way to enhance performance of LDFPN is to heuristically initialize these numbers. Let $H_{pn}(n)$ and $H_{dn}(n)$ be evaluation functions to initialize the proof and disproof number of leaf n. In LDFPN, $H_{pn}(n)$ and $H_{dn}(n)$ are defined as functions of the λ order of n, to reflect the property that a node with lower λ order should be searched with a higher priority (see Section 5.1 for details). DFPN with this enhancement is called DFPN+ in [Nagai, 2002].

Ladder Search

A λ^1 search for capturing problems of Go is the same as a *ladder* search, shown in Figure 1. This search is exceptionally easy, since the number of move candidates is very small (1 or 2 in most cases). In the implementation reported here, a special purpose *ladder* search is 10 times faster in terms of nodes per second than LDFPN. It can be used as a subroutine by LDFPN.

Simulation

Kawano's *simulation* [Kawano, 1996] saves search time by reusing the proof tree of similar board positions. If a board position A was (dis-)proven, a similar board position B is likely to be (dis-)proven by the same (dis-)proof tree.

LDFPN uses *simulation* only for pass moves at AND nodes. A pass move is searched at AND nodes, and if it results in a loss, its sibling nodes are checked using the disproof tree of the pass move. In this way, irrelevant moves which do not help the player can be disproven quickly.

5 Experimental Results

5.1 Setup of Experiments

The test suite consists of 217 capturing problems, including 110 from [Mas, 2002] and 107 modified ones. While the original problems all have a winning solution, the modified problems test the case where the first player loses. A typical example is shown in Figure 4. The attacker Black must capture the crucial white stones marked by triangles to save the black stones marked by squares. The correct answer is marked with a cross.

Each problem was searched in two ways: First to capture the stones marked with triangles, and second to defend the stones marked with squares. Some problems contained 2 *blocks* to be defended, and in some others, there were 2 *blocks* to be captured. Counting these as separate problems yields a total of 440 test problems.

The following three algorithms were tested:

- LDFPN: The LDFPN algorithm described in Section 4 with the highest λ order of 5. Several methods for initializing proof and disproof numbers were tested to tune H_{pn} and H_{dn}. As a result, H_{pn} and H_{dn} for a node n with λ order λ are defined as follows:

$$H_{pn}(n) = H_{dn}(n) = \max(1, 2^{\lambda-1}) \quad (0 \leq \lambda \leq 5).$$

- DFPN+: The DFPN algorithm with heuristic initialization. H_{pn} and H_{dn} use Go-specific knowledge, the distance to a target block. To allow a fair comparison, state of the art enhancements are incorporated into the implementation of the DFPN+ algorithm, such as techniques from [Kishimoto and Müller, 2003; 2005] and Kawano's simulation [Kawano, 1996].
- Pseudo λ search: Thomsen's original λ search uses an alpha-beta framework. For better comparison with LDFPN, *pseudo λ search* uses LDFPN with special parameter settings that mimic the search strategy of λ search, which expands trees strictly in increasing order of λ. A similar effect is achieved in LDFPN by letting the heuristic initialization grow very quickly with λ:

$$H_{pn}(n) = H_{dn}(n) = \max(1, 256^{\lambda-1}) \quad (0 \leq \lambda \leq 5).$$

Since the capturing problems tested are in open-ended areas, it is hard to restrict move generation and achieve provably correct results. In practice, most computer Go programs heuristically limit moves for their capture search engines and regard a block with a large enough number of liberties as escaped. Three types of move generators were tested: not only to generate all legal moves and also to accurately assess the life and death status.

- Heuristic generator: generates moves on the liberties of *surround blocks* [Thomsen, 2000], and of blocks near the target block. It also generates some more moves including the 2nd liberties of the target block.
- Heuristic+ generator: generates moves by the heuristic generator plus on all their adjacent points.
- Full board generator: All legal moves are generated.

For DFPN+, a liberty threshold to regard the block as escaped is passed as a parameter. LDFPN uses the failure of a search with the preset maximum λ order 5 to determine whether a target block has escaped. For DFPN+, the number of liberties of the target block was used as the threshold. LDFPN with maximum λ order of l evaluates a block with $l+2$ or more liberties as escaped. A liberty threshold of $l+2$ for DFPN+ is roughly comparable to a LDFPN search with maximum order l.

The parameters given to LDFPN are the target *blocks* to capture/defend, and the λ order.

A test case was considered solved, if the move returned was the solution given in [Mas, 2002].

Experiments were performed on an Opteron 870 at 2.0 GHz with a node limit of 1 million nodes per test position and a 200 MB transposition table.

