Title	証明数を用いたゲーム木探索の新たなパラダイム
Author(s)	張，嵩
Citation	
Issue Date	2019-06
Type	Thesis or Dissertation
Text version	ETD
URL	http://hdl.handle.net/10119/16064
Rights	
Description	Supervisor：飯田　弘之，先端科学技術研究科，博士

Doctoral Dissertation

A New Paradigm of Game Tree Search Using Proof Numbers

by

Zhang Song

Supervisor: Professor Dr. Hiroyuki Iida

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
Information Science

June, 2019

Supervised by

Prof.Dr.Hiroyuki Iida

Reviewed by

Prof. Shun-Chin Hsu

Prof.Dr.Tsan-Sheng Hsu

Dr.Kiyoaki Shirai

Dr.Minh Le Nguyen

Dr.Kokolo Ikeda

Abstract

Conspiracy number and proof number are two game-independent heuristics in a game-tree search. The conspiracy number is proposed in Conspiracy Number Search (CNS) which is a MIN/MAX tree search algorithm, trying to guarantee the accuracy of the MIN/MAX value of a root node. It shows the scale of "stability" of the root value. The proof number is inspired by the concept of conspiracy number, and applied in an AND/OR tree to show the scale of "difficulty" for proving a node. It is first proposed in Proof-Number Search (PN-search) which is one of the most powerful algorithms for solving games and complex endgame positions. The Monte-Carlo evaluation is another promising domain-independent heuristic which focuses on the analysis based on random sampling of the search space. The Monte-Carlo evaluation does not reply on any prior knowledge of human and has made significant achievements in complex games such as Go.

In this thesis, we select the conspiracy number search, the proof number search and the Monte-Carlo tree search as three example search algorithms with domain-independent heuristics to study its relations and differences, and finally propose a new perspective of the game tree search with domain-independent heuristics. The relations and differences of the three search algorithms mentioned can be summarized as follows. The Monte-Carlo tree search uses Monte-Carlo evaluations for the leaf nodes to indicate the most promising node for expansion. In other words, the Monte-Carlo evaluation can be regarded as a detector to obtain the information beneath the leaf nodes to forecast the promising search direction in advance. In contrast, the conspiracy number search and the proof number search tend to use the indicators corresponding to the structure or the shape of the search tree that has already been expanded. Therefore, it can be regarded as forecasting the promising search direction according to the information above the leaf nodes. As a natural induction of such understanding of the game tree search using domain-independent heuristics, we may get some improvements by combining the conspiracy number or the proof number idea with the Monte-Carlo evaluation into a search algorithm, which can be

considered as a combination of “the information above leaf nodes” and “the information beneath the leaf nodes”.

Our research focuses on applying or refining domain-independent heuristics such as the conspiracy number, the proof number and the Monte-Carlo evaluation to achieve such three purposes: (1) enhancing current search algorithm. For this purpose, we proposed the so-called Deep df-pn search algorithm to improve df-pn which is a depth-first version of PN-search by forcing a deeper search with a parameter. The experiments with Connect6 show a good performance of Deep df-pn. (2) Analyzing and visualizing game progress patterns for better understanding of games and master thinking way, such as showing the analysis of the game progress for learners in Chinese Chess tutorial system. For this purpose, we proposed the so-called Single Conspiracy Number method for long term position evaluation in Chinese Chess and obtained good results. (3) Studying the relations and differences between the conspiracy number, proof number and the Monte-Carlo evaluation and combining “the information above leaf nodes” and “the information beneath the leaf nodes” to propose a new search algorithm with domain-independent heuristics named probability-based proof number search. A series of experiments show that probability-based proof number search outperforms other famous search algorithms for solving games and endgame positions.

Keyword: combinatorial game, domain-independent heuristic, game solver, Monte-Carlo evaluation, game progress pattern

Acknowledgments

Firstly, I would like to express my special thanks of gratitude to my supervisor professor Hiroyuki Iida who is not only a outstanding scientist in computer science but also a grand master in Japanese Shogi. Under his supervision, I have obtained so much passion and interest in the study of games and artificial intelligence. During my PhD work, professor Iida not only gave me gold opportunities to communicate with other outstanding professors and researchers in the world, but also supported me a lot in my daily life whenever I had problem. I have learned so much from him, and I appreciate him so much.

Secondly, thanks professor Jaap ven den Herik who hosted me in Leiden University for my minor research and helped me so much on our research papers. Also thanks Matej Gruid and Victor Allis who encouraged me and gave me a lot of good advice on my research when I was in Leiden University.

Thirdly, thanks professor Kokolo Ikeda and secretary Setsuko Asakura for the kind help to my study and daily life in Iida lab. Thanks the committee members of my PhD defense: professor Tsan-Sheng Hsu, professor Kiyoaki Shirai and professor Minh Le Nguyen for their excellent work. Also thanks my postgraduate supervisor professor Deng Ansheng for his invitations to Dalian Maritime University. Thanks all the students in Iida lab: Mr Zuo Long, Dr Xiong Shuo, Mr Zhai He, Mr Ye Aoshuang and so on. Thanks all the staff of JAIST.

Next, I will thank my parents who gave me good education and great support in my life. They not only raised me up but also taught me how to be a better man. I love them.

Special Thanks to Ms Xue Yawen, my girl friend. She came to JAIST two years before me and waited for me until I enrolled. Without her, I would not start my PhD study and enjoy my life so much. She is the source of my confidence, courage and happiness. Thank you so much. You are my forever lover, best friend and soul mate. I love you.

Finally, thanks China Scholarship Council for the financial support of my PhD work.

Today is not easy. Tomorrow may be more difficult. But the day after tomorrow will be fantastic. Keep going and enjoy!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Games are classified by the following properties: (1) Zero-sum: Whether the reward to all players sums to zero (in the two-player case, whether players are in strict competition with each other). (2) Information: Whether the state of the game is fully or partially observable to the players. (3) Determinism: Whether chance factors play a part (also known as completeness, i.e. uncertainty over rewards). (4) Sequential: Whether actions are applied sequentially or simultaneously. (5) Discrete: Whether actions are discrete or applied in real-time [7]. Games that are zero-sum, perfect information, deterministic, discrete and sequential are described as combinatorial games. Combinatorial games [5] includes one-player combinatorial puzzles such as Sudoku [31] [17], and no-player automata, such as Conway's Game of Life [16], but largely confined to two-player games that have a position in which the players take turns changing in defined ways or moves to achieve a defined winning condition, such as chess [22], checkers [42], and Go [33]. Combinatorial games are of interest in artificial intelligence, particularly for automated planning and scheduling [14]. By refining practical search algorithms such as the alpha–beta pruning search [27], we can implement a strong AI in such type of games. Many studies [39] [27] [43] show that based on such search framework, the strength of AI highly depends on the quality of the heuristic. To obtain reliable heuristic information of games, one possible way is to use expert's knowledge, usually in the form of scoring game positions. We call such heuristic as domain-dependent heuristic, because such heuristic is limited

to work on a certain game and hard to be generalized into other domains. Sometimes, domain-dependent heuristics are not reliable and even cannot be obtained especially in some complex games such as Go. In such case, the domain-independent heuristic is more advanced as it is obtained automatically in a more general way. In this thesis, we will review three typical domain-independent heuristics: Monte-Carlo evaluation [12], conspiracy number [32] and proof number [2]. We discuss the connections and differences of these heuristics in order to give a new perspective of the game tree search with domain-independent heuristics. Based on such new perspective, several enhancements on the search algorithms corresponding to the domain-independent heuristics are achieved, and new applications of the domain-independent heuristics are proposed. Finally, the ultimate purpose is to combine the advantages of these domain-independent heuristics and propose a new advanced search algorithm: probability-based proof number search. We conduct experiments to evaluate the performance of the probability-based proof number search and obtain significant results in the end. The success of probability-based proof number search not only makes breakthrough in real applications, but also theoretically proves our understanding on the new perspective of game tree search in this thesis.

Starting from Deep Blue [9] defeating the Chess world champion in 1997, search algorithms in combinatorial games have been significantly developed from a brute-force large-scale search [30] to a very selective heuristic search. The search efficiency is significantly improved by both using the high-quality domain-dependent heuristic and extremely cutting off of the brunches of the game tree. While after AlphaGo [44] defeating the Go world champion in 2017, a new generation of the game tree search has been started. The Monte-Carlo tree search (MCTS) and its variants based on domain-independent sampling and simulations become more and more successful than other conventional approaches. The focus of Monte Carlo tree search is on the analysis of the most promising moves, expanding the search tree based on random sampling of the search space. The application of Monte-Carlo tree search in games is based on many playouts. In each playout, the game is played out to the very end by selecting moves at random. The final game result of each playout is then used to weight the nodes in the game tree so that better nodes are more likely to be chosen in future playouts. Different from the domain-dependent heuristic, such playout does not reply on any prior knowledge of human. Using such

domain-independent heuristic in a search algorithm combined with large-scale parallel computing and deep neural network, computer can master different games beyond human experts [46] [45]. In other words, the success of Monte-Carlo tree search in Go can be considered to be a success of domain-independent heuristics in the game tree search.

In fact, before Monte-Carlo tree search, there have been some successful game tree search algorithms using the domain-independent heuristics, such as conspiracy number search (CNS) [32] and proof number search (PN-search) [2]. The core idea of conspiracy number search is using a vector of conspiracy numbers to indicate the likelihood of the root taking on a particular value. More precisely, the conspiracy number is the minimum number of leaf nodes in the tree that must change their score (by being searched deeper) to result in the root taking on that new value [52]. For each iteration in the search process, the conspiracy number search always expands the most promising nodes indicated by the conspiracy numbers to make the game tree grow from an "unstable" state to a "stable" state in an expectantly fastest way. Different from the Monte-Carlo tree search, the conspiracy number search can be considered to be a search algorithm using both the domain-dependent heuristic (the scores of leaf nodes) and the domain-independent heuristic (the conspiracy number). Moreover, the conspiracy number is obtained, not based on the sampling or the simulations of the game, but based on the structure or the shape of a game tree that has already been expanded. The conspiracy number evaluates the "stability" of the root score, indicating a proper ending state of a search. Such theoretical concept is very promising, but suffers from a low efficiency and slow convergence in practical implementations, because it takes too much time and storage to compute and record the vector of conspiracy numbers for each node.

To incorporate the conspiracy number idea into a real application, the proof number search (PN-search) is proposed as a game solver searching on an AND/OR tree. Other than using the vector of conspiracy numbers, PN-search simplifies the conspiracy numbers into two numbers: a proof number (pn) and a disproof number (dn), showing the scale of difficulty in proving and disproving a node, respectively. More precisely, for all unsolved nodes (leaf nodes), the proof number and disproof number are 1. For a winning node, the proof number and disproof number are 0 and infinity, respectively. For a non-winning node, it is the reverse. For internal nodes, the proof number and disproof number are back

propagated from its children following MIN/SUM rules: at OR nodes, the proof number equals the minimum proof number of its children, and the disproof number equals the summation of the disproof numbers of its children. It is the reverse for AND nodes. For each iteration, going from the root to a leaf node, PN-search selects either the child with the minimum proof number at OR nodes, or the child with the minimum disproof number at AND nodes. Finally, it regards the leaf node arrived at as the most proving node for expansion. Without scoring the game positions, the proof number search completely uses the domain-independent heuristic based on the structure of an AND/OR tree. It becomes one of most successful search algorithms for solving games and endgame positions.

In this thesis, we select the conspiracy number search, the proof number search and the Monte-Carlo tree search as three example search algorithms with domain-independent heuristics to study the relations and the differences between the conspiracy number, the proof number and the Monte-Carlo evaluation. Based on the discussion, we will find a new perspective of the game tree search with domain-independent heuristics. According to the introduction above, the Monte-Carlo tree search uses Monte-Carlo evaluations as domain-independent heuristics for the leaf nodes to indicate the promising search node for expansion. In other words, the Monte-Carlo evaluation can be regarded as a kind of detector to obtain the information beneath the leaf nodes to forecast the promising search direction in advance. In contrast, the conspiracy number and the proof number are indicators based on the structure or the shape of the search tree that has already been expanded. In other words, the conspiracy number and the proof number forecast the promising search direction by using the information above the leaf nodes. As a natural induction of such understanding of game tree search with domain-independent heuristics, there should be potential improvements by combining the conspiracy number or the proof number idea with the Monte-Carlo evaluation in the search algorithm. It can be considered as a combination of using the "information above the leaf nodes" and the "information beneath the leaf nodes". A typical search algorithm among this category is the Monte-Carlo proof number search, using Monte-Carlo evaluations for the leaf nodes and proof number search rules to expand the search tree. However, the Monte-Carlo proof number search is not a good combination of the Monte-Carlo evaluation and the proof number search, because Monte-Carlo evaluation leads to real numbers but the proof num-

ber search updating rules are proposed for integer numbers, which causes the information loss. Based on such theoretical consideration, we propose a new search algorithm named probability-based proof number search (PPN-search). We conduct experiments to verify the effectiveness of the probability-based proof number search. Experimental results show that probability-based proof number search outperforms other existing approaches in solving games or endgame positions. The success of probability-based proof number search not only makes breakthrough in real applications, but also theoretically prove our new perspective on the game tree search with domain-independent heuristics.

