AND-OR Tree Search Algorithms for Domains with Uniform Branching Factors

分岐因子が一様な探索空間のための AND-OR 木探索アルゴリズム

by

Kazuki Yoshizoe

美添 一樹

A Doctor Thesis

博士論文

Submitted to
the Graduate School of the University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Abstract

An AND/OR tree consists of terminal nodes that have one of the logical values true or false, and inner nodes of two types: AND nodes and OR nodes. The goal of AND/OR tree search is to find the logical value of the root node by searching. Many problems can be represented as AND/OR trees. For such problems, algorithms that use proof numbers and disprove numbers have been especially successful. Depth-first proof number search (df-pn) is currently used as the standard algorithm for many problems such as checkmate solver for shogi (Japanese chess), endgame solver for checkers, and tsumego (life and death problem in Go) solvers. Today it is widely regarded that computers are superior to human beings in solving such complex AND/OR tree search problems that often appear in games.

Recently, mainly three groups of search algorithms are studied in the game AI field. These are Alpha–beta search, Monte Carlo Tree Search (MCTS), and AND/OR tree search. Our first goal is to classify target problems and clarify which algorithm is suitable for problems of which type. We drew a map showing the areas covered by these three algorithms. We discussed what characteristics of problems serve to separate areas.

We showed that groups of problems exist that are difficult for computers to solve, and also showed that common characteristics of these problems are large and roughly uniform branching factors. The area contains problems in which human beings have an advantage over computers. The most popular example is open border tsumego.

Existing best AND/OR tree search algorithms are based on the idea of Proof Number Search. This approach was successful for the solution of many problems. However, if the target search space has a large and uniform branching factors, then Proof Number Search presents a smaller advantage. Proof Number Search is slow for domains in which an assumption of “smaller branching factor is more promising” does not hold.

The nature of Proof Number Search is that it expands the node, which is expected to be the easiest to prove. This behavior is promising for many problems, but it is more difficult to be enhanced using additional knowledge, because they stubbornly expand the node with the smallest proof number. Therefore, a new approach is needed for search spaces with uniform branching factors, or more precisely, domains in which branching factors have no relation with the goodness of the search directions.

Two domain-independent approaches are proposed for search spaces with large and uniform branching factors. The first approach is to extract hidden hierarchy from seemingly plain search spaces. Existing approaches rely on static evaluation functions to control search directions in target search spaces. Our approach is to use the idea of λ search in the framework of df-pn. The search space will be separated into different hierarchies according to the λ order which can be explained as a distance from the win. Based on this approach, we developed λ df-pn algorithm. Results demonstrated that λ df-pn greatly improves the speed performance of the existing best algorithm df-pn in search space with roughly uniform branching factors, and also showed that λ df-pn is several times faster than df-pn, even in search spaces with non-uniform branching factors.

The second approach is to discard unpromising child nodes automatically during a search. Proof number search algorithms often encounter problems caused by the existence of overly numerous unpromising child nodes. The most unpromising nodes give no effective information to control the search direction. It would be effective if it were possible to discard only unpromising moves. Based on this approach, we developed dynamic widening, and showed that df-pn combined with dynamic widening is approximately 25 times faster for search space with uniform branching factors.

Open border problems of the game of Go were chosen as the targets of our experiments, not only because Go is a well known game, but also because local problems of Go are popular examples of search space with uniform branching factors. Moreover, it is an extremely difficult AND/OR tree search problem.

We combined dynamic widening with λ df-pn and measured the performance for solving Go problems. We also investigated the limits of our approach and clarified the conditions needed for our algorithm to achieve high performance. The results were satisfactory considering that no domain-dependent knowledge was used in our algorithm. Results show that our algorithms are promising for domains with uniform branching factors, which are difficult problems for existing algorithms.

In conclusion, we developed two algorithms on two approaches that use unused ability of proof number search. The resulting algorithms are suitable for the last remaining group of AND/OR tree search problems in the game AI field for which human beings are comparable or superior to computers.

論文要旨

AND/OR 木とは，末端の節点が true 又は false の 2 値を持ち，木の内部節点が AND 節点か OR 節点で構成されるような木のことである．AND/OR 木探索の目的は，根節点の論理値を探索によって求めることである．AND/OR 木で表現できる問題は多数存在する．そのような問題に対して，証明数/反証数を用いた探索アルゴリズムが非常に有効であることが知られている．特に df-pn(深さ優先証明数探索) は詰将棋，チェッカーの終盤，詰碁などに適用され，大きな成功を収めている．このように，ゲーム中にしばしば現れるような複雑な AND/OR 木を探索する能力において，今日のコンピュータは人間を上回っていると広く信じられている．

ゲーム AI の分野では近年, 主に alpha-beta 探索, モンテカルロ木探索 (MCTS), AND/OR 木探索の三種類の探索アルゴリズムが研究されている．我々の一つの目標は，探索問題を分類し，どの問題にどのアルゴリズムが適しているのかを明確にすることである．それぞれのアルゴリズムに適した問題を分類した図を描き，どのような特徴によって問題が分類されるのか検討を行った．

我々は，人間には解けるがコンピュータに取っては難しい一群の種類の問題がまだ存在することを示し，それらの問題に共通する特徴は，分岐因子が大きく，かつほぼ一様であることを示した. そのような問題では人間がまだコンピュータに対してほぼ互角であるかあるいは優位を保っている．代表的な例は，領域が開いており脱出の可能性のある詰碁である．

AND/OR 木探索のアルゴリズムでは，証明数探索が既存の最高のアルゴリズムであると考えられている．証明数探索には，探索対象の知識が少ない状態でも性能を発揮するという alpha-beta 探索には無い利点があり，実際に多くの問題で性能を発揮している．しかし探索対象が一様かつ大きな分岐因子を持つ場合を証明数探索は苦手としている．「分岐因子が小さい選択肢ほど見込みがある」という仮定が成り立たない問題に対しては証明数を用いたアルゴリズムは遅くなる．

証明数探索には，一番証明が簡単そうな節点から展開するという性質がある．これは多くの問題において有効な性質であるが，証明数が小さい方から探索するという性質はアルゴリズム自体が持つ性質であるため，その性質が有効で無い場合にヒューリスティックを加えて調整を行うことが困難である．そのために，分岐因子が一様な探索空間，より正確には分岐因子が探索の手がかりにならないような探索空間に対しては新たな技術を開発する必要がある．

我々は二つのアプローチに基づいて，分岐因子が大きくかつ一様な探索空間に適した探索アルゴリズムを提案した．一つのアプローチは一見すると平坦な探索空間から階層構造を抽出するものである．既存のアプローチは静的な評価関数を用いて探索の方向を制御しているが，我々のアプローチは実際に探索を行うことによってのみ判明する階層構造を利用する物である．我々は階層構造として λ 探索を用い，これを df-pn の枠組みに組み込むことによって λ df-pn アルゴリズムを開発した．これは，探索空間を λ order という概念に基づいて分類しながら探索を行うアルゴリズムである．λ order はゴール達成までのあ

る種の距離と言える概念である．我々は，分岐因子が大きくかつ一様な探索空間において既存の最高のアルゴリズムである df-pn と比較して λ df-pn が大幅に速度性能を向上させることを示した．また，分岐因子が一様でない探索空間においても λ df-pn は df-pn よりも数倍高速であった．

二番目のアプローチは，有望でない枝を探索中に自動的に無視する方法である．有望でない枝が大量に存在する木を探索する場合，証明数探索の探索順序が悪影響を受けることはしばしば観察される．もし有望でない枝を無視することが可能ならは性能向上を得ることができると期待される．我々は dynamic widening という簡単な手法によってそれを実現することが可能であると示した．実験により，df-pn と dynamic widening を組み合わせることによって分岐因子が一様かつ大きい探索空間に対しては df-pn の速度性能を約 25 倍高速化させることが可能であることを示した．

実験の対象としては，領域の開いた状態の囲碁の問題を用いた．囲碁を選んだ理由は，これが広く知られている問題であり，また分岐因子が一様かつ大きな問題の中でも既存のアルゴリズムにとって最も難しく，AND/OR 木探索の問題の中で最も難しい問題の一つであるからである．

我々は dynamic widening と λ df-pn を組み合わせて囲碁の問題を解き，性能を測定した．さらに我々のアプローチが性能を発揮するための条件も示した．探索対象に関するヒューリスティックな知識を全く用いない実験であったことを考えると，全体的な性能は満足すべき物であった．結果として，我々のアプローチは既存のアプローチでは難しい問題であった分岐因子が一様かつ大きい問題に対して有効なアプローチであると期待される．

我々は二種類のアプローチに基づいて，証明数探索の能力を利用した二つのアルゴリズムを開発した．結果として得られたアルゴリズムは，特にゲーム AI 分野の AND/OR 木探索問題の中で最後に残された人間がコンピュータに対抗できる一群の問題に対して適したアルゴリズムである．

Acknowledgements

In 2005, I left my work at Fujitsu Laboratories Inc. and started to work on game AI topics. Since that time, it has taken four years to earn my PhD. Without support from many people, I could not have finished this work. I would like to offer a few words of appreciation here.

First, I want to thank Professor Hiroshi Imai of the University of Tokyo. Despite the fact that I spent only one year and a half as a research student at his lab, and although he was not officially my supervisor, he has been my mentor since I left Fujitsu Laboratories. He was not only an excellent advisor in research topics, but also a splendid guide for my whole research career. He and the members of the laboratory spent much time reviewing my slides for the defense of my dissertation. I want to say thanks to the members of Imai lab, especially to Jun Hasegawa and Sonoko Moriyama, for giving me many useful comments.

In Japan, in only a few places can researchers of game AI have interesting discussions. The Game Programming Seminar provided me such a good place. I want to thank the members of the seminar, especially Tomoyuki Kaneko, Shunsuke Soeda and Prof. Tetsuro Tanaka. Shunsuke Soeda was working on an algorithm which closely resembled the λ df-pn algorithm. We exchanged ideas and refined it. I particularly thank Dr. Soeda for useful discussions, which nobody else in the world could have understood.

Before I decided to leave Fujitsu Laboratories, I asked several questions to Akihiro Kishimoto by email; he kindly gave me precise answers, although he was busy working on his own Ph.D. thesis at the University of Alberta. He is an expert of the df-pn algorithm and I learned much from his papers. Based on discussions with Dr. Kishimoto and Prof. Martin Müller (also from UofA) using AOL messenger, I wrote a paper about the λ df-pn algorithm, which is a main topic of this thesis. Prof. Müller gave me many insights into the algorithm. I also learned much from him about how to present ideas in papers, from the corrections and revisions he made to my draft. (AOL messenger and the development of internet technology are acknowledged also for providing me a chance to make discussions over the Pacific Ocean.) Later, in 2006, I visited the University of Alberta. He and the members of the computer Go group at the Computer Science Department of UofA showed me excellent hosipitality. I also want to thank the members of the group. I enjoyed my stay in Canada very much.

Dr. Kaneko, Dr. Kishimoto and I worked on an experimental Go program named Akebono. I learned much from their code. It helped me to achieve a good result for writing my thesis.

Lastly, I want to thank my family. When I became nervous, worn out, or suffering from illness, they patiently encouraged and supported me. Their support was the greatest reason that I was able to finish this difficult work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

This thesis specifically examines new approaches of AND/OR tree search. Many problems can be expressed as an AND/OR tree. Especially in two-player zero-sum perfect information games, it is necessary to search a complex AND/OR tree. Our approaches are suitable for a group of AND/OR tree search problems that remain unsolved, despite the fact that the size of the search space is sufficiently small for well trained human beings to obtain the answer deterministically.

1.1.1 Purpose of Game AI Study

Before starting the explanation of our approach, we start with the purpose of studying game AI topics, as they are often regarded just for fun and are not assigned much importance.

The ultimate goal of AI research can be explained as developing something as intelligent as human beings. Shannon claimed in his own paper about the possibility of chess, that the topic is interesting but will not be important. Actually, since the development of alpha–beta search, game AI topics are far from the ultimate goal of AI research. However, game AI remains an interesting target. An important purpose of game AI is that the field is an excellent test bed for algorithms.

In machine learning, techniques developed for other domains such as natural language processingand optimization controlhad been used to automatic generation of good evaluation functions. In search algorithms, techniques for speed improvement such as the efficient use of hash tables and both theoretical or heuristic pruning were developed especially for game tree searching. Many researchers tested techniques for games developed for other topics and gave feed back from knowledge obtained in game AI research to other topics. Search and machine learning are two major fields that are related to game AI. Examples of the feed-back from search algorithms are in research fields such as sequence alignment problemsSAT solverplanning and theorem prover.

Our interest is in search algorithms. Two-player zero-sum perfect information games can be solved optimally by searching a mini-max tree. However, the search space is so enormous that it is impossible to give the actual solution for popular two-player zero-sum board games such as chess, Othello, and Go. On the other hand, game playing programs require modest real time capability to react to moves of the opponent. Development of game-playing

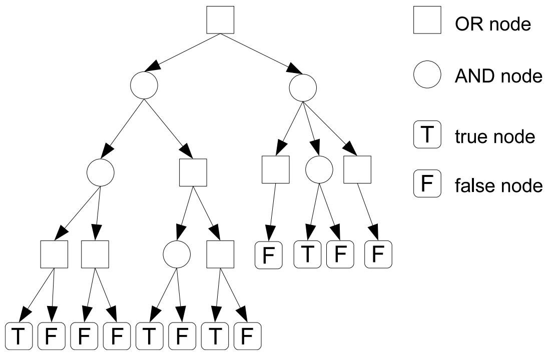


Fig. 1.1 General AND/OR tree example

computers is not too difficult but sufficiently difficult to be an interesting topic.

Because of its clear motivation (to beat a human champion) and clear evaluation (the stronger the better), many researchers have challenged the development of game AI. The rapid development yielded good results. The most famous example was in chess: Deep Blue beat the human champion Kasparov [57, 16].

It is also one area that clearly illustrates the ability of computers to society.

1.2 AND/OR tree

The target of this thesis is to search an AND/OR tree. An example of an AND/OR tree is portrayed in Fig. 1.1. An AND/OR tree is a tree that consists of terminal nodes with a logical value of either true of false, and internal nodes of two types: AND nodes and OR nodes. The logical value of an internal node is determined by calculating AND or OR of logical values of children. The goal of an AND/OR tree search is to determine the logical value of the root node.

In general, AND nodes and OR nodes are arbitrarily placed in trees. If AND nodes are OR nodes are placed in alternative layers, then AND/OR trees can naturally describe two-player zero-sum perfect information games, if we consider true as one player's win and false as the other player's win. Fig. 1.2 presents an example of such an AND/OR tree. The OR nodes stand for game positions with first player's turn, whereas AND nodes are positions for the other player's turn.

The goal of AND/OR tree search is to determine the logical value of the root node for given AND/OR trees.

Searching complex AND/OR trees are often needed in domains such as theorem prover, SAT solver and games because various types of complex AND/OR trees typically exist in endgames (e.g. chess, Shogi, checkers) and local problems (e.g. the capturing problem of Go).

1.3 Search algorithms and their fields

Mainly, three groups of search algorithms exist for the complex trees that often appear in games. This section presents discussion of the differences of these groups of algorithms and

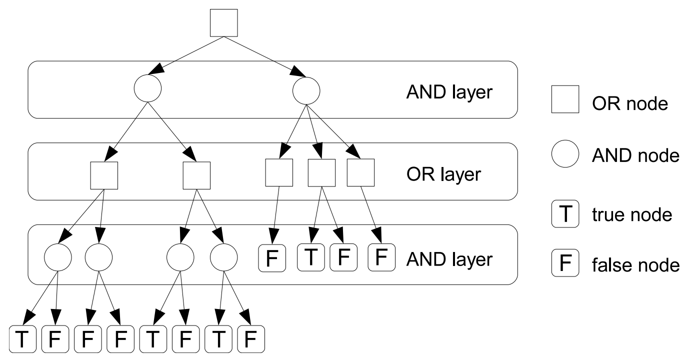


Fig. 1.2 AND/OR tree example for two-player zero-sum games

Table. 1.1 Strength of computers for two-player zero-sum games

Checkers	Defeated human champion in 1994. Proved to be a draw by optimal play in 2007.
Othello	Defeated human champion in 1997.
Chess	Defeated human champion in 1997.
Shogi	Strongest amateur level.
Go	Moderately strong amateur level.

clarifies the target area and goal of this thesis.

- Monte Carlo Tree Search
- AND/OR tree search
- Alpha–Beta search (Mini-Max search)

The areas covered by MCTS, AND/OR tree search, and Alpha–Beta search are portrayed in Fig. 1.3. The left side area is covered by the newly developing Monte Carlo Tree Search, and the other side is covered by AND/OR tree search.

The difference of these two algorithms might be readily apparent. The AND/OR tree search is applied to problems which have a small search space and search is expected to reach terminal nodes. However, MCTS, in contrast, is used for problems in which reaching the terminal nodes is not expected or is impossible.

As the name implies, Monte Carlo tree search is a probabilistic search algorithm. Therefore, it has the ability to search imperfect information games and non-zero-sum games. Details of MCTS will be precisely described in chapter 2. It has a great advantage that it requires no evaluation function. However, it also has an intrinsic weakness in finding a long narrow sequence which engenders a win.

As presented in Table 1.1, computers have achieved high strength in most two-player zero-sum perfect information games. All games in the table except Go can be played using alpha beta search. This fact shows that the combination of evaluation function and alpha–beta search is highly effective for a game tree search. Alpha–beta search covers a part of both areas for an AND/OR tree search and a Monte Carlo Tree Search. The lower area in Fig. 1.3 shows the field covered by alpha–beta search. The area of AND/OR tree can also be partly

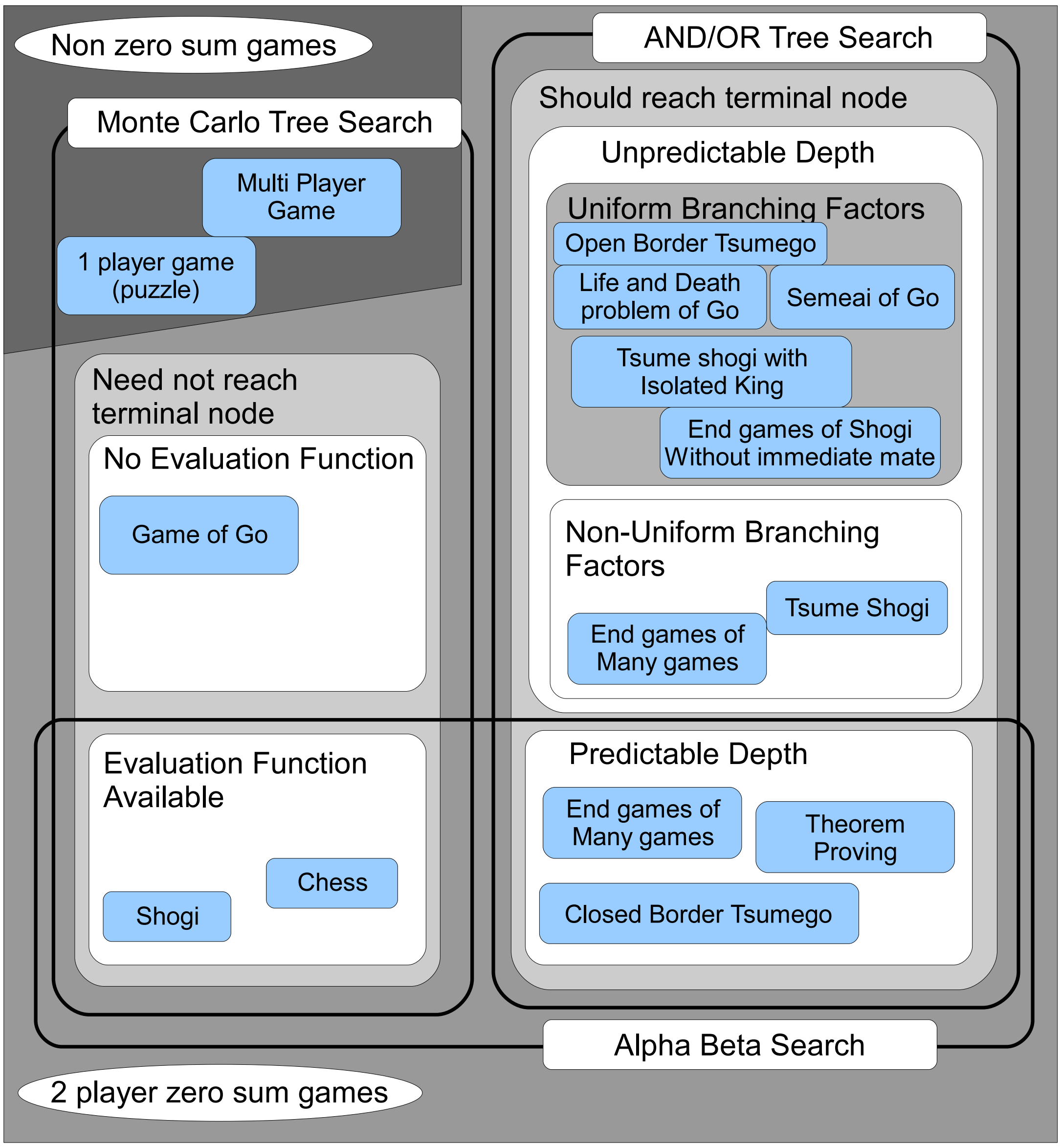


Fig. 1.3 Search algorithms and there fields

covered by alpha–beta. The area covered by both AND/OR tree search and alpha–beta search has a search space with "predictable" depth. Part of the area for MCTS is also covered by AND/OR tree search if it is possible to prepare good and fast evaluation functions.

Alpha–beta search took the main place in game AI studies for many years. However, since the development of the Monte Carlo Tree Search in 2006, interests of game AI researchers have shifted to MCTS. As depicted in Fig. 1.3, MCTS covers a large area of problems compared to alpha–beta search, and for problems for which MCTS is unsuitable, AND/OR tree search is more effective than alpha–beta search.

This thesis targets the remaining area, which is covered only by AND/OR tree search.

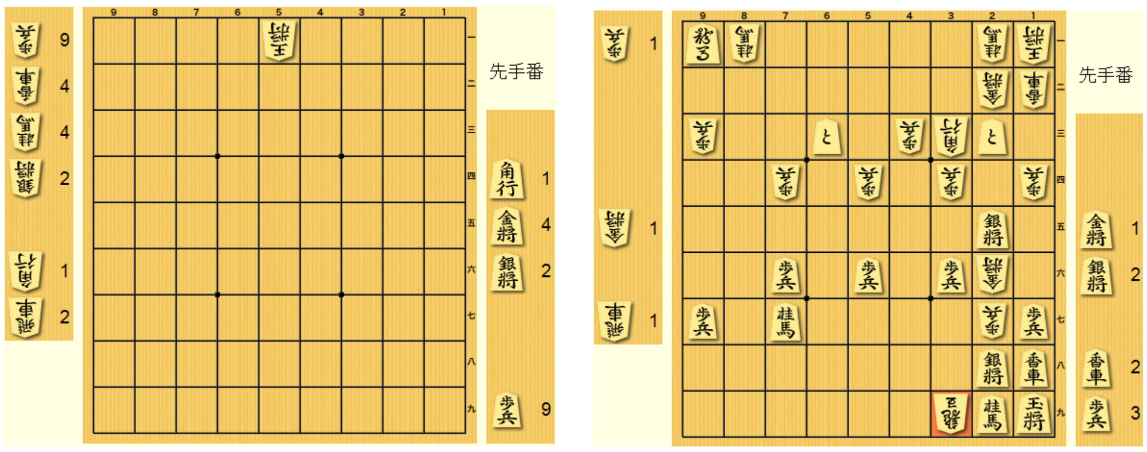


Fig. 1.4 Naked King tsume shogi and an end game of shogi

1.4 Remaining problems in AND/OR tree search

In the fields of AND/OR tree search, some groups of problems are difficult for existing algorithms to solve. The upper right area painted in gray in Fig. 1.3 shows the remaining difficult area. The area contains problems which can be expressed as AND/OR trees, and with which humans are superior to computers. The goal of this thesis is to analyze the difficulty of the area, and develop algorithms that are suitable to solve the problems in the area. We begin by an examination of what is causing the difficulty.

Two examples from *shogi* are depicted in Fig. 1.4. The left board portrays a *tsume shogi* problem. A *tsume shogi* means a checkmate problem of *shogi* (Japanese chess) in which only check moves are permitted for the attacker. It is widely regarded that today's computers are far superior to human beings in solving *tsume shogi*. However, computers are still not good at solving problems with the isolated king. The most illustrative type of that problem is called naked king *tsume shogi*, which is shown on the left of Fig. 1.4. The other example from *shogi* is endgame without direct checkmate. Direct mate means a checkmate after a sequence of checks. The position in the right of the figure is taken from a historical *shogi* game played between a human professional player and a *shogi* program. After the game with the program *bonanza*, Watanabe, the grand champion who had the Ryuoh title, said that "I found out that computers do overlook a move. I felt a little bit relieved because I thought computers are perfect in the end games." The move portrayed in the position is the move he expressed as the computer overlooked [*1].

From the game of Go, two *tsumego* problems are depicted in Fig. 1.5. The problem on the left is taken from a classical Go book hatsuyo-ron [96]. The original problem is difficult for existing solvers. Kishimoto reported in his thesis [34] that it can be solved by his solver if artificially enclosed as portrayed in the right of the figure. For human beings, no difference exists in the difficulty of these two problems. Simply put, the goal of this thesis is to fill the gap separating these two problems.

[*1] Computers never overlook a move. Developers often use the word "underestimate" instead of "overlook".

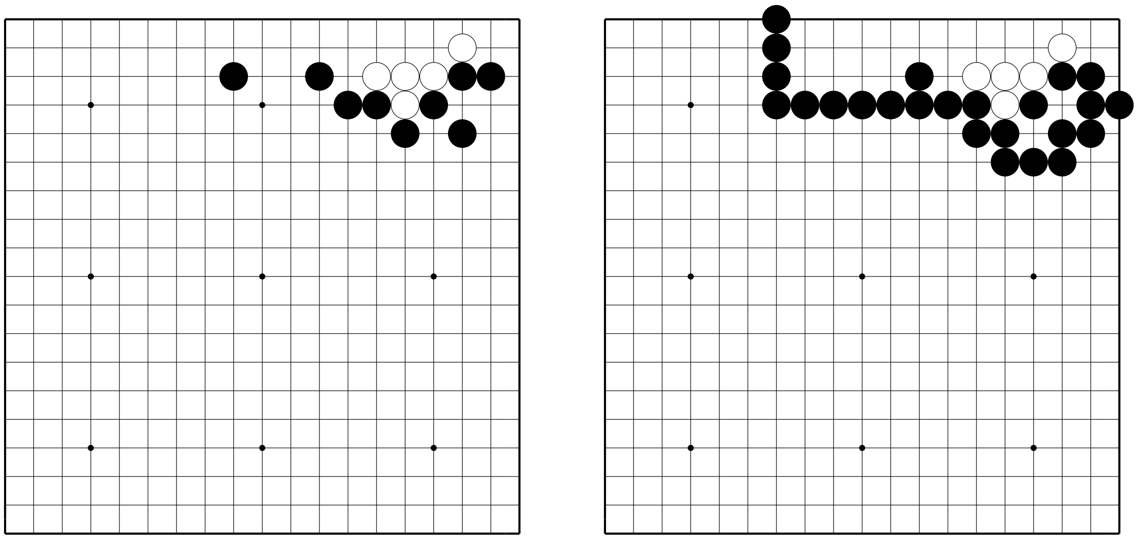


Fig. 1.5 Open tsumego and closed tsumego

The examples from shogi and Go are symbolic problems which remain difficult for computers They reside in the upper right area of Fig. 1.3. These are not special limited classes of problems but are natural problems. In the next section, we discuss the difficulty of the problems in that area.

1.5 Difficulty of Large and Uniform Branching Factors

One common characteristic of problems in the middle right area (Non-Uniform Branching Factors) of Fig. 1.3 is that the search spaces of these problems have small or non-uniform branching factors. On the contrary, difficult remaining problems have search spaces with large and roughly uniform branching factors. This difficulty is strongly related to the characteristics of a group of AND/OR tree search algorithms that are based on proof numbers. In fact, computer successes in the area of solved problems rely greatly on variants of Proof Number Search algorithms. Actually, PNS[8], PDS[52], and df-pn[53] are examples of such algorithms. These algorithms typically rely on proof numbers and disproof numbers.

How proof number search works can be explained as searching part of the search tree that is expected to be the easiest to prove or disprove. In other words, game players based on proof number search choose moves which minimize the number of the other player's options. This behavior has been effective for many problems including tsume shogi[53], tsumego[34], and endgame of checkers[67]. Variants of df-pn are currently used as the standard algorithms for end game solvers of many games.

However, to solve remaining difficult problems, search must go into parts of search trees in which the opponent would have numerous options. Proof number search hesitates to go in such a direction. This behavior can be explained as follows. The proof number search is slow if branching factors are not a good measure for controlling search direction. Therefore, problems that have uniform branching factors are difficult for the proof number search. Detailed discussions related to the proof number search are included in chapter 3.

These problems are natural subproblems of popular games. These games not only attract the interest of AI researchers but also have fairly large markets for computer players. Many reports have been published about computer Go and computer shogi. Nevertheless, despite efforts of researchers and developers, these problems remain as practically unsolvable problems.

1.6 Goal of the Thesis

Regarding the following analyses of remaining difficult problems, we have observed that the difficulty results from search spaces with large and uniform branching factors. Our goal is to develop search techniques that are suitable for such search spaces.

No algorithm would solve it if the search space were truly difficult. To solve an AND/OR tree search problem, a *proof tree* must be found. A *proof tree* is a subtree of the AND/OR tree that covers a sufficient part of the tree to prove the logical value of the root node. It is impossible to solve the problem if the size of the smallest *proof tree* is enormous. However, the difficult problems listed in the previous section can be solved by human experts, which means the *proof tree* of the problems have sizes that are sufficiently small for human beings to remember, but existing algorithms can not solve them efficiently. Therefore, in this context, the phrase *difficult problems* has a meaning that is close to problems which human beings can solve but which are difficult for computers.

These search space appear as plain and clueless to proof number search algorithms. To find answers from such search spaces efficiently, the ability to extract information from plain spaces is necessary.

Existing approaches used domain-dependent knowledge to somehow deal with difficult problems. Tsumego solvers represent the results of such efforts (gotools [91], Kishimoto's solver [34]). However, our goal is to develop a search algorithm for uniform branching factors based on domain-independent techniques.

The application area of search algorithms would be expanded to new domains if we were to succeed in developing such domain-independent techniques. Because remaining difficult problems are not special cases but instead include many natural problems, achieving this goal will be a valuable contribution to the development of search algorithms. It would also be ideal to cover the weakness of the Monte Carlo Tree Search.

Two approaches are proposed herein to tackle search spaces with large and uniform branching factors.

One approach is to extract a hidden hierarchy from search spaces. Existing approaches have used either handcrafted or automatically generated evaluation functions to control search directions that extract information as a single scalar value. Herein, we propose an approach based on a hierarchy that can be found by performing a specialized search algorithm for finding the hierarchy. This approach yielded an algorithm that we named λ df-pn. Fig. 1.6 shows a rough sketch of this approach. A search space that seems plain to proof number search algorithm can be searched efficiently using the hierarchy.

The other approach is to examine only promising subtrees from all possible subtrees. What is often observed in problems with large branching factors is that a large number of unpromising frontier nodes cause bad effects in the search direction. Frontier nodes are non-terminal

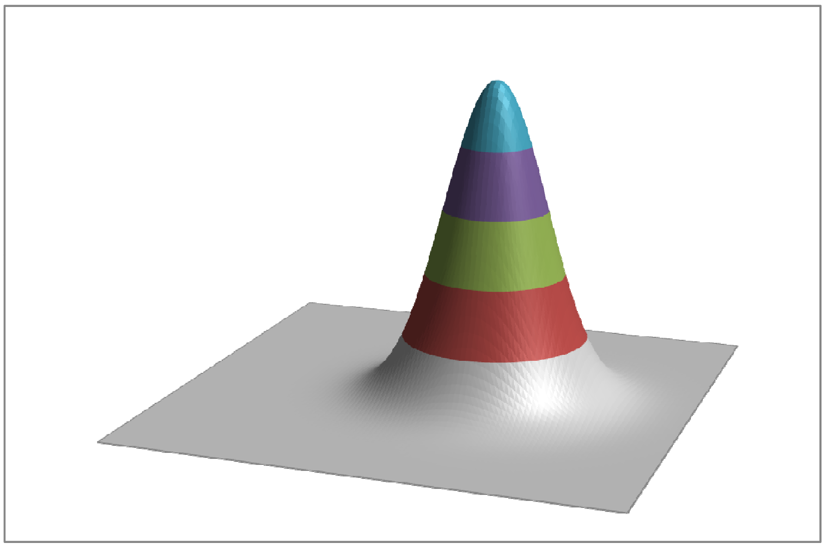


Fig. 1.6 Hierarchy in search space.

nodes in search trees whose children are not being searched at a moment. Based on this approach, we propose a search technique named Dynamic Widening.

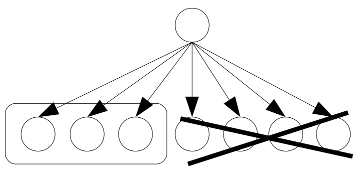


Fig. 1.7 Ignore unpromising moves.

1.7 Contributions of the Thesis

The main contributions of the thesis are the analysis of characteristics of the difficult problems and the development of search algorithms that are suitable for the problems.

Technical contributions in AND/OR tree search aspects are

1. Efficient solver for local problems (true/false result)
2. Efficient end game solver (terminal position requires no evaluation function)
3. Covering the weakness of MCTS (MCTS is weak for long narrow answering sequences)

Since the breakthrough of MCTS in 2006, the strength of computer Go players is improving rapidly. However, as described above, MCTS has one weakness in detecting long narrow winning sequences. Moreover, AND/OR tree search based on proof numbers are ideal algorithms for finding such sequences. Therefore, it is expected that the combination of MCTS and PN search will eliminate this weakness.

Furthermore, development of local problem solvers for Go are interesting topics themselves. Solvers can be used to evaluate board positions in games, and can be used to check the correctness of problems.

Based on the experience in the development of Go programs, it is widely regarded that how to combine local search result in overall play is not trivial. The main problem is that it is too time consuming to call a local search for all local targets. One solution for this problem is to preserve the local search results, although the results are valid. Often, local search affects only a region of the whole Go board. An algorithm was developed to detect such areas, which we call inversions. As long as an inversion remains unchanged, search results still hold.

1.8 Structure of the Thesis

Chapter 2 presents an explanation of the background of the studies in search algorithms for complex trees, mainly in games. This chapter specifically examines the explanation of the alpha–beta search and Monte Carlo Tree Search to provide readers sufficient knowledge to understand the difference of these two search algorithms and algorithms specifically developed for AND/OR tree searches.

Chapter 3 presents explanations of existing work about AND/OR tree searching: mainly proof number search algorithms. The comparison of proof number search, alpha–beta search, and MCTS are also presented in the chapter.

This thesis proposes two techniques. One is λ df-pn, which is explained in chapter 4. The other is Dynamic Widening, which is explained in chapter 5. The two techniques are combined and the performance of the overall algorithm is described in chapter 6. We also discuss the limit of our algorithms.

An additional result is related to *inversion* search. This technique isolates search areas from the other part of search space. It is useful to combine a local search with a global search. Chapter 7 presents an explanation of the technique.

The last chapter concludes the thesis and describes future work.

1.9 Publications

Actually, λ df-pn algorithm, which is described in chapter4 was first published in a paper titled “Lambda Depth-First Proof Number Search and Its Application to Go”. The paper was presented at the 20th International Joint Conference on Artificial Intelligence (IJCAI-07) [95]. Chapter 4 presents the results described in another paper titled “λ Search Based on Proof and Disproof Numbers”, which was published in the Journal of Information Processing Society of Japan (IPSJ) [75]. Chapter 5 is based on a paper titled “A New Proof-Number Calculation Technique for Proof-Number Search” which was presented at Conference on Computers and Games 2008 (CG 2008) [94]. Chapter 7 is based on the results in a paper “A Search Algorithm for Finding Multi Purpose Moves in Sub Problems of Go” presented at the Game Programming Workshop 2005 (GPW05) [93]. This chapter also includes additional ideas and results that are first presented in this thesis.