—	LDFPN	DFPN+	pseudo λ
Heuristic: num solved	294	272	288
Heuristic+: num solved	283	267	282
Full board: num solved	207	149	201

Table 1: Number of problems solved by each move generator.

5.2 LDFPN versus DFPN+

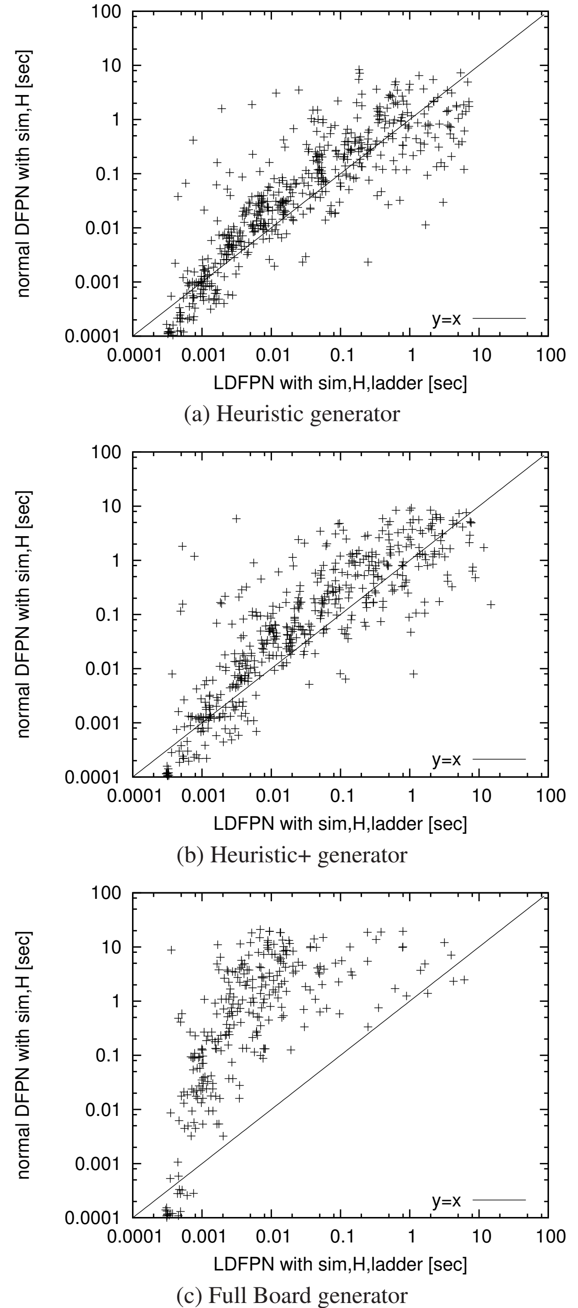
Table 1 summarizes the number of problems solved by LDFPN and DFPN+ with the three move generators. LDFPN solved more problems than DFPN+ in each case. The difference is largest for the full board move generator. This is not surprising, because threats based on the λ orders can drive LDFPN to focus on searching moves near the target block of the defender.

Figure 5 compares the performance of LDFPN and DFPN+ for each problem solved by both algorithms with the three move generators. In this Figure, the execution time of LDFPN was plotted on the horizontal axis against DFPN+ on the vertical axis on logarithmic scales. A point above $y = x$ indicates that LDFPN performed better. LDFPN outperforms DFPN+ for all three generators, with the largest difference for the full board move generator, showing the clear advantage of LDFPN on pruning irrelevant moves.

The example shown on the left in Figure 6 is solved faster by LDFPN. With the "heuristic" move generator, LDFPN searched 895 nodes to solve this problem in 0.007 seconds, while DFPN+ searched 3,143 nodes in 0.022 seconds. In this problem, black has to sacrifice 2 stones to win. If stones are captured, there are more empty points in a position (i.e. the points where the stones used to be placed), resulting in an increase in the number of legal moves. This increases the proof number of a node in DFPN+, which should have been easily proven. DFPN+ delays searching such a node with an apparently large proof number. LDFPN tends to search with a smaller set of moves based on the λ order, and this effect occurs less frequently and is less severe than with DFPN+.

One disadvantage of LDFPN is that it occasionally has to visit the same nodes in different λ orders. If the additional information of λ order is not effective, the advantage of LDFPN disappears. In particular, if the number of possible moves is small, DFPN+ works well.