1.2 Research Questions

Our research focuses on applying and refining domain-independent heuristics, such as the conspiracy number, the proof number and the Monte-Carlo evaluation, to achieve such three goals: (1) Enhancing current search algorithm. For this purpose, we propose the so-called Deep df-pn search algorithm to improve df-pn, an efficient variant of the proof number search by forcing a deeper search with a parameter. It shows a good performance in solving positions of Connect6. (2) Analyzing and visualizing game progress patterns for better understanding of games and master's thinking way. For this purpose, We propose the so-called single conspiracy number method for long term position evaluations in Chinese Chess and obtain good results. (3) Based on the theory of "the information above the leaf nodes" and "the information beneath the leaf nodes", we propose a new advanced search algorithm for solving games and endgame positions named probability-based proof number search to improve the efficiency of the original one. We conduct a series of experiments to confirm that the probability-based proof number search is more efficient than other search algorithms for solving games, such as the proof number search, the Monte-Carlo proof number search, the UCT solver and the pure MCTS solver.

1.3 Structure of the Thesis

The dissertation includes 6 chapters: introduction, literature review, Deep df-pn, single conspiracy number, probability-based proof number search, and conclusion. In this dissertation, one grand argument is proposed, then several new findings and further discussions

are incorporated to support this argument.

Chapter 1. Introduction. We select the conspiracy number search, the proof number search and the Monte-Carlo tree search as three example search algorithms with domain-independent heuristics to introduce the development of game-independent heuristics. Base on the introduction, we propose such argument: conspiracy number and proof number are two game-independent heuristics using "the information above leaf nodes" corresponding to the structure or the shape of the part of the search tree that has already been expanded to indicate the most promising node for expansion, while Monte-Carlo tree search method "the information beneath the leaf nodes" obtained by simulations to indicate the most promising node for expansion. There are high similarity and closed connection between the conspiracy number, the proof number and the Monte-Carlo evaluation. Therefore, we could get potential improvements by combining the conspiracy number or the proof number idea with Monte-Carlo evaluation. This can be regarded as the combination of 'the information above leaf nodes" and "the information beneath the leaf nodes".

Chapter 2. Literature Review. In this chapter, we introduce related works before our study. We classify the game-independent heuristics into three categories: "using the information above the leaf nodes", "using the information beneath the leaf nodes" and "combining the information above and beneath the leaf nodes". The "using the information above the leaf nodes" category includes the conspiracy number search, the proof number search, df-pn and Deep proof number search. The "using the information beneath the leaf nodes" category includes the Monte-Carlo tree search and the Monte-Carlo tree search solver which is a variant of Monte-Carlo tree search for solving endgame positions. The "combining the information above and beneath the leaf nodes" includes the Monte-Carlo proof number search which is a variant of proof number search combining with Monte-Carlo evaluations.

Chapter 3. Deep df-pn. Depth-first proof-number search (df-pn) is a powerful variant of proof number search algorithms, widely used for AND/OR tree search or solving games. However, df-pn suffers from the seesaw effect, which strongly hampers the efficiency in some situations. In this chapter, we propose a new depth-first proof number algorithm called Deep depth-first proof-number search (Deep df-pn) to reduce the seesaw effect in df-pn. The difference between Deep df-pn and df-pn lies in the proof number or

disproof number of unsolved nodes. It is 1 in df-pn, while it is a function of depth with two parameters in Deep df-pn. By adjusting the value of the parameters, Deep df-pn changes its behavior from searching broadly to searching deeply. The chapter shows that the adjustment is able to reduce the seesaw effect convincingly. For evaluating the performance of Deep df-pn in the domain of Connect6, we implemented a relevance-zone-oriented Deep df-pn that worked quite efficiently. The experimental results indicate that improving efficiency by the same adjustment technique is also possible in other domains.

Chapter 4. Single Conspiracy Number. Single Conspiracy Number (SCN) is a variant concept of conspiracy number and proof number which indicates the difficulty of a root node changing its MIN/MAX value to a certain score. It makes up the drawbacks of conspiracy number on computing complexity and storage cost, which can be easily applied into different search frameworks. This chapter explores the potential usage of SCN as a long-term position evaluation to understand in-depth game progress patterns. Chinese Chess is chosen as a test bed for this study, whereas a strong open source AI engine 'Xiangqi Wizard' is used. It is implemented with alpha-beta search and modified to produce SCNs during the search process. Experiments are conducted on different types of positions including tactical positions, drawn positions and opening positions. The experimental results show that SCN is more consistent and accurate for long-term position evaluation than the conventional way using evaluation function values only. One application of SCN is the Chess tutorial system. Besides the evaluation function, SCN provides another scalar axis showing the changes of the game progress. It helps the users obtain more information about the game. Using SCN together with evaluation function values enables us to better understand game progress patterns.

Chapter 5. Probability-based Proof Number Search. Probability-based proof number search (PPN-search) is a game tree search algorithm improved from proof number search (PN-search) [2], with applications in solving games or endgame positions. PPN-search uses one indicator named "probability-based proof number" (PPN) to indicate the "probability" of proving a node. The PPN of a leaf node is derived from Monte-Carlo evaluations. The PPN of an internal node is back propagated from its children following AND/OR probability rules. For each iteration, PPN-search selects the child with the maximum PPN at OR nodes and minimum PPN at AND nodes. This holds from the

root to a leaf. The resultant node is considered to be the most proving node for expansion. In this chapter, we investigate the performance of PPN-search on P-game trees [28] and compare our results with those from other game solvers such as MCPN-search [38], PN-search, UCT solver, and the pure MCTS solver [54]. The experimental results show that (1) PPN-search takes less time and fewer iterations to converge to the correct solution on average, and (2) the error rate of selecting a correct solution decreases faster and more smoothly as the iteration number increases.

Chapter 6. Conclusion and Future Works. We summarize all the introductions and discussions above, and make a conclusion. In addition, several possible future works are presented.

Chapter 2

Literature Review

In this chapter, we introduce related works before our study. We classify the game-independent heuristics into three categories: "using the information above the leaf nodes", "using the information beneath the leaf nodes" and "combining the information above and beneath the leaf nodes". The "using the information above the leaf nodes" category stands for the class of search algorithms using domain-independent heuristics based on the structure or the shape of the search tree that has already been expanded, including the conspiracy number search, the proof number search, df-pn and Deep proof number search. "using the information beneath the leaf nodes" category stands for the class of search algorithms using domain-independent heuristics based on the Monte-Carlo evaluation, including the Monte-Carlo tree search and the Monte-Carlo tree search solver. For the "combining the information above and beneath the leaf nodes", we introduce a typical approach: the Monte-Carlo proof number search as a benchmark.

2.1 Using the Information above the Leaf Nodes

2.1.1 Introduction

In this section, we select the conspiracy number search, the proof number search and its variants as example search algorithms with domain-independent heuristics using the information above the leaf nodes. All the mentioned search algorithms use the conspiracy number or the proof number as a kind of indicators based on the structure or the shape of the search tree that has already been expanded. In other words, the conspiracy number

and the proof number forecast the promising search direction by using the information above the leaf nodes. Here, We also introduce the seesaw effect that highly hampers the efficiency of the original proof number search. Then the Deep proof number search is introduced to solve this problem.

2.1.2 Conspiracy Number Search

Conspiracy Number Search (CNS) [32] is a MIN/MAX tree search algorithm, trying to guarantee the accuracy of the MIN/MAX value of a root node. The likelihood of the root taking on a particular value is reflected in that value's associated conspiracy number. The conspiracy number is the minimum number of leaf nodes in the tree that must change their score (by being searched deeper) to result in the root taking on that new value [41]. The tree is grown in a way that restricts the set of likely root values. The formalism of the conspiracy number is given below [41].

When n is a leaf node, if the evaluation of n is v, then the conspiracy number associated with v is 0 and for all other values is 1. If that leaf node is terminal (its value is absolute), then the alternative values can be viewed as requiring ∞ conspirators.

When n is an internal node, consider breaking the set of conspiracy numbers into two groups: the numbers required to increase a node's value ($\uparrow$ *needed*) and those to decrease it ($\downarrow$ *needed*). Values below the MIN/MAX score in $\uparrow$ *needed* and above the MIN/MAX score in $\downarrow$ *needed* are assigned 0. $\uparrow$ *needed* and $\downarrow$ *needed* of an interior node's children can be combined to form the parent's conspiracy numbers, CN, using the following rules (with v being a value and m being the MIN/MAX value):

(a) If $v = m$,

$$CN(v) = 0.$$

(b) For a MAX node:

$$CN(v) = \sum_{\text{all children i}} \downarrow needed_i(v), \text{ for all } v < m.$$

$$CN(v) = \underset{\text{all children i}}{Min} \uparrow needed_i(v), \text{ for all } v > m.$$

(c) For a MIN node:

$$CN(v) = \underset{\text{all children i}}{Min} \downarrow needed_i(v), \text{ for all } v < m.$$

$$CN(v) = \underset{\text{all children i}}{\sum} \uparrow needed_i(v), \text{ for all } v > m.$$

Fig. 2-1, Fig. 2-2 and Fig. 2-3 show an example of computing, expansion and updating conspiracy numbers for a MIN/MAX tree. For each iteration, the conspiracy number search always selects the leaf node corresponding to the minimum conspiracy number of the root to eliminate the unstable element in the tree. The conspiracy number is a very promising concept of the domain-independent heuristic, but suffers from a low efficiency in real applications. There are mainly two reasons: (1) it takes too much time and storage to compute and record the conspiracy numbers (2) the algorithm converges slowly, sometimes even cannot converge. As a result, the conspiracy number becomes a good theoretical benchmark to be improved.

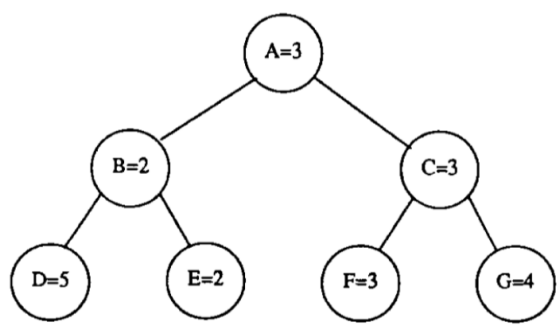


Figure 2-1: An example of MIN/MAX tree [40]

2.1.3 Proof Number Search

To incorporate the conspiracy number idea into a real application, the proof number search (PN-search) is proposed as a game solver searching on an AND/OR tree. Proof-Number Search (PN-search) [2] is a native best-first algorithm, using proof numbers and disproof numbers, always expanding one of the most-proving nodes. All nodes have proof and disproof numbers, they are stored to indicate which frontier node will be expanded, and updated after expanding. A proof (disproof) number shows the scale of difficulty in

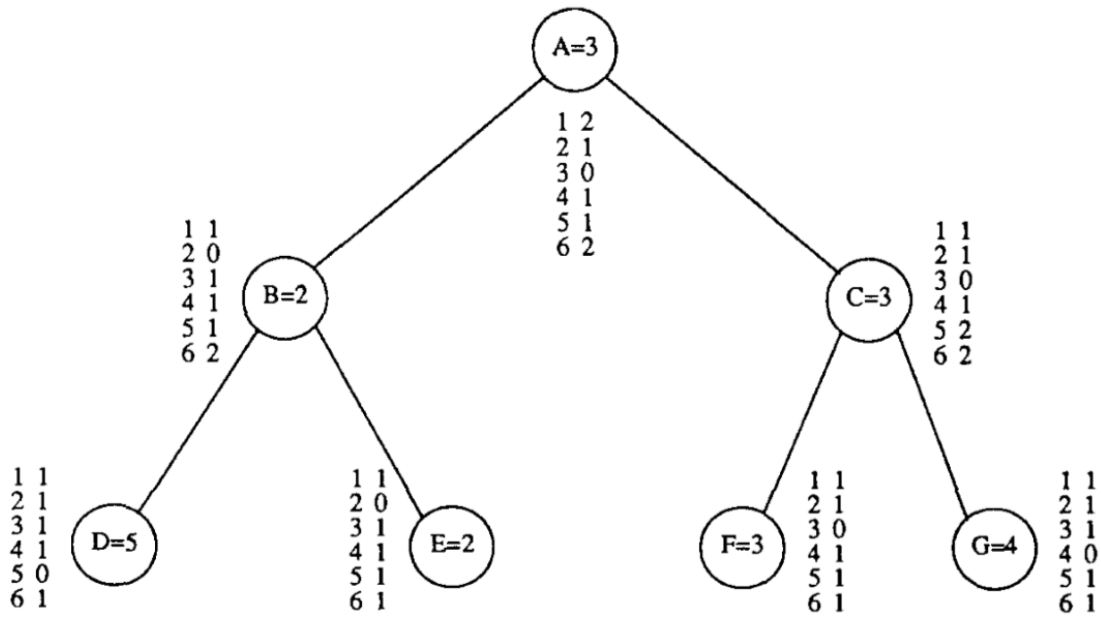


Figure 2-2: An example of conspiracy numbers (the left column of the number list of a node is the evaluation value, and the right column of the number list of a node is its corresponded conspiracy number) [40]

proving (disproving) a node. The expanded node is called the most-proving node, which is the most efficient one for proving (disproving) the root. By exploiting the search procedure, two characteristics of the search tree are established [50]: (1) the shape (determined by the branching factor of every internal node), and (2) the values of the leaves (in the end they deliver the game theoretic value). Basically, unenhanced PN-Search is an uninformed search method that does not require any game-specific knowledge beyond its rules [25]. The formalism is given below.