Chapter 2

Search Algorithms and AI

In this chapter, we will explain Alpha–beta search and Monte Carlo Tree Search (MCTS) in detail. These algorithms are compared with AND/OR tree search algorithms in the next chapter. To understand the comparison, readers not familiar with Alpha–beta search or MCTS are encouraged to read this chapter.

2.1 Alpha–beta Search

2.1.1 Basics of Alpha–beta Search

Two-player zero-sum perfect information games are considered to be the most fundamental example in game theory. It is well known that the optimal move can be determined by searching a minimax tree for such games.

The optimal move can be obtained if the game tree is sufficiently small for a brute force search. However, most interesting two-player zero-sum games have enormous search spaces for which it is practically impossible to determine the optimal move. In his Ph.D. thesis, Allis described a precise analysis of the size of search spaces for the 15 most popular two-player zero-sum games. Table 2.3 portrays a part of the list.

The Fig. 2.1 presents an example of a game tree. Each node stands for a game position. The root node stands for the initial position and the terminal nodes stands for positions at which the game has ended; each terminal node has a score. The first player tries to maximize the score, which is to reach a terminal node with the highest possible score. The other player tries to minimize the score. In this example, if both players choose their optimal moves, the game will end with a score of 50.

In a naive minimax tree search, all nodes of the tree will be searched. However, to obtain the score through optimal play, it is not necessary to search all nodes in the tree. It is known that it is possible to prune numerous nodes in the trees without changing the optimal score. Knuth and Moore [41] gave a precise analysis for the pruning technique for a minimax tree search. This enhancement is known as the famous Alpha–beta pruning, and a minimax search that uses Alpha–beta pruning is called *Alpha–beta search*.

A crucial advantage of Alpha–beta pruning is the ability of deep cut off. It was already known in the 1960s that Alpha–beta pruning can greatly reduce the number of nodes that must be evaluated during the search.

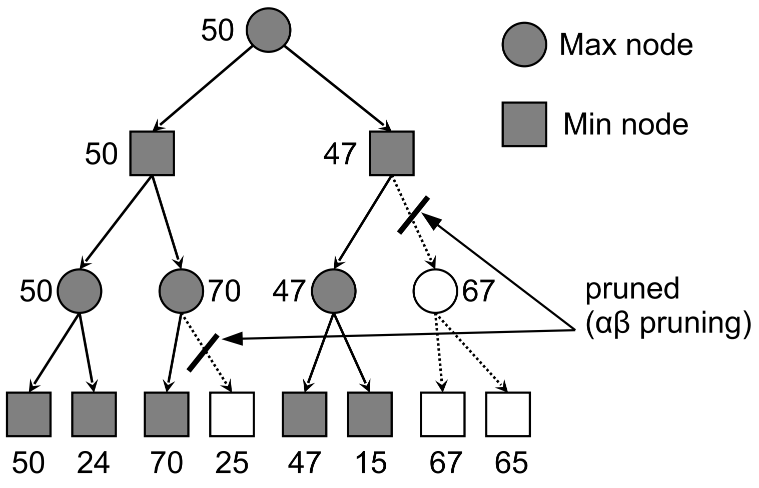


Fig. 2.1 Minimax Tree and Alpha–beta cut.

2.1.2 Enhancements for Alpha–Beta Search

In practical cases, the target tree is unbalanced, the depth is not known before the search and terminal nodes could not be reached in a practical time threshold. Mainly, three functions are needed for Alpha–beta pruning or Alpha–beta search to achieve high performance in such realistic cases. These functions are

- Move Ordering,
- Evaluation Function, and
- Depth Threshold Control.

Move Ordering

Let there be a game tree with a constant branching factor of b and depth d. The number of nodes searched in a simple minimax search will be b^d. However, if moves are ordered ideally from the best to the worst move, Alpha–beta pruning can reduce the number of searched nodes to $\lceil b^{\frac{d}{2}} \rceil + \lfloor b^{\frac{d}{2}} \rfloor - 1$. On the other hand, if the moves are searched in an order from the worst to the best, then the number of nodes would be identical to that of a minimax search. The paper [41], also gives a performance analysis for a tree with random values at terminal nodes. Their result shows that it is highly unlikely that the performance would become as bad as that of the worst case, but still *move ordering* is a key factor in the performance of Alpha–beta pruning in real applications.

Evaluation Function

The discussion related to the correctness of minimax search above subsumes that the terminal nodes give accurate scores of the game. However, this is only true when the value at the terminal nodes of the game tree represents the true score of the game.

In actual game play, it is almost impossible to reach terminal nodes except in the late end games. Therefore, it is a mandatory to prepare a guess of the game score at the inner nodes of the game tree. In practice, in game playing programs, Alpha–beta search is cut off at certain depth and the frontier nodes are treated as the terminal nodes. The values of the nodes are evaluated by the *evaluation function*, which returns the estimated values of the true score of the game.

Introductory books for chess always include a table that shows the values of the pieces (Table 2.1). A simple evaluation function is available for chess merely by calculating the sum of the values.

Table. 2.1 Value of chess pieces

pawn	bishop	knight	rook	queen	king
1	$3+\alpha$	3	5	9	∞

A fast and accurate evaluation function is necessary for the performance of Alpha–beta search. Evaluation functions can be either generated by hand-tuning or machine learning. Most evaluation functions used for chess variants are developed by adding evaluation of the board conditions to the values of the pieces. Values of the pieces, king safety, and relative mobility are typical elements of the evaluation functions. For Othello, evaluation functions can be generated by learning patterns on important part of the boards such as corners and sides.

Depth Threshold Control

Alpha–beta search is implemented as a depth-first search. As explained above, it requires a threshold at which to cut off the search. The method of defining the threshold is also an important factor for the strength of game playing programs.

Many reports describe how to control the search depth, so that the promising parts of the tree would grow deeper, while the unpromising parts are being pruned at shallower nodes. Normally, the value of the nodes are used to decide which node is more promising, resulting in highly unbalanced search trees.

Depth control is especially important when applying Alpha–beta search to goal-oriented AND/OR tree search, which returns binary scores (e.g. win/loss, true/false). However, these problems have less information than a normal game tree search for deciding which node is more promising because internal nodes have only two values. In these cases, to achieve good performance, Alpha–beta search must be combined with depth control, which is not based on the values of the internal nodes.

2.1.3 Other issues

These three functions are closely related to the fact that move ordering is often done based on evaluated values of nodes, and that the search depth is typically extended for promising moves according to the values of the evaluation functions.

Several other techniques are used as standard enhancements for Alpha–beta search. Probably, the most important remaining technique is iterative deepening.

Iterative deepening is a simple technique that is often used for search algorithms of many types. First, perform a search by depth threshold d, and continue the search by incrementing

d until a given time limit.

At first glance, iterative deepening seems to cause overhead by searching the same nodes repeatedly. However, at least two great advantages can be discerned.

One is the ability to play moves in a certain time threshold. It is difficult to predict the time need to perform an Alpha–beta search with a given depth threshold. The best move found in the previous iteration could be used as the best move so far if iterative deepening is used.

The other advantage is the improvement in move ordering. The best moves found in the last iteration are highly likely to be the best move again in the next iteration. It is a standard technique to store information of good moves and their best child node in a hash table and reuse them in the following iterations. The hash table used in the Alpha–beta search is called a transposition table.

Hand-tuned heuristics and machine learning techniques that are combined with iterative deepening yielded high-quality evaluation functions in real game playing. The best move determined by the value of the evaluation function turns outs to be the best move after the search with a probability as high as 90% in recent sophisticated chess programs.

2.2 Monte Carlo Tree Search (MCTS)

In this section, we explain the details and theoretical backgrounds of MCTS, a brand new algorithm first published in 2006. Therefore, most results explained in this section were done in this two and a half years. Currently, many AI researchers study MCTS intensively. It is a rapidly developing topic. For that reason, the description in this section might not remain valid for a long time.

2.2.1 A Revolution in Computer Go

For many board games, computer players have become as strong as or stronger than human champions, as presented in the most famous example in chess: Deep Blue's victory against human champion Gary Kasparov. The game of Go is the only popular two-player zero-sum perfection information game in which human beings retain a great advantage over computers.

Despite the efforts of many researchers for some decades, the game of Go resisted challenges from computer players. The difficulty of Go arises from the fact that it is difficult to create a fast and accurate evaluation function. Before 2006, the strength of the best program was about the same as moderate amateur players.

Table 2.2 shows the strength of computer players in popular board games. It can be said that the successes of computer players for all the games listed in the table are given by the Alpha–beta search, with the sole exception of the game of Go. The game of Go was regarded as a grand challenge of game AI for a long time [50, 10]. It was clear that Go has some characteristics that resisted the efforts of computer Go researchers.

However, in 2006, a totally novel search algorithm was developed that does not rely on evaluation functions. These algorithms are currently called Monte Carlo Tree Search (MCTS). Within only two and a half years since its development, MCTS was roundly successful in computer Go, which garnered several victories against professional Go players in small board (9×9 board) games, and considerable improvements in the strength for 19×19 board games also. Professional Go players played against Go programs on several occasions.

Table. 2.2 Strength of Computer Players (two-player zero-sum games)

Checkers	Computer "Chinook" defeated human champion Tinsley in 1994[64] (Draw is proven in 2007[67])
Othello	Logistello defeated human champion Murakami 6–0 in 1997 [14] (It is widely believed that a computer had already reached human champion level some years before 1997.)
Chess	IBM Deep Blue defeated champion Kasparov in 1997 [57, 16].
Shogi	At approximately the same strength as strongest amateur players in 2008.
Go	Around amateur *shodan* on normal 19×19 board and strong amateur level in 9×9 board in 2008.

Table. 2.3 Estimated State Space Size (Number of Possible Positions)

game	Size
Checkers	10^{20}
Othello	10^{28}
Chess	10^{50}
Shogi	10^{71}
Go (9×9)	10^{38}
Go (19×19)	10^{171}

The results reflect the rapid improvement of Go programs.

The motivation behind the development of Monte Carlo Tree Search is related strongly to the difficulty of Go. More detailed analyses of the difficulty of the game of Go will be described in the next section.

2.2.2 Difficulty of Go

Readers who are not familiar with the rules of Go should refer to Appendix B.

Size of State Space

A clear reason for the difficulty of Go is that it has numerous legal moves; moreover, each game has many moves.

It is possible to determine the optimal move if the search space of Go is sufficiently small. Go is officially played on 19×19 boards, but because of its simple rules, Go can be played on boards of any size. Actually, a complete analysis of Go on a 5×5 board was presented in an earlier paper [85].

However, as presented in Table 2.3, the state space of Go on 19×19 board is the greatest among all popular board games. It is impossible to find the optimal move by brute force search.

It is noteworthy that the difficulty of the games (for computers) has a strong relation with the size of search spaces. From the history of game AI in other games, it was known empirically that if two games have similar rules, computers are stronger for the game with the smaller search space. One counter example was the game of Go. As presented in the table, Go on a 9×9 board has a state space that is smaller than that of chess. However, before 2006, it was known that Go programs for a 9×9 board were merely as strong as the amateur shodan, which is close to the strength achieved for a 19×19 board. This fact shows that the difficulty of Go is inexplicable merely by the search space size.

2.2.3 Evaluation Function for Go

As explained above, Alpha–beta search requires an evaluation function to play games, except in the end games. Evaluation functions must be sufficiently accurate to be used for directing move ordering and depth threshold control. At the same time, the evaluation function must be sufficiently fast that it can be executed at least several tens of thousands of times per second because it is called every time the search reaches its depth threshold.

Development of an appropriate evaluation function for Go is known to be extremely difficult. Several reasons explain that difficulty. Unlike chess variants, each stone has equal value in Go. Unlike Othello, it is difficult to determine which part of the board is more important than other parts (this is especially true for a 19×19 board). The aim of Go is to occupy territory. Therefore, it seems a good idea to evaluate the size of territory. However, the territory can be only counted in late end games. Fig. 2.2 shows the difficulty of estimating territory size.

In addition, local best moves often turn out to be bad moves globally. For example, capturing opponent's stones are locally good moves, but it is a basic technique to sacrifice stones to get benefit in other places. The situation of the board changes during the game play such that often stones that were important become worthless. Abandoning such stones would give advantages to players. Fig. 2.3 presents a splendid example of sacrificing of stones.

In summary, no good criteria exist for an evaluation function in Go. It is difficult to make a quick and accurate evaluation function that consists of sum of simple elements. Despite the efforts of AI researchers and commercial programmers, currently available evaluation functions for Go are either inaccurate or unacceptably slow.

2.2.4 Example of Classic Go Program, GNU Go

Classical Go programs, Go programs developed before 2006, have somehow managed to reach the strength of amateur players with ranking of *shodan*. It is difficult to explain the strength of shodan but it is probably roughly comparable to chess rating 1200.

GNU Go [28] is a free Go program that was the strongest non-commercial Go program before 2006. GNU Go uses many quantities of complex evaluation functions to evaluate board positions.

How GNU Go works on each board position will be briefly explained. First, GNU Go examines the board position and evaluates the strengths of groups of stones. Secondly, it determines moves that match the GNU Go pattern database and assigns values to the moves

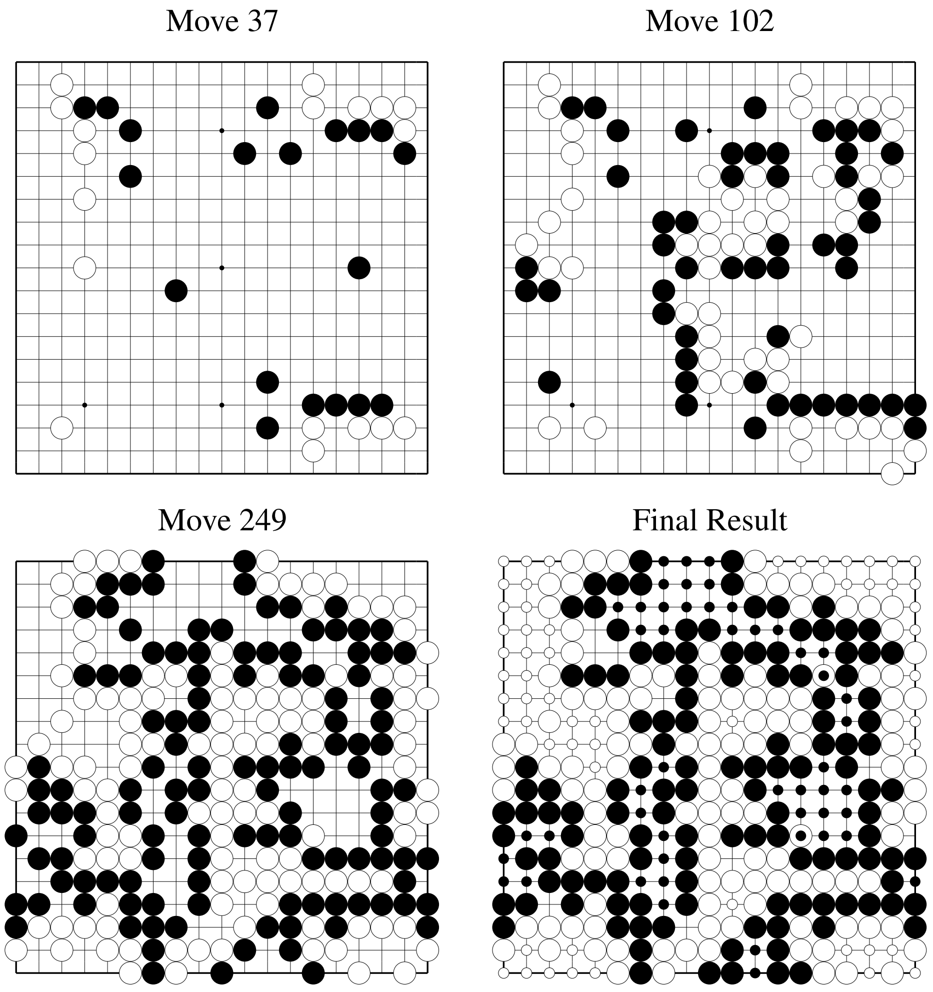


Fig. 2.2 Territory becomes clear only in the end games. This is a game played between professional players. Black is Masaki Takemiya; white is Cho Chikun. First, the middle right area seems to be the territory for black. However, as the game progresses, the situation becomes extremely complex. In the end, white won by 8.5 points (including *komi*).

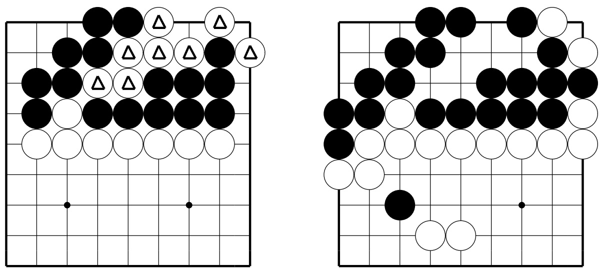


Fig. 2.3 Example of a successful sacrifice. White sacrificed stones marked with triangles to occupy the lower side of the board.

according to the patterns. Thirdly, it generates moves based on predefined goals. Examples of goals are the protection of one's own stones, attacking of opponent's stones, and enlarging one's own territory. Finally, it checks the dependencies of the values and plays the highest evaluated move. This method resembles the way human players play.

The values explained above have different meanings. It is difficult to find appropriate moves based on these values, and it requires subtle hand-tuning to produce a strong program. The source code of GNU Go is more than 80,000 lines, mainly written in C. Apart from the code, GNU Go has a good pattern database that consists of more than 52,000 lines in ASCII text. This pattern database includes not only simple patterns but complex patterns that rely on the result of simple capture search. In fact, GNU Go will weaken greatly if it is turned off. GNU Go is a result of the arduous efforts of many programmers.

In fact, GNU Go is slightly weaker than shodan on 19×19 boards, and has about the same strength also on a 9×9 board.

2.2.5 Monte Carlo Tree Search

As explained above, development of a good evaluation function of Go is difficult. However, if the game has ended, it is easy for computers to evaluate the score and the winner. Go programs based on Monte Carlo Tree Search are using this fact.

Primitive Monte Carlo Go

A group of researchers investigated the possibility of a Go program based on random moves from both players. This idea dates back to 1993 when Brügmann developed a Go program based on Monte Carlo simulation [13]. The original program by Brügmann was weak, but several researchers followed this idea and developed Go programs based on this idea. Programs developed by Bouzy[11] and Cazenave[20] are the examples.

In the game of Go, the number of legal moves decreases toward the end game. Because of this characteristic, if a constraint is added, it is possible for players who randomly choose a legal move to end the game properly.

As explained in the Go rules section in chapter B, stones will become *alive* if they have two or more *eyes*. The game will end properly even if both sides play randomly if both players merely avoid playing on their own eyes. In terms of the Monte Carlo Tree Search, it is called a *playout* to play a game until the end by two random players.

Primitive Monte Carlo Go programs are based on an extremely simple idea. First, perform a *playout* from a board position and get the result. Fig. 2.4 depicts a board position and a result of a playout. A simple playout is performed to evaluate the position shown on the left. Counting the score in the right board position would yield a score of black 34 and white 47, thereby white won [*1].

From a human Go players' perspective, this result is far from close. However, if the number of playouts is great, the evaluation would be more reasonable. As depicted in Fig. 2.5, many playouts are done for each move, and the move is selected according to the average score of the move.

However, primitive Monte Carlo Go has one intrinsic weakness. Programs using this

[*1] In this example, *area counting* is used. Correctly handling *territory counting* is a complex topic, and will not be described precisely in this thesis.

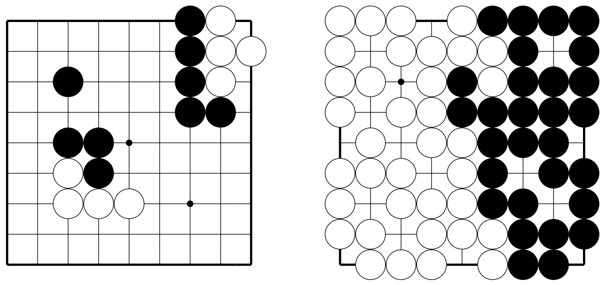


Fig. 2.4 Evaluating position by random play. The image is from slides used by Rémi Coulom in an invited talk given at Game Programming Workshop 2007

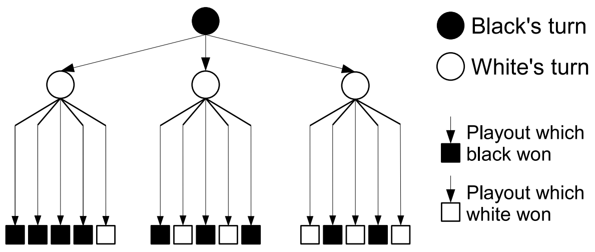


Fig. 2.5 Primitive Monte Carlo Go

method select moves based on the outcome of random plays. Therefore, they often overestimate moves which are bad moves but for which only a small number of correct countermeasures exist. If such a move exists, then it is unlikely that these programs correctly notice the situation. Therefore they play the move, and lose. For game trees with depth of two or higher, this weakness could not be solved even if infinite number of playouts were done. We investigated the diminishing returns of the strength and the number of playouts[92].

The term Primitive Monte Carlo Go was coined especially for use in this thesis, to clarify the difference of these programs and programs based on Monte Carlo Tree Search. In fact, the programs developed by Bouzy and Cazenave are based on a more complex and sophisticated approach. Nevertheless, they share the weakness explained above.

Go Programs based on MCTS

The Go program CrazyStone, developed by Rémi Coulom [24], won the 2006 Computer Olympiad [5]. That program and MoGo, developed by Sylvain Gelly et al. [27], are widely regarded as the two strongest Go programs developed since 2006 (and still are at the end of 2008). They are both based on the new algorithm: Monte Carlo Tree Search.

At the beginning of the search, MCTS works in the same way as primitive Monte Carlo Go programs. Playouts are performed on each move and the expected score is calculated. The first difference is that MCTS assigns more playouts on more promising moves as depicted in Fig. 2.6.

The second difference is that MCTS records the number of playouts on each node. If the number of playouts exceeds a given threshold, the node is expanded and the tree grows

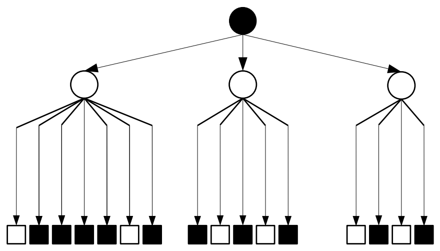


Fig. 2.6 Depth 1 MCTS / More playouts on promising moves

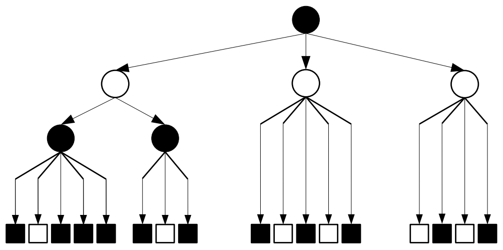


Fig. 2.7 MCTS / Tree grows on promising part

deeper. Because of this behavior, as the number of playouts increases, the tree would grow in an unbalanced manner: the promising part deeper and the unpromising part shallower (Fig. 2.7).

2.2.6 Theoretical Background of MCTS

Monte Carlo Tree Search algorithm was developed and used for CrazyStone by Rémi Coulom. Kocsis and Szepesvári improved the algorithm and provided a theoretical background [42]. MoGo used the improved algorithm Upper Confidence bound applied to Trees (UCT).

In this section, we will explain the theoretical background of the UCT algorithm.

Multi-Armed Bandit Problem

Multi-Armed Bandit problem is a problem that is studied in many fields such as statistics and machine learning. A Multi-Armed Bandit is an imaginary slot machine with multiple arms [*2].

The formal definition of Multi-Armed Bandit Problem is the following. Assume that more than one slot machine exists, each returning rewards according to an unknown probabilistic distribution. The goal is to gain the maximum reward with a given amount of coins (Fig. 2.8). The Multi-Armed Bandit is a problem that studies the strategy for choosing the appropriate machine.

If it is assumed that each machine stands for a move in Go, and that one coin stands for one playout, then a simple strategy that inserts the same number of coins to each machine is the strategy used for primitive Monte Carlo Go.

[*2] An one-armed bandit means a normal slot machine.

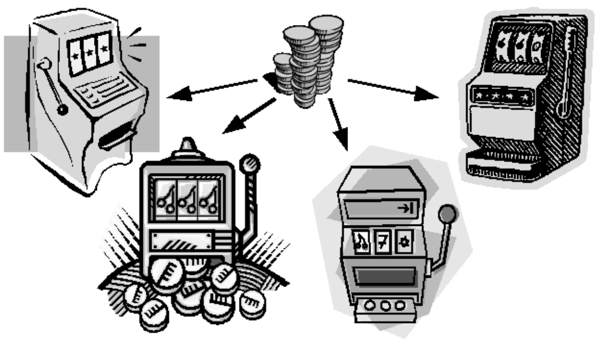


Fig. 2.8 Multi-Armed Bandit Problem. Choose the best arm/machine.

The asymptotic behavior of the optimal strategy is described in a classic paper written in 1985 by Lai and Robbins [45]. However, this paper does not describe a practical strategy that achieves good performance. Auer, Cesa-Bianchi, and Fischer gave simple strategies name Upper Confidence Bound (UCB) in 2002 [9]. These strategies were simple, but it is guaranteed that the the expected value of the difference between the total reward gained by UCB strategies and the optimal strategy is smaller than certain bounds.

The simplest strategy given in the paper is called UCB1. It calculates the UCB1 value for each machine given in eq. 2.1 and inserts a coin into the machine with the highest UCB1 value.

$$\bar{X}_j + c\sqrt{\frac{2\log n}{n_j}}. \tag{2.1}$$

Variable n signifies the total number of coins used so far, $\bar{X}_j$ stands for the average reward of the jth machine and n_j denotes the number of coins inserted in to j th machine so far. Constant c would adjust the behavior of the algorithm. In addition, c should be adjusted by experiments for each target domain.

This equation has the form of $(expectedvalue) + (bias)$ and the $bias$ for a machine becomes greater if the number of coins inserted into the machine is smaller. The concept of UCB is to give higher value for machines with higher expected value, but if the quantities of coins are small, to give higher biases them because the variances are great.

The behavior of the UCB1 strategy corresponds to Fig. 2.6 because it gives more coins to promising machines.

UCT (UCB applied to Trees)

First, MCTS used for CrazyStone used a formula based on a similar idea to that of the Boltzmann distribution[24]. It showed a great improvement in the strength of the Go program, but it was not based on UCB, nor did it have a theoretical proof that the algorithm would find out the optimal solution if given an infinite amount of time. Kocsis and Szepesvári improved the algorithm used by CrazyStone and developed UCT (UCB applied to Trees)[42]. Thereafter, UCT quickly spread among game AI researchers.

Starting from the root node, UCT follows the child node with the highest UCB1 value. If a frontier node is reached, it does a playout from the node. The node is expanded if the number

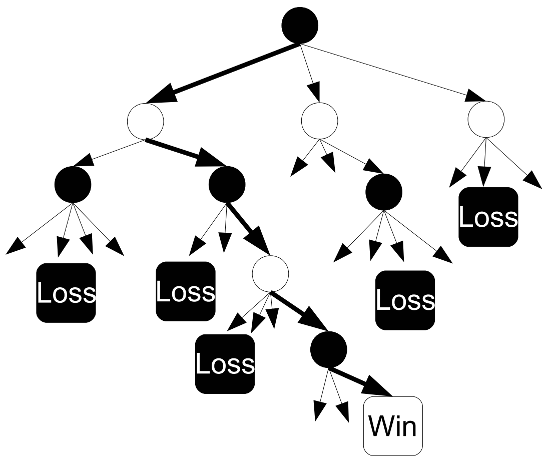


Fig. 2.9 MCTS is unlikely to reach the win.

of playouts at a frontier node exceeds a given threshold (Fig. 2.7).

The contributions of UCT are its higher performance than that of the original MCTS used by CrazyStone, in addition to the proof that UCT converges to the optimal answer if sufficient time is given. The order of the $bias$ term in UCB1 can be shown as $O\left(\log n/n\right)$ if the probability distribution obtained from playouts fulfills a certain assumption. Therefore, if the total number of playout n is sufficient, then an optimal solution can be found by following the child nodes with the highest average score.

The Monte Carlo tree search is a generic high-performance algorithm, but it is far from perfect. One intrinsic weakness is that it has difficulty in finding an answer after a long narrow sequence. As presented in Fig. 2.9, if only one WIN exists among many LOSSes, it is highly unlikely that the tree would grow sufficiently deep to find the WIN.

Another problem of MCTS is that it is only effective for problems in which random playouts are somewhat reasonable. The games of Go, Othello, and gomoku are good targets for MCTS because the number of legal moves converges to zero and the game naturally ends by random plays. On the other hand, random players in chess-like games are unlikely to reach a natural end game.

2.2.7 Development of MCTS

To address the weakness and enhance strengths, many approaches have been studied for MCTS. For the tree search part, most techniques developed for Alpha–beta search can be used also for MCTS. Interesting enhancements for MCTS are playout enhancements. Performance of MCTS using playouts enhanced with patterns are far superior to MCTS using entirely random playouts.

Actually, MCTS first proved its strength in the game of Go, but it was soon discovered that MCTS is highly effective for many other games. Many reports describe the successes of MCTS in two-player zero-sum perfect information games, such as Lines of Action[47], the game of Amazons[40, 46], and even in Shogi[62]. For multiplayer games, successes in card games have been reported in a paper about Hearts[77]. Even for a single player game, a good result is reported for SameGame[63]. A remarkable result is the development of a general game player. The General Game Player Competition takes place during the AAAI conference. A UCT based program, CADIA player [26] won the competitions in 2007 and 2008.

We used MCTS for a problem outside of the AI field. In fact, MCTS was used for security evaluation of biometrics recognition systems, yielding a good result[78, 79].

Parallelization of MCTS is not simple, but it proved to be easier than parallelization of Alpha–beta search. Many researchers are investigating parallelization of MCTS[23, 22, 32].

Summary

First presented in May 2006, MCTS is a brand new algorithm invented for use in a Go playing program. However, it now has many applications in game AI and other domains. We presented a brief picture of MCTS at the beginning of 2009, but because it is a rapidly developing area, the description in this chapter might soon become outdated.

Chapter 3

Search Algorithms for AND/OR Tree

In this chapter, we will describe existing studies of AND/OR tree search algorithms, and present a discussion about a comparison with Alpha–beta search and MCTS.

3.1 Proof Number Search Overview

Allis proposed a search algorithm named Proof Number Search (PNS) [8, 6, 7], which is based on the idea of *proof numbers*. Allis applied PNS to many games and proved that, in Gomoku, the first player can win through optimal play.

Seo's PN* was used for a tsume-shogi solver[72] and showed good performance. Nagai further enhanced the performance in Depth First Proof Number (Df-pn search), which uses both proof and disproof numbers [51, 52, 55, 54, 56, 53]. Kishimoto's Df-pn(r) adds a solution of GHI problem[59, 15, 12] to df-pn. The df-pn based algorithms are currently used as the standard algorithm for end games of many games and solvers for local problems. For example, the best algorithms for tsume-shogi solvers and tsumego solvers are both based on df-pn.

One remaining challenge for AND/OR tree search is open border tsumego. It is still necessary to modify original problems into enclosed problems before solving them using programs because the search space of the game of Go is unsuitable for (dis-)proof number based algorithms. Proof number based search is effective if the assumption of "promising moves have smaller (dis-)proof numbers" holds. However, in the game of Go, the branching factor is large and roughly uniform, giving little information about the goodness of moves. Solving open border tsumego is a remaining challenge for searching.

3.2 Proof-Number Search Variants

Two-player zero-sum perfect information games can be naturally expressed as AND/OR trees if a *proof* (= true) is regarded as a win for the first player and a *disproof* (= false) is regarded as a win for the opponent. In that case, OR nodes stand for the first player's turn, and AND nodes stand for the second player's turn.

Allis introduced proof and disproof numbers as estimates of the difficulty in finding proofs and disproofs in a partially expanded AND/OR tree[8, 6, 7]. The proof number of node n, $pn(n)$, is defined as the minimum number of leaf nodes that must be proved to find a proof for n, whereas the disproof number $dn(n)$ is the minimum number of leaf nodes to disprove for a disproof for n. In fact, $pn(n) = 0$ and $dn(n) = \infty$ for a proved terminal node n, and $pn(n) = \infty$ and $dn(n) = 0$ for a disproved terminal node. In addition, $pn(n) = dn(n) = 1$ 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:

$$pn(n) = \min_{i=1,\ddots,k} pn(n_i), \qquad dn(n) = \sum_{i=1}^{k} dn(n_i). \tag{3.1}$$

For an AND node n proof and disproof numbers are:

$$pn(n) = \sum_{i=1}^{k} pn(n_i), \qquad dn(n) = \min_{i=1,\ddots,k} dn(n_i). \tag{3.2}$$

Fig. 3.1 shows the calculation.

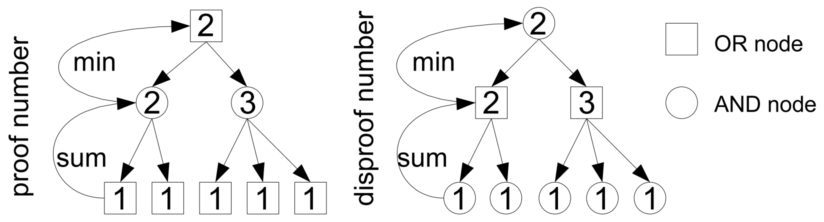


Fig. 3.1 Calculation of proof/disproof numbers

3.2.1 PNS

Proof-number search (PNS) [6] is a best-first search algorithm that maintains proof and disproof numbers for each node. Actually, PNS finds a leaf node from the root by selecting a child with smallest proof number at each OR node and one with smallest 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. Allis proved that gomoku ends with the first player's win if the game is played optimally.

3.2.2 C* Algorithm

Seo developed a C* algorithm and applied it to tsume-shogi problems. This algorithm is based on the conspiracy number.

The conspiracy number is the number of nodes which need to change their values to change the value of the root node.

3.2.3 df-pn

Depth-first proof-number (df-pn) search [53] is a depth-first reformulation of PNS that re-expands fewer interior nodes and capable of running in a small memory space.

To make a best-first search into a depth-first search, df-pn uses two thresholds, one for proof numbers and one for disproof numbers. Thresholds for proof and disproof numbers are gradually incremented and used to limit a depth-first search. Such techniques are well known, e.g. in IDA* [43] and Recursive Best-First Search [44]. The thresholds are adjusted so that the most proving node will be in the searched subtree.

The mechanism of df-pn is explained precisely in Appendix A.

df-pn+

Unlike alpha–beta search, df-pn has no evaluation function or move ordering.

The search order in df-pn can be adjusted by assigning values other than 1 to the initial value of (dis-)proof numbers. The original definition of (dis-)proof number means the lower bounds of the number of nodes needed to be (dis-)proved. Df-pn+ is an enhancement of df-pn that assign higher values to less promising nodes.

Nagai added shogi-specific knowledge to df-pn and solved almost all published tsume shogi problems. Many of today's best shogi programs include a checkmate search engine based on df-pn+.

df-pn(r)

The PN search works ideally for trees. However, if the target is a DAG or a cyclic graph, double counting of (dis-)proof numbers becomes a problem. Kishimoto provided a solution for double counting caused by cycles.

Search algorithms based on transposition table are known to cause Graph History Interaction (GHI) problem [59, 15, 12]. Kishimoto gave an efficient solution for GHI problems [37, 34]. Kishimoto implemented this solution to df-pn.

Df-pn with these enhancements are called df-pn(r). In fact, df-pn(r) is currently used for the best solver for the tsumego problem.

3.2.4 Branch Number Search (BNS)

Branch Number Search (BNS)[58] is an algorithm resembling df-pn. The difference is that unlike PN search, BNS only considers the branching number, which is the number of unsolved branches in the path from the root node to the frontier node.

The branching number is generally less informative to proof numbers because it lacks the information of the subtrees in the branches. However, ignoring the information from subtrees of branches gives BNS the ability to solve difficult problems of certain type. For normal proof number search algorithms, it is difficult to avoid double counting of proof numbers that occur in DAGs. The proof number would be at least 2^n during the search if double counting were to occur n times. The use of BNS avoids such exponential growth in the proof numbers.

Many difficult tsume shogi problems are known to have many interflows of the paths in the answering sequences; many problems can be solved only by BNS.