The right position in Figure 6 shows an example that DFPN+ solved more quickly than LDFPN. The answer is marked with a cross in this Figure. With the "heuristic" move generator, LDFPN searched 63,201 nodes to solve the problem in 0.44 seconds, and DFPN+ searched 10,871 nodes in 0.08 seconds. This is a typical problem, in which λ order has a negative effect. Both players have several suicidal moves with low λ orders. LDFPN therefore tends to search to disprove all suicidal moves before trying to prove the correct move. In DFPN+, the suicidal moves result in captures of non-target *blocks*, leading to larger proof numbers for these moves. DFPN+ therefore delays searching these suicidal moves and expands the correct move earlier.



(a) Heuristic generator

(b) Heuristic+ generator

(c) Full Board generator

Figure 5: LDFPN with ladder vs DFPN+

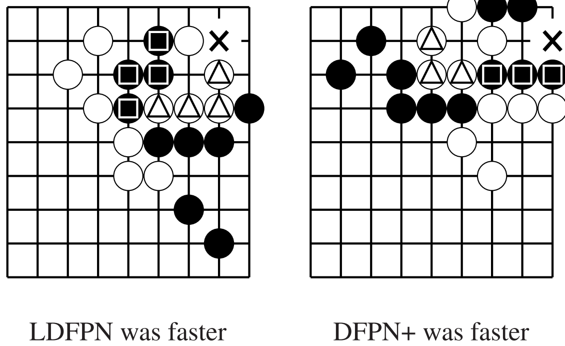


Figure 6: Problems which LDFPN solved faster/slower. (Black to play)

5.3 LDFPN versus Pseudo λ Search

Table 1 also compares the number of problems solved by LDFPN and pseudo λ search. For all types of move generators, LDFPN solved more problems than pseudo λ search.

This comparison shows that LDFPN is slightly more robust than pseudo λ search.

6 Conclusions and Future Work

This paper investigated an approach to combine proof number based search and threat based search. Results on applying LDFPN to the capturing problem are very promising. In particular, if the move generator generates a larger set of moves, LDFPN outperforms DFPN+ by a large margin with more robustness than the classical λ-search based approach. Since using a larger set of moves can improve the accuracy of solving the capturing problem, LDFPN can be a good choice for solving tactical problems in Go.

There are many topics to pursue for future work. First of all, the method should be explored for other domains, such as Hex. H_{pn} and H_{dn} of LDFPN can clearly include domain dependent knowledge to enhance performance as in DFPN+. The right balance of domain-dependent knowledge and λ order in H_{pn} and H_{dn} must be investigated. Additionally, since LDFPN can compute the λ-order of a goal, it can be applied to domains with more than one goal, in order to measure which of several goals can be achieved the fastest. Work is in progress on a semeai solver in Go, because this problem often involves multiple goals. Finally, integrating LDFPN with a complete Go-playing program will be an important topic to improve the strength of the programs.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

References

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

[Allis *et al.*, 1996] L. V. Allis, H. J. van den Herik, and M. P. H. Huntjens. Go-moku solved by new search techniques. *Computational Intelligence*, 12:7–23, 1996.

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

[Cazenave, 2002] T. Cazenave. A generalized threats search algorithm. In *Computers and Games 2002*, Edmonton, Canada, 2002.

[Donninger, 1993] C. Donninger. Null move and deep search: Selective search heuristics for obtuse chess programs. *ICCA Journal*, 16(3):137–143, 1993.

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

[Kishimoto and Müller, 2003] A. Kishimoto and M. Müller. Df-pn in Go: An application to the one-eye problem. In *Advances in Computer Games 10*, pages 125–141. Kluwer Academic Publishers, 2003.

[Kishimoto and Müller, 2005] A. Kishimoto and M. Müller. Search versus knowledge for solving life and death problems in Go. In *Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 1374–1379. AAAI Press, 2005.

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

[Mas, 2002] *Master of Semeai (Semeai no Tatsujin in Japanese)*. Japanese Go Association, 2002. ISBN: 4818204722.

[Nagai, 2002] A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, Dept. of Information Science, University of Tokyo, Tokyo, 2002.

[Schaeffer *et al.*, 2005] J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Solving checkers. In *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 292–297, 2005.

[Soeda *et al.*, 2005] S. Soeda, T. Kaneko, and T. Tanaka. Dual lambda search and its application to shogi endgames. To appear in Proceedings of Advances in Computer Games 11 (ACG11), Taipei, 2005.

[Soeda, 2006] Shunsuke Soeda. *Game Tree Search Algorithms based on Threats*. PhD thesis, The University of Tokyo, September 2006.

[Thomsen, 2000] T. Thomsen. Lambda-search in game trees - with application to Go. *ICGA Journal*, 23(4):203–217, 2000.

[Yoshizoe, 2005] K. Yoshizoe. A search algorithm for finding multi purpose moves in sub problems of Go. In *Game Programming Workshop 2005 (GPW05)*, pages 76–83, 2005.