Let $n.pn$ and $n.dn$ be the proof number and disproof number of a node n, respectively.

When n is a terminal node

(a) If n is a win for the attacker:

$$n.pn = 0$$

$$n.dn = \infty$$

(b) If n is a loss for the attacker:

$$n.pn = \infty$$

$$n.dn = 0$$

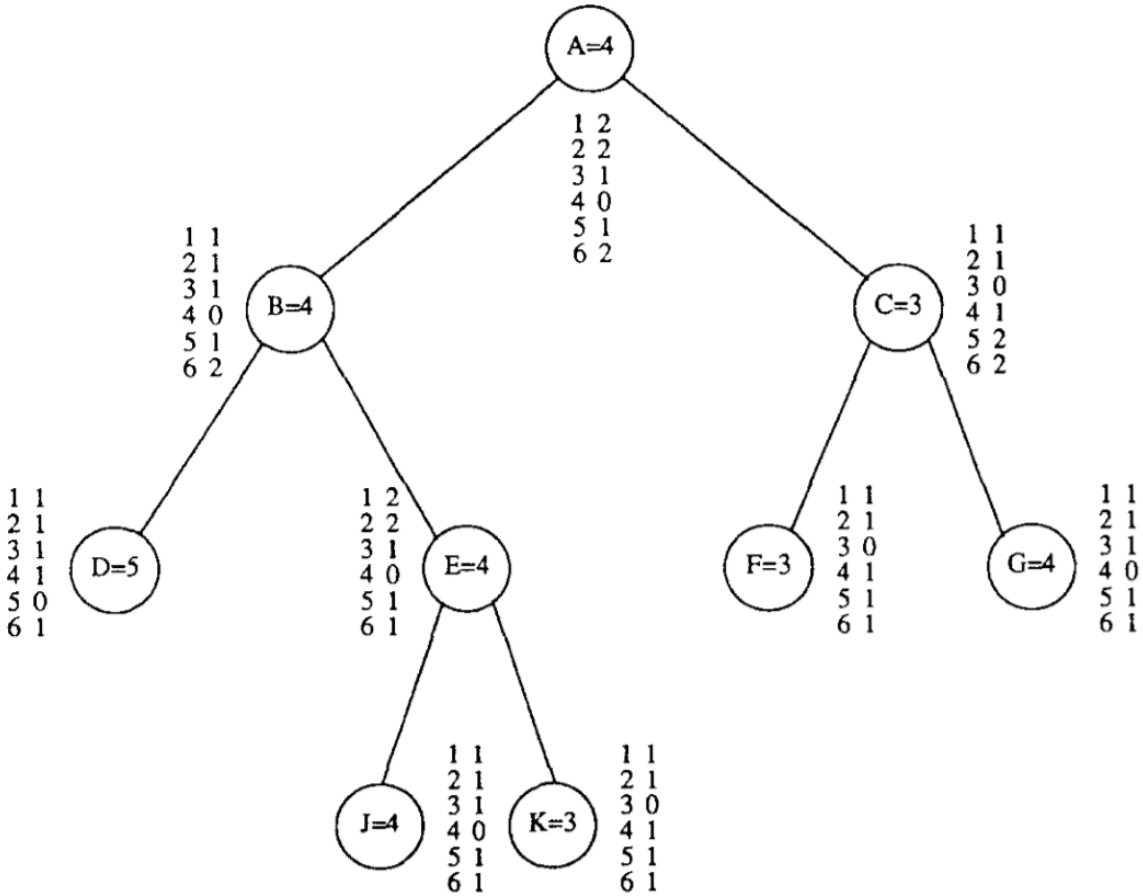


Figure 2-3: An example of the expansion and updating process in the conspiracy number search (the left column of the number list of a node is the evaluation value, and the right column of the number list of a node is its corresponded conspiracy number) [40]

(c) If the value of n is unknown:

$$n.pn = 1$$

$$n.dn = 1$$

When n is an internal node

(a) If n is an OR node:

$$n.pn = \underset{n_c \in \text{children of n}}{Min} n_c.pn$$

$$n.dn = \sum_{n_c \in \text{children of n}} n_c.dn$$

(b) When n is an AND node

$$n.pn = \sum_{n_c \in \text{children of n}} n_c.pn$$

$$n.dn = \underset{n_c \in \text{children of n}}{Min} n_c.dn$$

A most-proving node is a leaf node that is selected by tracing nodes from the root node in the following way.

- For each OR node, trace the child with the minimum proof number.
- For each AND node, trace the child with the minimum disproof number.

Note that Allis et al. [2] defined the most-proving node as the left-most one, if there is arbitrariness.

2.1.4 Df-pn

Although PN-search is an ideal AND/OR-tree search algorithm, it still has at least two problems. We mention two of them. The first one is that PN-search uses a large amount of memory space because it is a best-first algorithm. The second one is that the algorithm is not efficient as hoped for because of the frequently updating of the proof and disproof numbers. To solve the problems, Nagai [34] proposed a depth-first like algorithm using both proof number and disproof number. He called it df-pn (depth-first proof-number search). The procedure of df-pn can be characterized as (1) selecting the most-proving node, (2) updating the thresholds of proof number or disproof number in a transposition table, and (3) applying multiple iterative deepening until the ending condition is satisfied. Although df-pn is a depth-first like search, it has a same behavior as PN-search. The equivalence between PN-search and df-pn is proved in [34].

In df-pn, proof number and disproof number are renamed as follows.

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

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

Moreover, each node n has two thresholds: one for the proof number th_{pn} and the other for the disproof number th_{dn}. Similarly, th_{pn} and th_{dn} are renamed as follows.

$$n.th_{\phi} = \begin{cases} n.th_{pn} & \left(\text{ n is an OR node } \right) \\ n.th_{dn} & \left(\text{ n is an AND node } \right) \end{cases}$$

$$n.th_{\delta} = \begin{cases} n.th_{dn} & \left(\text{ n is an OR node } \right) \\ n.th_{pn} & \left(\text{ n is an AND node } \right) \end{cases}$$

Df-pn expands the same frontier node as PN-search in a depth-first manner guided by a pair of thresholds (th_{pn}, th_{dn}), which indicates whether the most-proving node exists in the current subtree [21]. The procedure is described below [34].

Procedure Df-pn

For the root node r, assign values for $r.th_{\phi}$ and $r.th_{\delta}$ as follows.

$$r.th_{\phi} = \infty$$

$$r.th_{\delta} = \infty$$

Step 1. At each node n, the search process continues to search below n until $n.\phi \geq n.th_{\phi}$ or $n.\delta \geq n.th_{\delta}$ is satisfied (we call it ending condition).

Step 2. At each node n, select the child n_c with minimum δ and the child n_2 with second minimum δ. (If there is another child with minimum δ, that is n_2.) Search below n_c with assigning

$$n_c.th_{\phi} = n.th_{\delta} + n_c.\phi - \sum n_{child}.\phi$$

$$n_c.th_{\delta} = \min\left(n.th_{\phi}, n_2.\delta + 1\right)$$

Repeat this process until the ending condition holds.

Step 3. If the ending condition is satisfied, the search process returns to the parent node of n. If n is the root node, then the search is totally over.

2.1.5 The Seesaw Effect

PN-search and df-pn are highly efficient in solving games. However, both are facing the drawback named as *seesaw effect* [18]. It can be best characterized as frequently going back to the ancestor nodes for selecting the most-proving node. Pawlewicz and Lew [36], and Kishimoto et al. [26] [24] showed one such weak point of df-pn. The weak point has been named the seesaw effect by Hashimoto [18].

To explain it precisely, we show, in Fig. 2-4, an example where the root node has two subtrees. The size of both subtrees is almost the same. Assume that the proof number of subtree L is larger than the proof number of subtree R. In this case, PN-search or df-pn will continue search in subtree R, which means that the most-proving node is in subtree R. After PN-search or df-pn has expanded the most-proving node, the shape of the game tree will change as shown in Fig. 2-4(b). By expanding the most-proving node, the proof number of subtree R becomes larger than the proof number of subtree L. So PN-search or df-pn changes its searching direction from subtree R to subtree L. In turn, when the search expands the most-proving node in subtree L, then the proof number of subtree L becomes larger than the one in subtree R. Thus, the search changes its focus from subtree L to subtree R. This change keeps going back and forth, which looks like a seesaw. Therefore, it is named as seesaw effect.

The seesaw effect happens when the two trees are almost equal in size. If the seesaw effect occurs frequently, the performance of PN-search and df-pn deteriorates significantly and cannot reach the required search depth. In games which need to reach a large fixed search depth, the seesaw effect works strongly against efficiency.

The seesaw effect is mostly caused by two issues: the shape of game tree and the way of searching. Concerning the shape of game tree, there are two characteristics: (1) a tendency of the newly generated children to keep the size equal and (2) the fact that many nodes with equal values exist deep down in a game tree. If the structure of each node remains almost the same (cf. characteristic 1), then the seesaw effect may occur easily. For characteristic 2, it is common in games such as Othello and Hex to search a large fixed number of moves before settling. This is also the case in connect-type games such as Gomoku and Connect6 which have a sudden death in the game tree. Therefore, it is necessary to design a new search algorithm to reduce the seesaw effect in these games.

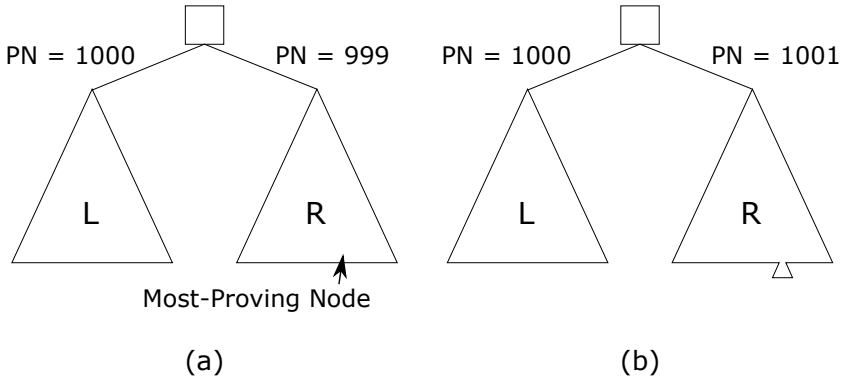


Figure 2-4: An example of the seesaw effect: (a) An example game tree (b) Expanding the most-proving node [19]

2.1.6 Deep Proof Number Search

This section is updated and abridged from the following publication.

- Ishitobi, T. (2016). Deep Proof-Number Search and Aesthetics of Mating Problems. JAIST Press.

In this section, we introduce a new search algorithm based on proof numbers named Deep Proof-Number Search (DeepPN). DeepPN is a variant of the original PN-search. Each node in the search tree has two indicators: the proof number and the disproof number. Additionally, for DeepPN, each node is assigned a so-called *deep value*. The deep values are determined and updated by the terminal node analogously to the proof and disproof numbers. DeepPN has been designed to: (1) combine the best-first and the depth-first search, and (2) to try and solve the problem of the seesaw effect. For evaluating the performance of DeepPN, we use endgame positions of Othello and Hex.

The Basic Idea of DeepPN

In the original PN-search, the most-proving node is defined as follows [2].

Definition 1. For any AND/OR tree T, a most-proving node of T is a frontier node of T, which by obtaining the value true reduces T's proof number by 1, while by obtaining the value false reduces T's disproof number by 1.