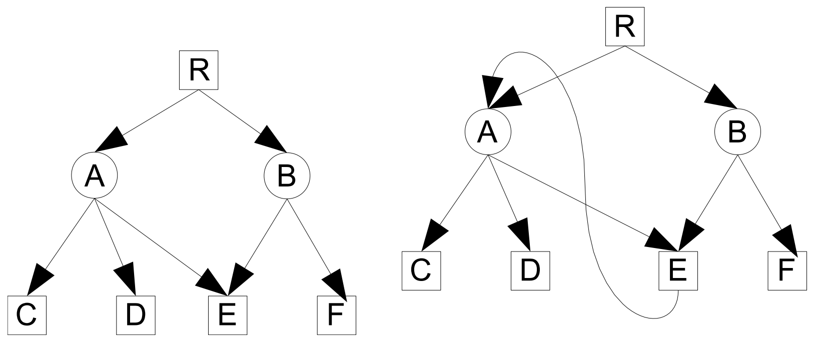


Fig. 3.2 DAG and DCG

3.2.5 Weak Proof Number Search

Ueda et al. proposed WPNS [84]. Actually, WPNS has a similar advantage to that of BNS, which has the ability to avoid double counting of proof numbers.

Instead of calculating the sum for (dis-)proof numbers, they only take the maximum value in the (dis-)proof quantities of the children, and add the number of children.

Calculation of the disproof number for OR node would be

$$dn(n) = \max_{i=1,\cdots,k} dn(n_i) + (k-1). \tag{3.3}$$

The proof number for AND node would be

$$pn(n) = \max_{i=1,\cdots,k} pn(n_i) + (k-1). \tag{3.4}$$

3.3 Proof Number Search, DAGs and Cyclic Graphs

Proof number search behaves well if the target of the search is limited to trees. However, if the targets are DAGs and cyclic graphs, a difficulty exists in the proof number calculation.

To prove the root node R in the left of Fig. 3.2, one set of nodes C, D, E, or E, F must be proved. These sets are called proof sets. For trees, the size of the smallest proof set is equal to the proof number. However, in DAGs, these differ because of double counting. In Direct Cyclic Graphs (DCGs), double counting also occurs as shown in the right of the figure. Cycles cause a more serious problem than interflows. Naive implementation of the proof number search often results in an infinite loop for cyclic graphs.

Schijf provided an analysis related to DAGs and DCGs in a paper[68], which also described practical proof number search algorithms for DAGs and DCGs. Müller developed the Proof-set search algorithm which directly uses the idea of the proof set[49]. However, practical implementation of the proof set search is difficult.

Kishimoto and Müller announced a proof of the completeness of df-pn on DAGs[35]. However, a solution for completely handling double counting caused by DAGs in a small overhead remains unknown. Nevertheless, it is possible to detect and handle cycles with small overhead. However, the proof for the completeness of df-pn for graphs with cycles is not yet given.

3.4 AND/OR Tree Search Examples: Application in Games

GoTools

GoTools[88, 90] was developed by Thomas Wolf in the early 1990s. It has been known as the best tsumego solver for more than a decade. It is a combination of Alpha–beta search and elaborated heuristics.

However, neither GoTools nor df-pn variants can solve open-border tsumego problems [89]. Problems must be modified to enclosed problems before being solved using existing solvers. The required size of the enclosed area is becoming larger, but it remains much smaller than a full Go board.

Tsume shogi solver

Shogi (Japanese chess) has the most complex end game among all chess variants because of the dropping rule. Checkmate problems of shogi are called tsume shogi. The first great success of df-pn was reported for tsume shogi [51, 53]. Nagai's solver solved most known tsume shogi problems and contributed in improving the strength of shogi playing programs.

Checkers

In 2007, it was proved that checkers will end in a draw through optimal play [67]. This result was obtained through the combined use of search, endgame database, and prover for confirming the result. In fact, df-pn(r) was used as the backend prover part of the proof [66, 65].

Lines of Action

Lines of Action is a board game played using black and white pieces. For the endgame of Lines of Action, Winnands used the PDS-PN algorithm [86, 87]. This is a combination of best first PN search and depth first Proof number and Disproof number Search.

3.5 Comparison with Alpha–beta search and MCTS

3.5.1 Searching AND/OR tree by Alpha–Beta search

An AND/OR tree search can be produced using Alpha–beta search by assigning numerical values to true, false, and unknown. Terminal nodes with a true value might have 1 and nodes with false might have -1, whereas internal nodes might have 0 if the logical value is not determined yet. If we limit our target to games, then one player's win would have the value of 1, a loss would have -1, and a draw would have 0. Consequently, one natural question would be, ”Can alpha–beta catch up with df-pn or not?”

The answer is yes, but it depends on the domain. For example, for tsume shogi, the performance of df-pn based solvers is far superior to that of solvers based on the Alpha–beta search. On the other hand, for closed boundary tsumego, the best Alpha–beta based solver GoTools [91] still has performance that is not so far behind the best solver that is based on df-pn by Kishimoto [34].

Therefore, the question is, "What is the difference between these two problems?" As explained in chapter 2, Alpha–beta search needs three functions to achieve high performance. The functions are good and fast evaluation, good move ordering, and depth threshold control.

It is easy to prepare an evaluation function based on the idea of (dis-)proof numbers. Preparing move ordering based on the evaluation function is also possible. Consequently, the difference of the difficulty between these two problems results from the third function: depth threshold control.

Alpha–beta search requires a scheme for deciding the depth threshold. Ideally, the given depth threshold should just cover the smallest proof tree (Fig. 3.3). A *proof tree* is a subtree of the AND/OR tree that covers a sufficient part of the tree to prove the logical value of the root node. This is possible if proof trees have a predictable shape.

For closed boundary tsumego, because of the nature of the rules of Go, the search space has somewhat uniform depth. For that reason, the shape is predictable. In contrast, for tsume shogi problems, the proof trees are highly unbalanced and the shape is not predictable. The most extreme example is a tsume shogi problem with an answering sequence that is 1,525 plies long [30].

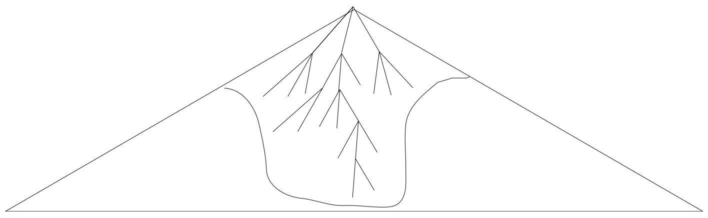


Fig. 3.3 Ideal depth threshold that covers a proof tree.

A survey paper by Grimbergen describes proof number search variants [29]. The paper used the phrase "variable-depth search", which is a good phrase to describe the advantage of proof number search.

3.5.2 Comparison with Monte Carlo Tree Search

Monte Carlo tree search could also be used for AND/OR tree search. However, the outcome can never be proved to be correct because even if terminal node can be reached, only a probabilistic or statistical answer can be given. In addition, as explained in chapter 2, MCTS is weak for long narrow sequences.

In fact, MCTS would be useful for problems for which a player chooses his first move correctly: then many ways to win can be found. After wrong moves, fewer ways to win present themselves. For such problems, it is possible to prepare AND/OR tree search problems for which MCTS works better than proof number search. However, in general, proof number search is better for solving AND/OR tree search problems.

The combination of proof number search and MCTS would be an ideal solution because the weak point of MCTS can be covered by proof number search.

Chapter 4
Lambda Df-pn Search

We describe the first approach for AND/OR tree search, which uses the hidden hierarchy in the search space of AND/OR trees.

In the framework of most existing search algorithms, the evaluation function extracts information from target search spaces, which is used to control the direction of the search. However, the evaluation function evaluates positions only statically. In contrast, our approach is to uncover a hidden hierarchy that could only be found as a result of search.

We implemented the λ df-pn algorithm, which uses λ order in Depth-First Proof Number search. In fact, λ order is an idea proposed by Thomsen, and used in his λ search algorithm. The λ df-pn algorithm can be explained as a combination of two algorithms: proof number search and threat-based search. Both are based on a domain-independent technique and have different advantages.

4.1 Overview

Search algorithms for AND/OR tree search based on proof numbers can be explained as simplest-first search. As explained in chapter 3, this approach is more domain-independent than Alpha–beta based approach which works well for many domains with less domain-specific knowledge of the target problems.

The PN search works based on the expectation that a small proof number means that it is easy to prove. It works well for problems for which the assumption "small proof number

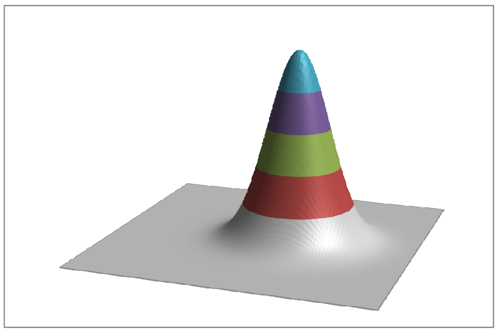


Fig. 4.1 Hierarchy in search space.

means promising" holds. However, if the branching factor is uniform or if the assumption does not hold, the PN search is slow.

Existing approaches for these types of problem rely on additional knowledge such as evaluation functions for Alpha–beta search, and heuristic proof number initialization [55, 53]. Most rely on domain-specific knowledge. However, because an advantage of the proof number search is its capability of searching without additional knowledge, our aim was to enhance the proof number search using more generic techniques.

Research in simplest first domain-independent tree search techniques produced another approach. The second family of algorithms uses null (pass) moves [25] for finding threats to achieve a goal [83, 18, 19]. Search then proceeds in a statically determined fashion from simpler to more complex threats.

Proof number search and threat-based search can both 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 a long series of simple threats is inferior to one with a small number of more complex threats. For proof-number based algorithms, overestimation of proof and disproof numbers delays the search of nodes that appear to be unpromising but which can be solved easily.

This chapter presents a study of a combined approach, which uses 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. This combination can extract information out of search spaces with uniform branching factor. The main contribution is the λ depth-first proof-number (λ df-pn) search algorithm, which synthesizes depth-first proof-number search [53] and λ search [83].

Results show that λ df-pn outperforms df-pn even if df-pn is combined with little domain specific knowledge and λ df-pn does not. Both algorithms are combined with state-of-the art enhancements for a fair comparison. We also showed that λ df-pn outperformed another search algorithm that we named classical λ df-pn search. Experimental results for capturing problems in the game of Go are provided. Go problems have roughly uniform branching factors and are therefore difficult for PN search.

The results of this chapter were first published in [95]. Although this thesis specifically addresses the problems of Go, this algorithm was proven to be effective for two other games—shogi and Shimpei[75]— in a paper written in collaboration with Soeda et al.

4.2 Threats, Threat-based Search, and λ Search

The idea of Lambda search is based on the order of *threats*. This idea has a close relation with the rule of the game of Go, but is also a generic idea that does not depend on the knowledge of the search domains.

Allis proposed threat-based search and solved gomoku in 1996[7]. The threat-based approach was generalized to λ search by Thomsen [83] and further by Cazenave [18].

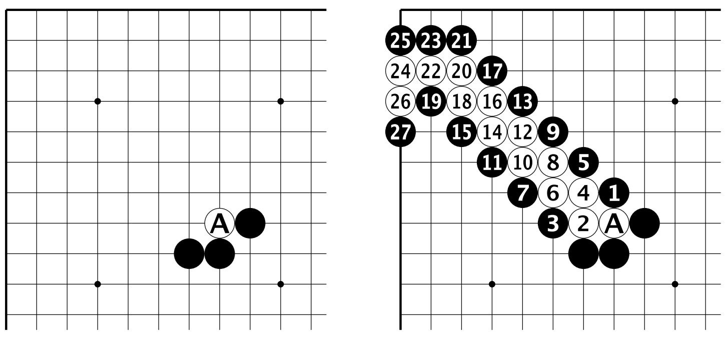


Fig. 4.2 Example of a ladder. The attacker can win by an order 1 threat, which is denoted in λ notation as $\lambda^1 = 1$.

4.2.1 λ Search Definitions

Threats can be found using null moves. Intuitively, a threat is a move that, if ignored, engenders a quick win. In the ladder example in Fig. 4.2, each black move is a threat to capture, and white has only one possible move to avert that threat. The initial position is on the left, and the winning sequence for black is shown on the right.

The player who tries to achieve a goal is called an *attacker*, whereas the other player is called a *defender*. The order of threats shows how quickly the attacker can win if the move was ignored by the defender.

Threats are useful for a 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. It is worth investigating it with high priority if a player has a threat to win. Threats severely restrict the defender's choice of replies to moves that can avert the threat, and often engender further follow-up threats.

While this chapter concentrates on a comparison with λ search, the ideas should apply to threat-based methods in general.

Table. 4.1 Examples of threats

order	gomoku	checkmate search	capturing of Go
0	make 5	capturing King	capturing move
1	make 4	check	atari

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. The λ search creates λ^n*-trees* consisting of λ^n*-moves*, which are defined recursively 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.

The capture problem of Go is a good example for explaining λ search. We will explain λ search using Go examples in which the attacker tries to capture the defender's block. In our examples, to ease understanding, the attacker is always black and the defender is always white.

It is denoted as $\lambda^n = 1$ if the attacker is able to win by a sequence of threats of order n or lower. Actually, $\lambda^n = 0$ indicates that the attacker cannot achieve the goal by threats of order n or lower. A λ^n-tree is a search tree that consists of λ^n-moves. That node is terminal and a loss for the player to move if no λ^n-move exists for a node. The definition of λ^n-moves is the following.

Definition 1 [83] A λ_a^n-move is an attacker's move such that if the defender passes in reply, there exists a λ^i-tree with $\lambda^i = 1$ $(0 \le i \le n-1)$. The attacker threatens to win within λ order lower than n.

Definition 2 [83] A λ_d^n-move is a defender's move such that after the move, no subsequent λ^i-tree exists with $\lambda^i = 1$ $(0 \le i \le n-1)$. The defender's move averts all lower order attacker's threats.

In capturing problems of Go, a λ^0-move occupies the last liberty of the target block and captures it. An attacker's λ_a^1 move is a move that threatens a defender's block to get captured, which is called an *atari* in Go terminology. A defender's λ_d^1 move is a move by which, after the move, no λ^0 move for the attacker exists. In other words, λ^1 tree consists of attacker's *ataris* and defender's evasion from *ataris*. This sequence results in a sequence called *ladder* as portrayed previously in Fig. 4.2.

By definition, if an attacker plays a λ_a^n moves and the defender ignores the move, then there must be a way for the attacker to win in λ order less than n. Therefore, in capturing problems of Go, if black plays an *atari* move (λ_a^1 move) and the defender ignores it, a λ^0 move (attacker's winning move) exists. If the attacker plays a more indirect attack, which, if ignored by the defender, allows the attacker to win by a ladder (which means $\lambda^1 = 1$), then the indirect attack is a λ_a^2 move.

Furthermore, by definition, a defender's λ_d^n move is a move that prevents the attacker's win within λ order less than n. Evasion from *atari* is a λ_d^1 move because it prevents the λ^0 move. Prevention from being captured using a ladder is a λ_d^2 move because it prevents λ_a^1 move.

4.2.2 λ Search Example

It is easy to understand how λ^1 search works. In the capturing problem of Go, the attacker only tries *atari* moves. In chess, meaning that the attacker only tries checks to search for a checkmate.

To elucidate how λ search works for order higher than λ^2, we explain use of a Go example.

Fig. 4.3 portrays a basic capturing technique in Go, which is called a *net*. Directly attacking the white stone by *ataris* does not work as depicted in the lower left figure (The white stone on the upper left corner of the board is called a *ladder breaker* in such cases.). However, an indirect attack portrayed in the lower right board captures the white stone. This is an indirect attack with λ order 2. We will explain how λ search finds out this move.

Fig. 4.4 portrays how λ search prunes moves according to the order of the threat. This is an example of λ^n search in which the attacker searches for a win in order n or lower. The pruning is done by searching the result of the defender's pass. In the figure, from the root node marked λ_a^n, an attacker tries four moves. For each move, the defender passes and the

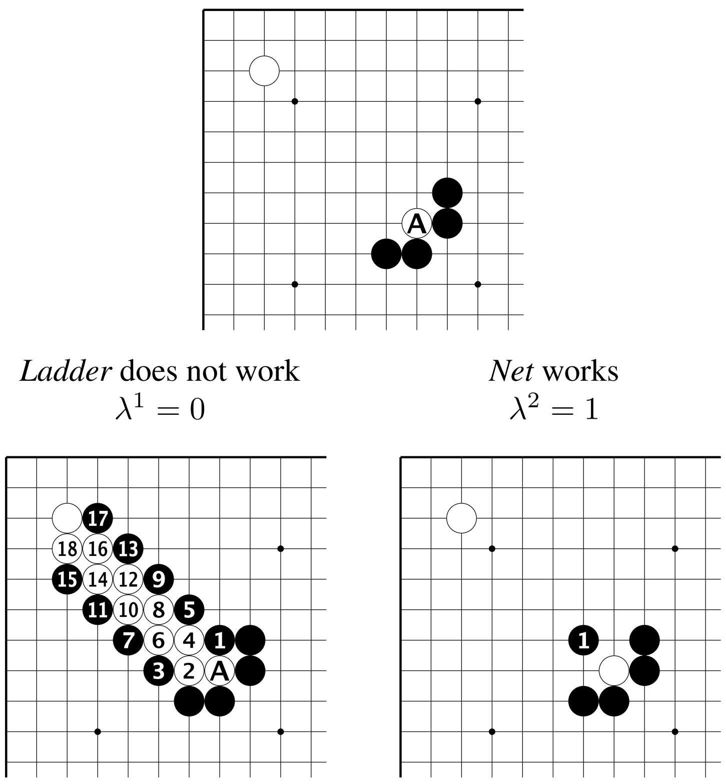


Fig. 4.3 To capture a white stone marked with "A", *ladder* does not work ($\lambda^1 = 0$), and *net* works ($\lambda^2 = 1$).

attacker tries to win by order λ^{n-1}. The node is a part of λ^n tree if the attacker can win by order $n-1$ or lower (which is denoted as $\lambda_a^{n-1} = 1$). If $\lambda_a^{n-1} = 0$, then the node is pruned because the attacker's move from the root node is not an order n threat, by definition 1.

Fig. 4.5 portrays a defender's node. For each child node, λ^{n-1} search is done first. The child node is pruned if the result is a win for the attacker in order $n-1$.

Fig. 4.6 depicts an attacker's λ^2 example with Go positions. For each attacker's move, λ^1 search is done, and according to the results, move B, C, D, and F are λ^2 moves because if white passes $\lambda_a^1 = 1$. Move E is pruned because it is still $\lambda_a^1 = 0$ after a pass of white.

Fig. 4.7 presents a defender's move after the attacker played move C from the previous figure. All the defender's moves other than moves marked with triangles are insufficient to prevent the attacker from winning in the λ^1 search. Therefore, all other moves are pruned.

4.2.3 Correctness and Efficiency

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 smaller than λ^n trees in general. Therefore λ search is expected to prune moves according to the λ order with little effort.

A λ search is especially efficient for problems for which the λ order is meaningful. Capturing problems in Go presents an ideal case because lower bounds on the λ order can be derived from game-specific knowledge, the number of liberties of a block [83].

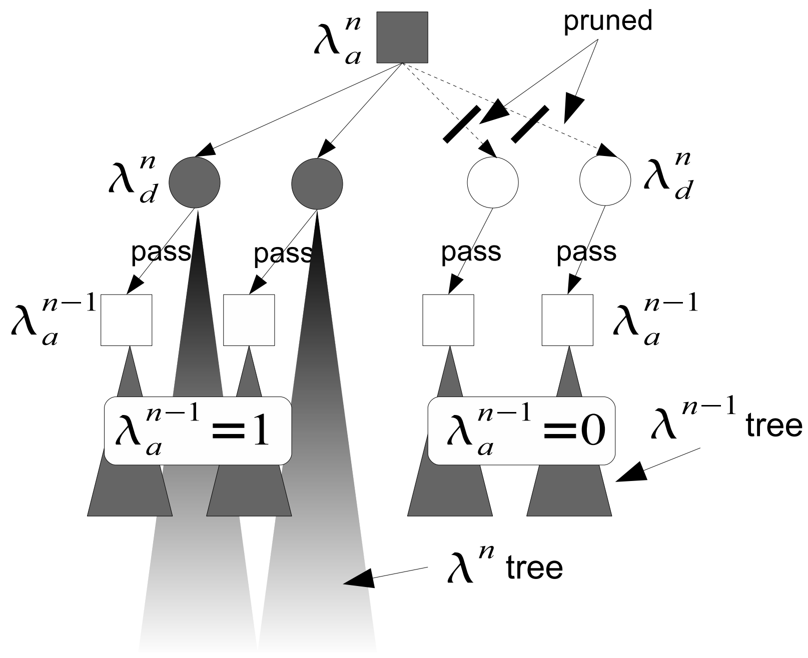


Fig. 4.4 Lambda Search example at an attacker's node. Only the nodes marked in gray are λ^n nodes.

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

$$\text{if} \quad \lambda^n = 1 \quad \text{then} \quad \lambda^i = 1 \quad (n \leq i) \tag{4.1}$$

$$\text{if} \quad \lambda^n = 0 \quad \text{then} \quad \lambda^j = 0 \quad (0 \leq j \leq n) \tag{4.2}$$

A positive result of a λ search, $\lambda^n = 1$, is correct provided that a pass is allowed or *zugzwang* is not a motive in a game. A zugzwang is a position in chess in which a pass is the only way to win. Because λ search relies on defender's pass to switch to lower order λ trees, zugzwang would lead to incorrect results. The probability of zugzwang and its effect on the correctness is not analyzed fully for Go. Through our analysis of a simple board game called Shimpei, no wrong result was caused by zugzwang [75]. Shimpei is a small board game that is sold by Bandai Co. Ltd., and is already solved completely by Tanaka [80]. It is known that Shimpei have zugzwangs in the game tree. However, analyses for more complex games have not been done.

A negative search result, $\lambda^n = 0$, means that either no solution exists, or that a solution will be found at a higher order $n' > n$. If the highest probable λ order is known and the limit of λ order at the root node is set to the order, then the result of λ is proved to be identical to the search result of a minimax search. However, if such a bound is not known, it is difficult to determine how to limit the λ order at the root node.

4.2.4 Problem of λ Search

The original λ search has a problem that degrades the overall speed performance. To prune the tree, the λ search must prove or disprove the lower order subtree. However, it is impossible to predict the time needed for proof or disproof; especially, disproof often takes a long time.

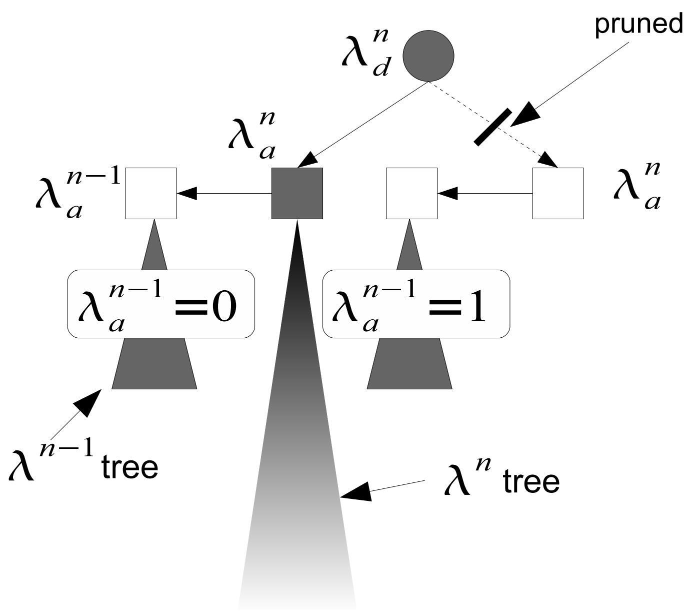


Fig. 4.5 Lambda Search example at defender's node.

Therefore, if some of the lower order subtrees requires a long time for disproof (and that is likely to happen) λ search takes a long time.

Thomsen provided only the performance of the pruning rate in his paper. He did not describe the precise speed performance [82]. In addition, the test problems publicly available from Thomsen's homepage [81] consist only of order 2 or 3 problems. It is presumed that the original λ search was only able to solve up to λ^3 problems within a reasonable time.

4.3 λ df-pn Search Algorithm

4.3.1 Details of λ df-pn

λ df-pn is a proof number based λ search. Proof/disproof numbers for AND/OR nodes are initialized and propagated as in df-pn. 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 Fig. 4.8. 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 (4.1), the attacker must prove only one of the pseudo-nodes. Then the proof number of n would be the minimum of the proof numbers of the pseudo-nodes. However, disproving any lower order λ tree does not help disproving the λ^l tree as in eq. (4.2). Therefore, the disproof number of the attacker's node n is the same as for c_l.

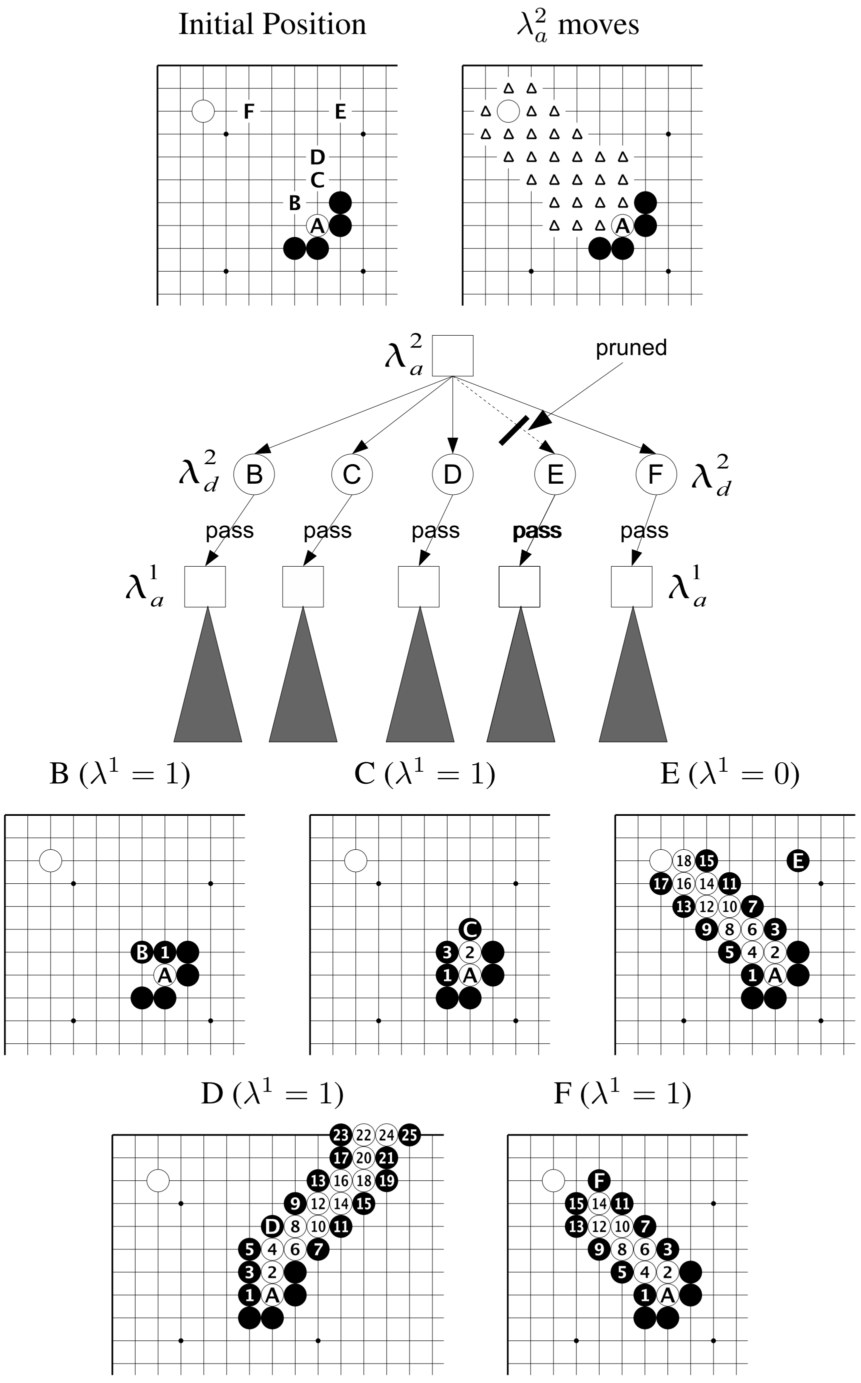


Fig. 4.6 Lambda Search example for Attacker's node with Go example

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

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

A defender node n is shown at the bottom of Fig. 4.8. The defender's aim is to disprove this node by showing $\lambda^l = 0$. Using eq. (4.2), proofs of lower order λ subtrees are irrelevant

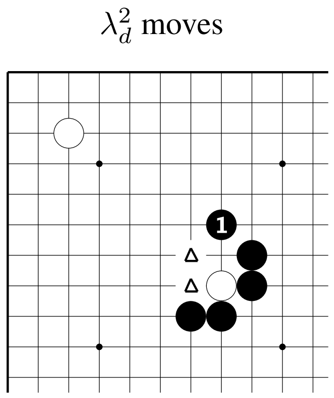


Fig. 4.7 Lambda Search example for the Defender's node with the Go example

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. As shown 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 perspective, the purpose for searching this pseudo-node c_{l-1} is to check the attacker's previous move m, which leads to node n, was a λ_a^l move or not. The search can immediately return to the parent of n if $\lambda^{l-1} = 0$ is proved. Because 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 as

$$pn(n) = pn(c_l), \qquad dn(n) = \min(dn(c_{l-1}), dn(c_l)). \tag{4.3}$$

The definition of $pn(n)$ is based on the fact that a lower order pass does not affect the proof, and called maximum order strategy [75]. However, during the search for order l sub tree at AND nodes, subtrees for $l-1$ passes are searched. Therefore there can be a different definition of the proof number, which is called a summation strategy. $pn(n) = pn(c_l) + pn(c_{l-1})$. Soeda used this definition for his shogi problem solver [74, 73, 75]. This definition is particularly addressing the search effort whereas the definition in eq. 4.3 is focusing on the number of nodes needed for proof. Preliminary experiments showed that maximum order strategy was better for Go problems, but the result might differ for other domains.

As does df-pn, λ df-pn uses two thresholds for proof and disproof numbers. Here, λ df-pn selects the leaf node with the smallest proof/disproof number, irrespective of the λ order. The main advantage of λ df-pn over λ search is the ability of λ df-pn to switch between child nodes seamlessly with different λ orders. The search no longer must wait for a proof or disproof of all lower λ order subtrees. Because disproofs are often more difficult than proofs, the ability to skip difficult disproofs of λ^{n-1} subtrees and starting search of a λ^n subtree improves the search behavior.

Annotated pseudo-code of λ df-pn is presented in appendix A.

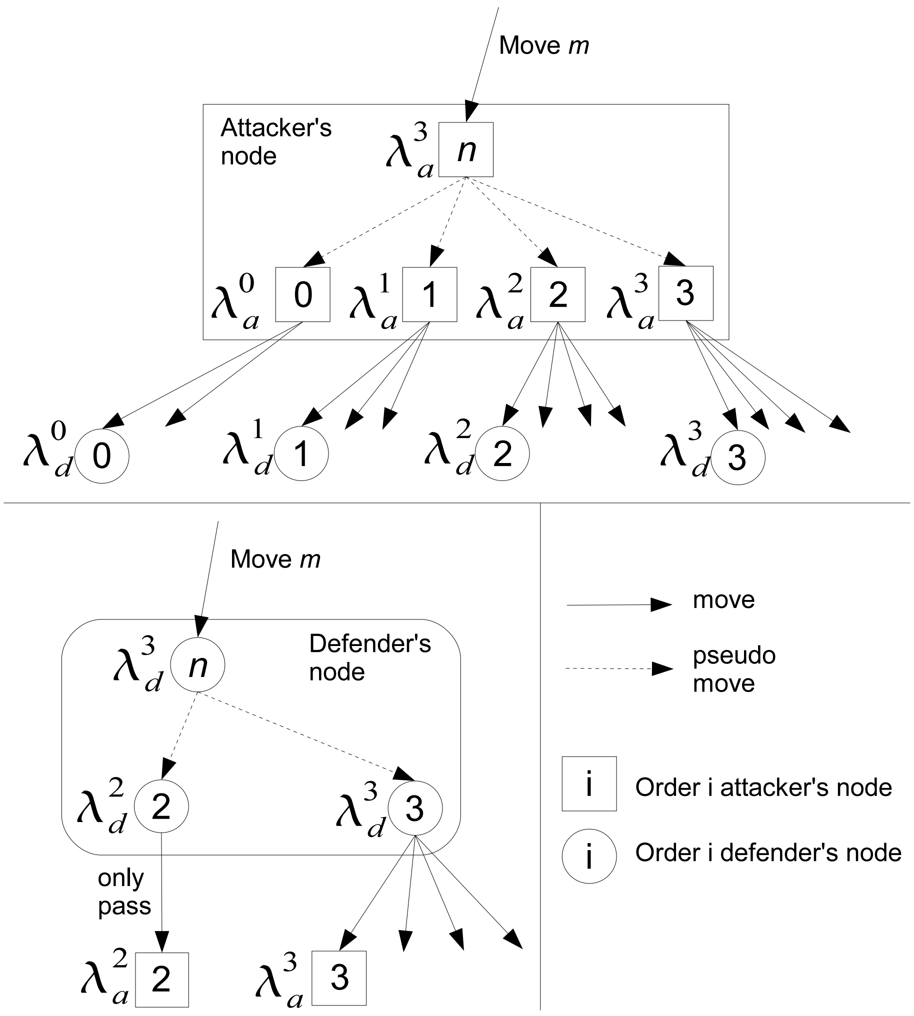


Fig. 4.8 Attacker's node and Defender's node

4.3.2 Search Enhancements

Heuristic Initialization of Proof and Disproof Numbers

The basic df-pn algorithm initializes proof and disproof numbers of an unproven leaf node to 1. As in [8, 53], one way to enhance the performance of λ df-pn is to initialize these values heuristically. Let $H_{pn}(n)$ and $H_{dn}(n)$ be the evaluation functions to initialize the proof and disproof number of leaf n. In λ df-pn, $H_{pn}(n)$ and $H_{dn}(n)$ are defined as functions of the λ order of n, to control the search direction so that a node with lower λ order would be searched with a higher priority (see section 4.6.1 for details). Df-pn with this enhancement is called df-pn+ [55, 53].

Simulation

Kawano's *simulation* [33] saves search time by reusing the proof tree of similar board positions. A similar board position B is likely to be (dis-)proved by the same (dis-)proof tree if a board position A was (dis-)proved. Simulation is a technique to reuse the proof tree for other nodes which have similar board positions. Soeda proposed this method as *Proof Tree Tracing* because the term *simulation* is often confusing.

The λ df-pn uses *simulation* only for pass moves at AND nodes. A pass move is searched at AND nodes; if it yields a loss, then its sibling nodes are checked using the disproof tree of the pass move. In this way, irrelevant moves that do not help the defender can be disproved quickly.

4.4 Implementation Techniques

This section presents an explanation of other techniques used in our implementation. These techniques seem trivial one by one, but without these techniques, λ df-pn would not achieve high performance.

4.4.1 Candidate Move Recording

While searching through the graphs for real game play, the edges of the graphs are not prepared as data structures, meaning that we must generate possible moves of the game when we visit a new node. Move generation is a time consuming part of a search.

Especially when the search algorithm revisits the same node many times, recording the moves in a hash table will save execution time. However, few programs do so because of the great memory consumption.

In λ df-pn, we decided to record the moves in the hash table. The first reason is that df-pn frequently re-expands previously visited nodes. The second, more important reason is that higher order λ moves can be known only from the result of lower order λ search, and would cause a loss of time if we were to discard the information of high-order λ moves.

In our implementation, we recorded λ moves using bitboard structures. Therefore, we needed one bitboard for one λ order. Our data structure used 128 bits for a 9×9 board. The size of one hash table entry was 160 to 180 bytes for a 9×9 board when the maximum λ order was set to five. This size is approximately 10 times larger than the memory size required for normal df-pn.

It is possible to reduce the size by recording only higher order λ moves or by discarding unpromising moves, but such techniques are not discussed further in this thesis.