This definition implies that the most-proving node sometimes exists in a plural form in a tree, i.e., there are many fully equivalent most-proving nodes. For example, if the

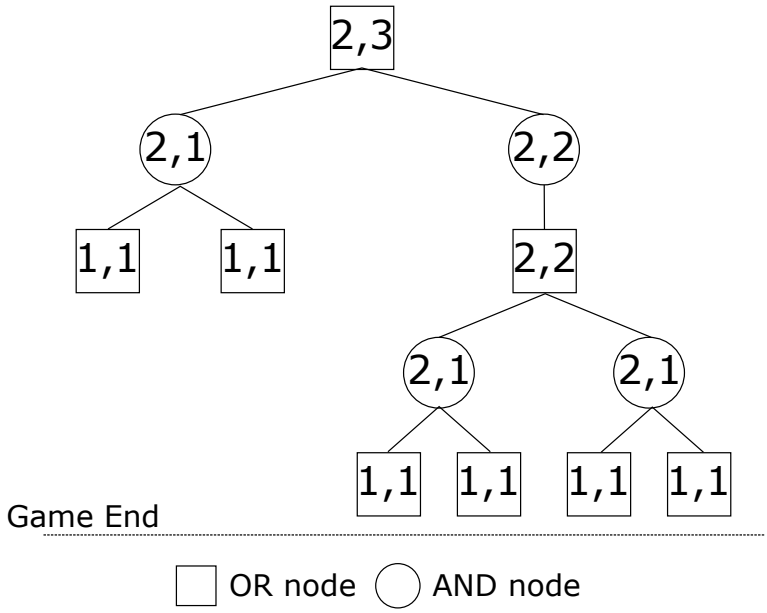


Figure 2-5: An example of a suitable tree for an Othello end-game position. This game tree has a uniform depth of 4, and the terminal nodes are reached at game end. [19]

child nodes have the same proof or disproof number then both subtrees have each a most-proving node. The situation that the child nodes has the same proof (disproof) number in an OR (AND) node is called a tie-break situation. Now, we have the question about which most-proving node is the best for calculating the game-theoretical value. PN-search chooses the leftmost node with the smallest proof (disproof) number, also in a tie-break situation. In particular, the proof and disproof number do not take other information into account, and therefore PN-search cannot choose a more favorable most-proving node in a tie-break situation.

Determining the best most-proving node in a tie-break situation is a difficult task, because the answer depends on many aspects of the game. However, when focusing on games which build up a suitable tree, we may develop some solutions. In a suitable tree, the "best" most-proving node is indicated by its depth number. See the example given in Fig 2-5.

This game tree is from the Othello [8]. The end of the game is shown by "Game End" in Fig 2-5. All level-two nodes are most-proving nodes, because the proof numbers of child nodes under the root node are the same (i.e., 2). So, we have a tie-break situation. Now, in the next search step, PN-search will focus on the most-proving node that exists in left side as produced by the original PN-search algorithm. However, if the search focuses

immediately on the most-proving node of the right side, then the search will be more efficient, because the nodes on the left side do not reach the game end and their value cannot be found yet. In contrast, nodes that exist at the right side reach the game end, and if we try to expand these nodes, then the game value of each node is known. In this example, we follow the idea that a most-proving node in the deepest tree of a suitable game tree, is the best.

To test this idea, we performed a small experiment. We prepared an original PN-search and a modified PN-search. In a tie-break situation, PN-search focuses on a most-proving node that exists in the leftmost node, and the modified PN-search focuses on the *deepest* most-proving node. For checking performance, we prepared 100 Othello endgame positions. The performance of the modified PN-search is better than the results of the original PN-search (about 10% reduction). These results suggest that the *deepest* most-proving node works advantageously for finding the game-theoretical value.

In addition, the example of Fig 2-5 shows the essence of the seesaw effect. If the game end exists and has a depth of more than 4, then the search for a proof number goes back and forth between the two subtrees. Even if the game end is of depth 4, then the search that focuses on the right subtree will change its focus on the left subtree. But, when modifying PN-search, the small seesaw effect is suppressed. This phenomenon of modifying PN-search suggests a new heuristic. The search depth of nodes can be used for solving the seesaw effect in a suitable game tree. In fact, this is what $1 + \epsilon$ trick [36] in effect tries to accomplish, to stay deep in a suitable game tree. For more detail of $1 + \epsilon$ trick, see section 3.3.4 in chapter 3. Now, let us try to think of a new technique. For instance, consider the moves that the modified PN-search plays when finding the deepest most proving node. We noticed that these moves combined best-first with depth-first behavior. The modified PN-search works in a best-first manner, and in a tie-break situation, PN-search work depth-first for the most-proving nodes. Depending on how often tie-breaks occur, the algorithm works more frequently best-first than depth-first. The resulting improvement, when measured in number of iterations and nodes leads to a small result. Thus, we will design a new algorithm that can change the ratio of best-first manner and depth-first manner. Its description is an follows. This system is named Deep Proof-Number Search (DeepPN). Here, $n.\phi$ means proof number in OR node and disproof

number in AND node. In contrast, $n.\delta$ means proof number in AND node and disproof number in OR node.

1. The proof number and disproof number of node n are now calculated as follows.

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

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

2. When n is a terminal node

   (a) When n is proved (disproved) and n is an OR (AND) node, i.e., OR wins

$$n.\phi = 0$$

$$n.\delta = \infty$$

   (b) When n is disproved (proved) and n is an AND (OR) node, i.e., OR does not win

$$n.\phi = \infty,$$

$$n.\delta = 0$$

   (c) When n is unsolved, i.e., its value is unknown

$$n.\phi = 1,$$

$$n.\delta = 1$$

   (d) When n is terminal node, then n has deep value

$$n.deep = \frac{1}{n.depth} \tag{2.1}$$

3. When n is an internal node

(a) The proof and disproof number are defined as follows

$$n.\phi = \min_{n_c \in \text{children of n}} n_c.\delta$$

$$n.\delta = \sum_{n_c \in \text{children of n}} n_c.\phi$$

(b) The deep values, DPN(n) and n.deep are defined as follows.

$$n.deep = n_c.deep \text{ where } n_c = \arg\min_{n_i \in \text{unsolved children}} DPN(n_i) \tag{2.2}$$

$$DPN(n) = (1 - \frac{1}{n.\delta + 1})R + n.deep(1 - R) \qquad (0.0 \leq R \leq 1.0) \tag{2.3}$$

The proof and disproof number are the same as in the original PN-search. The improvement is the new term, i.e., the concept of the *deep* value. The deep value in a terminal node is calculated by formula (2.1). The deep value is designed to decrease inversely with depth. In an internal node, calculating the deep value has only a limited complexity. First, we define a function named DPN (see formula 2.3). DPN has two features: (a) $n.\delta$ is normalized and designed to become larger according to the growth of $n.\delta$ and (b) a fixed parameter R is chosen. R has a value between 0.0 and 1.0. If R is 1.0 then DeepPN works the same as PN-search, and if R is 0.0 then DeepPN works the same as a primitive depth-first search. Therefore, the normalized δ fulfills the role of best-first guide and the *deep* acts as a depth-first guide. This means that by changing the value of R, the ratio of best-first and depth-first search of DeepPN can be adjusted. Second, in an internal node, the deep value is updated by its child nodes using formula (2.2). The deep value of node n is decided by a child node n_c which has smallest $DPN(n_c)$. A point to notice is that the updating value is only *deep*, not $DPN(n_c)$. Additionally, when n_c is solved, then the deep value of n_c is ignored in arg min.

In DeepPN, an expanding node in each iteration is chosen as follows.

$$select\_expanding\_node(n) := \arg\min_{n_c \in \text{children of n except solved}} DPN(n_c) \tag{2.4}$$

This sequence is repeated until the terminal node is reached. That terminal node is the node that is to be expanded. If $R = 1.0$, then this expanding node is the most-proving node.

Performance with Othello

For measuring the performance of DeepPN, we prepared a solver using the DeepPN algorithm and Othello endgame positions. We configured a primitive DeepPN algorithm for investigating the effect of DeepPN only, without any supportive mechanisms such as transposition tables and ϵ-thresholds. We prepared 1000 Othello endgame positions. They are constructed as follows. The positions are taken from the 8 x 8 board. We play 44 legal moves at random from the begin position. This implies that 48 squares from the 64 are covered. So, the depth of the full tree to the end is 16.

In all our experiments DeepPN is applied to these 1000 endgame positions. Our focus is the behavior of R (see formula (2.3)). For $R = 1.0$, DeepPN works the same as PN-search and shows the same results. For $R = 0.0$, DeepPN works the same as a primitive depth-first search. When R is between 1.0 to 0.0, then DeepPN behaves as a mix between best-first and depth-first. We changed R from 1.0 to 0.0 by increments of 0.05. We focus on the values of two concepts, viz. the number of iterations and the number of nodes. The number of iterations is given by counting the number of traces of finding the most-proving node from the root node. This value indicates an approximate execution time unaffected by the specifications of a computer. The number of nodes is an indication of the total number of nodes that are expanded by the search. This value is an approximation of the size of memory needed for solving. We show the results in Fig 2-6 and 2-7.

Figure 2-6 shows the variation of (1) the number of iterations and (2) the number of nodes. Each point is as mean value calculated from the results of 1000 Othello endgame positions. $R = 1.0$ shows the results of PN-search, and this value is the base for comparison. As R goes to 0.8, the number of iterations and nodes decrease almost by half. From $R = 0.8$ to 0.6, the number of iterations stops decreasing, but the number of nodes decreases slowly. From $R = 0.6$ to 0.4, the decrease stops, and the number of iterations starts increasing again slowly. In $R = 0.35$, both numbers increase rapidly. We see that for R of around 0.4, the balance between depth-first and best-first behavior appears to be

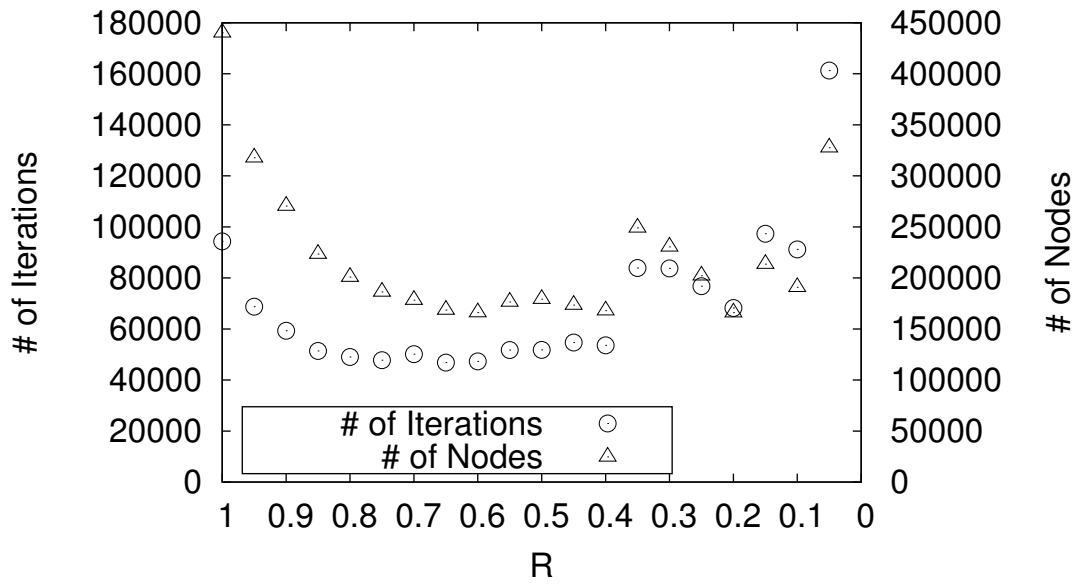


Figure 2-6: The variation in Othello: The number of # of Iterations and # of Nodes. $R = 1.0$ is PN-search, $R = 0.0$ is depth-first search, and $1.0 > R > 0.0$ is DeepPN. Lower is better. [19]

optimal. We surmise that DeepPN is stuck in one subtree and cannot get away since the algorithm is too strongly depth-first. For $R = 0.35$ to 0.2, the number of iterations and nodes is decreasing. Around $R = 0.2$, the balance was broken again, and is decreasing towards 0.1. Finally, DeepPN performs worse when R approaches 0.0 closely. In $R = 0.0$, almost no Othello end game position can be solved, and this value is omitted from Fig 2-6.

In Figure 2-6, the scale of the number of iterations and nodes are different. To ease our understanding, Figure 2-7 shows the amount of the reduction rate. This reduction rate is normalized by the result of PN-search, i.e., the reduction rate of $R = 1.0$ is 100%. Each point is the mean value of the reduction rate calculated by the results of 1000 Othello endgame positions. The results of 2-7 show almost the same characteristics as 2-6. There is a different point where the number of iterations decreases after $R = 0.8$ and the number of nodes decreases after $R = 0.6$. In Figure 2-7, the number of iterations decreased about 50% in $R = 0.4$ and the number of nodes decreased about 35% in $R = 0.4$. Thus, DeepPN reduced the number of iterations ($\approx$ time) to half and the number of nodes ($\approx$ space) to one-third. In $R = 0.05$, the number of iterations increased to over 100%, which is not shown.

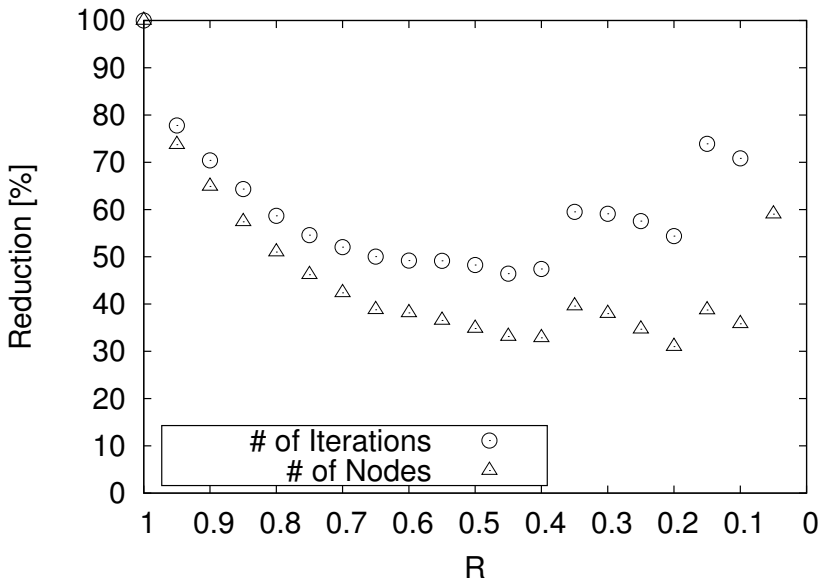


Figure 2-7: The reduction rate in Othello: The number of # of Iterations and # of Nodes. $R = 1.0$ is PN-search, $R = 0.0$ is depth-first search, and $1.0 > R > 0.0$ is DeepPN. Lower is better. [19]

Finally, we show two graphs about the changes in *reducing* and *increasing* cases in Othello endgame positions in Fig 2-8 and 2-9. Please note that in Fig 2-8 and 2-9 we showed the number of iterations and number of nodes.

The plots for reducing cases give the number of Othello endgame positions which are solved efficiently compared to PN-search, i.e., the reduction rate is under 100%. In contrast, the plots for increasing cases give the number of Othello endgame positions that have a reduction rate over 100%. The vertical axis shows the number of Othello endgame positions. Figure 2-8 shows the number of iterations by which the reducing cases decrease slowly from $R = 0.95$ to 0.4. Likewise, for number of nodes the graph decreases slowly from $R = 0.95$ to 0.4. Around $R = 0.4$, the trend is broken, and the number of increasing cases increases rapidly. From $R = 0.35$ to 0.2 and from 0.15 to 0.1, the number of cases does not change much. This result indicates that the reason of decreasing from $R = 0.35$ to 0.2 is shown in Fig 2-6 and 2-7. As the number of cases is not changed, the decreasing number of iterations and nodes of the Othello end game positions are caused by reducing cases. In brief, some Othello end positions can be handled efficiently as R is reduced. But, for some Othello end game positions a changing R causes an increase. Therefore, Othello end game positions can be categorized in relation to R. The first group belongs to

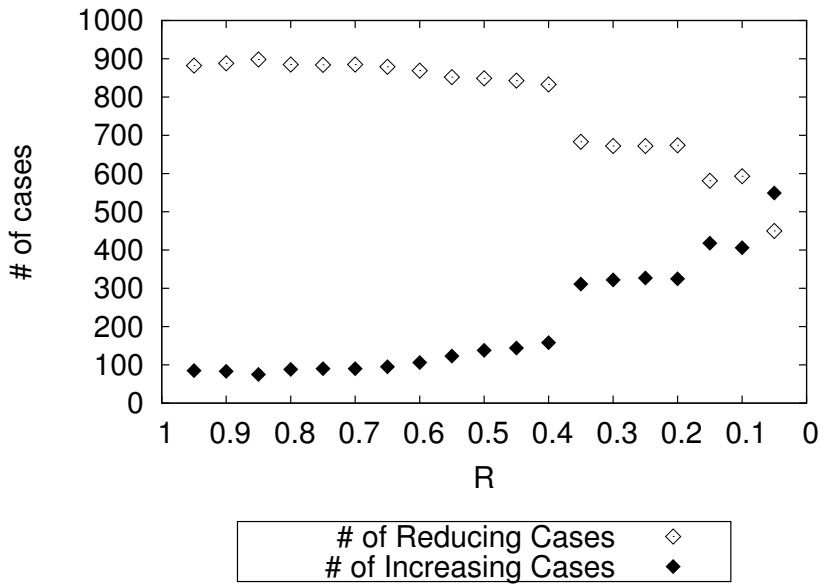


Figure 2-8: # of Iterations in Othello: The changes of Reducing and Increasing Cases for # of Iterations and # of Nodes [19]

$R = 0.95$ to 0.05. This group does not react to changes in R, they do not switch between the reducing case and increasing case. We can see this group clearly from $R = 0.95$ to 0.40. The second group belongs to $R = 0.35$ to 0.2. This group fitted from $R = 0.95$ to 0.4, and they could not keep efficiency work after $R = 0.4$. The third group belongs to $R = 0.15$ to 0.1, and the characteristics of this group are the same as for the second group. In either group, the cases are not efficiently close to $R = 0.0$.

The question remains when DeepPN works most efficiently in the Othello endgame position for 16-ply. The answer depends on the group of Othello endgame positions. However, if we have to choose the best R, then a value of around 0.65 is a good compromise for most cases.

Performance with Hex

For measuring the performance of DeepPN, we also prepared a solver for Hex [3]. As for the experiments of Othello, we created a primitive DeepPN algorithm for checking the effect of DeepPN only. The Hex program is a simple program that does not have any other mechanisms such as an evaluation function. Our Hex program uses a 4 x 4 board (called Hex(4)), and tries to solve that board using DeepPN. Our focus is on the behavior of R (see formula 2.3). Concerning the characteristics of R, please see section

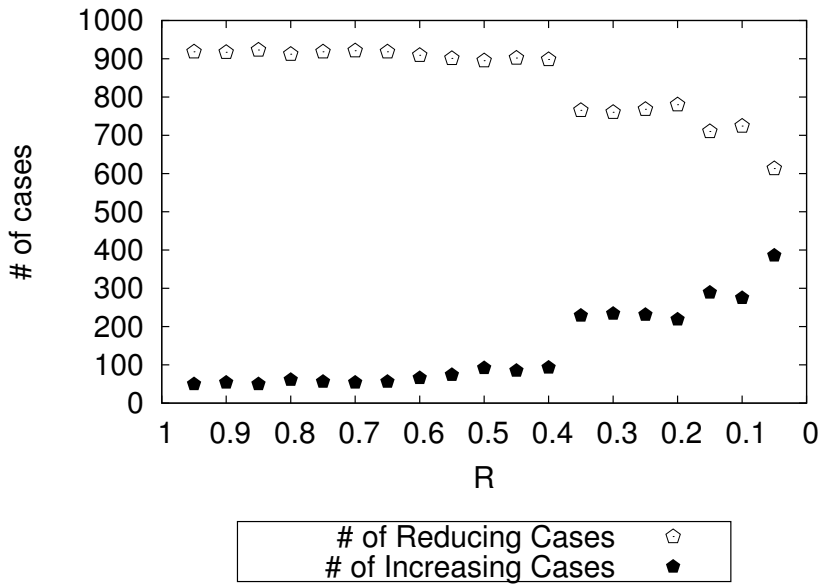


Figure 2-9: # of Nodes in Othello: The changes of Reducing and Increasing Cases for # of Iterations and # of Nodes [19]

2.1.6 or 2.1.6. We changed R from 0.0 to 1.0 by 0.5, and tried to solve Hex(4) 10 times in each R. The legal moves of Hex are sorted randomly in every configuration, viz. there is the possibility that each result is different. The results in each R are calculated by the average of the 10 experiments. Next we focused on two concepts: (1) number of iterations and (2) number of nodes. About the characteristics of both values, please see the section 2.1.6. The experimental data are given in Fig 2-10 and 2-11.

Figure 2-10 shows the changes in the number of iterations and nodes. We can see that the results of DeepPN decrease (improve) in some positions compared by PN-search. This is not the case for $R = 0.0$, because we cannot solve Hex(4) for this R when we limit ourselves to 500 million nodes. For ease of understanding, we prepare another graph in Fig 2-11. There we show the reduction rates normalized by the result of PN-search, i.e., the result of PN-search has 100% reduction rate.

Figure 2-11 shows that the number of iterations and nodes is reduced by a 30% reduction rate between $R = 0.95$ and $R = 0.5$. The result has two downward curves: from $R = 1.0$ to 0.7 and from $R = 0.7$ to $R = 0.0$. The first curve starts from $R = 1.0$ and decreases toward 0.95. After $R = 0.95$, the results start to increase and grow to over 100% after 0.85. The second curve starts from $R = 0.7$ and the results starts to decrease again. At around $R = 0.5$, the results reach about 50%. Finally, the results are increasing again

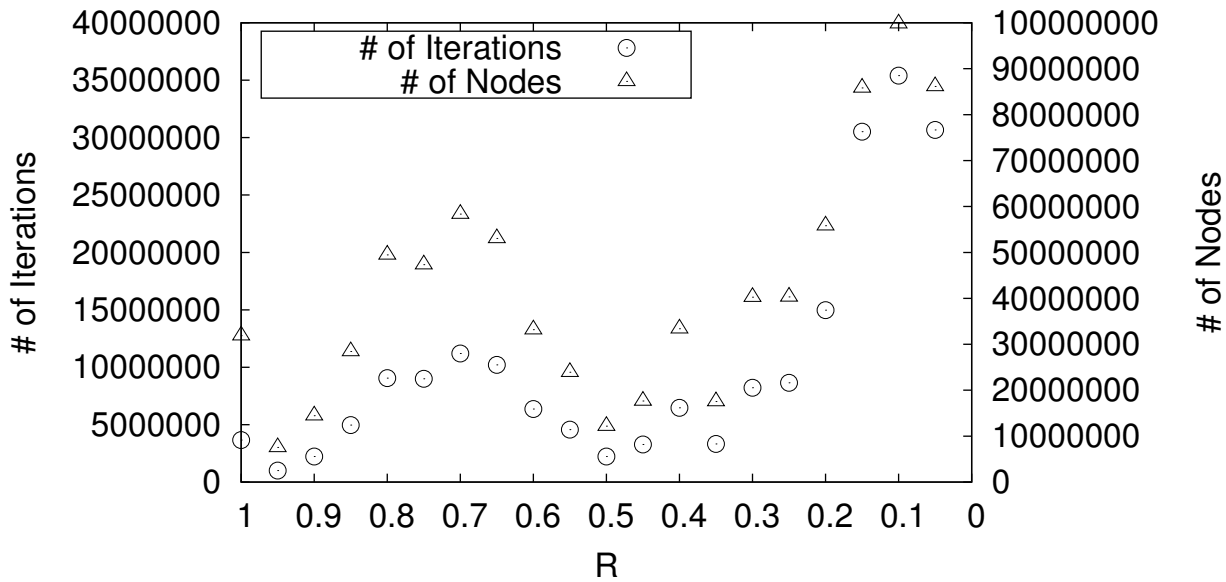


Figure 2-10: The variation in Hex: The changes of Reducing and Increasing Cases for # of Iterations and # of Nodes [19]

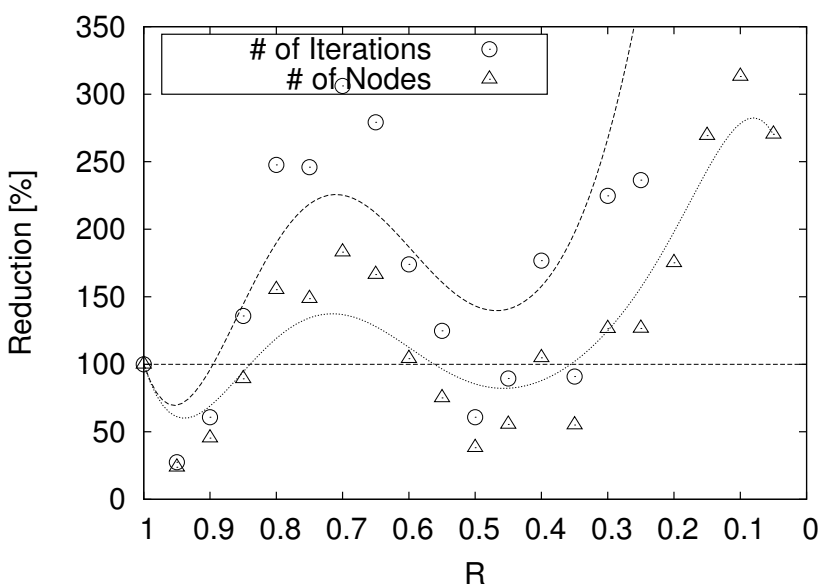


Figure 2-11: The reduction rate in Hex: # of Iterations and # of Nodes for Hex(4). $R = 1.0$ is PN-search, $R = 0.0$ is depth-first search, and $1.0 > R > 0.0$ is DeepPN. Lower is better. [19]

toward $R = 0.0$, like Othello.

For understanding the details of how DeepPN works around $R = 0.95$, we tried to change R by 0.1 between from 1.0 to 0.9. The results are shown in Fig 2-12.