4.4.2 Reducing Number of Re-expansions

It is a mandatory for df-pn to record the nodes into a hash table because it must revisit nodes. Nagai described a technique to reduce the number of re-expansions in his thesis [53]. He added the average of the (dis-)proof numbers to the thresholds, which delays returning from the subtrees. The average of (dis-)proof numbers was thought to be a unit of the search difficulty at the moment.

Pawlewicz and Lew proposed another technique called 1+ε trick [60]. This technique simply multiplies 1+ε to the thresholds for (dis-)proof numbers. Their experiments were done for df-pn and PDS, and the target problems were Go and Lines of Action. The actual value of ε was set to $1/4$ for df-pn and $1/16$ for PDS.

We did not implement these techniques in our implementations, but combinations of these techniques with other enhancements should be studied further.

4.5 Related Work

Cazenave proposed several approaches to adjust the behavior of λ search [17, 18, 19]. The problem of original λ search was that it stays in a subtree until it is proved or disproved. These works provides a systematic means of returning from subtrees before proof or disproof is given.

Our approach can be explained as using different search algorithms and switching among them, using (dis-)proof as the measure. In that sense, the approach by Winands et al. [87] has a similar idea. They alternatively used PDS (depth-first search) and PN search (best-first search).

It is natural for shogi program researchers to use checkmate search as a tool for searching indirect checkmate. Nagai used his checkmate solver to search for order 2 checkmates in shogi endgame problems. Soeda extended the idea to shogi endgame problems in which both kings are involved [76]. They solved difficult shogi endgame problems[74]. Soeda's idea inspired a similar algorithm to our λ df-pn, which is named df-pn driven λ search[73]. Later we published a paper in collaboration with Soeda [75].

In fact, λ df-pn can also be thought of as an extension of meta search. Meta-search is often used as a generic noun meaning a search algorithm that relies on other search algorithms, such as a web search engine based on other (2 or more) search engines. The λ search itself is a combination of search algorithms with different orders. In this chapter, we propose the use of a fast search algorithm for low-order λ search to improve the performance. In meta-search, the mode of adjusting the time in search algorithms will be the problem. However, in our approach, the efforts put into search algorithms are adjusted automatically based on the framework of the proof number search.

4.6 Experimental Results

4.6.1 Setup of Experiments

We used the same two test sets explained in chapter 5.4, Master of Semeai and Thomsen's test sets. The following three algorithms were tested:

- λ df-pn: The λ df-pn algorithm described in section 4.3 with the highest λ order set to 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.

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

- df-pn+: The df-pn algorithm with heuristic initialization. H_{pn} and H_{dn} use Go-specific knowledge, the distance to a target block. To allow for a fair comparison, state-of-the art enhancements are incorporated into the implementation of the df-pn+ algorithm, such as techniques described in [36, 39] and Kawano's simulation [33].

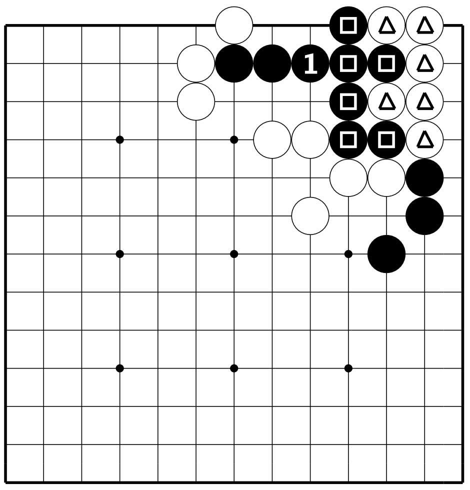


Fig. 4.9 Sample test problem (Black to play)

- Classical λ df-pn search: Thomsen's original λ search uses an Alpha–beta framework. For better comparison with λ df-pn, we developed classical λ df-pn search based on df-pn, which expands trees strictly in increasing order of λ. Which means λ^n search will not start before λ^{n-1} search is finished.

Because the capturing problems in our test sets are open boundary problems, it is difficult to restrict move generation and achieve provably correct results. In practice, most computer Go programs heuristically limit moves for their capture search engines and assume a block with a sufficiently large number of liberties is completely safe. Move generators of three types with different level of forward pruning were tested: not only to generate all legal moves, but also to assess the life and death status accurately.

- Strong Forward Pruning: generates moves on the liberties of the *surround blocks* [83], and the blocks near the target block. It also generates some more moves including the second liberties of the target block.
- Weak Forward Pruning: generates moves by the Strong Forward Pruning generator plus on all their adjacent points.
- No Forward Pruning: All legal moves are generated.

The parameters given to λ df-pn are the target *blocks* to capture/defend, and the maximum λ order. In fact, λ df-pn has the ability to give up capturing naturally if no way exists for the attacker to win within the given maximum λ order.

However, for df-pn+, a liberty threshold used to assume the block as safely escaped must be passed as an additional parameter. Here, λ df-pn with the maximum λ order of l evaluates a block with $l+2$ or more liberties as escaped. The df-pn+ with the liberty threshold of $l+2$ is roughly comparable to a λ df-pn search with maximum order l. We used 2–5 for the preset maximum λ order for λ df-pn. Correspondingly, we tested df-pn+ with the liberty thresholds of 4–7. We also tested df-pn+ with the liberty threshold set to ∞, which means that df-pn+ will never give up capturing.

A test case was considered solved if the move returned by the algorithm was identical to the prepared solution in the test suite.

4.6.2 λ df-pn versus df-pn+

We define the true λ order of problems as the smallest λ order which can solve the problem. To achieve the highest performance, the λ order at the root node should be set equal to the true lambda order of the problem. We solved the problems with classical λ df-pn search and found the minimum λ order which could solve each problem to find out the true λ orders of the problems. The difficulty of problems greatly depends on the true λ order. Therefore, we classified our test problems according to the true λ order.

Tables 4.2, 4.3, and 4.4 show the number of problems solved using λ df-pn and df-pn+ with three move generators.

Results show that λ df-pn solved more problems than df-pn+ in each setting. The difference is greatest for the no forward pruning move generator. This is not surprising because threats based on the λ orders can drive λ df-pn to examine, specifically, those moves near the target block of the defender. Two causes for misses (wrong answers) exist.

1. Wrong forward pruning (which does not occur for the "No Forward Pruning" generator)
2. Problems results in Ko (Ko handling is not discussed)

Solutions for Ko are described in [88, 34]. However, in this thesis we do not discuss Ko detection. The reason is that the Thomsen test set requires no Ko detection, and Ko detection requires a second search, which makes it difficult to compare the speed performance.

Figs. 4.10 and 4.11 present comparisons of the performance of λ df-pn and df-pn+ for each problem solved by either algorithm with the three move generators. In this figure, the execution time of λ df-pn is shown on the horizontal axis against df-pn+ on the vertical axis, both in logarithmic scales. Each point above the $y = x$ line represents a problem λ df-pn solved faster. The comparison is between order n λ df-pn which is denoted as λ^n df-pn, and normal df-pn+ with liberty threshold $n + 2$ which is denoted as df-pn^n+. Results show that λ df-pn outperforms df-pn+ for all three generators, with the largest difference for the no forward pruning move generator, underscoring the clear advantage of λ df-pn on pruning irrelevant moves.

The problems shown on the left in Fig. 4.12 are solved faster by λ df-pn. In the problem portrayed at the top left, black must sacrifice two stones to win. If stones are captured, then more empty points are 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 df-pn+, which should have been proved easily. df-pn+ delays searching such a node with an apparently large proof number. Here, λ df-pn 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 df-pn+. The bottom left problem is difficult both for computers and human beings. In this case, the ability of λ df-pn, which specifically examines lower order threats, works as expected and provides a remarkably higher speed than that of df-pn+.

One disadvantage of λ df-pn is that it occasionally must visit the same nodes in different λ orders. The advantage of λ df-pn disappears if the additional information of λ order is ineffective. In particular, df-pn+ works well for few possible moves.

Table. 4.2 Solver ability: "Strong Forward Pruning" move generator

(a) Master of Semeai

—	lib. thr.	solved	missed	lost	exceeded
df-pn+	4	50	17	70	4
df-pn+	5	66	24	31	20
df-pn+	6	69	25	17	30
df-pn+	7	65	25	9	42
df-pn+	∞	63	23	0	55
—	λ thr.	solved	missed	lost	exceeded
λ dfpn	2	35	7	99	0
λ dfpn	3	67	19	52	3
λ dfpn	4	80	20	17	24
λ dfpn	5	75	20	5	41
—	λ thr.	solved	missed	lost	exceeded
c-λ dfpn	2	37	4	100	0
c-λ dfpn	3	65	20	53	3
c-λ dfpn	4	77	23	20	21
c-λ dfpn	5	78	19	5	39

(b) Thomsen test set

—	lib. thr.	solved	missed	lost	exceeded
df-pn+	4	301	12	109	11
df-pn+	5	394	16	0	23
df-pn+	6	389	16	0	28
df-pn+	7	385	14	0	34
df-pn+	∞	375	15	0	43
—	λ thr.	solved	missed	lost	exceeded
λ dfpn	2	273	6	154	0
λ dfpn	3	402	18	0	13
λ dfpn	4	398	15	0	20
λ dfpn	5	398	14	0	21
—	λ thr.	solved	missed	lost	exceeded
c-λ dfpn	2	277	2	154	0
c-λ dfpn	3	402	19	3	9
c-λ dfpn	4	397	20	0	16
c-λ dfpn	5	398	18	0	17

The right problems in Fig. 4.12 present an example that df-pn+ solved more quickly than λ df-pn. The answer is marked with a cross in this figure. These are typical problems in which λ order has a negative effect. In these problems, both players have several suicidal moves with low λ orders. Therefore, λ df-pn tends to search to disprove all suicidal moves before trying to prove the correct move. In df-pn+, the suicidal moves result in captures of non-target *blocks*, leading to larger proof numbers for these moves. Therefore, df-pn+ delays searching these suicidal moves and expands the correct moves earlier.

Table. 4.3 Solver ability: “Weak Forward Pruning” move generator

(a) Master of Semeai

—	lib. thr.	solved	missed	lost	exceeded
df-pn+	4	51	13	66	11
df-pn+	5	67	20	26	18
df-pn+	6	64	18	9	50
df-pn+	7	60	18	5	58
df-pn+	∞	58	14	0	69
—	λ thr.	solved	missed	lost	exceeded
λ dfpn	2	34	8	99	0
λ dfpn	3	68	15	47	11
λ dfpn	4	73	18	9	41
λ dfpn	5	73	14	5	49
—	λ thr.	solved	missed	lost	exceeded
c-λ dfpn	2	36	5	100	0
c-λ dfpn	3	70	15	51	5
c-λ dfpn	4	77	14	9	41
c-λ dfpn	5	72	14	5	50

(b) Thomsen test set

—	lib. thr.	solved	missed	lost	exceeded
df-pn+	4	298	12	102	21
df-pn+	5	375	17	0	41
df-pn+	6	363	14	0	56
df-pn+	7	347	17	1	68
df-pn+	∞	341	17	0	75
—	λ thr.	solved	missed	lost	exceeded
λ dfpn	2	272	7	154	0
λ dfpn	3	406	16	0	11
λ dfpn	4	390	17	0	26
λ dfpn	5	391	18	0	24
—	λ thr.	solved	missed	lost	exceeded
c-λ dfpn	2	275	4	154	0
c-λ dfpn	3	409	14	0	10
c-λ dfpn	4	391	18	0	24
c-λ dfpn	5	386	17	0	30

4.6.3 λ df-pn Performance and Lambda Order

The advantage of λ df-pn is strongly affected by the true λ order of the target problems.

Comparison of the answering ability and search time (in number of node expansions) are presented in Tables 4.5 and 4.6. This is the result for the “Strong Forward Pruning” move generator and the node expansion limit is 2M. The comparison is between λ^n search, and

Table. 4.4 Solver ability: "No Forward Pruning" move generator

(a) Master of Semeai

—	lib. thr.	solved	missed	lost	exceeded
df-pn+	4	31	9	55	46
df-pn+	5	29	8	10	94
df-pn+	6	26	7	6	102
df-pn+	7	26	6	4	105
df-pn+	∞	24	6	0	111
—	λ thr.	solved	missed	lost	exceeded
λ dfpn	2	34	7	100	0
λ dfpn	3	37	10	14	80
λ dfpn	4	37	10	6	88
λ dfpn	5	37	10	4	90
—	λ thr.	solved	missed	lost	exceeded
c-λ dfpn	2	35	6	100	0
c-λ dfpn	3	33	10	16	82
c-λ dfpn	4	28	8	6	99
c-λ dfpn	5	28	8	5	100

(b) Thomsen test set

—	lib. thr.	solved	missed	lost	exceeded
df-pn+	4	213	1	75	144
df-pn+	5	147	2	0	284
df-pn+	6	111	2	0	320
df-pn+	7	102	2	0	329
df-pn+	∞	93	1	0	339
—	λ thr.	solved	missed	lost	exceeded
λ dfpn	2	268	3	146	0
λ dfpn	3	276	6	0	151
λ dfpn	4	270	6	0	157
λ dfpn	5	270	6	0	157
—	λ thr.	solved	missed	lost	exceeded
c-λ dfpn	2	273	2	152	6
c-λ dfpn	3	218	4	0	211
c-λ dfpn	4	207	3	0	223
c-λ dfpn	5	205	3	0	225

normal df-pn+ with liberty threshold $n+2$ because λ^n search is roughly comparable to df-pn that gives up capturing blocks if the liberties of blocks reach $n+2$.

Results show that λ df-pn has an advantage for problems with small true λ order, but df-pn+ has an advantage in high λ order problems. However, the result shows that the performance penalty for estimating higher λ order than the true λ order, is smaller than using unnecessarily high liberty threshold for df-pn+. Therefore, if true λ orders of problems are not known, then λ df-pn is more robust than normal df-pn.

Table. 4.5 Answering ability for each λ order. Actually, df-pn^n + 2+ has similar behavior to that of λ^n df-pn.

true λ order		λ^2 df-pn	λ^3 df-pn	λ^4 df-pn	λ^5 df-pn
of problems		df-pn^4+	df-pn^5+	df-pn^6+	df-pn^7+
2 (38 probs)	df-pn^n+	36	35	35	35
2 (38 probs)	λ^n df-pn	35	37	37	37
3 (35 probs)	df-pn^n+	12	24	23	21
3 (35 probs)	λ^n df-pn	—	30	29	28
4 (16 probs)	df-pn^n+	2	7	11	9
4 (16 probs)	λ^n df-pn	—	—	13	10

Table. 4.6 Total node expansions for each λ order.

true λ order		λ^2 df-pn	λ^3 df-pn	λ^4 df-pn	λ^5 df-pn
of problems		df-pn^4+	df-pn^5+	df-pn^6+	df-pn^7+
2 (38 probs)	df-pn^n+	794,389	2,562,534	3,043,912	3,060,501
2 (38 probs)	λ^n df-pn	75,070	608,462	536,656	1,256,725
3 (35 probs)	df-pn^n+	—	10,161,116	13,490,750	16,896,894
3 (35 probs)	λ^n df-pn	—	3,607,117	9,199,982	12,816,495
4 (16 probs)	df-pn^n+	—	—	5,854,416	11,832,045
4 (16 probs)	λ^n df-pn	—	—	8,283,126	11,287,398

The reason that the advantage of λ df-pn diminishes for high λ orders can be explained as follows.

- For λ^n search, overhead is incurred by dividing one normal node into n pseudo-nodes.
- The heuristic implemented only for df-pn+ is more effective for higher λ order search.

4.6.4 λ df-pn versus Classical λ df-pn Search

Tables 4.2, 4.3, and 4.4 also present comparisons of the number of problems solved by λ df-pn and classical λ df-pn search. Figs. 4.13 and 4.14 present comparisons of the speed performance. The performance is similar for "Strong Forward Pruning" and "Weak Forward Pruning" move generators, but λ df-pn has a great advantage for "No Forward Pruning" move generator in the answering ability. This comparison shows that λ df-pn achieves better performance in stability if the move candidates are more numerous, especially for λ^4 and λ^5.

4.7 Discussion

This chapter presented an investigation of an approach to combine the proof number based search and threat based search. Results of applying λ df-pn to the capturing problem are promising. In particular, λ df-pn outperforms df-pn+ by a large margin with more robustness than the classical λ df-pn search if the move generator generates a larger set of moves.

Because a larger set of moves can improve the accuracy of solving the capturing problem, λ df-pn can be a good choice for solving tactical problems in Go, by virtue of its ability to extract non-uniformities from a search space with seemingly uniform branching factors.

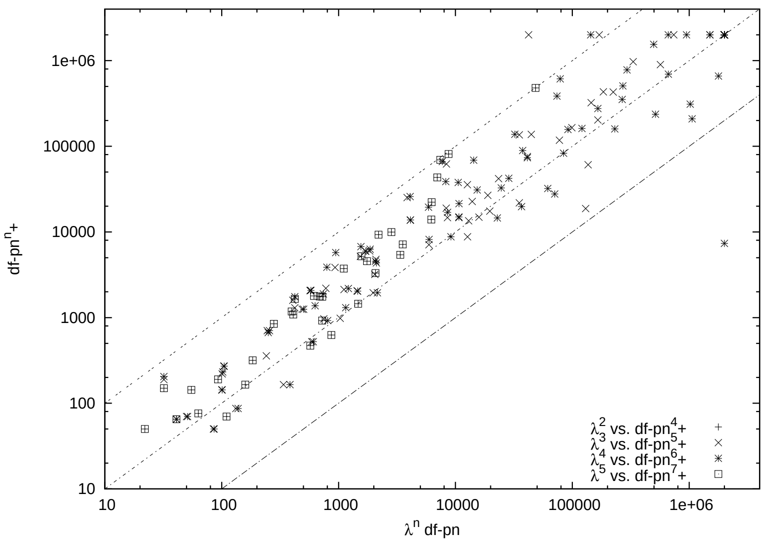
The difficulty of problems can be estimated by the true λ order of the problems. We first implemented classical λ df-pn only for the comparison, but it has a purpose in finding the true λ order of the problems. Classical λ df-pn search finds the true λ order in a more efficient way than the original λ search.

Many topics can be pursued as future work. First, methods should be explored for use with other domains, such as Hex or endgame of Shogi without λ^1 checkmate. The H_{pn} and H_{dn} of λ df-pn can clearly include domain- dependent knowledge to enhance performance, as in df-pn+. The right balance of domain-dependent knowledge and λ order in H_{pn} and H_{dn} must be investigated. Additionally, because λ df-pn can compute the λ-order of a goal, it is applicable 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.

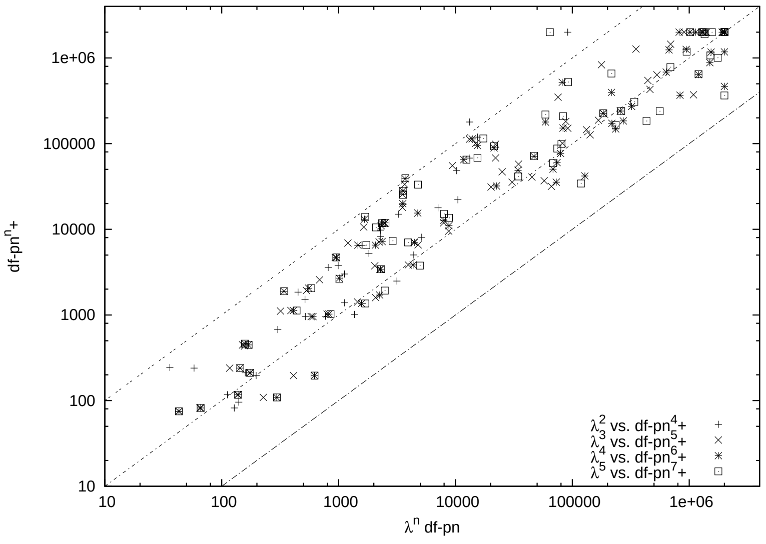
The capture problem is a good target for λ search because it is easy to detect capturing of blocks. Using λ df-pn for tsumego problem in a straightforward manner seems to present difficulty in determining the λ order because the defender's blocks require many attacker's moves to get captured even after it is clearly killed. However, if we use sufficiently fast subroutines for detecting death/life instead of the ladder search we used for λ^1 search, we can apply λ df-pn for tsumego. Probably, λ df-pn can be used for problems for which it is difficult to determine terminal positions.

Finally, integrating λ df-pn with a complete Go-playing program will be an important topic to improve the strength of the programs. We propose a method for this purpose in chapter 7.

(a)
“Strong Forward Pruning”
move generator



(b)
“Weak Forward Pruning”
move generator



(c)
“No Forward Pruning”
move generator

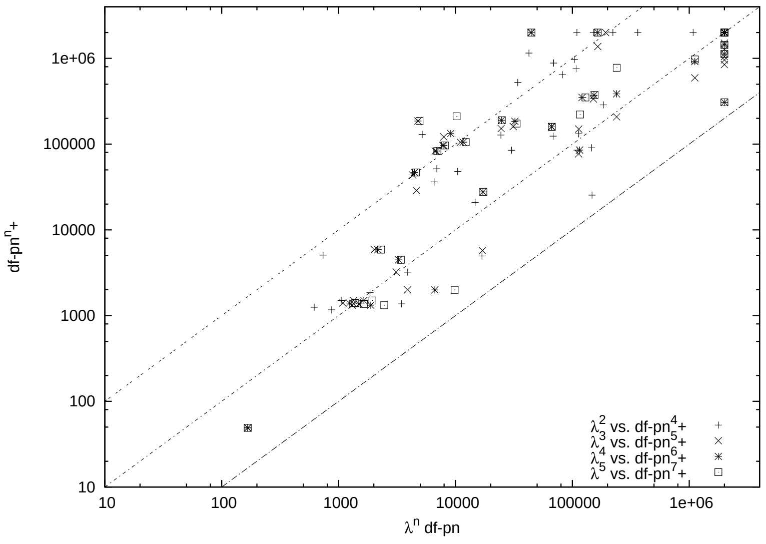
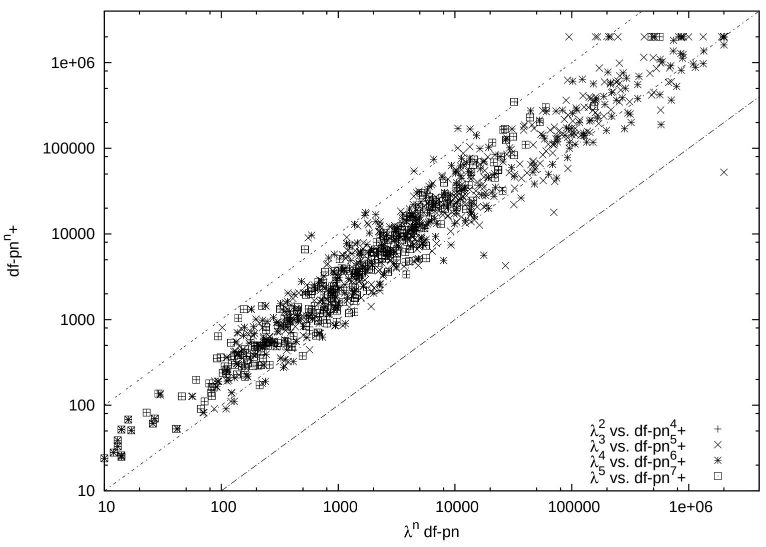
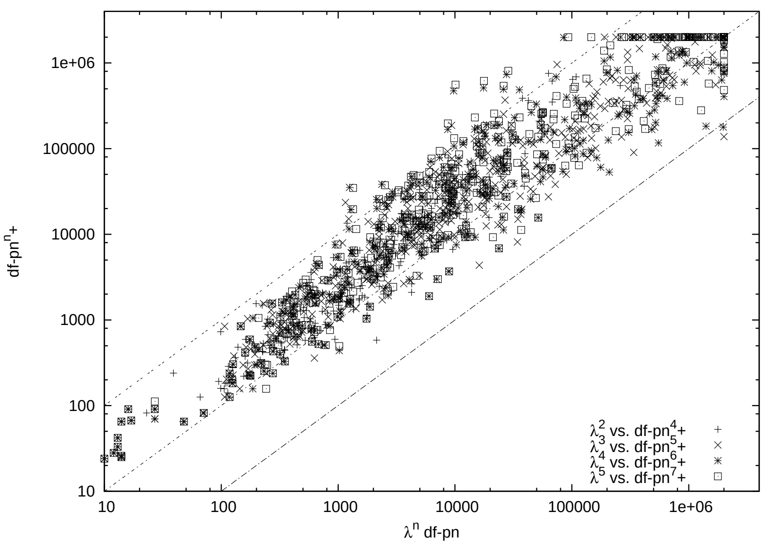


Fig. 4.10 Speed comparison: Master of Semeai: df-pn+ vs. λ df-pn

(a)
“Strong Forward Pruning”
move generator



(b)
“Weak Forward Pruning”
move generator



(c)
“No Forward Pruning”
move generator

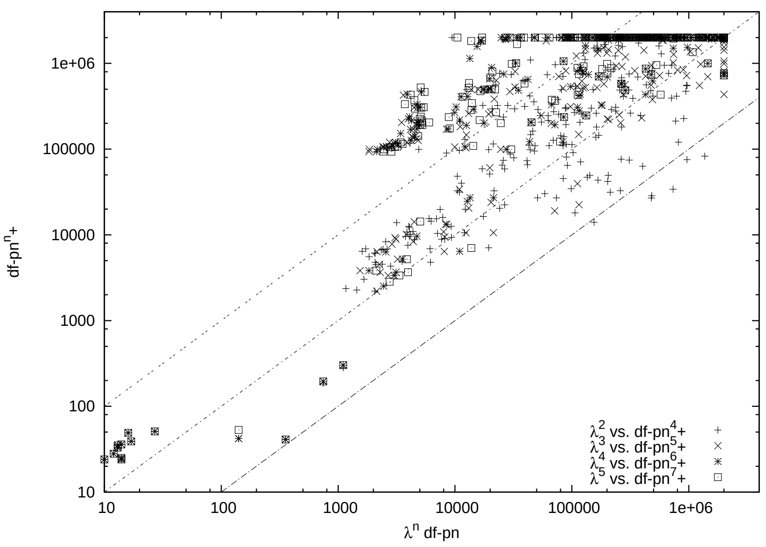


Fig. 4.11 Speed comparison: Thomsen test set: df-pn+ vs. λ df-pn

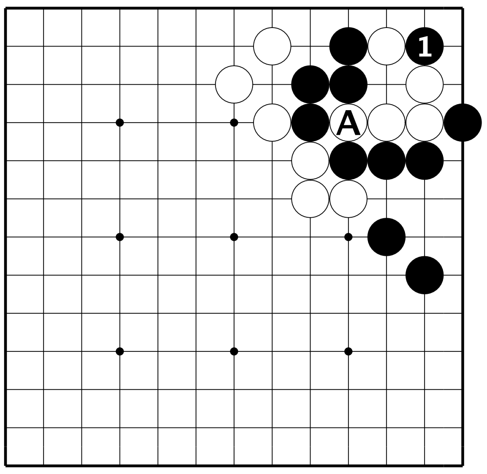
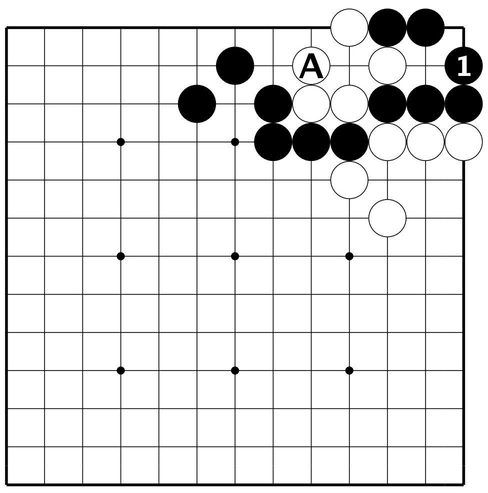
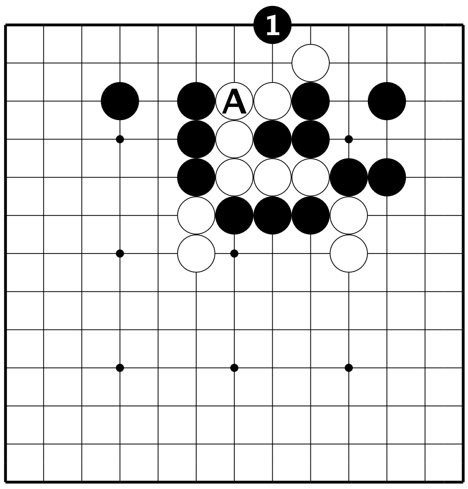
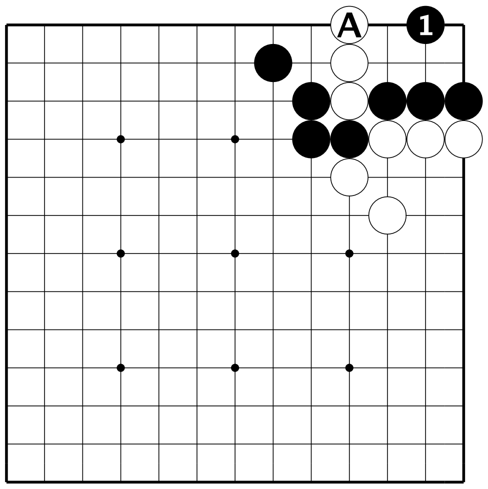
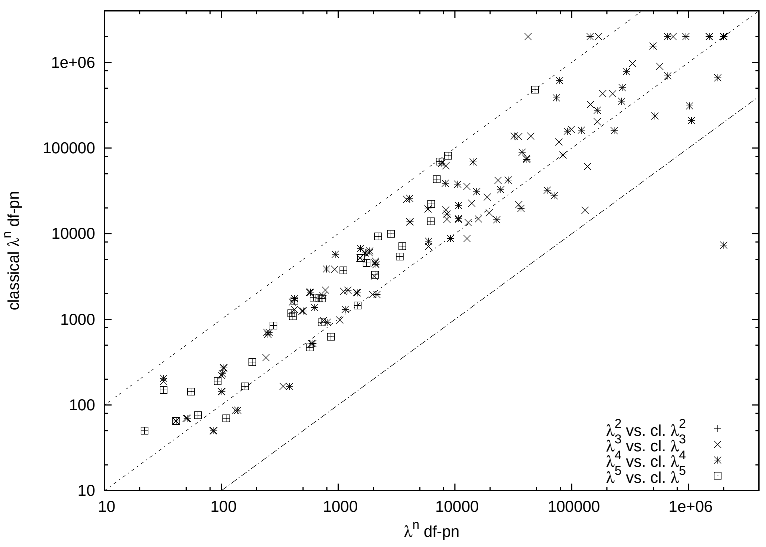
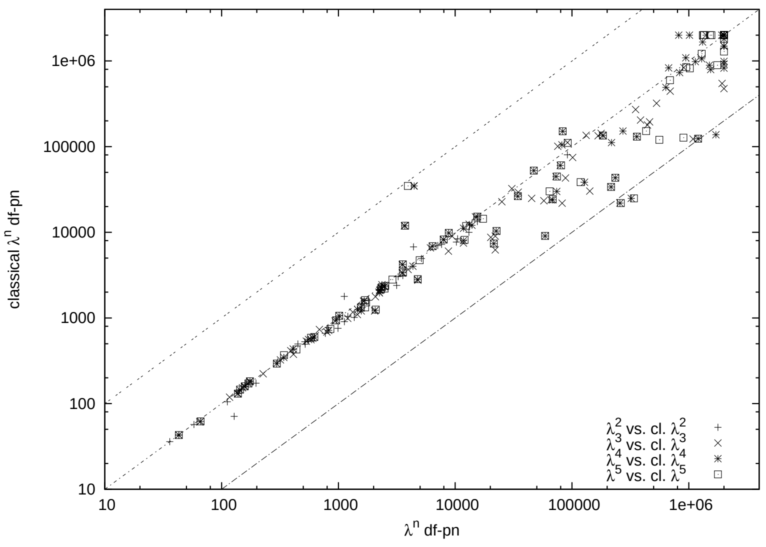
	
df-pn^4+ 4,559 nodes λ^2 df-pn 1,147 nodes	df-pn^6 19,739 nodes λ^4 df-pn 51,760 nodes
df-pn^5+ 6,012 nodes λ^3 df-pn 1,117 nodes	df-pn^7 21,640 nodes λ^5 df-pn 58,798 nodes
	
df-pn^4+ 480,847 nodes λ^2 df-pn 28,872 nodes	df-pn^6 159,722 nodes λ^4 df-pn 139,832 nodes
df-pn^5+ exceeded 2M limit λ^3 df-pn 429,255 nodes	df-pn^7 197,673 nodes λ^5 df-pn 190,967 nodes
λ df-pn was faster	df-pn+ was faster

Fig. 4.12 Problems for which λ df-pn solved faster/slower. (Black to play)

(a)
“Strong Forward Pruning”
move generator



(b)
“Weak Forward Pruning”
move generator



(c)
“No Forward Pruning”
move generator

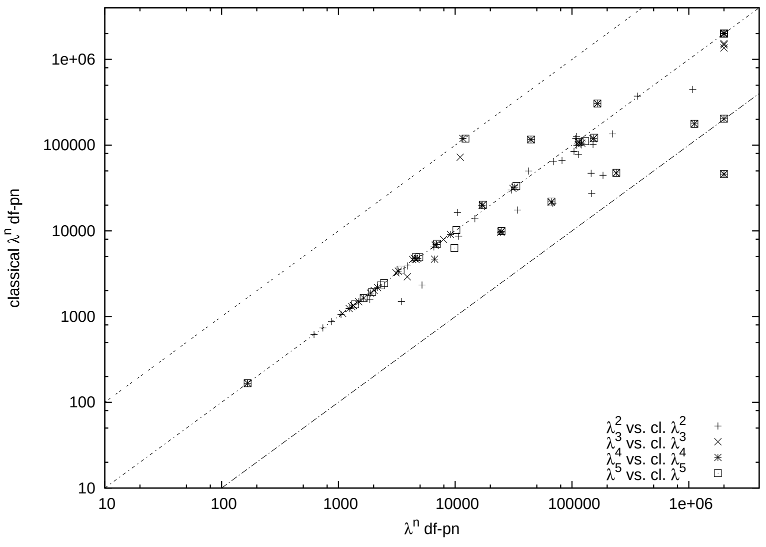
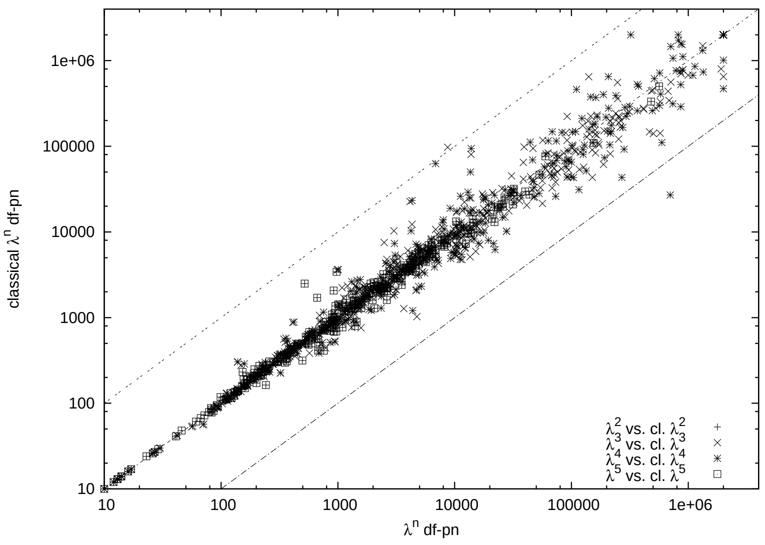
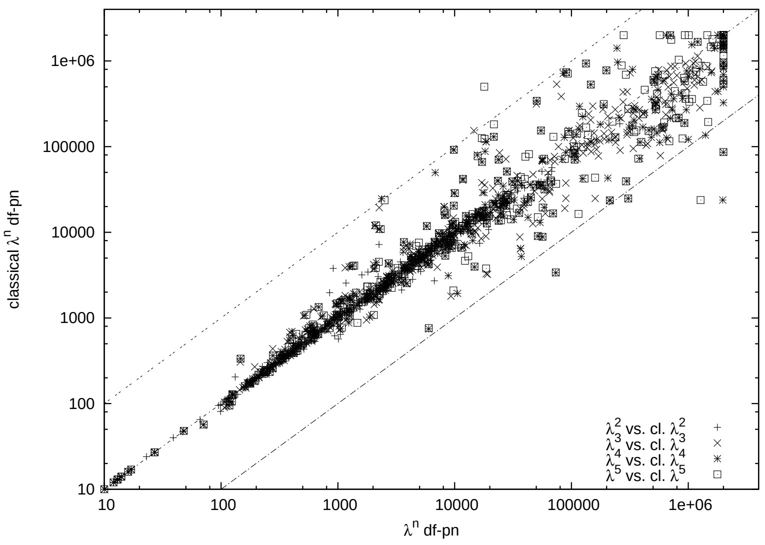


Fig. 4.13 Speed comparison: Master of Semeai with classical λ df-pn vs. λ df-pn

(a)
"Strong Forward Pruning"
move generator



(b)
"Weak Forward Pruning"
move generator



(c)
"No Forward Pruning"
move generator

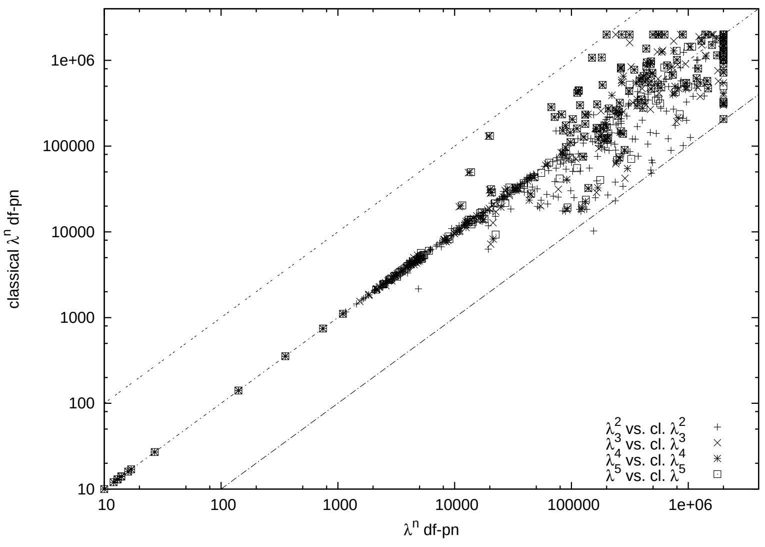


Fig. 4.14 Speed comparison: Thomsen with classical λ df-pn vs. λ df-pn

Chapter 5

Dynamic Widening

Our second approach is to limit the target of the search to promising moves using a simple dynamic technique, not a static evaluation function. This approach was developed as *dynamic widening* technique and was used in combination with df-pn.