By looking at the results, we can see that DeepPN works almost twice as good as PN-search from $R = 0.99$ to 0.95. Form $R = 0.95$ to 0.90, we have a small curve like

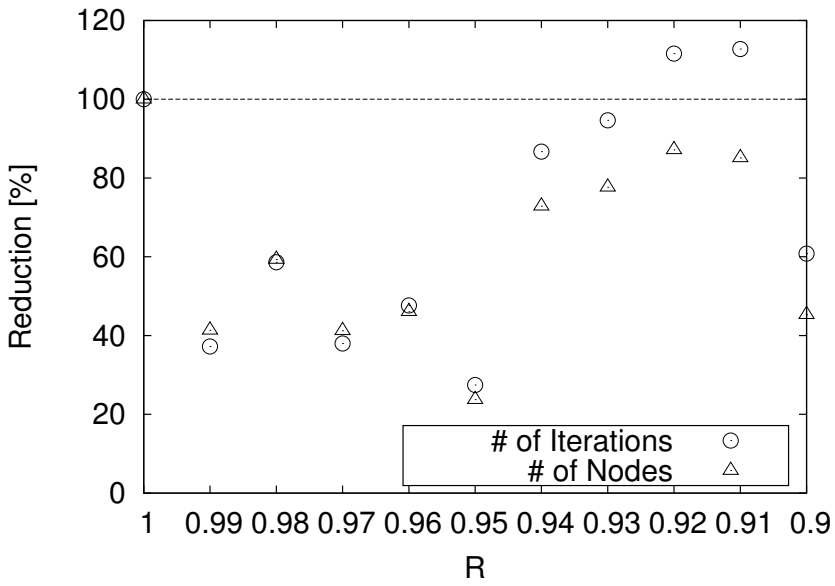


Figure 2-12: Hex: The detail of Fig 2-11. This figure is zoomed $1.0 \leq R \leq 0.9$. The lower is better. [19]

figure 2-11.

In Hex(4), the optimum value of R is around $R = 0.95$ (and perhaps $R = 0.5$). We can see that depth-first does not work so well for Hex(4) as it does for Othello, although there is an improvement over pure best-first.

Discussion

DeepPN works efficiently in 16-ply Othello endgame positions, and in Hex(4). It can reduce the number of iterations and nodes almost by half compared to PN-search. It must be noted that the optimum balance of R is different in each game and for each size of game tree. We can see that for both games a certain amount of depth-first behavior is beneficial, but the changes are not the same. The precise relation is a topic of future work.

Both in Othello endgame positions and in Hex(4), we encountered positions that showed increasing (worse) results. We suspect that a reason for this problem may be (1) the holding problem and (2) the length of the shortest correct path. Concerning (1), the depth-first search can remain stuck in one subtree (holding on to the subtree). If this holding subtree cannot find the game-theoretical value, then the number of iterations and nodes become meaningless. When DeepPN employed a strong depth-first manner, then

we found many increasing results in Othello endgame positions. Also, in Hex(4), DeepPN cannot work efficiently around $R = 0.0$. Finding an optimal R is a topic of future work.

Concerning (2), the problem is related to (1). In Othello, the shortest correct path is almost the same for each position, because Othello has a fixed number of depth to the end. However, in Hex(4), the shortest winning path may exist before a depth of 16. If we happen to find a balance between depth and best-first, then DeepPN will change the subtree it focuses on time. For example, when $R = 0.95$, then DeepPN quickly finds the shortest path. But after $R = 0.95$, DeepPN misses that path and arrives in regions that are more deeply in the trees. Finding a good value of R in Hex is more difficult than in Othello.

2.2 Using the Information beneath the Leaf Nodes

2.2.1 Introduction

In this section, we select Monte-Carlo tree search (MCTS) and Monte-Carlo tree search solver (MCTS solver) as example search algorithms with domain-independent heuristics using the information beneath the leaf nodes. The Monte-Carlo tree search and solver use Monte-Carlo evaluations as domain-independent heuristics for the leaf nodes to indicate the promising search node for expansion. In other words, the Monte-Carlo evaluation can be regarded as a kind of detector to obtain the information beneath the leaf nodes to forecast the promising search direction in advance.

2.2.2 Monte-Carlo Tree Search

The Monte-Carlo method is a statistical physics used to obtain approximations to intractable integrals. The basic idea of Monte-Carlo method is to define a domain of possible inputs, generate inputs randomly from a probability distribution over the domain, perform a deterministic computation on the inputs and aggregate the results. The Monte-Carlo method is first applied to solve deterministic problems using a probabilistic analog. Abramson [1] demonstrated that this sampling might be useful to approximate the game-theoretic value of a move. The notation is adopted by Gelly and Silver that the Q-value of an action is simply the expected reward of that action:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} I_i(s,a) z_i$$

where $N(s,a)$ is the number of times action a has been selected from state s, $N(s)$ is the number of times a game has been played out through state s, z_i is the result of the ith simulation played out from s, and $I_i(s,a)$ is 1 if action a was selected from state s on the ith play-out from state s or 0 otherwise.

Monte-Carlo evaluation provides a new approach to get game heuristic without human knowledge. An intuitive application of Monte-Carlo evaluation in combinatorial games is to obtain the heuristic value for the MIN/MAX tree. This method is known as the Monte-Carlo planning with MIN/MAX value updating. However, the framework is not proved or verified to be more efficient than other alternatives. To challenge such intuition, Coulom [15] proposes a new framework of Monte-Carlo tree search. Instead of backing-up the min-max value close to the root, and the average value at some depth, a more general backup operator is defined that progressively changes from averaging to MIN/MAX value as the number of simulations grows. More concretely, for each iteration, Coulom's Monte-Carlo tree search only takes one simulation and expands one node. Then the information of the leaf node can be back propogated frequently to the root by taking the average of all the branches. As a result, this type of Monte-Carlo tree search converges faster to MIN/MAX value than the Monte-Carlo planning with MIN/MAX value updating. Based on such framework, Kocsis et al. [29] propose the UCT, a MCTS variant applying bandit [4] ideas to guide Monte-Carlo planning. UCT is based on the UCB1 formula derived by Auer, Cesa-Bianchi, and Fischer [4] and the provably convergent AMS (Adaptive Multi-stage Sampling) algorithm first applied to multi-stage decision making models (specifically, Markov Decision Processes) by Chang, Fu, Hu, and Marcus [10]. Kocsis et al. recommend to choose in each node of the game tree the move for which has the highest UCT value. In [29], Kocsis et al. compare the performance of the UCT, the pure Monte-Carlo tree search, Monte-Carlo planning with MIN/MAX value update and alpha-beta search in P-game trees [47]. Experimental results show that in several domains, UCT is significantly more efficient than its alternatives.

After reviewing the development of the Monte-Carlo tree search, Chaslot et al. [11] finally put forward Monte-Carlo Tree Search as a novel, unified framework to game AI.

The basic framework of the Monte-Carlo tree search involves iteratively building a search tree until some predefined computational budget (typically a time, memory or iteration constraint) is reached, at which point the search is halted and the best performing root action returned. Each node in the search tree represents a state of the domain, and directed links to child nodes represent actions leading to subsequent states [7]. Generally, the basic algorithm of the Monte-Carlo tree search includes four steps: (see also Fig 2-13)

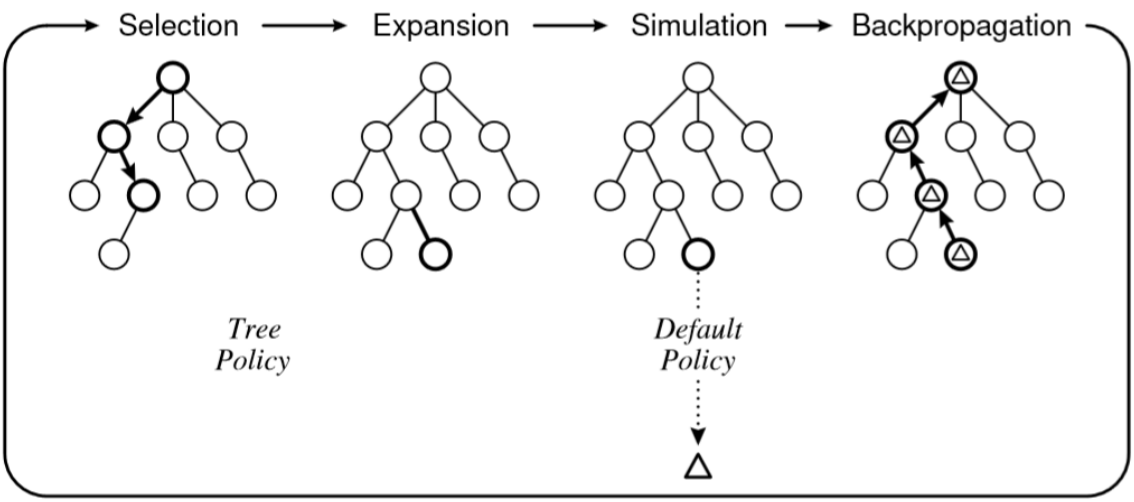


Figure 2-13: One iteration of the general MCTS approach [7]

(1) Selection: Selection picks a child to be searched based on the previously gained information. For pure MCTS when going from the root to a leaf node, the child with the largest simulation value will be selected. For UCT, an enhanced version of MCTS, it controls the balance between exploitation and exploration by selecting the child with the largest UCT (Upper Confidence Bounds applied to Trees) value:

$$v_i + \sqrt{\frac{C \times \ln n_p}{n_i}},$$

where v_i is the simulation value of the node i, n_i is the visit count of child i, and n_p is the visit count of current node p. C is a coefficient parameter, which has to be tuned experimentally. Winands et al. also consider other strategies to optimize the selection based on UCT, such as progressive bias (PB). But in this paper, to make it easy to follow, we only apply the UCT strategy.

(2) Expansion: Expansion is the strategic task that decides whether nodes will be added to the tree. In this paper, we expand one node for each iteration.

(3) Simulation: The simulation step begins when we enter a position that is not a part of the tree yet. Moves are selected in self-play until the end of the game. This task might consist of playing plain random moves.

(4) Backpropagation: Backpropagation is the procedure that propagates the result of a simulated game back from the leaf node, through the previously traversed node, all the way up to the root. A usual strategy of UCT or pure MCTS is taking the average of the results of all simulated games made through this node.

2.2.3 Monte-Carlo Tree Search Solver

Monte-Carlo Tree Search (MCTS) [13] is a best-first search guided by the results of Monte-Carlo simulations. In the last few years, MCTS has advanced the field of computer Go substantially. Although MCTS equipped with the UCT (Upper Confidence Bounds applied to Trees) formula which enables the evaluations to converge to the game-theoretic value, it is still not able to prove the game theoretic value of the search tree. This is even more true for sudden-death games, such as Chess. In this case, some endgame solvers (i.e., PN-search) are traditionally preferred above MCTS. To transform MCTS to a good game solver, Winands et al. introduced an MCTS variant called MCTS solver [54], which has been designed to prove the game-theoretical value of a node in a search tree. The MCTS solver includes the following four strategic steps.

(1) Selection: Selection picks a child to be searched based on the previously gained information. For pure MCTS when going from the root to a leaf node, the child with the largest simulation value will be selected. For UCT, an enhanced version of MCTS, it controls the balance between exploitation and exploration by selecting the child with the largest UCT value:

$$v_i + \sqrt{\frac{C \times \ln n_p}{n_i}},$$

where v_i is the simulation value of the node i, n_i is the visit count of child i, and n_p is the visit count of current node p. C is a coefficient parameter, which has to be tuned experimentally. Winands et al. also consider other strategies to optimize the selection based on UCT, such as progressive bias (PB). But in this paper, to make it easy to follow, we only apply the UCT strategy. To transform UCT and pure MCTS to a solver, a node

is assumed to have the game-theoretical value ∞ or $-\infty$ that corresponds to a proved win or not win, respectively. In this paper, we consider all the drawn games as proved to be not win games to make the experimental results more easy to interpret. When a child is a proven win, the node itself is a proven win, and no selection has to take place. But when one or more children are proven to be not a win, it is tempted to discard them in the selection phase. In this paper, to make it easy to compare, i.e., we do not consider the proved win or the proved not win node in the play-out step, because such technique can similarly be applied into PPN-search and MCPN-search. Moreover, for the final selection of the winning move at the root, often, it is the child with the highest visit count, or with the highest value, or a combination of the two. In the UCT solver or in the pure MCTS solver, the strategy is to select the child of the root with maximum quantity $v + \frac{A}{\sqrt{n}}$, where A is a parameter (here, set to 1), v is the node's simulation value, and n is the node's visit count.

(2) Play-out: The play-out step begins when we enter a position that is not a part of the tree yet. Moves are selected in self-play until the end of the game. This task might consist of playing plain random moves.

(3) Expansion: Expansion is the strategic task that decides whether nodes will be added to the tree. In this paper, we expand one node for each iteration.

(4) Backpropagation: Backpropagation is the procedure that propagates the result of a simulated game back from the leaf node, through the previously traversed node, all the way up to the root. A usual strategy of UCT or pure MCTS is taking the average of the results of all simulated games made through this node. For the UCT solver and the pure MCTS solver (in addition to backpropagating the values 1,0,-1) the search also propagates the game-theoretical values ∞ or $-\infty$. The search assigns ∞ or $-\infty$ to a won or lost terminal position for the player to move in the tree, respectively. Propagating the values back in the tree is performed similar to negamax in the context of MIN/MAX searching in such a way that we do not need to distinguish between MIN and MAX nodes.

2.3 Combining the Information above and beneath the Leaf Nodes

2.3.1 Introduction

Based on the perspective of the domain-independent heuristic with "the information above the leaf nodes" and "the information beneath the leaf nodes", a natural induction is that there should be potential improvements by combining the conspiracy number or the proof number idea with the Monte-Carlo evaluation in the search algorithm. It can be considered as a combination of using "the information above the leaf nodes" and "the information beneath the leaf nodes". For this aspect, we introduce the Monte-Carlo proof number search. In chapter 5, we will give a new contribution named probability-based proof number search (PPNS) to this aspect, which is proved to be more efficient than the Monte-Carlo proof number search.

2.3.2 Monte-Carlo Proof Number Search

Monte-Carlo proof number search (MCPN-search) [38] is an enhanced proof number search by adding the flexible Monte-Carlo evaluation to the leaf nodes. It has exactly the same rules as PN-search except that the proof number and the disproof number of unsolved nodes are derived from Monte-Carlo simulations. This method makes MCPN-search more efficient than PN-search especially in balanced tree games. The formalism is presented as follows.

Let $n.pn$ be the proof number of a node n and $n.dn$ be the disproof number of n. There are three types of nodes to be discussed below.

(1) Assume n is a terminal node

(a) If n is a winning node,

$$n.pn = 0.$$

$$n.dn = \infty.$$

(b) If n is not a winning node,

$$n.pn = \infty.$$

$$n.dn = 0.$$

(2) Assume n is a leaf node (not terminal), and R is the winning rate computed by applying several playouts from this node. Take θ as a small positive number smaller than 1 and closed to 0.

(a) If $R \in (0, 1)$,

$$n.pn = 1 - R.$$

$$n.dn = R.$$

(b) If $R = 1$,

$$n.pn = \theta.$$

$$n.dn = R - \theta.$$

(c) If $R = 0$,

$$n.pn = \theta.$$

$$n.dn = R - \theta.$$

(3) Assume n is an internal node, using AND/OR probability rules of independent events.

(a) If n is an OR node:

$$n.pn = \underset{n_c \in \text{children of n}}{Min} n_c.pn$$

$$n.dn = \sum_{n_c \in \text{children of n}} n_c.dn$$

(b) When n is an AND node

$$n.pn = \sum_{n_c \in \text{children of n}} n_c.pn$$

$$n.dn = \underset{n_c \in \text{children of n}}{Min} n_c.dn$$

The similar to PN-search, the MCPN-search search also includes the following four steps.

(1) Selection: for all nodes from the root to a leaf node, do select the child with the minimum proof number at OR nodes and the child with the minimum disproof number at AND nodes, while regarding it as the most proving node for expansion.

(2) Expansion: expanding the most proving node.

(3) Play-out: The play-out step begins when we enter a position that is not a part of the tree yet. Moves are selected in a randomly self-play mode until the end of the game. After several play-outs, the proof number and the disproof number of the expanded nodes are derived from Monte-Carlo evaluations.

(4) Backpropagation: updating the proof number and disproof number from the extended nodes to the root, while following the MIN/SUM rules given above.

2.4 Chapter Conclusion

In this chapter, we introduced related works before our study. We classified the game-independent heuristics into three categories: "using the information above the leaf nodes", "using the information beneath the leaf nodes" and "combining the information above and beneath the leaf nodes". In the rest of the thesis, we will improve these previous studies respectively.

Chapter 3

Deep df-pn

This chapter is an updated and abridged version of the following publication.

- Zhang S., Iida H., van den Herik H.J. (2017) Deep df-pn and Its Efficient Implementations. In: Winands M., van den Herik H., Kosters W. (eds) Advances in Computer Games. ACG 2017. Lecture Notes in Computer Science, vol 10664.

3.1 Introduction

Proof-Number Search (PN-search) [2] is one of the most powerful algorithms for solving games and complex endgame positions. PN-search focuses on AND/OR tree and tries to establish the game theoretical value in a best-first manner. Each node in PN-search has a proof number (pn) and disproof number (dn). This idea was inspired by the concept of conspiracy numbers, the number of children that need to change their value to make a node change its value [32]. A proof (disproof) number shows the scale of difficulty in proving (disproving) a node. PN-search expands the most-proving node, which is the most efficient one for proving (disproving) the root.

Although PN-search is an effective AND/OR-tree search algorithm, it still has some problems. We mention two of them. The first one is that PN-search uses a large amount of memory space because it is a best-first algorithm. The second one is that the algorithm is not efficient as hoped for because of the frequently updating of the proof and disproof numbers. So, Nagai [34] proposed a depth-first algorithm using both proof number and disproof number based on PN-search, which is called depth-first proof-number search (df-

pn). The procedure of df-pn can be characterized as (1) selecting the most-proving node, (2) updating thresholds of proof number or disproof number in a transposition table, and (3) multiple iterative deepening until the ending condition is satisfied. Nagai proved the equivalence between PN-search and df-pn [34]. He noticed that df-pn always selects the most-proving node as PN-search does in the searching path. Moreover, its depth-first manner and the use of a transposition table give df-pn two clear advantages: (1) df-pn saves more storage, and (2) it is more efficient than PN-search.

Yet, both PN-search and df-pn suffer from the seesaw effect which can be characterized as frequently going back to the ancestor nodes for selecting the most-proving node, as described in [36] [26] [24]. They showed that the seesaw effect works strongly against the efficiency in some situations. In Ishitobi et al. [20], the seesaw effect was discussed in relation to PN-search. The authors arrived at a DeepPN search. However, DeepPN has in turn still at least two drawbacks: (1) it suffers from a big cost of storage as PN-search, and (2) DeepPN spends much time on updating the proof and disproof number, which makes DeepPN actually not an efficient algorithm. This chapter proposes a Deep depth-first proof-number search algorithm (Deep df-pn) to reduce the seesaw effect in df-pn. The difference between Deep df-pn and df-pn lies in the proof number or disproof number of unsolved nodes. In df-pn the proof number or disproof number of unsolved nodes is 1, while in Deep df-pn it is a function of depth with two parameters. By adjusting the value of parameters, Deep df-pn changes its behavior from searching broadly to searching deeply. It will be proved in this chapter that doing so will be able to reduce the seesaw effect convincingly.

To evaluate the performance of Deep df-pn, we implement a relevance-zone-oriented Deep df-pn to make it work efficiently in the domain of Connect6 [56]. The concept of relevance zone in Connect6 is introduced by Wu and Lin [55]. It is a zone of the board in which the defender has to place at least one of the two stones, otherwise the attacker will simply win by playing a VCDT (victory by continuous double threat) strategy. Such a zone indicates which moves are necessary for the defender to play. It helps to cut down the branch size of the proof tree. With a relevance zone, Deep df-pn can solve positions of Connect6 efficiently. Experimental results show its good performance in improving the search efficiency.

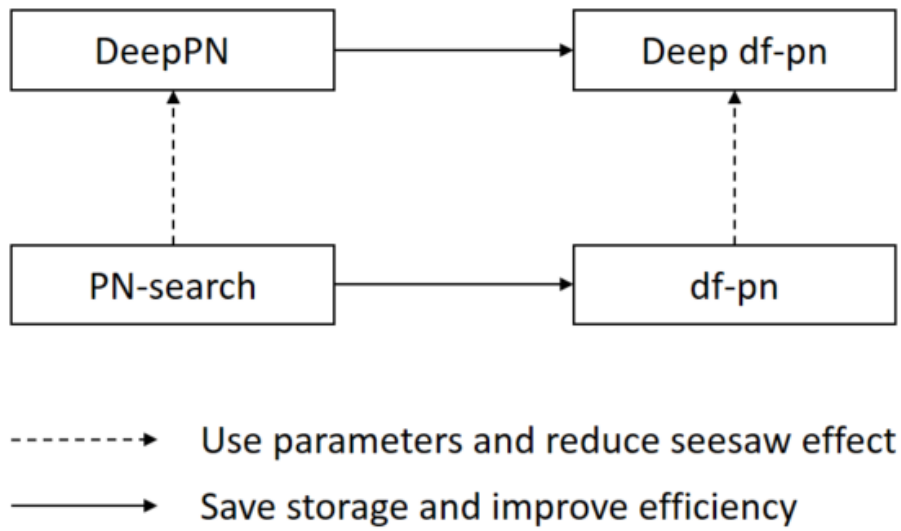


Figure 3-1: Relationship between PN-search, df-pn, DeepPN and Deep df-pn

The remainder of the chapter is as follows. Definitions of Deep df-pn and its characteristics are presented in Section 3.2. In Section 3.3, we introduce the relevance-zone-oriented Deep df-pn for Connect6. Then, we conduct experiments to show its better performance in reducing the seesaw effect in Section 4.3. Finally, concluding remarks are given in Section 5.5.

3.2 Basic Idea of Deep df-pn

In this section, we propose a new proof-number algorithm based on df-pn to cover the shortage of DeepPN (see section 1), named as Deep Depth-First Proof-Number Search or Deep df-pn in short. It not only extends the improvements of df-pn on (1) saving storage and (2) efficiency, but also (3) reduces the seesaw effect. Fig. 3-1 shows the relationship between PN-search, df-pn, DeepPN, and Deep df-pn.

Similar to DeepPN, the proof number and disproof number of unsolved nodes are adjusted in Deep df-pn by a function of depth with two parameters. By adjusting the values of the two parameters, Deep df-pn can change its behavior from searching broadly to searching deeply (and vice versa). Definitions of Deep df-pn are given below.

In Deep df-pn, the proof number and disproof number of node n are calculated as given in section 2.1.6 (here repeated for readability).

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

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

When n is a leaf node, there are three cases.

(a) When n is proved (disproved) and n is an OR (AND) node, i.e., OR wins

$$n.\phi = 0$$

$$n.\delta = \infty$$

(b) When n is proved (disproved) and n is an AND (OR) node, i.e., OR does not win

$$n.\phi = \infty$$

$$n.\delta = 0$$

(c) When the value of n is unknown

$$n.\phi = \underline{Ddfpn}\,(n.depth)$$

$$n.\delta = \underline{Ddfpn}\,(n.depth)$$

When n is an internal node, the proof and disproof number are defined as follows

$$n.\phi = \underset{n_c \in \text{children of n}}{Min} n_c.\delta$$

$$n.\delta = \sum_{n_c \in \text{children of n}} n_c.\phi$$

Definition 2. $\underline{Ddfpn}\,(x)$ is a function from $\mathbb{N}$ to $\mathbb{N}$, which

$$\underline{Ddfpn}\,(x) = \begin{cases} E^{D-x} & (D > x \wedge E > 0) \\ 1 & (D \leq x \wedge E > 0) \\ 0 & (E = 0) \end{cases}$$

where E and D are parameters on $\mathbb{N}$, E denotes a threshold of branch size and D denotes a threshold of depth.

The complete algorithm of Deep df-pn is presented in the appendix. Table 3.1 shows the behavior of Deep df-pn with different values of E and D. When $E = 0$, Deep df-pn is a depth-first search. When $E > 1$ and $D > 1$, Deep df-pn is an intermediate search procedure between depth-first search and df-pn. For other cases, Deep df-pn is the same as df-pn. Deep df-pn focuses on changing the search behavior of df-pn. The procedure of selecting the most proving node in df-pn is controlled by the thresholds of proof number and disproof number. So changing the search behavior of df-pn can be implemented by two methods: (1) changing the thresholds of proof number and disproof number (such as $1 + \epsilon$ trick [36]); (2) changing the proof number and disproof number of unsolved nodes. Deep df-pn implements the method (2). If E or D becomes smaller, Deep df-pn tends to search more broadly usually with more seesaw effect. If E or D becomes larger, Deep df-pn tends to search more deeply usually with less seesaw effect. Below we will prove that Deep df-pn helps reduce the seesaw effect in df-pn.

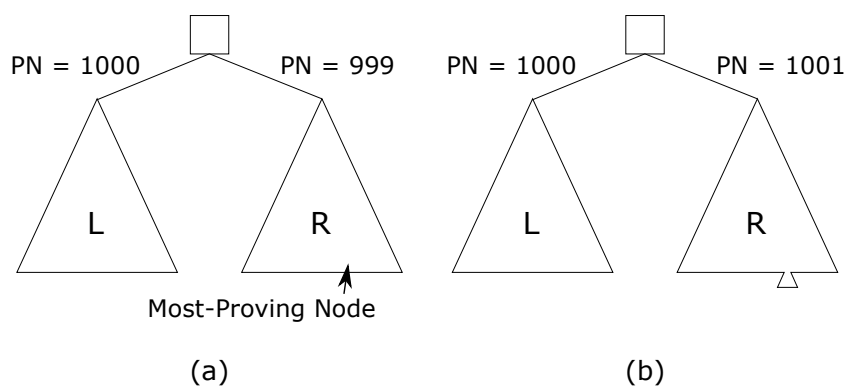


Figure 3-2: An example of the seesaw effect: (a) An example game tree (b) Expanding the most-proving node [19]

Theorem 1. Deep df-pn outperforms df-pn in reducing the seesaw effect.