Dynamic widening improved the speed performance of df-pn approximately 25 times on search spaces with uniform branching factors.

5.1 Overview

The proof number search achieves high performance if the goodness of moves greatly affects the amount of proof numbers. Tsume shogi is an ideal domain in this aspect. However, typically when the branching factors are great, df-pn (or other PN search algorithms) tends to have less advantages compared to other approaches such as Alpha–beta search.

When many legal moves exist and most of are likely to be unpromising moves, (dis-)proof numbers become great, which hides information about the goodness of moves. We can explain simply that the difference between 3 and 4 might have some meaning but difference between 100 and 101 is highly likely to be just a result of coincidence.

It is often observed for problems that df-pn does not perform well, and that (dis-)proof numbers become great when numerous unpromising moves exist. Existing solutions to this problem are to specify unpromising moves and ignore them unless all other more promising moves are disproved. At the beginning of the search, unpromising moves are pruned and ignored. When results show that all promising moves do not work, unpromising moves appears in the search tree. This approach is called *widening* by several researchers [17, 19].

These approaches are based on domain-dependent knowledge founded upon hand-crafted heuristics or machine learning. Instead of relying on domain-dependent knowledge, we propose a new simple domain-independent technique for Proof Number Search [94].

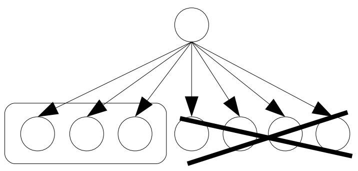


Fig. 5.1 Automatically specifically examine promising moves

5.2 Background

As explained in chapter 3.2, proof-number-based search algorithms expand the nodes with the smallest (dis-)proof number. Immediately after the expansion of a node, usually (dis-)proof numbers are simply proportional to the number of the candidate moves. An intuitive explanation of proof number search is that it is an algorithm that is intended to minimize the number of an opponent's legal moves.

For problems such as those of tsume-shogi, the proof number is effective because moves which limit opponent's legal moves often turn out to be good. However, for problems in games such as Go, the legal moves are vastly numerous and appear to be similar no matter which move we choose. For that reason, the initial values of (dis-)proof numbers do not reflect the goodness of the moves. In general, negative effects are often observed on a move ordering based on (dis-)proof numbers when numerous (typically unpromising) candidate moves exist.

To alleviate the bad effects in the search direction caused by a large number of unpromising moves, a technique that delays the generation of unpromising moves is used in existing solvers if it is possible to distinguish unpromising moves from other moves. For tsume-shogi solvers, defensive *dropping moves* [*1] are typically delayed. For tsumego, searching for a pass move is often delayed. Therefore, these are not included in calculation of (dis-)proof numbers unless other promising moves are disproved. However, it remains difficult for today's best tsumego solvers [90, 34] to solve open-border tsumego problems. Typically, a human player must modify the problems to enclosed problems before using the solvers.

A simple technique, named Dynamic Widening, is proposed to address this difficulty. Our method is to use only the best few moves to calculate (dis-)proof numbers. The underlying idea is that, even given numerous legal moves, only the few best moves are important to evaluate the goodness of nodes.

We combined this approach with df-pn [55, 53] and measured the performance for Go, capturing problems on 19×19 boards. The speedup over normal Df-pn+ was more than 30 times when we searched all legal moves.

This approach was ineffective when we used heuristic forward pruning. Searching all legal moves with Df-pn+ and Dynamic Widening was only a fourth as fast as normal Df-pn+ with forward pruning. However, our advantage is that we can guarantee the correctness of answers because heuristic forward pruning results can yield in wrong answers.

Hereinafter, we designate a player who tries to prove the tree an *attacker*; the other player is a *defender*.

5.3 Dynamic Widening

In normal proof number search algorithms, the disproof number of an OR node is calculated as the sum of the disproof numbers of the children (Fig. 5.2). In this section, our explanation specifically addresses OR nodes, unless described explicitly. For AND nodes, algorithms are identical if proof numbers and disproof numbers are exchanged.

[*1] Reuse of captured opponent pieces.

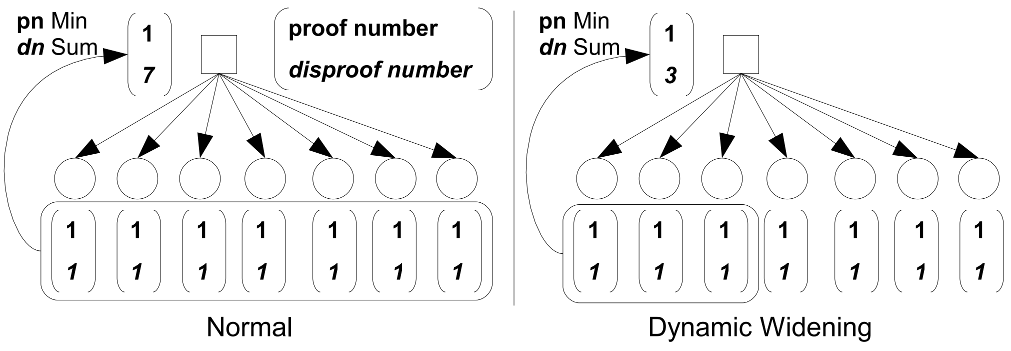


Fig. 5.2 Dynamic Widening: Immediately after expansion (OR node example)

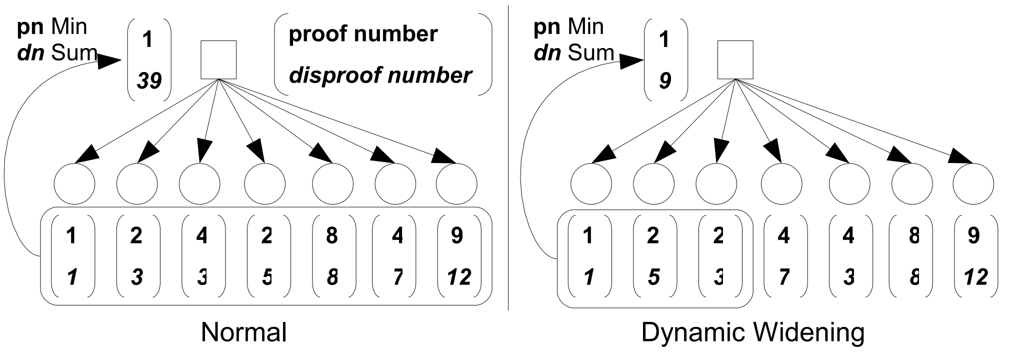


Fig. 5.3 Dynamic Widening: Search in progress (OR node example)

The technique we propose in this chapter is to sum up the disproof numbers of only the best group of the children. As depicted in the right tree of the figure, we calculate the disproof number using only a part of the child nodes. Fig. 5.2 presents an OR node immediately after the expansion. Initial values of (dis-)proof numbers are set to 1.

As the search continues, the (dis-)proof numbers will be updated (Fig. 5.3). Children are sorted according to the proof numbers before calculating the (dis-)proof numbers.

Let n be an OR node, and $n_1, \cdots, n_k$ be its children. For OR nodes, n_i are sorted according to their proof numbers in increasing order; only top j children are used to calculate (dis-) proof numbers. The (dis-)proof number of the OR node n would be

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

For an AND node, children are sorted in increasing order of disproof numbers. The (dis-)proof number will be

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

If a node is disproved, then the proof number will be set to ∞. After sorting, the disproved node will be passed to the end of the list and another node will become the target of disproof number calculation. Therefore, the search order will be changed, but the correctness of our technique is maintained.

Various ways of defining j can be used. Several methods were introduced and their performance was measured. Mainly, we compared the performance of the following two methods.

- Set j to a fixed constant ($TOPn$)
- Set j as $1/n$ of all children ($RATEn$)

We also tested some other methods for Dynamic Widening, such as preparing a threshold according to (dis-)proof numbers. However, defining j according to proof number threshold was ineffective. If it happens that more numerous child nodes are within the threshold, these nodes would be regarded as unpromising.

As already explained in the preceding chapter, the search speed might also be improved by assigning small (dis-)proof numbers to promising moves, and large (dis-)proof numbers to unpromising moves. We combined our methods and this technique (proposed in Df-pn+[55]).

With this combination, our algorithm will initially use only the promising moves for (dis-)proof number calculation. As the search progresses, the algorithm will shift to less-promising moves. Techniques that gradually increase the target scope of the search are called widening [19]. We designate our technique as Dynamic Widening because the scope for the search changes dynamically during the search.

5.4 Experimental Results

We performed experiments of Dynamic Widening to find the best parameter settings for the algorithm.

5.4.1 Experimental Conditions

We tested our algorithm for capturing problems of Go on a full 13×13 and a 19×19 board. We prepared two test sets: Master of Semeai and Thomsen's test set.

The first test set consists of 141 capturing problems, all taken from a Go problem book "Master of Semeai"[4]. This is a book of *semeai* problems. A semeai means a capturing race between two or more blocks. A typical example is portrayed in Fig. 4.9. The black attacker 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. We placed the problems in a 13×13 board. The book contained 148 problems but some problems in the book were complex problems that could not be solved merely by seeking a capture. Those problems were omitted from the test set.

The other test set includes problem sets distributed by Thomsen[81]. This problem set does not include problems that result in a Ko. Original problems included double-goal problems that require capture of either one of two blocks. Therefore, we selected only single-goal problems, resulting in 434 problems. For each problem, only one move was shown as the answer. For that reason, we added answer moves for some problems that had multiple answers.

These two problems are both capture problems of Go, but the characteristic is different. For semeai problems, two or more blocks are in danger and fighting for their lives. For the Thomsen's test set, the attacker has little worry about a counterattack by the opponent.

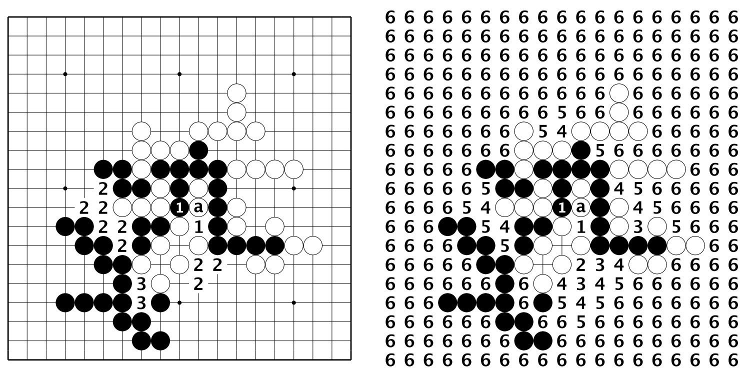


Fig. 5.4 Examples of heuristic move candidates (left) and all legal move candidates (right). The block marked with **"a"** is the capturing target.

Our algorithm was tested for move generators of two types. The first generator generates all legal moves except when the target block is in atari (generates 300 moves on average); the other is a move generator with heuristic forward pruning (generates 20 moves on average). The heuristic uses the idea of *surrounding-stones* [83] and consists of the liberties of up to *5-surrounding-stones*, liberties of adjacent friend stones, and some other moves including second liberties of target stones.

Fig. 5.4 portrays the difference of the candidate moves. The block marked with an "a" is the target to capture. The intersections marked with a number are used as candidate moves for Black. The number shows the initial value of (dis-)proof numbers used in Df-pn+. Promising moves have smaller (dis-)proof numbers. Let the number in the figure be n. The actual initial (proof number, disproof number) were, for OR nodes, $(2^{(n-1)}, 1)$ for AND nodes, $(1, 2^{(n-1)})$.

We limited the number of node expansions to 2,000,000 nodes. The searched nodes are fewer than this because of re-expansion.

Normally for a capture search, a liberty threshold to give up capturing will be set. However, in this experiment, we set the limit to ∞ so that it never gives up capturing.

5.4.2 Sorting Priority

Before performing the experiment, we have done a preliminary experiment to decide one parameter for the sorting method.

Two methods of sorting can be used when proof numbers are equal. The first is to compare the proof number. If the proof number is equal, then select the node with the "greater" disproof numbers. We designate this as the "less greater" method. The other method is to select the node with a smaller disproof number if the proof number is equal ("less less"). (For AND nodes, please flip proof/disproof number in this explanation.)

The performance comparison is depicted in Fig. 5.1 and Fig. 5.5. The limit of node expansions was set to 200,000. A great gap separates the answering ability and the speed performance. According to this experiment, we decided to use a "less greater" method for dynamic widening.

Despite the fact that the "less greater" method is choosing subtrees with greater branch-

Table. 5.1 Answering ability of the sorting methods.

sorting method	solved
less greater	152
less less	128

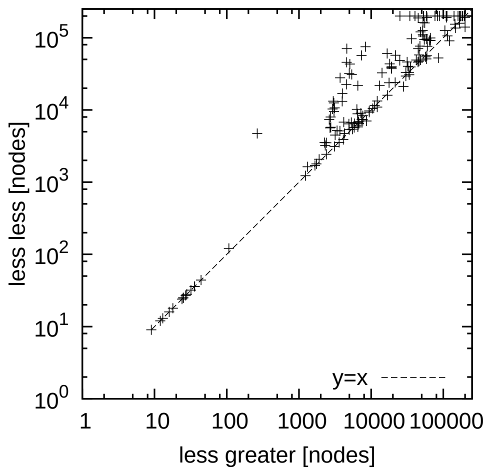


Fig. 5.5 Speed comparison of the sorting methods.

ing factors, the performance was markedly superior. This result can be explained simply as a selection of promising moves is the good strategy, because at an OR node, a smaller proof number is promising for the *attacker*; a larger disproof number is unpromising for the *defender*. However, this observation should be checked for domains other than Go.

5.4.3 Answering Ability

The answering abilities of methods are presented in Table 5.2. The first row shows the move generation method: "AL" means all legal moves are searched; "FP" means that a move generator with heuristic forward pruning is used. The second row shows the name of the method and parameter used for Dynamic Widening: $TOPn$ means that the best n moves are used in (dis-)proof number calculation; $RATEn$ means that the top $1/n$ moves are used.

The row named "solved" presents the number of problems solved within 2,000,000 node expansions; "exceeded" represents the number of problems which were not solved within the threshold. In addition, "miss" signifies the number of problems for which the algorithm returned an answer that turned out to be a wrong move. By searching all legal moves, the returned answers are always right.

These results show that Dynamic Widening is effective if we generate all legal moves. $TOPn$ is improved as n gets smaller until n=5: however, for n=2, the result is poor. With forward pruning, Dynamic Widening has a negative effect. For smaller n, the performance is worse. The FP move generator tends to generate fewer candidate moves after promis-

Table. 5.2 Answering ability with 2M nodes limit

cand.	DW method	solved	miss	exceeded
AL	none	94	0	340
AL	$TOP20$	182	0	252
AL	$TOP10$	206	0	228
AL	$TOP5$	191	0	243
AL	$TOP2$	165	0	269
AL	$RATE2$	77	0	357
AL	$RATE5$	97	0	337
AL	$RATE10$	150	0	284
AL	$RATE20$	183	0	251
AL	$RATE40$	199	0	235
AL	$RATE80$	190	0	244
FP	none	341	17	76
FP	$TOP20$	319	15	100
FP	$TOP10$	294	10	130
FP	$TOP5$	239	5	190

ing moves. Therefore the assumption of "a smaller (dis-)proof number means promising" does hold. We presumed that the worsening performance results from the fact that Dynamic Widening merely ignores this information.

5.4.4 Parameter for Top n

Based on the results shown in the previous section, we decided to examine the Top n method for dynamic widening specifically because the performance was the most promising in terms of speed and answering ability.

We set n as 2–20 and observed the answering ability and speed. The limit of node expansions is again set to 2,000,000. Answering ability is simply the number of problems solved right. Speed comparison is done by the total node expansions for the problems solved by at least one parameter setting.

For the Thomsen testset, Figs. 5.6 and 5.7 show the performance with and without forward pruning. For Master of Semeai, Figs. 5.8 and 5.9 show the performance. The horizontal line "default" shows the performance of normal df-pn without dynamic widening.

Results show the degree to which the performance declines with forward pruning as n decreases. However, the performance gain is obtained only when forward pruning is turned off. The node expansions show different characteristics for two test sets. However, the number of solved problems shows that n=7 is the best parameter for Top$n$. We concluded that Top7 is the best setting for these test sets.

For more precise analyses, we show the number of node expansions needed to solve each problem. The charts show a comparison of the speed performance of df-pn without DW and df-pn DW (top5 to top20). Fig. 5.10 shows that the overall performance of dynamic widening is good, with the best case at top 7. However, as the plot shows, the improvements are widely

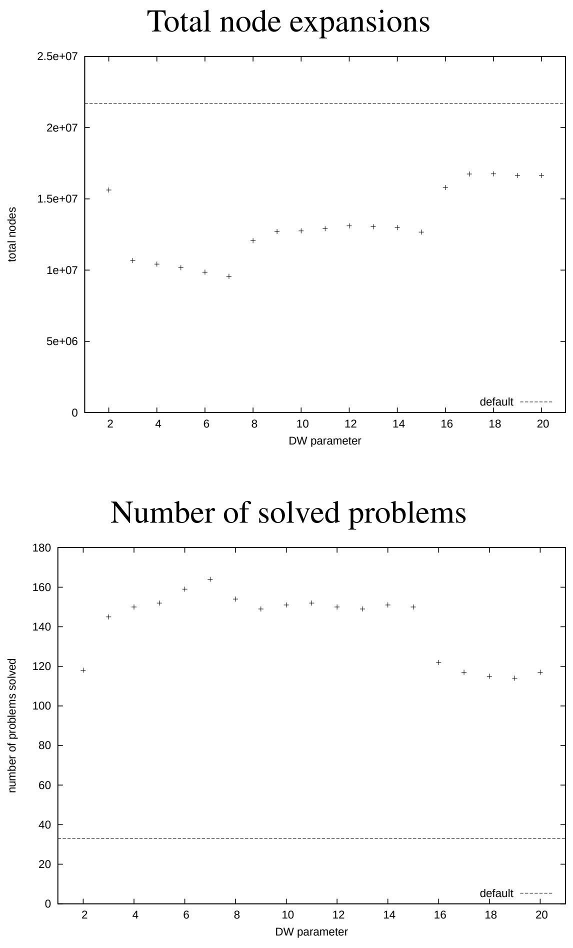


Fig. 5.6 DW parameter and performance: Thomsen test set with No Forward Pruning

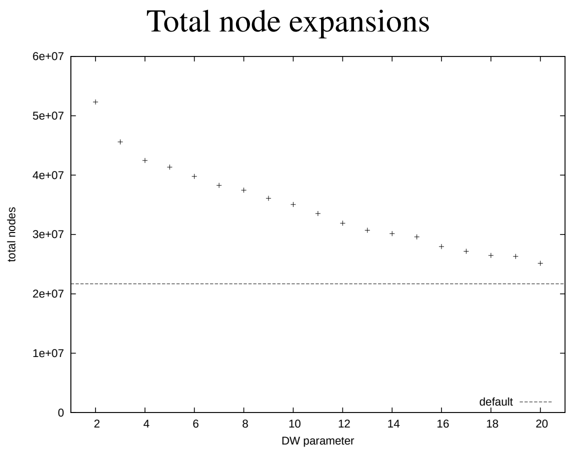
ranged. For smaller n, improvements are more unstable. Fig. 5.11 confirms that dynamic widening does not improve the performance if combined with heuristic forward pruning.

5.4.5 Overall Analysis

To compare the respective overall performances of the three methods listed below, we compared the total number of nodes needed to solve the problems that were solved using all three methods.

- Default Df-pn+ (all legal)
- $TOP7$ dynamic widening Df-pn+ (less greater)
- Forward pruned Df-pn+

The result presented in Table 5.3 shows that $TOP7$ DW makes Df-pn+ more than 25 times faster, but it is still slower than Df-pn+ using forward pruning. Nevertheless, the advantage of Dynamic Widening is that it can confirm the correctness of the returned answer because the 3% of the answers returned by the heuristic forward pruning version were wrong.



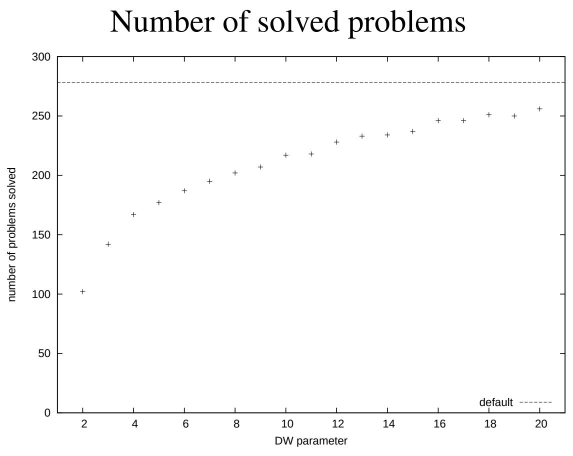


Fig. 5.7 DW parameter and performance: Thomsen test set with Forward Pruning

Table. 5.3 Overall Performance (AL, all legal; FP, Forward Pruning)

method	solved	total nodes
Df-pn (AL, DW off)	94	39,880,893
Df-pn (AL, DW on)	213	1,668,379
Df-pn (FP, DW off)	341	205,874

For precise speed performance, we again show the number of node expansions needed to solve problems. The chart in Fig. 5.12 shows the log–log plot of the performance of Df-pn+ with heuristic forward pruning, and with $TOP7$ dynamic widening when all legal moves are used. This chart shows that the performance is stable but inferior. These results are those of Thomsen test sets. The chart for Master of Semeai was omitted because the results were similar.

The children must be sorted before selecting the child to expand. However, only the top n nodes need to be sorted. Once sorted, the order will not be disturbed much from the second time thereafter. Therefore, we expected that sorting is not so time consuming. In this

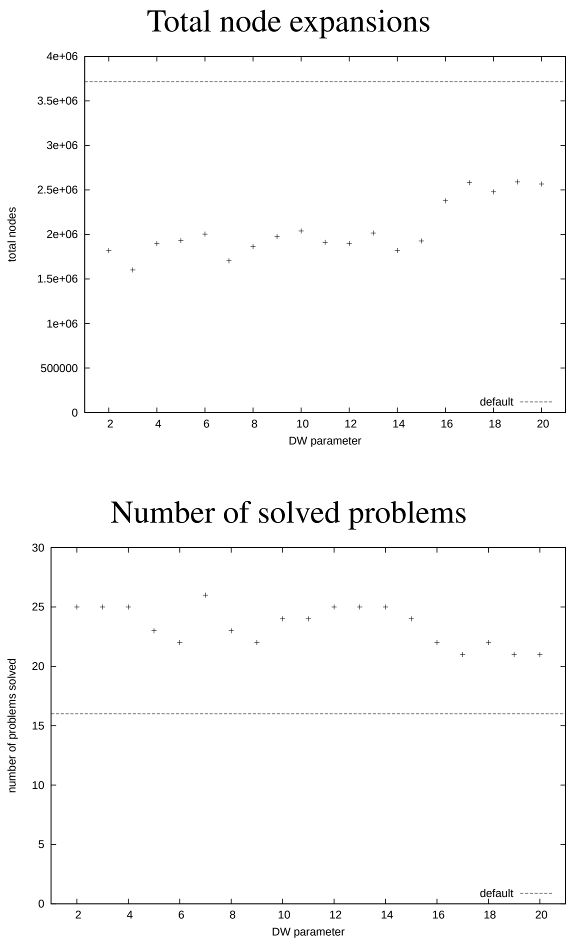


Fig. 5.8 DW parameter and performance: Master of Semeai with No Forward Pruning

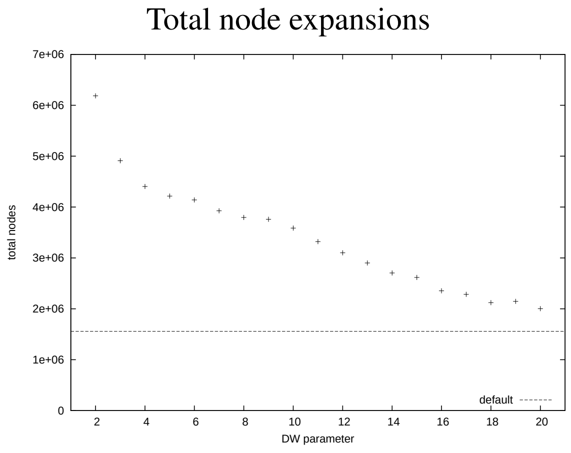
experiment, the time needed for sorting was 1–2% of the total execution time. (The ratio differs for different problems.)

In conclusion, Dynamic Widening is effective for capturing problems of Go if all legal moves are considered. It can be implemented easily with small execution time overhead. It was ineffective if combined with heuristic forward pruning, but it will be useful for problems with large branching factors such as those of Go. Dynamic Widening is especially effective for problems for which it is necessary to guarantee the correctness or those for which heuristic pruning is difficult.

Moreover, from these results, we infer that only 7 moves are meaningful to control search direction for the problems used in the experiment. We provided a new perspective to analyze the characteristics of target domains for proof number search algorithm. However, the number is unstable for each problem. Further study is needed in other domains.

5.5 Related Work

Dynamic Widening can be explained thusly. Discarding a part of the information we have would result in higher performance in certain circumstances. A similar existing approach in



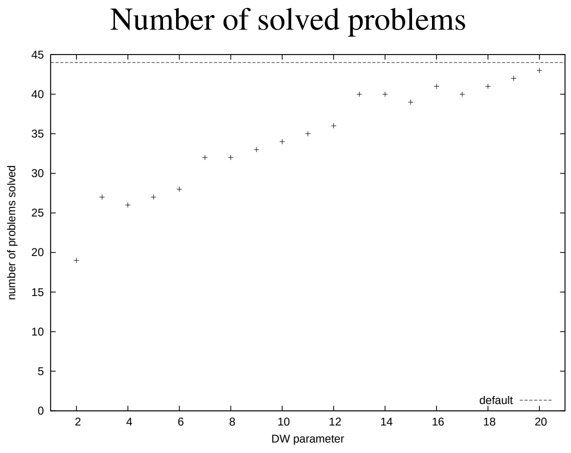


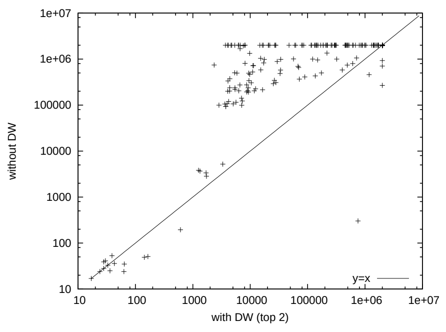
Fig. 5.9 DW parameter and performance: Master of Semeai with Forward Pruning

this direction is Branch Number Search (BNS) developed by Okabe[58].

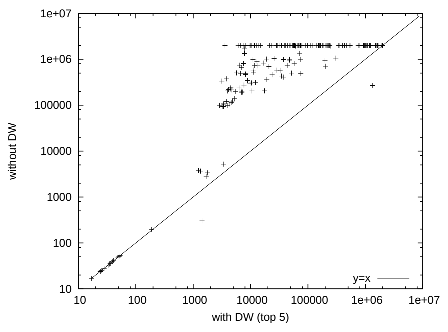
In fact BNS is currently known as the best tsume-shogi solver for a class of difficult problems. The idea is similar to that of PN search. When selecting the most proving node, PN search considers the total number of unsolved nodes. On the other hand, BNS considers only the number of unsolved branches in the sequence from the root node to the edge, which is leading to the node.

The advantage of BNS is in its ability to solve problems that have frequent interflows in the proof tree. When PN search tries to solve such graphs, interflowing nodes cause double counting of proof numbers. The answering sequence of a difficult long move tsume shogi problem often reaches some hundreds of moves and often includes many interflows. The value of the (dis-)proof number would be exponential to the number of interflows. In fact, BNS is not affected by double counting caused by interflows. It solved many previously unsolved tsume shogi problems.

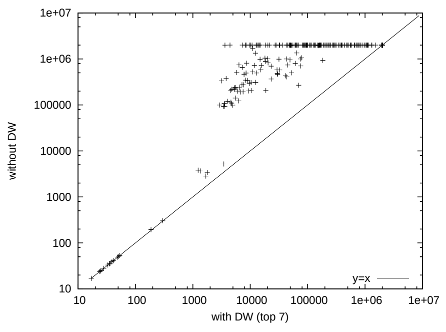
Weak proof number search[84] uses a similar approach to that of dynamic widening. When calculating the sum of (dis-)proof numbers of the child nodes, they use maximum instead of the sum. Their goal was to avoid negative effects from interflows in DAGs. The resulting algorithm shows a similar behavior to that of dynamic widening.



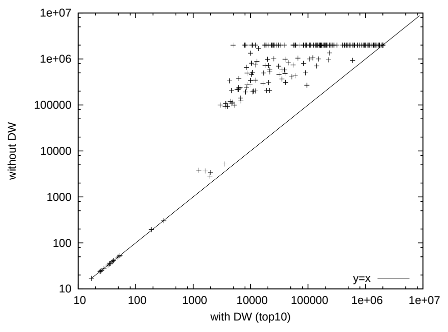
top2



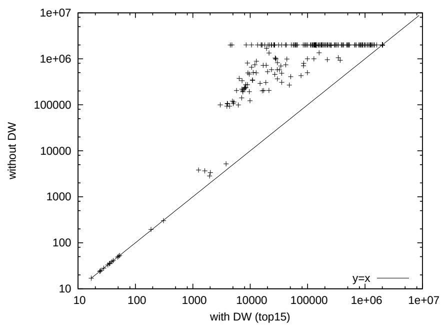
top5



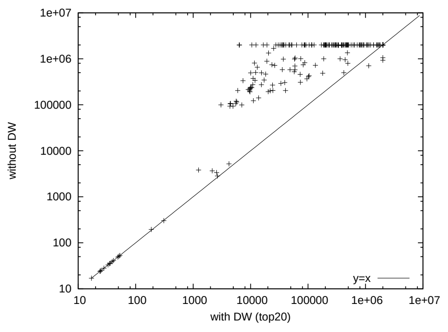
top7



top10

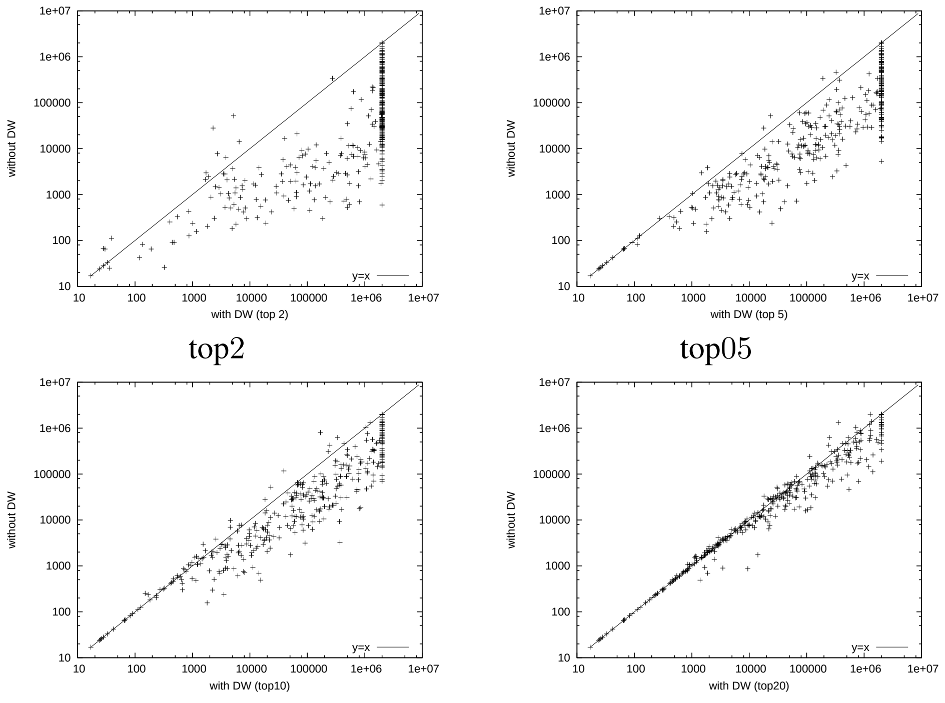


top15



top20

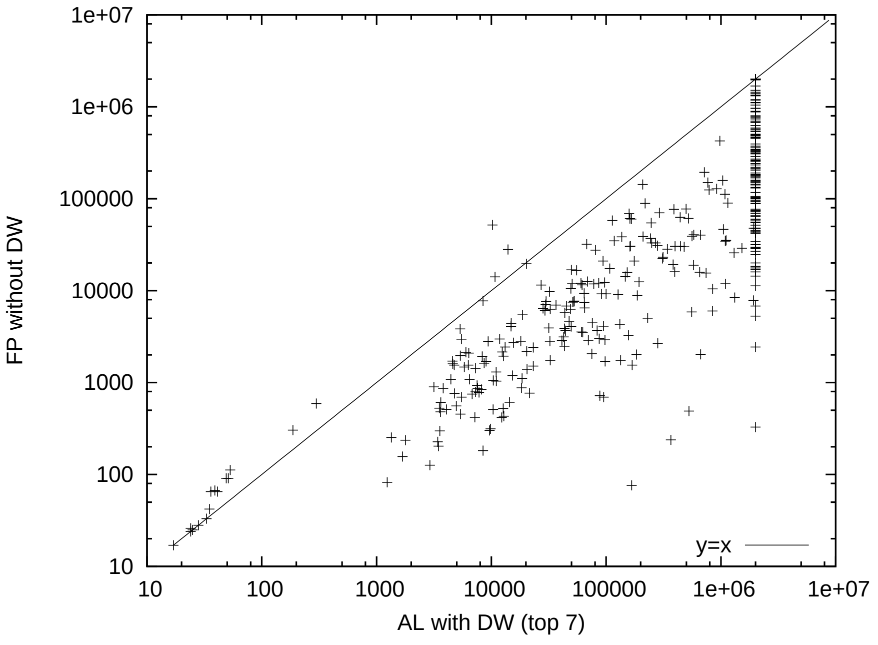
Fig. 5.10 Dynamic Widening with all legal moves.



top2 top05

top10 top20

Fig. 5.11 Dynamic Widening with forward pruning.



FP without DW vs. AL with DW (top7)

Fig. 5.12 Still cannot catch up with FP.

Chapter 6

Overall Solver Ability

We tested our techniques and examined the limits of their capabilities.

6.1 Overview

In chapters 5 and 4, we described two techniques: dynamic widening and λ df-pn. Now we test the overall ability of our two techniques. We combined λ df-pn with dynamic widening and tested its ability for solving classic Go problems. The problems are taken from famous classic Go books described in chapter B.4.2 and a Japanese Go training book [4].

6.2 Semeai Solver and Tsumego Solver

Theoretically, tsumego solver can be used as a semeai solver. What tsumego solvers have in addition to semeai solvers is the ability to detect stones having two or more eyes as alive. Eye shape detection needs pattern matches during a search, which slows the execution time. Therefore, if only capturing is concerned, semeai solver is generally faster for solving problems. Nevertheless, it is extremely time consuming to use semeai solver for tsumego problems without the knowledge that two eyes make a life.

6.3 Ko Resolution

For local problem solvers of Go, positions resulting in *ko* are always a problem.

In terms of Go problems, problems are regarded to have three answers win, loss, and ko. This is true because, in real games, global ko threats must be considered if the problem results in ko locally. For that reason, ko differs from a local win or loss.

Our solvers currently do not handle ko locally. The solver merely solves the problem as if no other stones exist on the board. Only local ko threats are played.

Ko solutions complicate the comparison because of the second search, especially if a win by ko makes it possible to win in lower λ order. Fig. 6.1 shows an example of such a problem. Therefore, we did not compare our algorithm with ko solutions.

However, solutions for Ko are known and the implementation is not difficult.

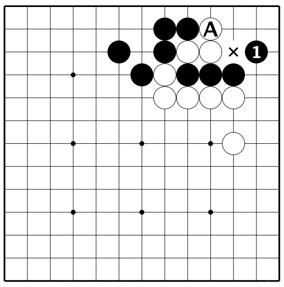


Fig. 6.1 Playing on the cross results in λ^2 win by ko, while an unconditional win is possible in λ^3.

6.3.1 Ko Solution by Kishimoto

Kishimoto's papers[34] describe a simple and effective solution that is valid for most problems. It requires two searches: one to determine the temporal winner, and a second to detect ko.

1. Determine the temporal winner.
2. Perform second search. During the second search, the temporary loser is allowed illegal ko captures at any time.
3. The problems will result in ko if the winners differ in the two searches.

The time needed for a second search can be greatly reduced by reusing the proof tree produced for the first search. This solution is effective for most problems, but it does not detect which side has an advantage for ko.

6.3.2 Ko Solution by GoTools

The solution used by GoTools[91] also requires a second search. The difference is that it counts illegal ko captures needed for the temporary loser to win. This solution evaluates the *approach ko* correctly, and distinguishes favorable kos and disadvantageous kos. However, this solution is more time consuming than the previous solution.

6.4 Combination of Dynamic Widening and λ df-pn

The idea of dynamic widening can be easily combined with other proof number search algorithms. We tested variations of top$n$ dynamic widening combined with λ df-pn. The methods for deciding the number of child nodes used in proof number calculation are listed below.

1. Limit the number of child nodes to a constant, irrespective of λ order.
2. Use all moves with λ order 2 or lower, and limit the number of moves over λ^2 to a constant.
3. Limit moves over λ^3 to a constant.
4. Limit moves over λ^4 to a constant.

We performed preliminary experiments and chose the parameters for dynamic widening for λ df-pn with DW. With strong forward pruning, the top 15 for over λ^3 was the best. With weak forward pruning, the top 25 for over λ^4 was the best. Unfortunately, we were able to solve only a few problems with no forward pruning.

6.5 Ladder Search for λ^1

A λ^1 search for capturing problems of Go is the same as a *ladder* search. A ladder search is exceptionally simple because the number of move candidates is small (1 or 2 in most cases). In the implementation reported here, a special purpose *ladder* search is approximately 15 times faster in terms of nodes per second than λ df-pn. It is useful as a subroutine by λ df-pn.

We can use a λ search for other purposes if we replace the ladder search with other search algorithms. The advantage of λ df-pn over the original λ search is that the search can switch seamlessly between search algorithms based on proof/disproof numbers, which makes it feasible to adjust efforts for each search component.

6.6 Preparing Test Problems

We tested the answering ability of the algorithm for semeai problems taken from classic Go books and a recently published Go training book.

Problems for test sets are taken from

- “center part” of Xuánxuán Qíjīng (XXQJ, 玄玄碁経) [2]
- Selected problems with true lambda order of up to λ^5. of “semeai part” of Gokyo Shumyo (Shumyo, 碁経衆妙) [3].
- Selected problems with true lambda order of up to λ^5 from Master of Semeai (Master) [4].

However, to select the appropriate test problems, we wanted to know the true λ order of the problems. We tried to solve all problems with the classical λ df-pn solver or λ df-pn solver, given 32M node extensions as the limit. We only used problems that at least one of our solvers was able to solve. Therefore, the true λ order is known. We listed the problems solved by at least one of the algorithms.

6.7 Solving λ^2 Problems

Problems with true λ order one or two are easy. We demonstrate the ability of our solver for solving two λ^2 problems portrayed in a classic Go book. These two problems are listed in the “center chapter” of Xuánxuán Qíjīng (玄玄碁経), and are the most difficult λ^2 problems in our test set.

The result presented in Table6.1 shows that λ df-pn with ladder enhancement solves both problems quickly. From this result, we concluded that any practical Go problem that fits into a 19×19 board with λ order one or two can be solved quickly with λ df-pn.

Table. 6.1 Node expansions for λ^2 problems.

problem	heuristic	heuristic+	full board
XXQJ Nos. 30	2566	3440	94715
XXQJ Nos. 31	2570	3102	72511

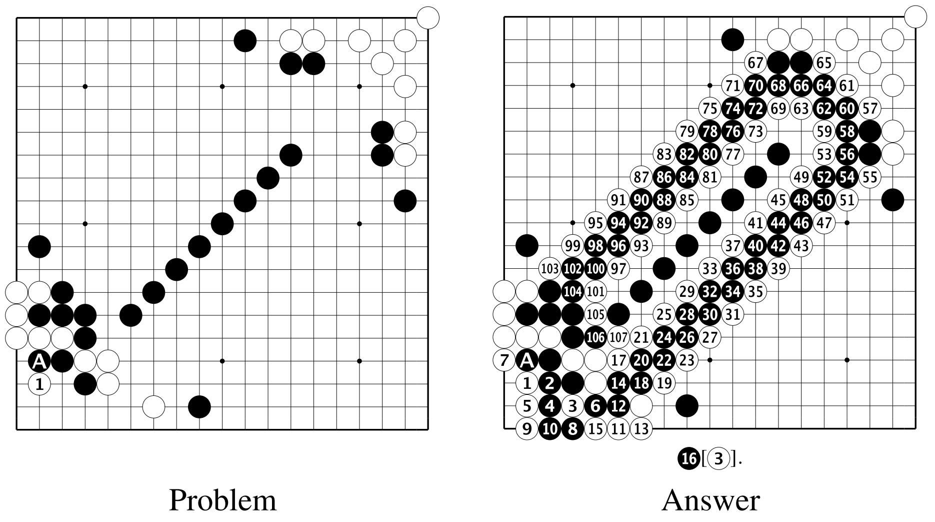


Fig. 6.2 Xuánxuán Qíjīng No. 30 千層寶塔勢 (One thousand floor treasure tower)

6.8 Solving Problems λ^3 or Higher

To test the solver ability for problems with λ order three or higher, we selected such problems from Xuánxuán Qíjīng[2], Gokyo Shumyo[3], and Master of Semeai[4], by selecting problems solved using the classical λ df-pn. Problems are listed in Tables C.3, C.4, and C.5.

Tables 6.2 and 6.3 show the node expansions for solving the problems. Entries denoted as — are problems that could not be solved within the 32M node limit. It is difficult to choose the best algorithm based on the results, but considering the answering ability, λ df-pn and λ df-pn combined with dynamic widening are slightly better than df-pn+ and classical λ df-pn.

However, as the charts show, df-pn+, λ df-pn and λ df-pn with DW have similar speed performance relative to the number of node expansions for each problem.

As described already in chapter 4, df-pn+ and λ df-pn have similar performance in λ^4 and λ^5 problems. To solve existing classic Go problems, further study is needed, such as using heuristic knowledge in λ df-pn and combining heuristic forward pruning with widening techniques.

6.9 Conclusion

The performance of λ df-pn DW was similar to the performance of df-pn+. As discussed in chapter 4, λ df-pn presents less of an advantage if the true order of the problem is higher than

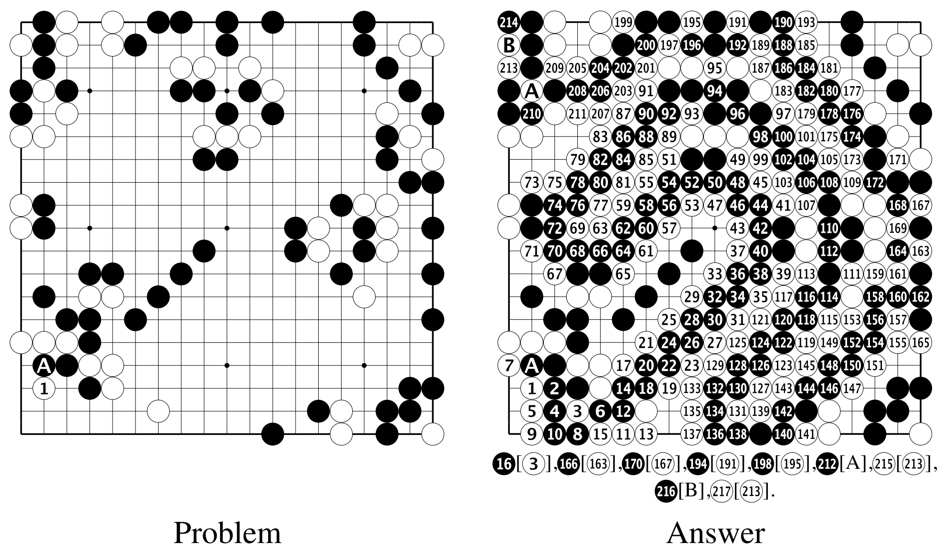

16[3], 166[163], 170[167], 194[191], 198[195], 212[A], 215[213], 216[B], 217[213].
Problem
Answer

Fig. 6.3 Xuánxuán Qíjīng No. 31 明皇遊月宮勢 (Emperor Xuanzong plays on the moon castle)

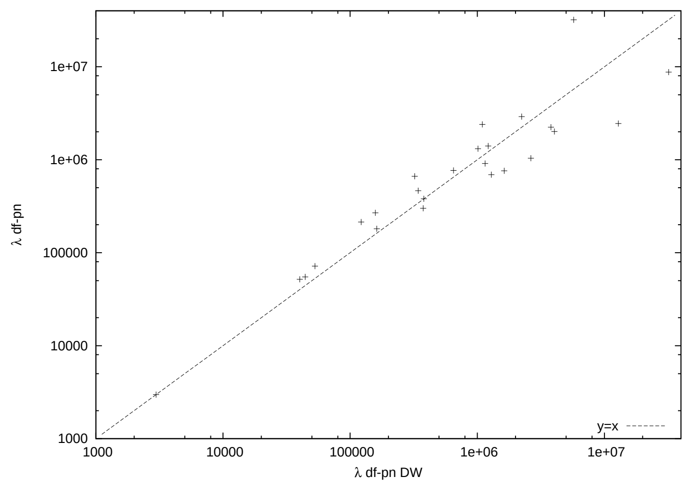


Fig. 6.4 λ df-pn vs. λ df-pn DW (Top15, λ^3) with Strong Forward Pruning

3 and the order of the target problem is known. Furthermore, df-pn has an advantage because, in the experiments described in this chapter, the true λ order is known. (It is also discussed that it is unlikely that the true λ order is known in advance.)

The results presented in this chapter show that this is the limit of λ df-pn and dynamic widening with no domain-dependent heuristic. However, considering the fact that λ df-pn DW did not use any domain-dependent knowledge and that df-pn+ used a simple domain-dependent heuristic, a good chance exists for λ df-pn DW to overcome df-pn.

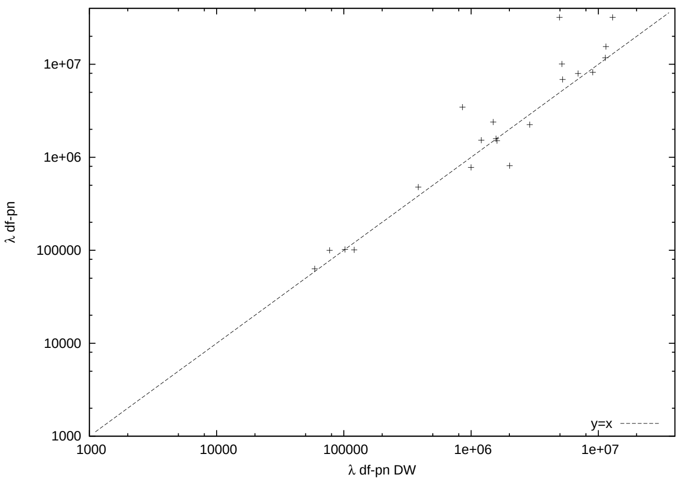


Fig. 6.5 λ df-pn vs. λ df-pn DW (Top15, λ^3) with Weak Forward Pruning

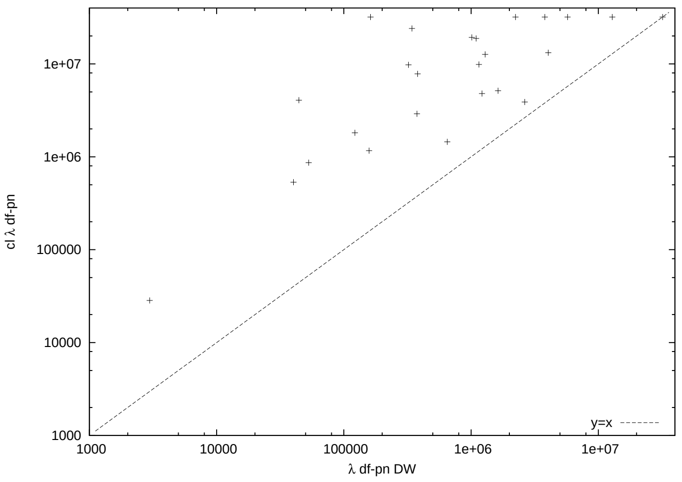


Fig. 6.6 cl λ df-pn vs. λ df-pn DW (Top15, λ^3) with Strong Forward Pruning

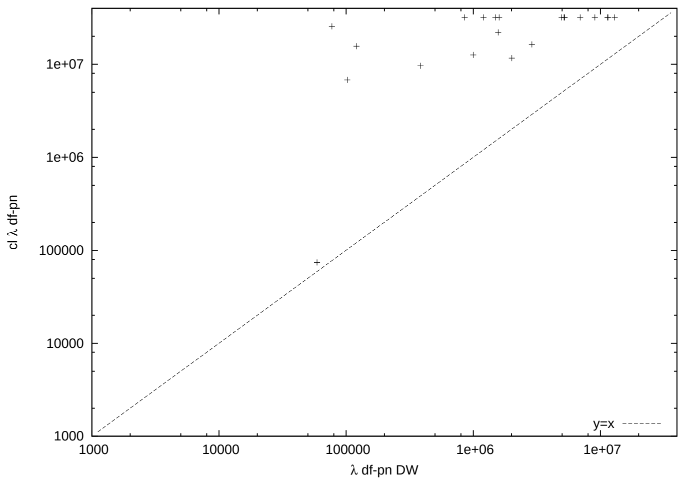


Fig. 6.7 cl λ df-pn vs. λ df-pn DW (Top15, λ^3) with Weak Forward Pruning

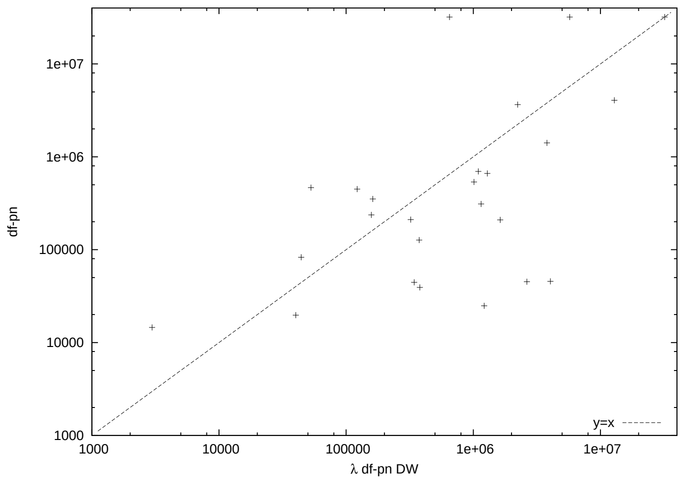


Fig. 6.8 df-pn vs. λ df-pn DW (Top15, λ^3) with Strong Forward Pruning

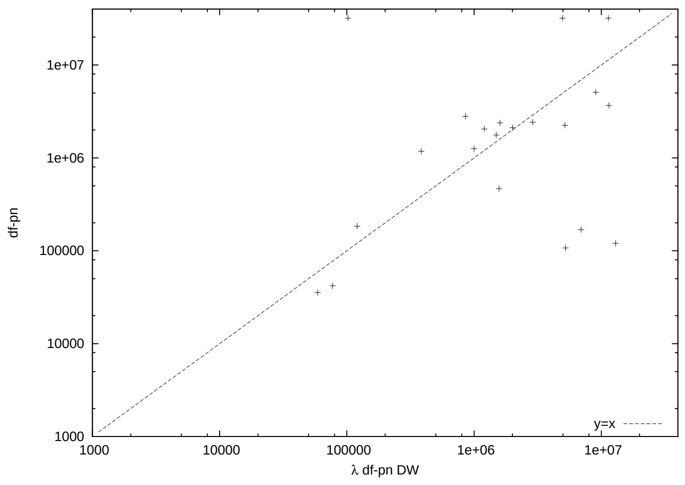


Fig. 6.9 df-pn vs. λ df-pn DW (Top15, λ^3) with Weak Forward Pruning

Table. 6.2 Node expansions for λ^4 problems (Strong Forward Pruning)

problem	df-pn	λ df-pn	λ df-pn DW	cl λ df-pn
XXQJ007	210558	663357	322044	9768703
XXQJ012	45661	2014334	4039971	13200141
XXQJ018	1411946	2237325	3796383	—
Gokyo023	—	—	5736906	—
Gokyo073	535301	1313033	1013611	19309755
Gokyo075	45172	1038034	2639814	3888861
Master001	127032	300723	375478	2902610
Master010	19739	51760	40186	533410
Master024	351948	180921	161981	—
Master034	310580	910418	1151863	9876043
Master043	696123	2402224	1096347	18812413
Master045	82978	54915	44280	4068768
Master093	14566	2973	2973	28376
Master106	236902	268661	157876	1162692
Master107	661721	691557	1291090	12671900
Master115	—	767037	650085	1448925
Master119	465823	71712	52923	864015
Master121	44545	464444	342803	24092189
Master123	209058	759525	1627948	5139865
Master136	24882	1404747	1216989	4800390
Master142	448797	213901	122105	1813347

Table. 6.3 Node expansions for λ^5 problems (Strong Forward Pruning)

problem	df-pn	λ df-pn	λ df-pn DW	cl λ df-pn
XXQJ004	39256	379919	379882	7822823
Gokyo004	—	8763619	—	—
Gokyo046	3646377	2908575	2464074	—
Master071	4065446	2453974	4326252	—

Chapter 7

Searching Inversions

7.1 Overview

The AND/OR tree search algorithms were developed specifically for solving complex AND/OR tree searches. They have the ability to find out the answer from long narrow sequences using little domain specific knowledge. However, the algorithms can be only applied to a problem with a well defined goal that is easy to determine as true or false. On the other hand, MCTS is a generic and high-performance algorithm that works well for problems of many types, but has a weakness in finding an answer after a long narrow sequence.

Therefore it would be ideal if an AND/OR tree search could be combined to cover the weakness of MCTS. For the combination, AND/OR tree search must have the ability to solve deterministic local problems that occur frequently while performing MCTS. λ df-pn is suitable for this purpose because it is a generic AND/OR tree search algorithm.

To combine these two algorithms, we developed an algorithm to efficiently reuse the search results of local searches. For reusing the past local search results, it is necessary to detect and clarify the conditions needed for them to remain valid. In other words, we must find out how to isolate a local problem from nother parts of the board.

In this chapter, we describe an algorithm that searches *inversions* of local searches. An inversion is a set of moves of a player, that can invert the search results if the other player ignores the move. In the game of Go, a set of moves can be described by an area of the board. Although the main objective for inversion search is isolating a local problem from other parts of the board, inversion can also be used for resolving dependencies between different subgoals.

The inversion search algorithm has been tested on our test problems and compared with the results with past similar approaches. The performance was improved in terms of accuracy and flexibility, while maintaining reasonable search speed. We first used df-pn for the search algorithm in a published paper [93], and then improved the result in this thesis using λ df-pn.

7.2 Capturing Problem of Go and Inversion

The advance of search algorithms is especially apparent in the game playing area. Only if a single goal exists with a well defined evaluation function can the way to the goal be sought efficiently. For example, it was possible to construct one good evaluation function for chess, which approximates the global goal (to mate the opponent's king while protecting

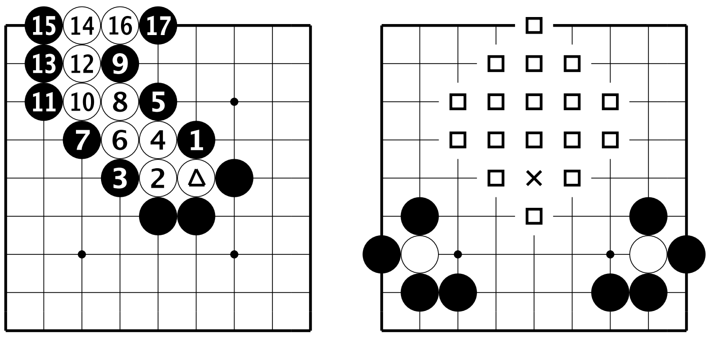


Fig. 7.1 a *ladder*, and a double *ladder*

one's own king). This approach achieved good results in many games. However, the game of Go remains as a great challenge for computers [10, 50] because Go is a game for which it is extremely difficult to construct a fast and accurate evaluation function, except in the late endgame. Human players rely on a combination of partial local subgoal searches. It is believed that most competitive Go playing programs developed before 2006 also relied on subgoal searches to generate moves.

As explained in chapter B, knowing if a *block* can be captured is the most important subgoal for playing Go. It has an apparent evaluation function, and can be efficiently sought only if the target *block* is independent. For this reason, we specifically examine stone capture problems of Go.

If a search result can be preserved along with the *inversion*, then the search result would remain unchanged if no change exists in the *inversion* area. Especially on a 19 $\times$ 19 board, This can save search time greatly.

Another purpose of *inversions* is to find double-purpose moves. In the right of the Fig. 7.1, the two white stones on the board are nearly captured by black. If we search for them independently, then we can immediately find that both cannot escape from the corresponding *ladder* even if white plays first. However, white has an option to play a *ladder breaker*, which is a move to prevent a *ladder* if the other player ignores the move. This is a clear example of an inversion.

White has the possibility of saving one of the two stones if a *ladder breaker* for both ladders exists. We can find a double *ladder breaker* by searching for the overlap of inversions (marked with squares). Additionally, in fact, the intersection marked with a cross is a double *ladder breaker*. We can find that such a candidate moves automatically.

7.3 Resolving Dependencies

To search for a way to reach a goal of a given problem, generally we rely on evaluation functions that show the preferable path to the goal. Many sophisticated search algorithms exist. An efficient search for the way to the goal can be done only if the goal is single and has a fast and accurate evaluation function. However, we often must search for problems with multiple partial goals with dependencies.

In such cases, there would be two solutions. One is to manage to find out one global goal which resembles the combination of the multiple goals. This solution is efficient, but in many cases, it is difficult to construct a global goal and/or the constructed goal is inaccurate. Another more accurate and intuitive solution is to search for each partial goal and then resolve the dependencies.

We can detect the dependencies by finding overlapped inversions. In this chapter, we describe an algorithm for detecting inversions and finding the dependencies of multiple goals. Resolving the dependencies of different local searches makes it possible to limit the search space of compound goal search problems because we can limit the candidate moves using inversions. This is critical for search speed: if we search for two atomic subgoals in a naive algorithm, then the number of candidate moves and the search depth are both doubled, yielding a time-consuming algorithm.

7.4 Related Work

The relevancy zone is defined for capture problems of Go. It is an area in which a played stone might affect the search result. Our idea of inversion is related strongly to the relevancy zone [83].

Cazenave solved transitive connection problems of Go in [21]. This is the first paper describing solving compound subgoals by searching subgoals and resolving dependencies. Cazenave used the generalized threats search to obtain a *trace*. *Traces* are the intersections involved with the tests performed during the local subgoal search. They used *trace* to detect the dependencies between two connections, and then solved transitive connection problems.

In [61], the relevancy zone is extended and used to solve compound goals for connections, captures, and living. However, the paper does not precisely describe how to ascertain relevancy zones.

In the game of Go, Kishimoto and Müller tried isolating a problem into isolated sub problems [38]. They divide a problem based on theoretical analysis of the problem. Their approach was theoretically more strict than our approach, but it requires the board position to enclose the sub problems which would less likely occur in real game play.

7.5 Searching Inversions

7.5.1 Algorithm for Searching Inversions

This section presents a description of the algorithm for finding the inversion of a given subgoal.

First, we do a normal search for the subgoal, and find the winner and the loser, and the relevancy zone. As depicted in Fig. 7.2, we recursively reduce the set of candidate moves for inversion.

An inversion is defined as a set of the loser's moves which can enable the loser to win if the winner ignores the moves. A weak inversion is defined as a set of the loser's moves for which, if followed using a winner's pass, the result of the subgoal search becomes UNKNOWN.

As portrayed in Fig. 7.2, we recursively reduce the set of candidate moves for inversion.

```
enum Player { BLACK = 0, WHITE = 1, UNKNOWN = 2};
struct result {
  Player winner;
  MoveSet rz;
};
void inversion(node n, int inv_max_nodes,
              MoveSet &inv, MoveSet &weak_inv, int λorder)
{
  result orig_result;
  orig_result = LambdaDfpn(n, MAX_NODES, λorder);
  MoveSet inv_cand_rest = orig_result.rz;
  MoveSet inv_cand_checked = ∅;
  while ( inv_cand_rest is not ∅ ) {
    // select one from inv_cand
    move m = select_from(inv_cand_rest);
    makeMove(m);
    result inv_res = LambdaDfpn(n, inv_max_nodes, λorder);
    unmakeMove();
    inv_cand_checked = inv_cand_checked +{m};
    if (inv_res.winner == orig_result.winner) {
      // m is not an inversion
      inv_cand_rest =
        inv_cand_rest ∩ inv_res.rz − inv_cand_checked;
    } else if (inv_res.winner == UNKNOWN) {
      // m is a weak inversion
      weak_inv = weak_inv+{m};
      inv_cand_rest = inv_cand_rest −{m};
    } else {
      // m is a strong inversion
      inv = inv +{m};
      inv_cand_rest = inv_cand_rest −{m};
    }
  }
}
```

Fig. 7.2 Pseudo-code for inversion search

7.6 Experimental Results and Analysis

Tables 7.3 and 7.4 show the experimental results. The subgoal is to either capture or save the target *block* that is marked by a triangle.

An important parameter for searching inversion is the threshold to give up search and return UNKNOWN. If the number of node expansions (which is identical to the number of function calls to LambdaDfpnMIDAt and LambdaDfpnMIDDf in Fig. A.8 and A.9) exceeds the threshold, then the λ df-pn returns UNKNOWN.

The candidate was a strong inversion if the result was inverted. The candidate was a weak inversion if the result was UNKNOWN. Strong inversions are marked with crosses; weak inversions are marked with squares. The number of node expansions is also shown. The threshold marks when to give up the search for a move and treat the move as a weak inversion.

We have tested node expansion thresholds of 1000 and 10000. Observations show that, if λ^3 capture search succeeds, then the number of node expansions rarely exceeded 10000, and for λ^2 capture search, node expansions were mostly less than 1000. The board below "thr1000" shows the result of threshold 1000, and "thr10000" shows the result of threshold 10000.

For **prob01** and **prob02**, the union of weak and strong inversions seems meaningful for Go players within the threshold of 1000. However, for the remainder of the problems, it seems that 1000 is insufficient. The results of all problems seem meaningful for Go players if the threshold is 10000. For problems to find an inversion that makes it possible to capture a safe *block*, we should choose to use only the strong inversions. Additionally, for problems to find an inversion that makes a captured *block* able to escape, the union of both weak and strong inversion should be used.

To disprove a capture, the number of node expansions can become enormous, as in **prob04**. Nevertheless, even in such cases, a useful inversion can be found in a reasonable search nodes.

7.7 Conclusions and Future Work

We developed an algorithm for finding inversions and applied it to the capture problems of Go. Our first attempt was based on df-pn[93]. As described this chapter, we improved the performance further by using λ df-pn search.

We designed λ df-pn independently of the subgoals. It will be tested for other subgoals such as connecting stones and living.

Observations indicate that the use of weak inversion can save time for inversion search. For using this in real game play, we should search for a set of moves that changes the search result from UNKNOWN to WIN/LOSS. We temporarily call this a pseudo-inversion. Pseudo-inversions should be sought for problems for which the original search result is UNKNOWN within a search threshold.

This result is useful for future work for combination with MCTS and efficient implementation of the compound goal search.

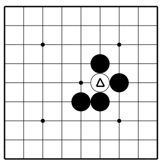
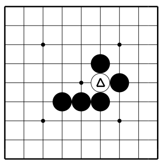
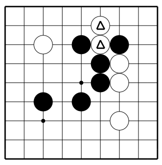
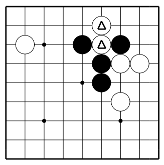
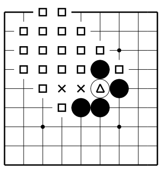
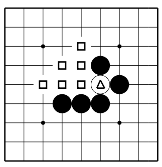
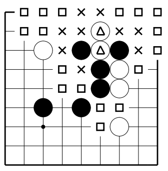
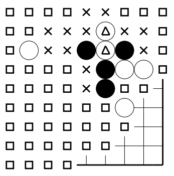
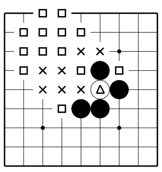
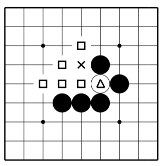
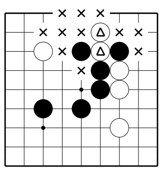
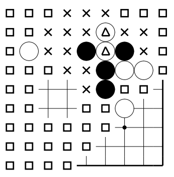
prob01	**prob02**	**prob03**	**prob04**
λ^2save Loss 41 nodes 	λ^2save Loss 41 nodes 	λ^2capture Loss 1691 nodes 	λ^2capture Loss 23782 nodes 
thr1000 18204 nodes 	thr1000 9656 nodes 	thr1000 18906 nodes 	thr1000 49836 nodes 
thr10000 157858 nodes 	thr10000 57824 nodes 	thr10000 36375 nodes 	thr10000 418758 nodes 

Fig. 7.3 List of Search Results for Inversions (1/2)

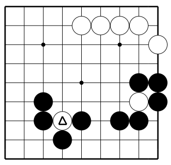
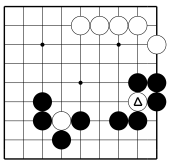
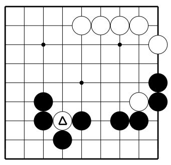
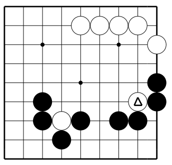
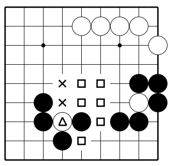
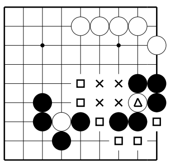
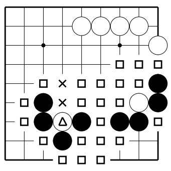
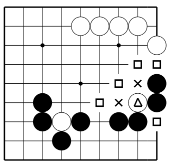
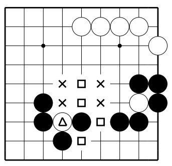
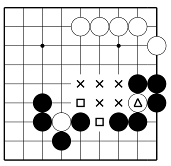
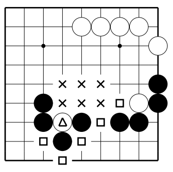
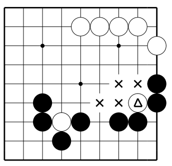
prob05A	**prob05B**	**prob06A**	**prob06B**
λ^2save Loss 28 nodes 	λ^2save Loss 651 nodes 	λ^2save Loss 784 nodes 	λ^2capture Win 882 nodes 
thr1000 7848 nodes 	thr1000 14395 nodes 	thr1000 30113 nodes 	thr1000 7119 nodes 
thr10000 48011 nodes 	thr10000 37946 nodes 	thr10000 117423 nodes 	thr10000 9688 nodes 

Fig. 7.4 List of Search Results for Inversions (2/2)

Chapter 8

Conclusions

8.1 Overview

As explained in the Introduction, a group of problems have proof trees sufficiently small for human beings to solve, but they remain difficult for existing computer programs. The difficulty of this group of problems is attributable to their large and uniform branching factors.

We developed two algorithms for such search spaces based on two approaches. One is based on finding a hidden hierarchy, which is actualized as the λ df-pn algorithm. The other is based on automatic selection of promising moves, which was formulated into *dynamic widening*. Both approaches are based on a proof number framework which uses the unused ability of the proof number search.

The capabilities of these two algorithms were tested and the limits of these approaches were measured.

8.2 Technical Contribution

We showed that the framework of proof number and disproof number can be used to switch among different search algorithms. This ability was used to implement the λ df-pn algorithm, which necessitated seamless switching between different search spaces with different hierarchies.

We proposed that discarding unpromising moves is possible using a simple technique: dynamic widening. We discovered that discarding information given by unpromising moves can improve the performance of the Proof Number search algorithm. In addition, experimental results revealed that, for capture problems of Go, only 7 best moves provide valid information for controlling search directions. The remaining legal moves give no useful information. This number differs for different target domains. This observation gives a new measure for examining the characteristics of search spaces.

An inversion search was also implemented. As explained in chapter B in the game of Go, it is difficult to strengthen overall play using results of local searches.

From the viewpoint of df-pn, we developed two domain-independent techniques that improve the df-pn performance. Actually, $lambda$ df-pn improves df-pn for almost all instances for most settings in the experiments; and dynamic widening works for cases with large and uniform branching factors. Especially for several difficult problems, the combination of $lambda$ df-pn and dynamic widening improves df-pn. Experimental results of capture prob-

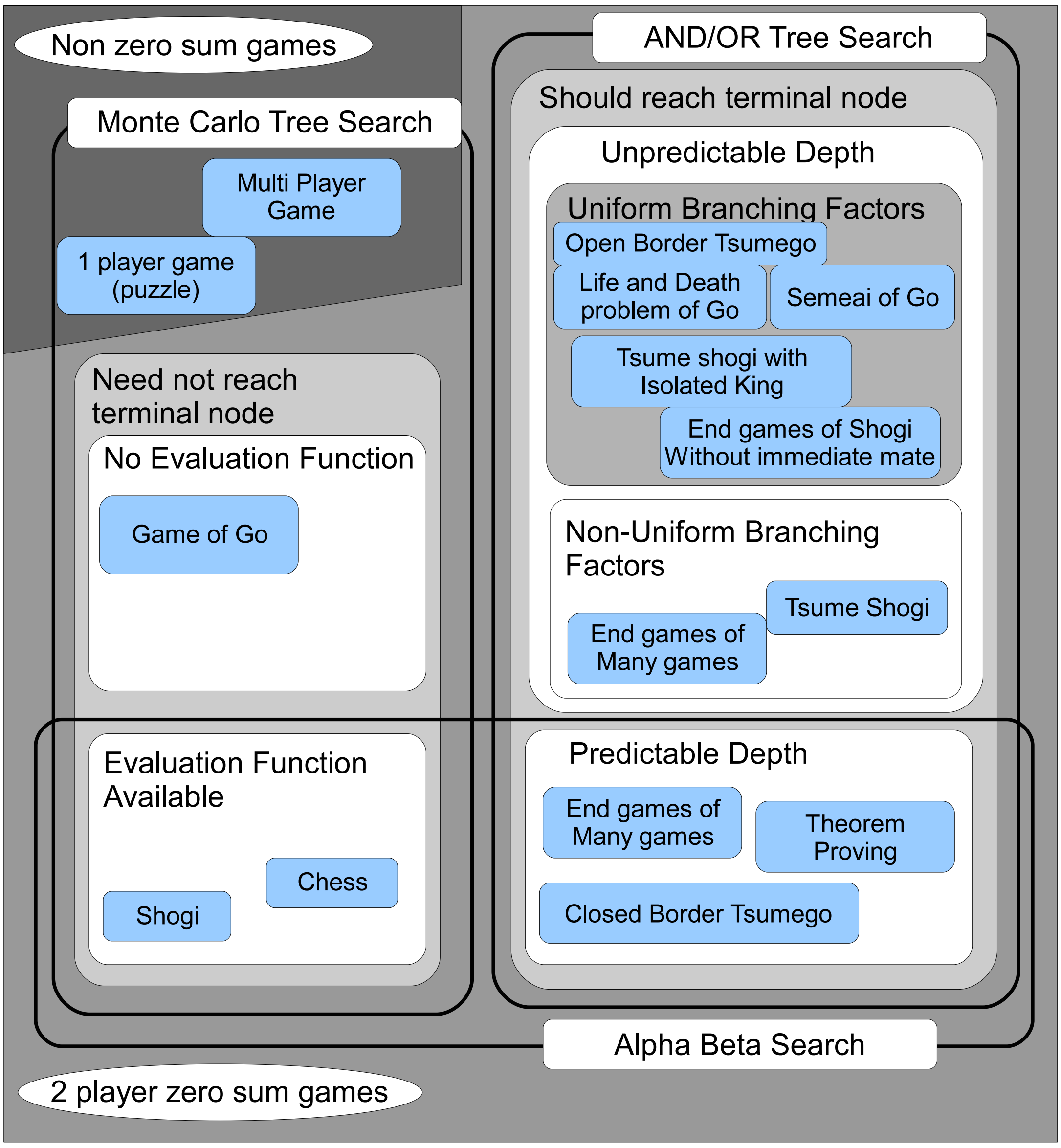


Fig. 8.1 The main contribution is the development of algorithms, which is suitable for the remaining difficult areas (upper right oval area shaded in gray).

lems of Go, show that our algorithms are promising for AND/OR trees in the game of Go, especially if no forward pruning is done.

8.3 Summary

Results clarified that, in domains of AND/OR tree search, there exist a group of problems that are difficult for computers to resolve. Problems of this group have a search space with large and roughly uniform branching factors. Existing AND/OR tree search algorithms showed weakness in solving the search space with uniform branching factors.

Two domain-independent techniques were proposed to tackle this weakness: two algorithms based on the approaches and measured the performance. The results were promising. We expect that our algorithm can be useful to solve other remaining difficult problems in the group.

Appendix A

Annotated Pseudo-codes

The pseudo-codes for df-pn and λ df-pn are provided in this appendix.

A.1 df-pn

For the attacker's nodes, ϕ stands for proof numbers and δ stands for disproof numbers. For defender's nodes, these roles are exchanged. (A similarity exists between this notation and minimax and nega-max.)

This notation is confusing at first glance, but can reduce unnecessary branches in codes, and is used in our implementation. However, it is instructive for readers who are unfamiliar with df-pn to try writing program without exchanging ϕ and δ in each iteration.

The Multiple Iterative Deepening (MID) function is the main function. The algorithm relies on the hash table. When all the entries in the hash table are used up, some data must be deleted. It is known that df-pn combined with good replacement scheme works well with limited amount of memory. The replacement scheme itself is an interesting topic. Nagai implemented *SmallTreeGC*[52] and used for his tsume-shogi solver[53].

The behavior of df-pn is illustrated in Figs. A.4 and A.5. The pseudo-codes and figures are helpful for implementing df-pn. However, this code is likely to work well only for trees or DAGs, and cycle detection must be implemented for games with repetitions. Although it is not shown in our pseudo-codes, we are using cycle detection described in Kishimoto's thesis[34].

A.2 λ df-pn

In λ df-pn, exchanging ϕ and δ are no longer necessary because the functions for the attacker and the defender are asymmetric.

Main iteration works as portrayed in Fig. A.6. For the attacker's node with λ order n, λ df-pn searches child nodes from order 0 to n.

However, because of the dominance portrayed in Eq. A.2, only order n moves are searched for the defender's node with λ order n, with one additional child node, which is the pass move of order $n-1$ and all moves of order n.

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

```
struct node { $\phi$; $\delta$; };                        //(dis-)proof numbers in
                                                          // dual notation
void MID(node $n$, int $\Phi$, int $\Delta$) {            //the main function
                                                          //Multiple Iterative Deepening
   ttLookup($n$);                                         //read node $n$ from the hash table
   if ($n.\phi >= \Phi$ || $n.\delta >= \Delta$) return;  //$\Delta$ and $\Phi$ are the thresholds
   if (IsTerminal($n$)) {                                 //if node $n$ is terminal,
      if (CurrentPlayerWin($n$)) {
         $n.\phi = 0$; $n.\delta = \infty$                //current player win
      } else {
         $n.\phi = \infty$; $n.\delta = 0$                //current player loss
      }
      ttStore($n$);                                       //record winner in the hash table
      return;                                             //and return
   }
   GenerateMoves($n$);                                    //prepares child nodes
   while ($\Phi >$ DeltaMin($n$) &&                       //search while within threshold
           $\Delta >$ PhiSum($n$)) {
      $n_c =$ SelectChild($n$, $\delta_2$);               //$n_c$: current best child
      $\Phi_c = \Delta + n_c.\phi -$ PhiSum($n$);         //$\Phi_c$, $\Delta_c$ are thresholds
      $\Delta_c =$ min($\Phi$, $\delta_2$+1);             // for next iteration
      MID($n_c$, $\Phi_c$, $\Delta_c$);                   //call next iteration
   }
   $n.\phi$ = DeltaMin($n$); $n.\delta$ = PhiSum($n$);    //update (dis)proof number
   ttStore($n$);                                          //store the result in the hash table
}
```

Fig. A.1 Pseudo-code of df-pn (1 of 3)

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

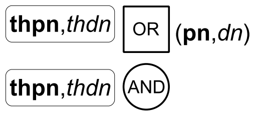
```
int DeltaMin(node $n$) {
    int $min = \infty$;
    for (each chilld $n_{child}$ of $n$) {
        ttLookup($n_{child}$);
        $min = \min(min, \delta)$;
    }
    return $min$;
}

int PhiSum(node $n$) {
    int $sum$ = 0;
    for (each child $n_{child}$ of $n$) {
        ttLookup($n_{child}$);
        $sum = sum + n_{child}.\phi$;
    }
    return $sum$;
}
```

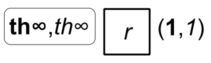
Fig. A.2 Pseudo-code of df-pn (2 of 3)

```
void SelectChild(node $n$, int &$\delta_2$) {           //returns the current best child
    node $best\_child$;                                   //best child is the child node
    $\delta = \delta_2 = \infty$; $\phi = 0$;             // with the smallest $\delta$
    for (each child $n_{child}$ of $n$) {
        ttLookup($n_{child}$);                            //read all $n_{child}$ from the hash table
        if ($n_{child}.\delta < \delta$) {                //find the smallest $\delta$
            $best\_child = n_{child}$;
            $\delta_2 = \delta$; $\delta = n_{child}.\delta$; $\phi = n_{child}.\phi$;   //and second smallest $\delta_2$
        }
        else if ($n_{child}.\delta < \delta_2$)
            $\delta_2 = n_{child}.\delta$;
        if ($n_{child}.\phi = \infty$)
            return $best\_child$;
        }
        return $best\_child$;
    }
```

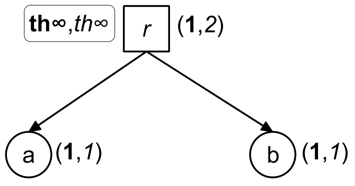
Fig. A.3 Pseudo-code of df-pn (3 of 3)



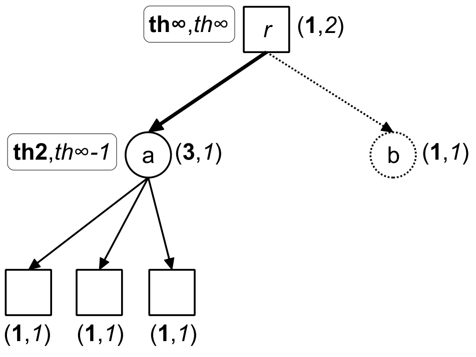
Squares signify OR nodes and circles denote AND nodes. On the left, thresholds for (dis-)proof numbers are shown as rounded squares. On the right, the (dis-)proof numbers are shown in brackets. Proof numbers (**pn**) are presented in **bold** and disproof numbers (*dn*) are shown in *italic*.



This is the root node.
First, the thresholds are both ∞. The df-pn would search until WIN/LOSS is determined. Here, ∞ **proof number** signifies an attacker's LOSS, and ∞ *disproof number* signifies an attacker's WIN.

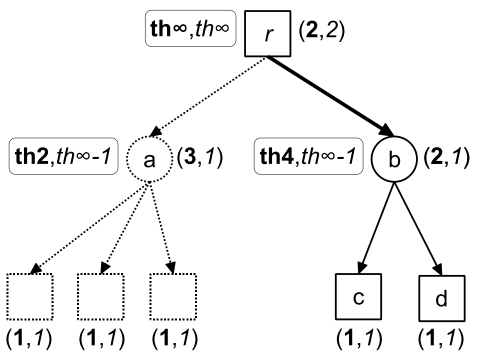


Expanded the root node *r*. For the child nodes, proof numbers (**pn**) and disproof numbers (*dn*) are initialized to (**1**, *1*).

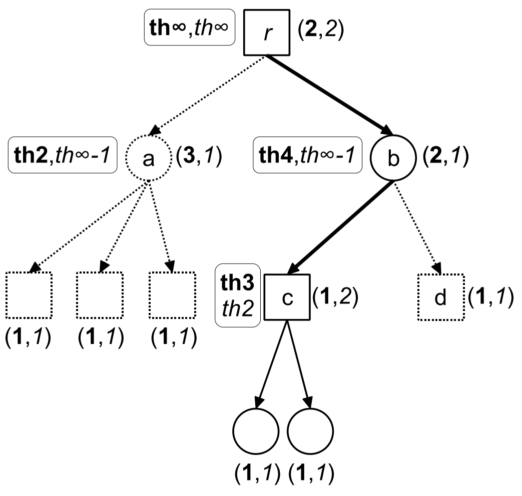


The attacker will choose the frontier node with the smallest proof number. (In this case, **pn** is **1** for both child.)
The **pn** of node **b** is **1**; therefore, the attacker will expand **b**, if **pn** of **a** reaches **2**. Consequently, the threshold for **pn** is set to **2**.
The threshold of *dn* at *r* is ∞, and the sum of *dn* of **a** and **b** must not reach ∞. Therefore, the threshold for *dn* is $(\infty - 1)$.

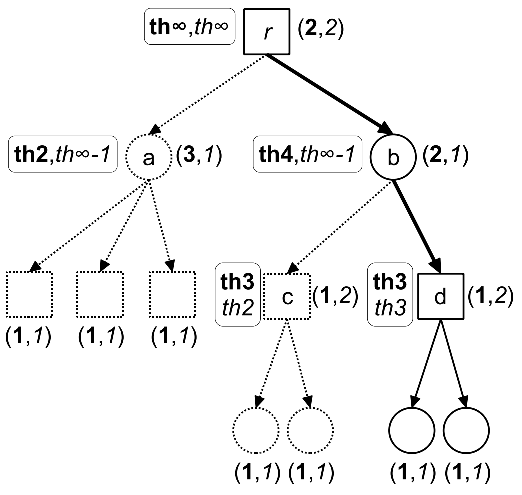
Fig. A.4 Mechanism of df-pn (1/2)



Now **pn** of node **a** is **3**, and had reached the threshold **2**, search will return to parent node *r*, and then go to node **b**.
The proof number of **a** is **3**; if **pn** of **b** reaches **4**, then the search should go to **a**. Therefore, the **pn** threshold is set to **4**.



The **pn** of node **b** is now **2** and is less than the threshold **4**. Search will expand the node **c**.
The *dn* of node **d** is *1*. Therefore, if *dn* of **c** reaches *2*, the defender is expected to expand **d**. Then the threshold for *dn* is set to *2*.
The sum must be less than **4** if the sum of the **pn** of node **c** and **d** reaches the **pn** threshold **4** of node **b**. Therefore, the**pn** threshold is set to **4-1=3**.



The *dn* of node **c** is now *2*, and reached the threshold *2*. Search now returns to **b**. At node **b**, both thresholds are OK, and node **d** is expanded.
The *dn* threshold will be *3* because the *dn* of **c** is *2*.
The sum of **pn** of **c** and **d** should be within the **pn** threshold at node **b**. Therefore, the threshold is set to **4-1=3**.

Fig. A.5 Mechanism of df-pn (2/2)

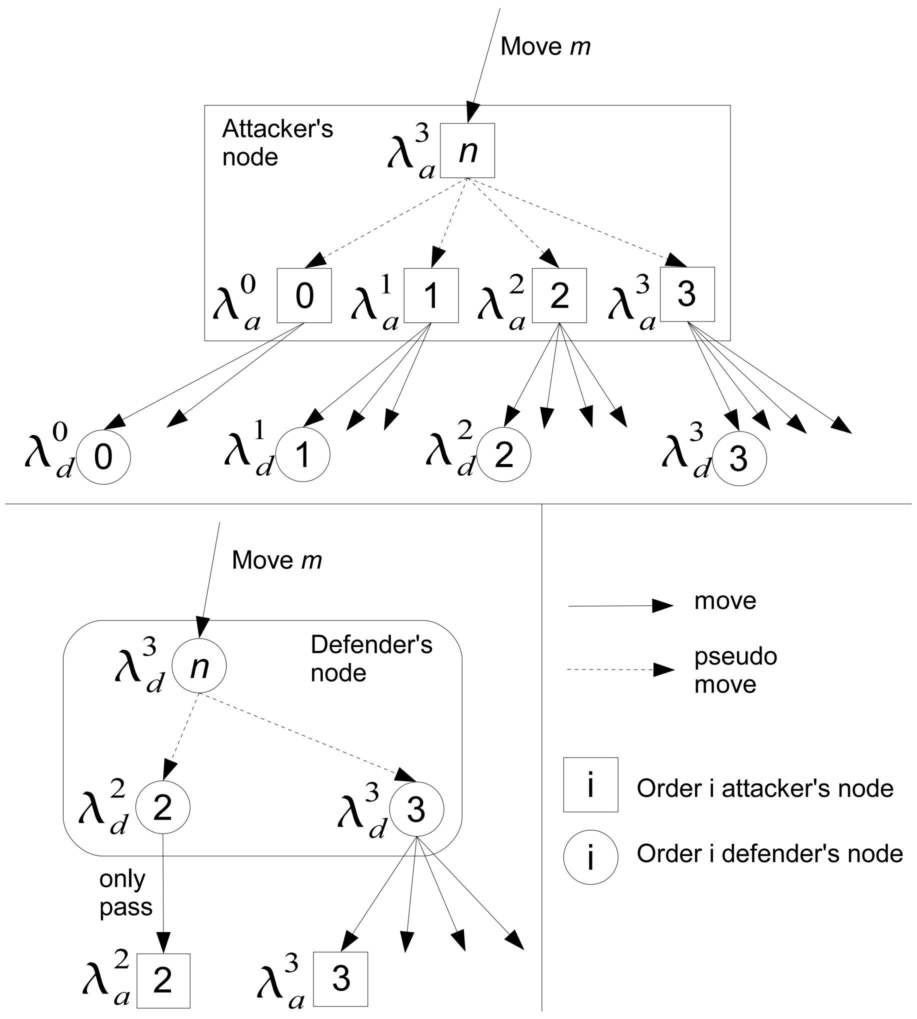


Fig. A.6 Attacker's node and Defender's node (revisited)

```
struct node { $\phi$[MAX$\lambda$+1]; $\delta$[MAX$\lambda$+1]; };   // Each node has (dis-)proof
                                                                    // for each order.
void Lambda0(node $n$) {                                            // Check $\lambda^0$
  if (attackerWin()) {                                              // If the Attacker had won,
    for (int $l$=0; $l$<=MAX$\lambda$; $l$++) {                      // $\lambda^0 = 1$,
      $n.\phi[l]$ = $\infty$; $n.\delta[l]$ = 0;
    }
  } else { $n.\phi[0]$ = 0; $n.\delta[0]$ = $\infty$; }             // else $\lambda^0 = 0$.
  ttStore($n$);
}
```

Fig. A.7 Pseudo-code of the λ df-pn algorithm. Node structure and λ^0 search.

```
void λDfpnMIDAt(node $n$, int $\lambda$, int $\Phi$, int $\Delta$) {    // Main function for attacker
   ttLookup($n$);                                                        // Create $n$ entry if needed
   if ($n.\phi[\lambda]$ >= $\Phi$ || $n.\delta[\lambda]$ >= $\Delta$) return;   // Return if threshold is exceeded
   generateMoves(order 0 to l);                                          // Search all moves with
                                                                         // order $l$ or lower
   while( DeltaMinAt($n$, $\lambda$) < $\Phi$ &&
            SumPhiAt($n$, $\lambda$) < $\Delta$ ) {
      node $n_c$ = SelectChildAt($n$, $\lambda$, $\phi$, $\delta$, $\delta_2$, $\lambda_c$);   // Select the best child
      $\Phi_c$ = $n_c.\phi[\lambda_c]$ + $\Delta -$ SumPhiAt($n$, $\lambda$);          // Calculate threshold
      $\Delta_c$ = min($\Phi$, $\delta_2 + 1$);
      λDfpnMIDDf($n_c$, $\lambda_c$, $\Phi_c$, $\Delta_c$);                         // $\lambda_c$ can be any order
   }                                                                     // lower than or equal to $l$
   $n.\phi[\lambda]$ = DeltaMinAt($n$, $\lambda$);                                // Update (dis-)proof numbers
   $n.\delta[\lambda]$ = SumPhiAt($n$, $\lambda$);
   ttStore($n$);                                                         // and store to the hash table.
}
int DeltaMinAt(node $n$, int $\lambda$) {                                // Return smallest proof number
   int min = $\infty$;
   for(int l=0;l<$\lambda$;l++) {                                        // irrespective of the order.
      for( each child $n_c$ of λorder==l ) {
         ttLookup($n_c$);
         min = min(min, $n_c.\delta[l]$);
      }
   }
   return min;
}
int SumPhiAt(node $n$, int $\lambda$) {                                  // Return sum of disproof number
   int sum = 0;
   for(int l=0;l<$\lambda$;l++) {                                        // irrespective of the order.
      for( each child $n_c$ of λorder==l ) {
         ttLookup($n_c$);
         sum += $n_c.\phi[l]$;
      }
   }
   return sum;
}
```

Fig. A.8 Pseudo-code of the λ df-pn algorithm Attacker's main iteration (MID: Multiple Iterative Deepening)

```
void $\lambda$DfpnMIDDf(node $n$, int $\lambda$, int $\Phi$, int $\Delta$) {   // Main function for defender
  ttLookup($n$);                                       // Create $n$ entry if needed
  if ($n.\phi[\lambda] >= \Phi$ || $n.\delta[\lambda] >= \Delta$) return;      // Return if threshold exceeded
                                                       // Search only order $\lambda^l$ moves
  generateMoves($l-1$ pass and all $l$ moves);         // and $\lambda^{l-1}$ pass.
  while( DeltaMinDf($n$, $\lambda$) $< \Phi$ &&
        SumPhiDf($n$, $\lambda$) $< \Delta$) {
    node $n_c$ = SelectChildDf($n$, $\lambda$, $\phi$, $\delta$, $\delta_2$, $\lambda_c$);   // Select the best child
    $\Phi_c$ = $n_c.\phi[\lambda_c]$ + $\Delta-$ SumPhiDf($n$, $\lambda$);     // Calculate threshold
    $\Delta_c$ = min($\Phi$, $\delta_2+1$);
    if($\lambda_c$ == 0) Lambda0($n_c$);                   // $\lambda_c$ can be $l$ or $l-1$
    else $\lambda$DfpnMIDAt($n_c$, $\lambda_c$, $\Phi_c$, $\Delta_c$);
  }
  $n.\phi[\lambda]$ = DeltaMinDf($n$, $\lambda$);                  // Update (dis-)proof numbers
  $n.\delta[\lambda]$ = SumPhiDf($n$, $\lambda$);
  ttStore($n$);                                        // and store them to the hash table.
}
int DeltaMinDf(node $n$, int $\lambda$) {                      // Smallest disproof number
  int min = $\infty$;
  for( each child $n_c$ of $\lambda$order==l                   // within $\lambda^l$ moves and $\lambda^{l-1}$ pass.
      and pass of $\lambda$order==l-1) {
    ttLookup($n_c$);
    min = min(min, $n_c.\delta[l]$);
  }
  return min;
}
int SumPhiDf(node $n$, int $\lambda$) {                        // Return sum of proof number
  int sum = 0;
  for( each child $n_c$ of $\lambda$order==$\lambda$ ) {                // within only $\lambda^l$ moves.
    ttLookup($n_c$);
    sum += $n_c.\phi[\lambda]$;
  }
  return sum;
}
```

Fig. A.9 Pseudo-code of the λ df-pn algorithm Defender's main iteration (MID: Multiple Iterative Deepening)

```
node SelectChildAt(node $n$, int $\lambda$, int &$\phi$,
                   int &$\delta$, int &$\delta_2$, int &$\lambda_c$) {
  node best_child;
  $\delta = \delta_2 = \infty$;
  for(int $l$=0;$l$<=$\lambda$;$l$++) {
    for( each child $n_c$ of $\lambda$order==$l$ ) {
      ttLookup($n_c$);
      if($n_c.\delta[l] < \delta$) {
        best_child = $n_c$;
        $\lambda_c = l$; $\delta_2 = \delta$; $\phi = n_c.\phi[l]$; $\delta = n_c.\delta[l]$;
      } else if ($n_c.\delta[l] < \delta_2$) $\delta_2 = n_c.\delta[l]$;
      if ($n_c.\phi[l]$==$\infty$) return best_child;
    }
  }
  return best_child;
}
```

```
node SelectChildDf(node $n$, int $\lambda$, int &$\phi$,
                   int &$\delta$, int &$\delta_2$, int $\lambda_c$) {
  node best_child;
  $\delta = \delta_2 = \infty$;
  for( each child $n_c$ of $\lambda$order==$l$
       and pass of $\lambda$order==$l$-1) {
    ttLookup($n_c$);
    if($n_c.\delta[l] < \delta$) {
      best_child = $n_c$;
      $\lambda_c = l$; $\delta_2 = \delta$; $\phi = n_c.\phi[l]$; $\delta = n_c.\delta[l]$;
    } else if ($n_c.\delta[l] < \delta_2$) $\delta_2 = n_c.\delta[l]$;
    if ($n_c.\phi[l]$==$\infty$) return best_child;
  }
  return best_child;
}
```

Fig. A.10 Pseudo-code of the λ df-pn algorithm select child node. The attacker selects the best child irrespective of the order. The defender selects the best child from λ^l moves and λ^{l-1} pass.

Appendix B

The game of Go

B.1 Overview

The game of Go is a two-person zero-sum game with perfect information. The exact origin of Go is not known but the game was already popular in China 2000 years ago. It is called Go or Igo in Japanese, baduk in Korean, and wéiqí in Chinese.

Go is one of the few two-player games for which many professional players exist. Among various two-player zero-sum perfect information games, only five games have a number of professional players. These are chess, checkers, Shogi (Japanese chess), Chinese chess, and Go.

Go is mainly played in east Asian countries. Recently, it is spreading among other countries in the world, largely because of the efforts of Asian professionals and the development of internet Go servers.

B.2 Rules

One professional Go player has stated that “Shogi is the best game ever invented by human beings, Go was invented by the God.” The simplest set of rules results in the most complex two-player perfect information board game.

The equipment used to play Go consists of a board with intersections, black stones, and white stones. In the official rules, a 19×19 board is used but the rules of Go do not rely on the board size. It can be played on a board of any size.

Rules of Go consist of the following six rules.

1. Players take turns placing stones on the intersections of a grid. The first player uses black stones; the second player uses white stones.
2. The aim of the game is to occupy more territory than the opponent.
3. Stones completely surrounded by opponent stones are captured and are removed from the board.
4. Placing a stone on an intersection is prohibited if it is already surrounded completely by the opponent.
5. There is an exceptional case in the move [*1]. prohibition rule.

[*1] In board games like chess, a “move” means a play by a player. However, in Go, stones would never move once placed on the board. Therefore, it is sometimes criticized that the word “play” should be used instead of

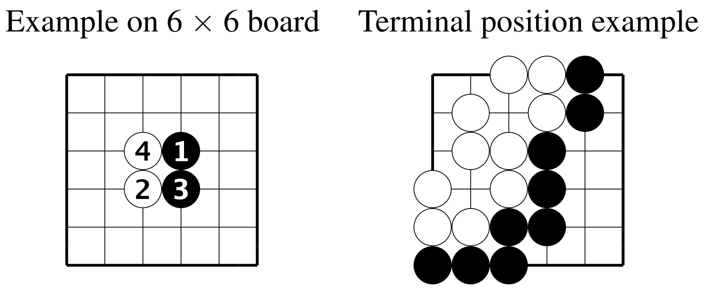


Fig. B.1 Example of a game on 6 × 6 board.

6. Repetition is prohibited.

B.2.1 Placing Stones

Players take turns placing stones on the intersections of a grid. The first player uses black stones; the second player uses white stones. A stone placed on the board remains in its position unless the opponent captures and removes it. The left board in Fig. B.1 shows an opening of 6 × 6 board game.

B.2.2 Territory

The aim of the game is to occupy more territory than the opponent. Two different rules exist in scoring.

The Japanese rules [70] use territory scoring, which counts of empty intersections surrounded only by each color of stones. In the right position portrayed in Fig. B.1, black has 11 points and white has 6 points on the board. However, in this scoring rule, stones that are captured during the game are kept as prisoners and added to the points. White captured 2 stones during the game and has two additional points. In the end, Black wins by 3 points.

The Chinese rules use area scoring [69] which counts the sum of total number of stones and size of surrounded areas. In the same position, black has 20 points and white has 16 points. Black has won by 4 points.

An additional discussion is provided about scoring rules later in this chapter.

B.2.3 Capturing Rule

The key rule of Go is the capturing of 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. Captured stones are removed from the board, and in Japanese rules, they are kept by the capturer as prisoners.

The position in the left of Fig. B.2 shows examples of captures. If black plays on A or B, it can capture white stones. Note that two stones are captured at once by playing on B.

"move". However, in this thesis, the word "move" is used to avoid confusion among game AI researchers.

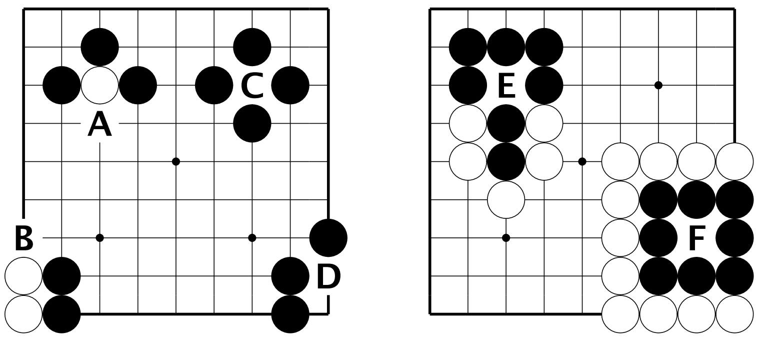


Fig. B.2 Examples of captures and prohibited moves.

B.2.4 Prohibited Moves

In consequence of the capturing rule, prohibited moves are defined. A move by a player that occupies the last liberty of one of his own blocks is prohibited. White is not allowed to play on C in the left of Fig. B.2. White can play on D because it has one more liberty.

B.2.5 Exception in Prohibited Moves

One exception exists in the move prohibition rule. That move is permitted if placing a stone on the last liberty of his own block results in capturing an opponent's block(s).

Point E in the right of Fig. B.2 is surrounded by black stones, but white can play on E because white can capture two black stones. Playing on F is also possible because all black stones around F will be captured by placing a white stone on F.

B.2.6 Avoiding Repetition (Ko Rule)

Repetitions of board positions are prohibited. A simple pattern that results in local repetition is depicted in Fig. B.3. These patterns are called *Ko*. Black can capture a white stone by playing on either one of A, B or C in the left position. Then, white can recapture the black stone by playing on A, B or C in the right position. To avoid repetition, white must play one move at some other points before recapturing the black stone.

Repetition positions could occur in more complex form than simple Kos These positions are called super Ko. Ko frequently occurs in real games, but super Ko is rare in that it appears less than once in 1,000 professional games. Most games can be played normally by merely prohibiting the repetition caused by Ko patterns.

A rule that prohibits any repetition is called the super Ko rule. The super Ko rule was used in Chinese rules, but it requires great care by the players and the referee to detect super Ko in human games. Therefore, super Ko rule is currently (in 2008) not used in official rules.

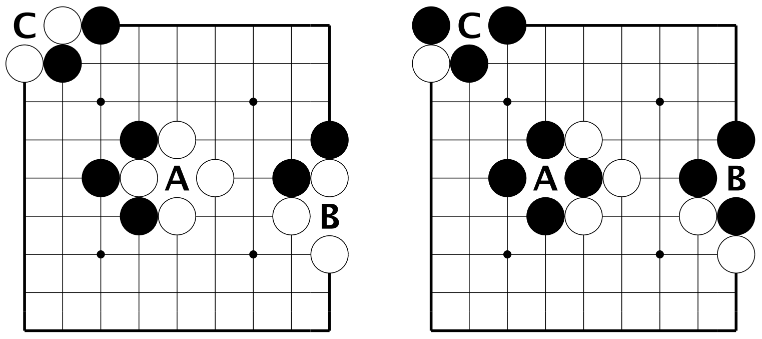


Fig. B.3 Examples of ko

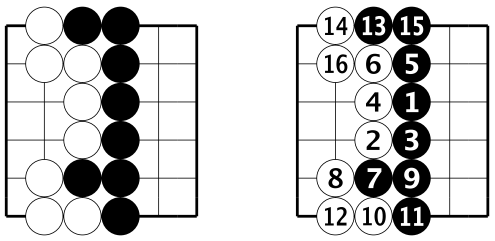


Fig. B.4 Normal end game of Go and the game record.

B.3 Development of the rule of Go

It remains difficult for a beginner to play Go properly immediately after learning the rules explained in the previous section. It is said that the time immediately after learning the rules is the most difficult point for human Go players.

We will try to explain what causes that difficulty. Additionally, we explain a brief history of the rule of Go, and discuss why the original simple rule was changed into seemingly complex rules.

B.3.1 Difficulty of the Go Endgame

One reason is the seemingly vague definition of the end game. Fig. B.4 presents an example of an end game position. However, it is difficult for beginners to see why the game has ended. One typical question from beginners is whether the game portrayed in Fig. B.5 is ended as black's win. Actually it did not. We will try to answer this question.

B.3.2 Life and Death of Stones

If the number of stones represents the score of the game, and each player places one stone at a time, then the difference of the score could only result from the capturing rule. Therefore, at the end of the game, the board is almost fully covered by stones which could never be

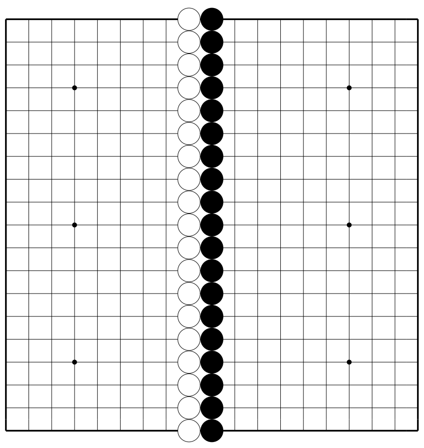


Fig. B.5 Is this game finished?

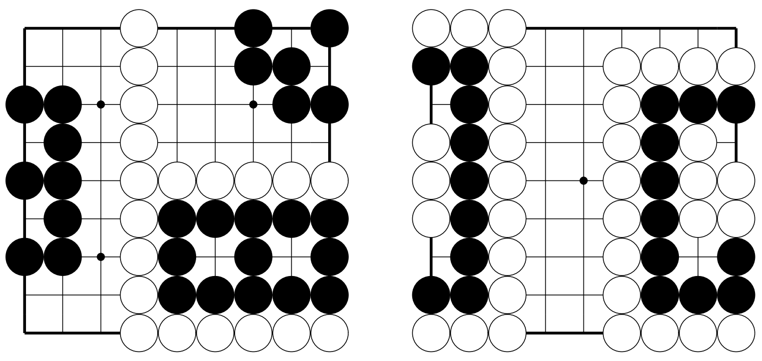


Fig. B.6 Stones living by two eyes and seki.

captured by the opponent. The only way to make stones safe is to make them *alive*.

One important fact that follows the rules of Go is that stones occasionally become alive. Living stones are stones that would never get captured (unless suicidal moves are played).

The black blocks of stones in the left of Fig. B.6 are examples of living stones. Each block has two eyes.

An eye is an empty point that is surrounded by stones of the same color. Playing on an enemy's eye is prohibited by the rule, unless the eye is the last liberty of the block. The block that has two or more eyes could never be captured because the opponent can only place one stone at a time. This is the normal mode of making stones *alive*.

Usually, stones become alive by having two eyes, but complex situations often occur in games. Sometimes stones can live without having two eyes. Stones in the right of Fig. B.6 are alive by *seki*. Seki can be said to be a kind of ZugZwang. Understanding the difference of dead stones and live stones in seki is often difficult for beginners.

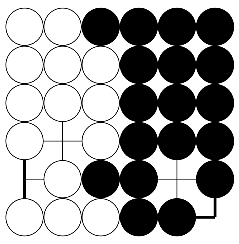


Fig. B.7 End game of pure Go.

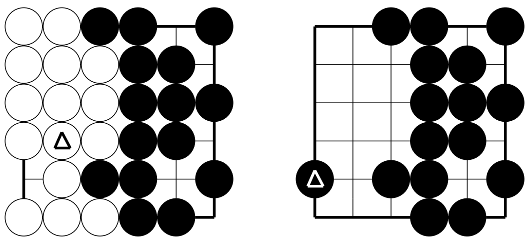


Fig. B.8 A suicidal move is played by white.

B.3.3 Pure Go Rule

It is presumed that in the original rule of Go, the aim of the game was to place as many stones on the board as possible. Recently, a group of Japanese professional Go players have proposed using *stone-filling-Go* or a *pure Go* rule instead of normal Go rule[31, 48]. The rule of *pure go* is thought to be close to the rule of the original form of Go. Fig. B.7 presents an example of end game of pure Go. A game of pure Go ends by passes from both players. The winner is decided by simple counting of stones.

Passes are played when no beneficial move remains. In Fig. B.7, if either player plays one more stone on the board, that would be a suicidal move. A whole white block would be captured (as shown on the right) if white plays the move marked with a triangle, as presented in Fig. B.8. Therefore, typically in an endgame of pure Go, all blocks remaining on the board have two eyes each.

B.3.4 Territory Counting and Area Counting

Soon Go players realize that playing until the apparent endgame is a waste of time if the pure Go rule is used. At some point in the game, reasonable moves remaining on the board would be simple moves that fill their own territory one by one. As depicted in Fig. B.9, after the left position, both players fill in their territory, and after the right position white passes several times while black continues the fill-in process.

Therefore, it is natural to end the game by agreement at the left position in the figure and determine the winner by adding the number of remaining empty points to the number of each stones. This is thought to be the origin of the area scoring rule, which is currently used

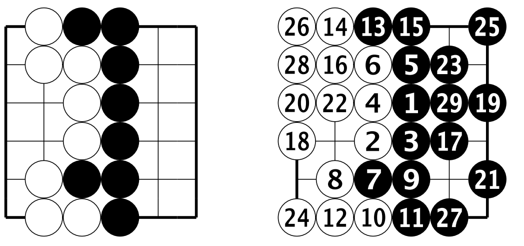


Fig. B.9 In the endgame of pure Go, both players merely fill in their territories.

in Chinese rules. The number of stones played by each player is basically equal (or black has one more stone). If the number of captured stones is recorded then the number of the difference of black and white stones can be calculated easily. Consequently, just counting the empty points of both sides results in a score difference that is about the same as that by area scoring. This is the territory scoring rule, which is used in the Japanese rule.

Only a small difference would occur in the score calculated according to the pure Go rule and the score calculated by area scoring or territory scoring because, in pure Go, each block must have two eyes each. For example, if all black stones are connected and white stones are separated into two blocks, then black would only have to keep two eyes, but white must keep four eyes in all. Despite this change in counting, area scoring or territory scoring is widely used because of its convenience.

In all rules, a territory is a zone in which all of an opponent's invasions can be prevented by capturing stones played in the zone. The stone would be captured if an opponent places a stone in a well surrounded area. To make a stone safe, typically stones must have two eyes, and in the left of the Fig. B.9 it is easy to disturb white to make two eyes in black's region. In other words, a region becomes a territory only if the owner of the region can capture any opponent stone played inside the region. Consequently, the difference of the score is caused solely by the capturing rule.

It is often difficult to determine if an invading stone can be captured or not. If pure Go rules are used, then the game ends by passes of both sides, therefore no difficulty arises in defining the end of a game. However, in today's rules, the endgame of Go is difficult for beginners, especially for the Japanese rules.

B.4 Miscellaneous

B.4.1 Ranking System

Unlike chess, an official rating system is not popular in Go because the strength of players can be roughly measured by the number of handicap stones. A ranking system of Go is based on *kyu* and *dan*.

A beginner who has just learned the rules of Go would be ranked as 25 kyu; the number decreases as the player improves. Two players with different kyu can play a roughly even game with handicap stones with the same number as the difference of kyu. For example, a 4-kyu player can play an even game against a 1-kyu player by placing 3 handicap stones.

The player would be shodan (which means 1 dan) if a 1-kyu player becomes stronger by 1 rank. Becoming a dan ranked player is the first goal for most Go players. After becoming a dan player, the number increases as the player improves.

B.4.2 Classic Books

Many classic Go books have been written in China and Japan starting from Wangyou Qingle Ji (忘憂清楽集), which was written around 1100 AD.

Wángyōu Qīnglé Jí (忘憂清楽集)

The oldest existing Go book was edited by Li Yimin (李逸民) around 1100 AD [1]. The title means "The Collection of the Pure Happiness of Forgotten Worries". Forgotten worries (忘憂) is a well known nickname for the game of Go.

It includes game records, josekis, tsumegos, and tactical problems. The book has important information related to the history of Go rules.

Xuánxuán Qíjīng (玄玄碁経)

Xuánxuán Qíjīng (玄玄碁経) is the second oldest existing Go book, published in 1347 [2]. The Japanese pronunciation is Gengen Gokyo. The editors were Yan Defu (厳徳甫) and Yan Tianzhang (晏天章), who were known as strong Go players of the age.

The book comprises several chapters.

1. Introduction to Go, including the history and development of Go, and about the book itself.
2. Introduction to playing the game of Go.
3. Joseki and game records. It has only historical value: in that era, two black and white stones were placed on the star points of the board before the game was started.
4. Tsumego and tactical problems. This is the most valuable part of the book. It is said that most important tactical techniques are described in the book. This chapter comprises three sections: a corner part, a side part, and a center part. Most of the problems in the last section "center part" are semeai or capturing problems, which we used as a part of our test sets.

Guānzǐ Pǔ (官子譜)

Guānzǐ Pǔ (官子譜) is another Chinese classic Go problem collection. Guānzǐ (官子) means endgame, or invasion. First edited and published in 1660 by Guo Bailing (過百齢) and Tao Shiyu (陶式玉) edited again and published in 1689.

It is a large collection of problems that consist of tsumego, endgame and tactical problems resulting in a total of 1478 problems. It includes several problems with no answer (seemingly promising moves do not work because of good reactions by the opponent). These problems are good for training human players because the positions are more realistic and occur in real games. However, it is difficult to use them directly for test sets for computer solvers because the goal of the problems is not clear.

Some problems are taken from Xuánxuán Qíjīng (玄玄碁経).

Gokyo Shumyo (碁経衆妙)

Gokyo Shumyo, written by Hayashi Genbi (林元美), was published in 1812 [3].

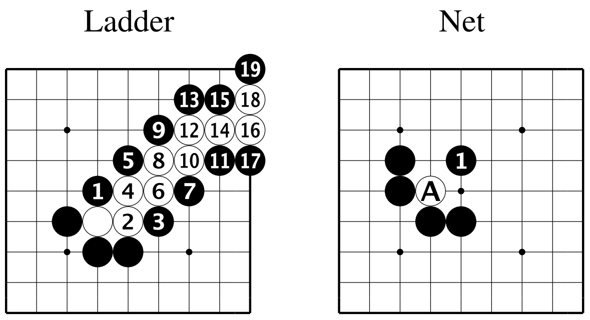


Fig. B.10 Ladder and net.

Hayashi Genbi was an 8-dan professional Go player and was the leader of Hayashi house, one of the four Go houses sponsored by the Japanese government during the Edo era.

Gokyo Shumyo consists of tsumego problems and tactical problems. Unlike other most classical Go books, which mainly consist of difficult problems for training strong players, Gokyo Shumyo focuses on more fundamental problems for novice Go players.

1. 生之部 (life) includes 103 tsumego problems for making life
2. 死之部 (death) includes 71 tsumego problems for killing
3. 劫之部 (ko) includes 90 problems that result in ko
4. 攻之部 (attack) includes 96 semeai problems
5. 追落之部 (attack to kill) includes 40 capturing/semeai problems
6. 盤之部 (connect) includes 74 connection problems
7. 挟之部 点之部 続之部 断之部 征之部 tactical and tsumego problems of specific classes

B.4.3 Local Problems in Go

Apart from overall play, making solvers for subproblems of Go presents interesting challenges for computers. Wolf [88, 90] and Kishimoto [34] developed a life and death solver; Thomsen developed a tactical problem solver[83].

It was effective for several games to decompose overall problems into subproblems and use search results of subproblems. Unfortunately for Go, searching local problems themselves often presents a difficult problem. For example, life and death problems remain difficult for computers if problems are not enclosed. Furthermore, it was not clear how to use results of local searches.

These search problems can be represented as AND/OR trees. Therefore, two-player zero-sum games are a good testbed for AND/OR tree research.

B.5 Go Terminology

For Go terms not listed herein, the sensei's library provides excellent information about Go [71].

approach ko A ko that requires an approach move(s) for one player to become a normal ko. An approach ko that requires one approach move is called a one-move approach ko; if two approach moves are required, it is a two-move approach ko, and so on.

atari A block that has only one remaining liberty is said to be in atari.

block Stones connect vertically or horizontally, and form blocks. Furthermore, they can be called *strings* or *dragons*.

board Go is played on a board with, officially, a 19×19 grid. However, 9×9 boards and 13×13 boards are used for beginners or short games.

dan Ranking of Go. Stronger than kyu players. Becoming shodan (= 1 dan) is the first goal of amateur Go players. Stronger players have a higher number of dans, with 8 dan being the strongest amateur player. Players can roughly play even games with the same number of handicap stones as the difference of dan of each player. Professional players also have dan rankings, with 9 dan the highest, but the rating system differs from that used for amateur players.

death A block that has no way to avoid being captured is said to be dead.

double ko These are called double ko if two mutually related ko are made. Double ko can be used as an infinite source of ko threats.

eye A 1×1 empty point surrounded by stones of the same color.

ko A simple typical case of repetition. An example is presented in Fig. B.3.

ko fight Fight for a ko. A sequence of moves that consists of capturing of ko and ko threats.

ko threats It is prohibited to recapture the stone in ko positions immediately. He must play a forcing threatening move at other position on the board if a player wants to recapture the target stone. These moves are called ko threats.

kyu Ranking of Go. According to the Japanese Go Association, a beginner of Go is rated as 25 kyu. The number decreases for stronger players. Players can roughly play even games with the same number handicap stones as their difference of kyu. For example, a 5-kyu player can play a 1-kyu player with a 4 stone handicap. Players stronger than 1 kyu would be rated as dan class players.

ladder A simple capturing technique portrayed in Fig. B.10.

liberty Empty points adjacent to blocks are called liberties.

life A block that could never be captured unless either of the players played a suicidal move is said to be alive. Typically a block with two or more eyes is alive.

net A simple capture technique depicted in Fig. B.10.

semeai A capturing race between two or more blocks. Semeai is an important sub-problem of Go.

super ko Repetition of board positions, including patterns other than simple ko.

super ko rule A rule that prohibits repetition of board positions. The Situational Super Ko (SSK) rule is used for most computer Go competitions, which regards positions as identical if the positions and player color are the same. The Positional Super Ko (PSK) rule regards positions as identical only if the board positions are the same. Actually, PSK is rarely used because it would yield counterintuitive results.

tesuji Tactical techniques; it sometimes simply means the correct move.

tsumego Problem of living or killing one or more blocks.

Appendix C
List of Problems

Problems are categorized according to λ order and are listed here. XXQJ denotes problems taken from Xuánxuán Qíjīng (玄玄碁経), Gokyo signifies problems taken from Gokyo Shumyo (碁経衆妙) and Master stands for problems taken from Master of Semeai.

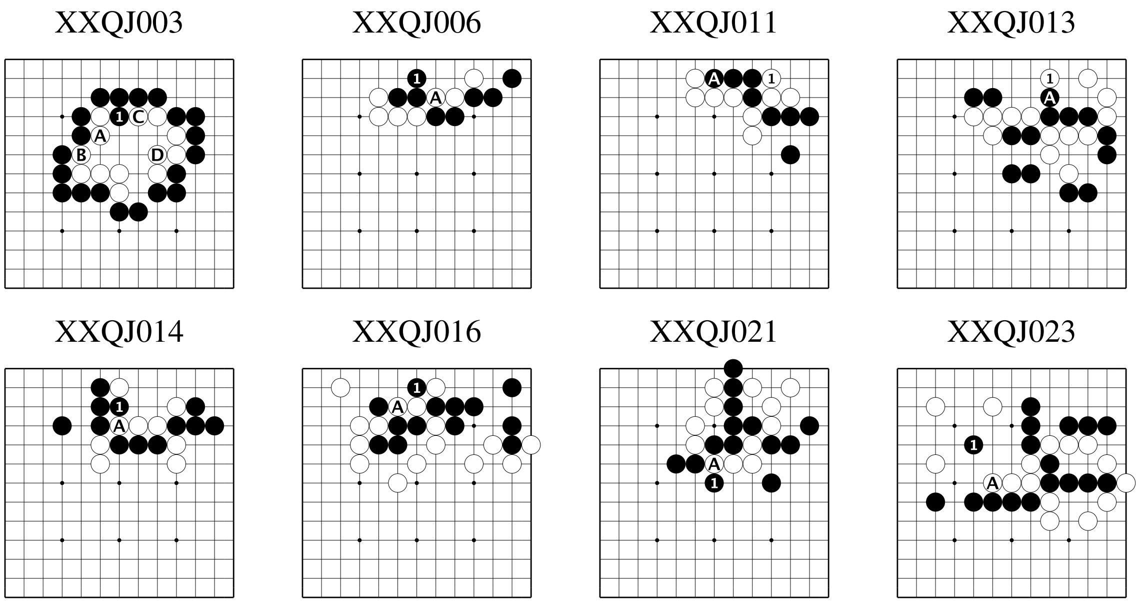
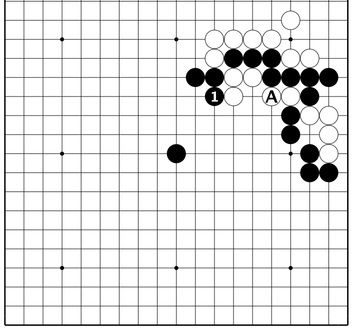


Fig. C.1 Classic λ^3 problems (1/2)

XXQJ024



XXQJ029

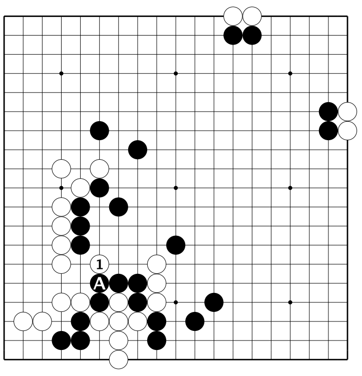
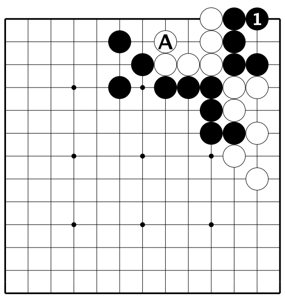
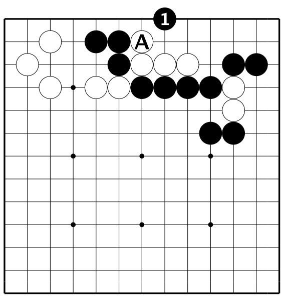


Fig. C.2 Classic λ^3 problems (2/2)

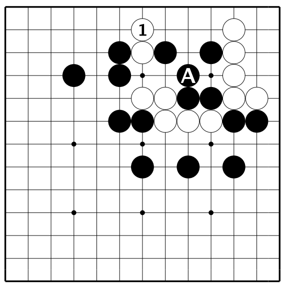
XXQJ007



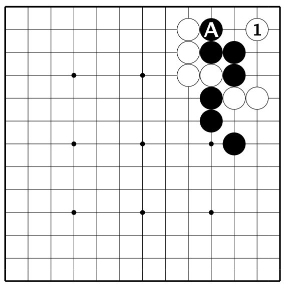
XXQJ012



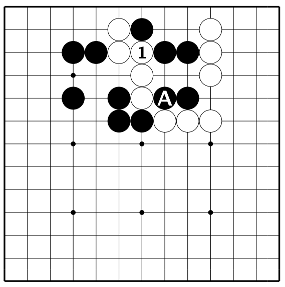
XXQJ018



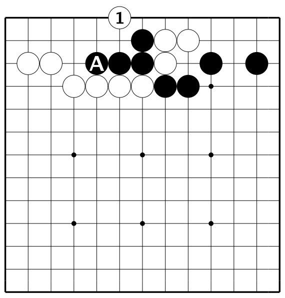
Gokyo023



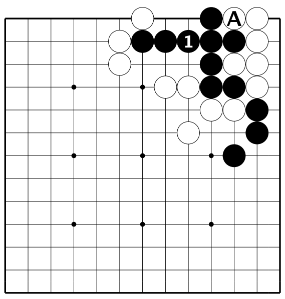
Gokyo073



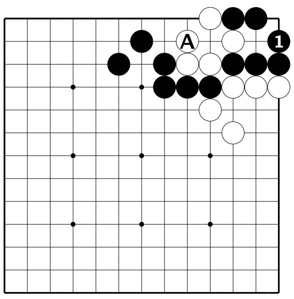
Gokyo075



Master001



Master010



Master024

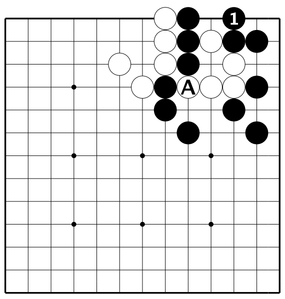


Fig. C.3 λ^4 problems (1/2)

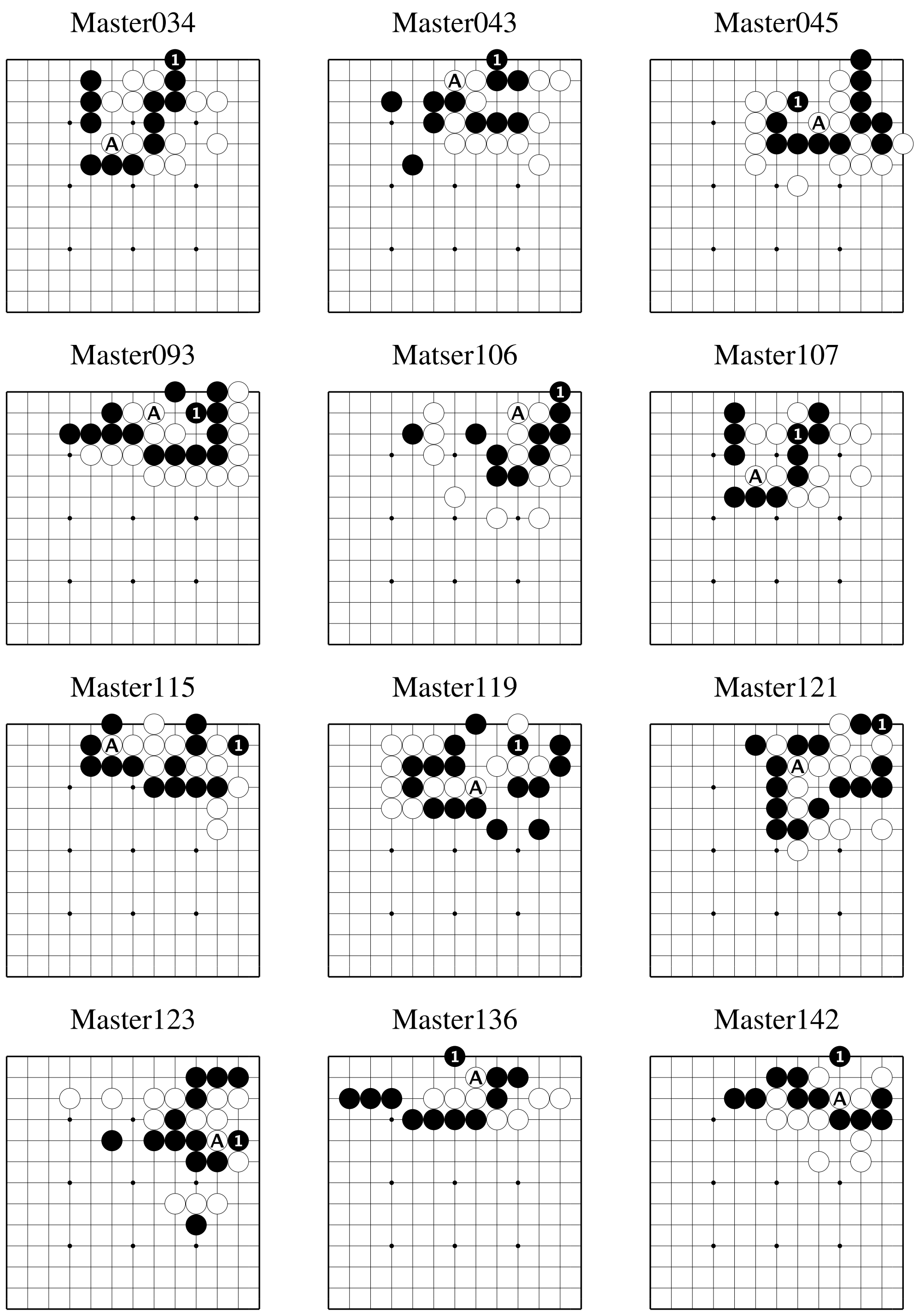
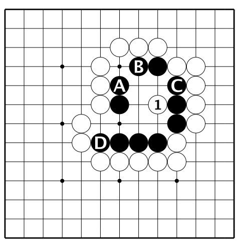
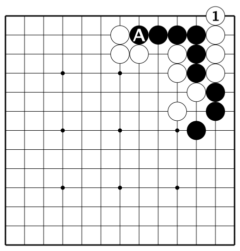


Fig. C.4 λ^4 problems (2/2)

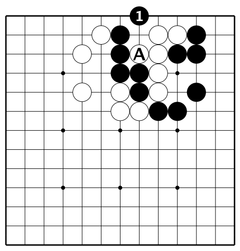
XXQJ004



Gokyo004



Gokyo046



Master071

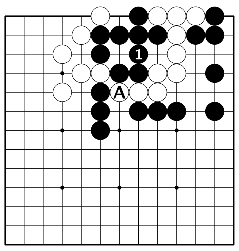


Fig. C.5 λ^5 problems

Bibliography

[1] Wàngyōu Qīnglè Jí (忘憂清楽集) (1100), (in Chinese).

[2] Xuánxuán Qíjīng (玄玄碁経) (1347), (in Chinese).

[3] Gokyo Shumyo (碁経衆妙) (1812), (in Japanese).

[4] Master of Semeai (攻め合いの達人), Japanese Go Association (2002), ISBN: 4818204722.

[5] Official homepage of the 11th Computer Olympiad (2006), `http://www.cs.unimaas.nl/olympiad2006/`.

[6] ALLIS, L. V. *Searching for Solutions in Games and Artificial Intelligence*, PhD thesis, University of Limburg, Maastricht, the Netherlands (September 1994).

[7] ALLIS, L. V., HERIK, VAN DEN H. J. and HUNTJENS, M. P. H. Go-Moku Solved by New Search Techniques., *Computational Intelligence*, **12** (1996), 7–23.

[8] ALLIS, L. V., MEULEN, VAN DER M. and HERIK, VAN DEN H. J. Proof-number search, *Artificial Intelligence*, **66**, 1 (1994), 91–124.

[9] AUER, P., CESA-BIANCHI, N. and FISCHER, P. Finite-time analysis of the multi-armed bandit problem, *Machine Learning*, **47** (2002), 235–256.

[10] BOUZY, B. and CAZENAVE, T. Computer Go: An AI-Oriented Survey, *Artificial Intelligence*, **132**, 1 (2001), 39–103.

[11] BOUZY, B. and HELMSTETTER, B. Monte Carlo Go developments, Proc. 10th Advances in Computer Games Conference (2003).

[12] BREUKER, D. M., HERIK, VAN DEN H. J., UITERWIJK, J. W. H. M. and ALLIS, L. V. A Solution to the GHI Problem for Best-First Search, Computers and Games (CG1999), Vol. 1558 of *Lecture Notes in Computer Science*, Springer (1999).

[13] BRÜGMANN, B. Monte Carlo Go (1993), Unpublished technical note.

[14] BURO, M. The Othello Match of the Year: Takeshi Murakami vs. Logistello, *ICGA Journal*, **20(3)** (1997), 189–193.

[15] CAMPBELL, M. The graph-history interation: on ignoring position history, Proc. 1985 ACM Annual Conference (1985).

[16] CAMPBELL, M., JR., A. J. H. and HSU, HSIUNG F. *Deep Blue*, ELSEVIER (2002), 97–123.

[17] CAZENAVE, T. Iterative Widening, IJCAI-01 Proceedings, Vol. 1 (2001).

[18] CAZENAVE, T. A Generalized Threats Search Algorithm, Computers and Games 2002, Vol. 2883 of *Lecture Notes in Computer Science*, Springer (2002).

[19] CAZENAVE, T. Generalized Widening, Proc. 16th European Conference on Artificial Intelligence (ECAI'2004) (2004).

[20] CAZENAVE, T. and HELMSTETTER, B. Combining tactical search and Monte-Carlo in the game of go, IEEE Symposium on Computational Intelligence and Games (2005).

[21] CAZENAVE, T. and HELMSTETTER, B. Search for transitive connections, *Information*

Sciences, **175**, 4 (November 2005), 284–295.

[22] CAZENAVE, T. and JOUANDEAU, N. A Parallel Monte-Carlo Tree Search Algorithm, Computers and Games (CG2008), Vol. 5131 of *Lecture Notes in Computer Science*, Springer (2008).

[23] CHASLOT, G. M. J. B., WINANDS, M. H. M. and HERIK, VAN DEN H. J. Parallel Monte-Carlo Tree Search, Computers and Games (CG2008), Vol. 5131 of *Lecture Notes in Computer Science*, Springer (2008).

[24] COULOM, R. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search, Proc. 5th International Conference on Computer and Games (2006).

[25] DONNINGER, C. Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs, *ICCA Journal*, **16**, 3 (1993), 137–143.

[26] FINNSSON, H. and BJÖRNSSON, Y. Simulation-Based Approach to General Game Playing, Proc. 23rd AAAI Conference on Artificial Intelligence, AAAI 2008 (2008).

[27] GELLY, S., WANG, Y., MUNOS, R. and TEYTAUD, O. Modification of UCT with patterns in Monte-Carlo Go, Technical Report 6062, INRIA (2006).

[28] GNU Go Homepage `http://www.gnu.org/software/gnugo/`.

[29] GRIMBERGEN, R. A Survey of Tsume-Shogi Programs Using Variable-Depth Search, Computers and Games, Vol. 1558 of *Lecture Notes in Computer Science*, Springer (1999).

[30] HASHIMOTO, K. Microcosmos (1986).

[31] ISHIKURA, N., UMEZAWA, Y., KUROTAKI, M. and HYODO, T. Go Lecture in the University of Tokyo (東大教養囲碁講座), Koubun sha (2007), (in Japanese).

[32] KATO, H. and TAKEUCHI, I. Parallel Monte-Carlo Tree Search with Simulation Servers, Proc. 13th Game Programming Workshop (GPW08), Vol. 2008 of *IPSJ Symposium Series* (2008), (in Japanese).

[33] KAWANO, Y. Using Similar Positions to Search Game Trees, Games of No Chance (ed.Nowakowski, R. J.), Vol. 29 of *MSRI Publications*, Cambridge University Press (1996).

[34] KISHIMOTO, A. *Correct and Efficient Search Algorithms in the Presence of Repetitions*, PhD thesis, University of Alberta (March 2005).

[35] KISHIMOTO, A. and MÜLLER, M.

[36] KISHIMOTO, A. and MÜLLER, M. Df-pn in Go: An Application to the One-Eye Problem, Advances in Computer Games 10, Kluwer Academic Publishers (2003).

[37] KISHIMOTO, A. and MÜLLER, M. A General Solution to the Graph History Interaction Problem, Proc. 19th National Conference on Artificial Intelligence (AAAI'04), AAAI Press (2004).

[38] KISHIMOTO, A. and MÜLLER, M. Dynamic Decomposition Search: A Divide and Conquer Approach and its Application to the One-Eye Problem in Go, Proc. IEEE Symposium on Computational Intelligence and Games (CIG'05) (2005).

[39] KISHIMOTO, A. and MÜLLER, M. Search versus knowledge for solving life and death problems in Go, Proc. 20th National Conference on Artificial Intelligence (AAAI-05), AAAI Press (2005).

[40] KLOETZER, J., IIDA, H. and BOUZY, B. The Monte-Carlo Approach in Amazons, Proc. Computer Games Workshop 2008 (2007).

[41] KNUTH, D. E. and MOORE, R. W. An analysis of Alpha–Beta Pruning, *Artificial Intelligence*, **6**, 4 (1975), 293–326.

[42] KOCSIS, L. and SZEPESVÁRI, C. Bandit Based Monte-Carlo Planning, 17th European Conference on Machine Learning (ECML 2006), Vol. 4212 of *Lecture Notes in Computer Science*, Springer (2006).

[43] KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, **27**, 1 (1985), 97–109.

[44] KORF, R. E. Linear-Space Best-First Search, *Artificial Intelligence*, **62**, 1 (1993), 41–78.

[45] LAI, T. L. and ROBBINS, H. Asymptotically efficient adaptive allocation rules, *Advances in Applied Mathematics*, **6** (1985), 4–22.

[46] LORENTZ, R. J. Amazons Discover Monte-Carlo, Computers and Games (CG2008), Vol. 5131 of *Lecture Notes in Computer Science*, Springer (2008).

[47] M. H. M. WINANDS, Y. B. and SAITO, J.-T. Monte-Carlo Tree Search Solver, Computers and Games (CG2008), Vol. 5131 of *Lecture Notes in Computer Science*, Springer (2008).

[48] MEIEN, O. Pure Go (純碁), Weekly Go (newspaper), from Jun. 28th to Sept. 13th 1999 (1999), (in Japanese).

[49] MÜLLER, M. Proof-set search, Technical Report TR-99-20, Electrotechnical Laboratory, Tsukuba, Japan (1999).

[50] MÜLLER, M. Computer Go, *Artificial Intelligence*, **134**, 1–2 (2002), 145–179.

[51] NAGAI, A. A new AND/OR Tree Search Algorithm Using Proof Number and Disproof Number, Complex Games Lab Workshop (1998).

[52] NAGAI, A. A new depth-first search algorithm for AND/OR trees, Master's thesis, Department of Information Science, University of Tokyo (1999).

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

[54] NAGAI, A. and IMAI, H. Proof for the equivalence between some best-first algorithms and depth-first algorithms for AND/OR trees, Proc. Korea–Japan Joint Workshop on Algorithms and Computation.

[55] NAGAI, A. and IMAI, H. Application of df-pn+ to Othello endgames, Proc. Game Programming Workshop 99 (1999).

[56] NAGAI, A. and IMAI, H. Application of df-pn Algorithm to a Program to Solve Tsume-shogi Problems, *Transactions of Information Processing Society of Japan*, **43**, 6 (June 2002), 1769–1777, (in Japanese).

[57] NEWBORN, M. *Kasparov versus deep blue: computer chess comes of age*, Springer–Verlag New York, Inc. (1997).

[58] OKABE, F. Application for solving tsume shogi problem by route branch number, Proc. Game Programming Workshop 2005 (GPW05), No. 15 (November 2005), in Japanese.

[59] PALAY, A. J. *Searching with probabilities*, PhD thesis, Carnegie-Mellon University (1983).

[60] PAWLEWICZ, J. and LEW, L. Improving Depth-First PN-Search: $1 + \varepsilon$ Trick, Proc. 5th International Conference on Computers and Games (CG 2006), Vol. 4630 of *Lecture Notes in Computer Science* (2007).

[61] RAMON, J. and CROONENBORGHS, T. Searching for compound goals using relevancy zones in the game of Go, Fourth International Conference on Computers and Games (eds.Herik, van den J., Björnsson, Y. and Netanyahu, N.), Ramat-Gan, Israel (2004), ICGA.

[62] SATO, Y. and TAKAHASHI, D. A Shogi Program Based on Monte-Carlo Tree Search, Proc. 13th Game Programming Workshop (GPW08), Vol. 2008 of *IPSJ Symposium Series* (2008), (in Japanese).

[63] SCHADD, M. P. D., WINANDS, M. H. M., HERIK, VAN DEN H. J., CHASLOT, G. M. J. B. and UITERWIJK, J. W. H. M. Single-Player Monte-Carlo Tree Search, Computers and Games (CG2008), Vol. 5131 of *Lecture Notes in Computer Science*, Springer (2008).

[64] SCHAEFFER, J. *One Jump Ahead*, Springer (1997).

[65] SCHAEFFER, J. Game over: Black to play and draw in checkers, *ICGA Journal*, **30**, 4 (2007).

[66] SCHAEFFER, J., BJÖRNSSON, Y., BURCH, N., KISHIMOTO, A., MÜLLER, M., LAKE, R., LU, P. and SUTPHEN, S. Solving Checkers, Proc. 19th International Joint Conference on Artificial Intelligence (IJCAI-05) (2005).

[67] SCHAEFFER, J., BURCH, N., BJÖRNSSON, Y., KISHIMOTO, A., MÜLLER, M., LAKE, R., LU, P. and SUTPHEN, S. Checkers Is Solved, *Science* (July 2007), 1144079+.

[68] SCHIJF, M., ALLIS, L. V. and UITERWIJK, J. W. H. M. Proof-Number Search and Transpositions, *ICCA Journal*, **17**, 2 (1994), 63–74.

[69] Chinese Rules section in Sensei's Library `http://senseis.xmp.net/?ChineseRules`.

[70] Japanese Rules section in Sensei's Library `http://senseis.xmp.net/?JapaneseRules`.

[71] Sensei's Library `http://senseis.xmp.net/`.

[72] SEO, M. The C^* Algorithm for AND/OR Tree Search and its Application to a Tsume-Shogi Program, *IPSJ SIG Notes. ICS*, **95**, 23 (1995), 103–110, (in Japanese).

[73] SOEDA, S. *Game Tree Search Algorithms based on Threats*, PhD thesis, The University of Tokyo (September 2006).

[74] SOEDA, S., KANEKO, T. and TANAKA, T. *Dual Lambda Search and its Application to Shogi Endgames* (2005).

[75] SOEDA, S., YOSHIZOE, K., KISHIMOTO, A., KANEKO, T., TANAKA, T. and MÜLLER, M. λ Search Based on Proof and Disproof Numbers, *Information Processing Society of Japan (IPSJ) Journal*, **48**, 11 (2007), 3455–3462, in Japanese.

[76] SOEDA, S., YOSHIZOE, K. and TANAKA, T. Efficient Implementation of the Lambda Search based on Proof Numbers, Proc. 11th Game Programming Workshop (2006), (in Japanese).

[77] STURTEVANT, N. R. An Analysis of UCT in Multi-player Games, Computers and Games (CG2008), Vol. 5131 of *Lecture Notes in Computer Science*, Springer (2008).

[78] TANABE, Y., YOSHIZOE, K. and IMAI, H. A Study on Security Evaluation Methodology in Finger Vein Authentication Systems, 2008 Symposium on Cryptography and Information Security (2008), Distinguished paper award, (in Japanese).

[79] TANABE, Y., YOSHIZOE, K. and IMAI, H. Extension of Wolf Search Algorithm in Finger Vein Authentication Systems, Proc. 2009 Symposium on Cryptography and Information Security (2009), (in Japanese).

[80] TANAKA, T. Complete analysis of a board game "SIMPEI", Transactions of Information Processing Society of Japan, Vol. 48 (2007), (in Japanese).

[81] THOMSEN, T. MadLab website `http://www.t-t.dk/madlab/problems/`

index.html.

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

[83] THOMSEN, T. Lambda-Search in Game Trees — with Application to Go, Computers and Games (CG 2000), Vol. 2063 of *Lecture Notes in Computer Science*, Springer (2002).

[84] UEDA, T., HASHIMOTO, T., HASHIMOTO, J. and IIDA, H. Weak Proof-Number Search, Computer and Games (CG 2008), Vol. 5131 of *Lecture Notes in Computer Science*, Springer (2008).

[85] WERF, VAN DER E., HERIK, VAN DEN J. and UITERWIJK, J. Solving Go on Small Boards, *ICGA Journal*, **26**, 2 (2003), 92–107.

[86] WINANDS, M., UITERWIJK, J. and HERIK, VAN DEN H. PDS-PN: A New Proof-Number Search Algorithm: Application to Lines of Action, Computers and Games (eds.J. Schaeffer, M. M. and Björnsson, Y.), Vol. 2883 of *Lecture Notes in Computer Science*, Springer (2003).

[87] WINANDS, M., UITERWIJK, J. and HERIK, VAN DEN H. An Effective Two-Level Proof-Number Search Algorithm, *Theoretical Computer Science*, **313**, 3 (2004), 511–525.

[88] WOLF, T. The Gotools program and its computer-generated tsume go database, Proc. 1st Game Programming Workshop, Vol. 1994, Hakone, Japan (1994).

[89] WOLF, T. About problems in generalizing a tsumego program to open positions, Proc. 3rd Game Programming Workshop, Vol. 1996, Hakone, Japan (1996).

[90] WOLF, T. Forward pruning and other heuristic search techniques in tsume go, *Special issue of Information Sciences*, **122**, 1 (2000), 59–76.

[91] WOLF, T. and SHEN, L. Checking Life & Death Problems in Go I: The Program ScanLD, *ICGA Journal*, **30**, 2 (2007), 67–74.

[92] YOSHIMOTO, H., YOSHIZOE, K., KANEKO, T., KISHIMOTO, A. and TAURA, K. Monte Carlo Go Has a Way to Go, Proc. 21st National Conference on Artificial Intelligence (AAAI-06) (2006).

[93] YOSHIZOE, K. A Search Algorithm for Finding Multi-Purpose Moves in Sub Problems of Go, Proc. 10th Game Programming Workshop (GPW05), No. 15 (11 2005).

[94] YOSHIZOE, K. A New Proof-Number Calculation Technique for Proof-Number Search, Computers and Games (CG 2008), Vol. 5131 of *Lecture Notes in Computer Science*, Springer (2008).

[95] YOSHIZOE, K., KISHIMOTO, A. and MÜLLER, M. Lambda Depth-First Proof Number Search and Its Application to Go., Proc. 20th International Joint Conference on Artificial Intelligence (IJCAI-07) (2007).

[96] INOUE INSEKI(井上因碩) Igo Hatsuyo Ron (囲碁発陽論) (1713).