Proof Assume that node n is a most-proving node in a seesaw effect (see Fig. 3-2(b)). Without loss of generality, n is an AND node in subtree L. According to the feature of the seesaw effect, after expanding n, its proof number becomes larger, which makes the proof number of subtree L larger. Then df-pn changes its focus on subtree R and the seesaw effect happens.

From the definitions of Deep df-pn, the proof number of n is given by: $\underline{Ddfpn}\,(n.depth)$. After expanding n, its proof number is given by

$$\sum_{\text{children of n}} \underline{Ddfpn}\,(n.depth+1) = E^{'} \cdot \underline{Ddfpn}\,(n.depth+1)\,.$$

where $E^{'}$ denotes the number of children of n. If $E^{'} \leq E$ and $n.depth + 1 < D$, then we have $E^{'} \cdot \underline{Ddfpn}\,(n.depth+1) = E^{'} \cdot E^{D-(n.depth+1)}$ and $E^{'} \cdot E^{D-(n.depth+1)} \leq E^{D-depth}$. So we obtain the following inequation

$$\sum_{\text{children of n}} \underline{Ddfpn}\,(n.depth+1) \leq \underline{Ddfpn}\,(n.depth)\,.$$

Therefore, Deep df-pn continues focusing on subtree L and the seesaw effect does not occur. For a certain proof trees of which the maximum branching factor is larger than E, the degree of reducing the seesaw effect increases as the value of E increases. As a result, Deep df-pn outperforms df-pn in reducing the seesaw effect.

Table 3.1: Different behaviors by changing parameters

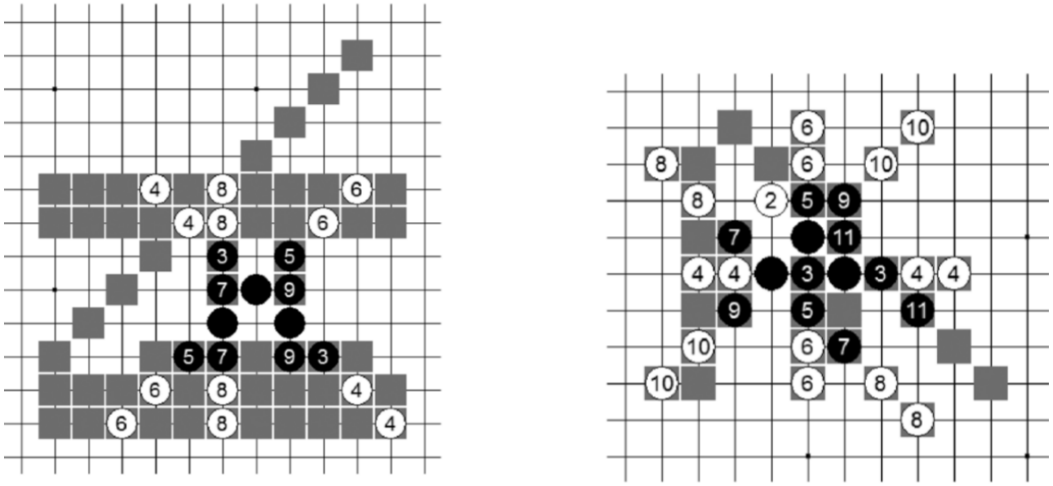
	$E = 0$	$E = 1$	$E > 1$
$D = 0$	Depth-first	Df-pn	Df-pn
$D = 1$	Depth-first	Df-pn	Df-pn
$D > 1$	Depth-first	Df-pn	Intermediate

3.3 Deep df-pn in Connect6

In this section, we implement a relevance-zone-oriented Deep df-pn and make Deep df-pn work efficiently in Connect6. We first introduce the game of Connect6, then introduce the concept of relevance zone. Finally, we present the structure of relevance-zone-oriented Deep df-pn.

3.3.1 Connect6

Connect6 is a two-player strategy game similar to Gomoku. It is first introduced by Wu and Huang [56] as a member of the connect games family. The game of Connect6 is played as follows. Black (first player) places one stone on the board for its first move. Then both



(a) relevance zone Z (b) relevance zone Z'

Figure 3-3: An example of (a) relevance zone Z and (b) relevance zone Z' [55]

players alternatively place two stones on the board at their turn. The player who first obtains six or more stones in a row (horizontally, vertically or diagonally) wins the game. Connect6 is usually played on a (19×19) Go board. Both the state-space and game-tree complexities are much higher than those in Gomoku and Renju. The reason is that two stones per move results in an increase of branching factor by a factor of half of the board size. Based on the standard used in [51], the state-space complexity of Connect6 is 10^{172}, the same as that in Go. If a larger board is used, the complexity is much higher. So finding a way to cut down the branching size of the proof tree is important for solving positions of Connect6.

As Wu and Huang mentioned in [56], threats are the key to winning Connect6. A player A has t and only t threats, if and only if t is the smallest number of stones that the opponent B needs to place to prevent B from losing the game on the next move. A move is called a single-threat move if the player who makes the move has one and only one threat after the move; it is a double-threat move if the player who made the move has two and only two threats after the move, a triple-threat move if precisely three threats exist, and a non-threat move if none threat exists. In [56], Wu and Huang showed a type of winning strategy by making continuously double-threat moves and ending with a triple-or-more-threat move or connecting up to six in all variations. This is called victory by continuous double-threat-or-more moves (VCDT). Therefore, using a VCDT solver is

a key method to reduce the complexity of solving a position of Connect6.

3.3.2 Relevance-zone-oriented Deep df-pn

The implementations of a relevance-zone-oriented Deep df-pn (i.e., a Deep df-pn procedure and a VCDT solver) is used to find winning strategies and to derive a relevance zone for Deep df-pn to cut down the branch size. According to the description in [55], the relevance zone is a zone of the board in which the defender has to place at least one of the two stones, otherwise the attacker will simply win by playing a VCDT strategy. Such a zone indicates which moves are necessary for the defender to play. It helps to cut down the branch size of the proof tree. The relation between Deep df-pn and the relevance zone is as follows. When Deep df-pn generates new moves for the defender, it first generates a null move which means that the defender places no stone for this move. Then the VCDT solver is started for the attacker. If a winning strategy is found, the VCDT solver derives a relevance zone Z (it is a zone where defense is necessary). Subsequently, the defender places one stone on each square s in Z to generate seminull moves. For each seminull move, the VCDT solver starts to derive a relevance zone Z' corresponding to this seminull move. As a result, all the necessary moves of the defender are generated by setting one stone on square s in Z and another on one square in Z' corresponding to the seminull move at s. The size of generated defender moves is far smaller than the one without relevance zone. Fig 3-3 shows an example of relevance zone Z and Z'. For the next step, the VCDT solver starts to analyze the best move for each new position derived from these defender moves. If VCDT solver finds a winning strategy, then it returns a win to Deep df-pn. If not, Deep df-pn is continued recursively.

In this section we chose Connect6 as a benchmark to evaluate the performance of Deep df-pn. We first present the experimental design, then we show and discuss the experimental results. Next, we compare the performance of Deep df-pn and $1 + \epsilon$ trick [36]. Finally, we propose a method to find the relatively optimized parameters of Deep df-pn.

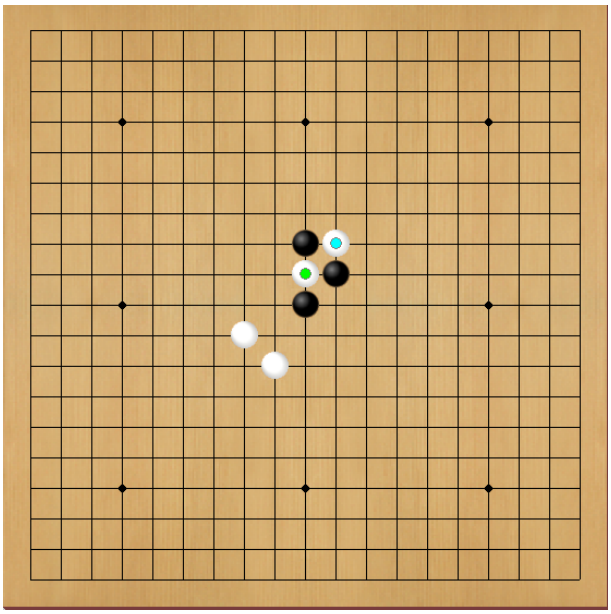


Figure 3-4: Example position 1 of Connect6 (Black is to move and Black wins)

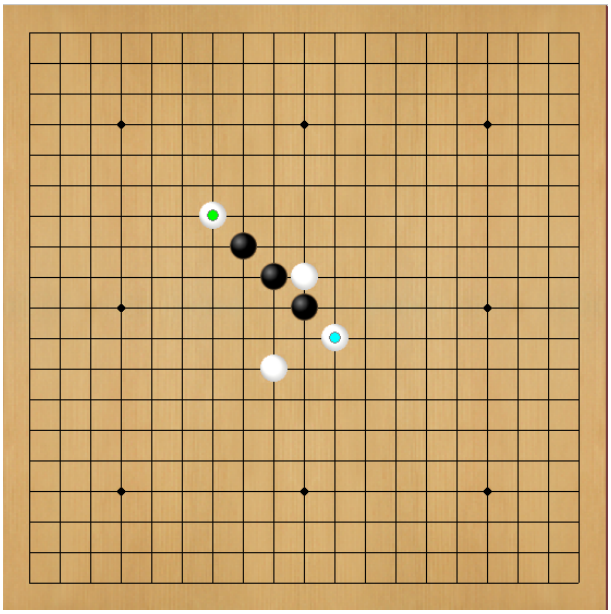


Figure 3-5: Example position 2 of Connect6 (Black is to move and Black wins)

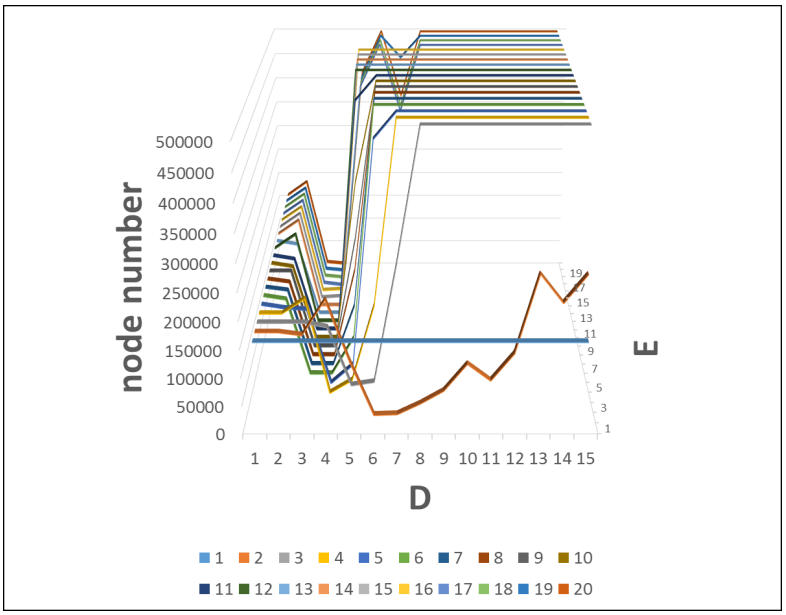


Figure 3-6: Deep df-pn and df-pn compared in node number (including repeatedly traversed nodes) with various values of parameter E and D for position 1 (Df-pn when $D = 1$)

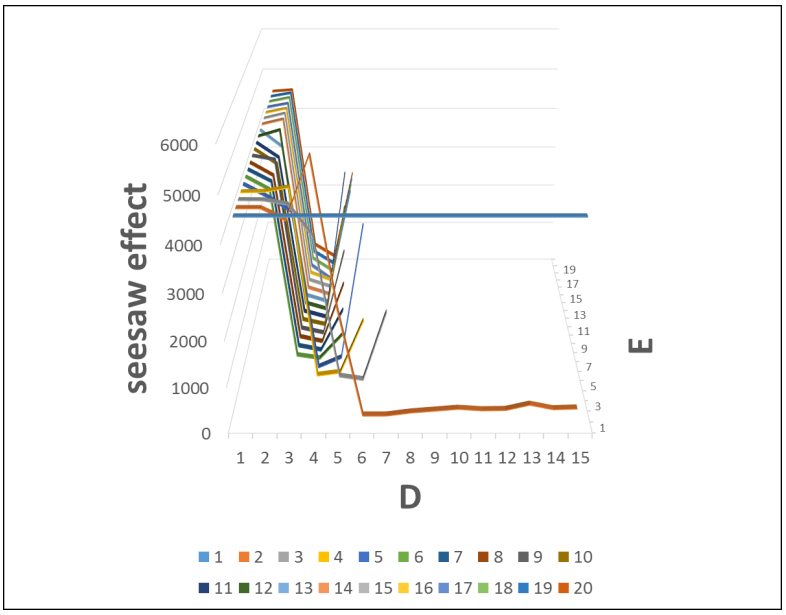


Figure 3-7: Deep df-pn and df-pn compared in seesaw effect number with various values of parameter E and D for position 1 (Df-pn when $D = 1$)

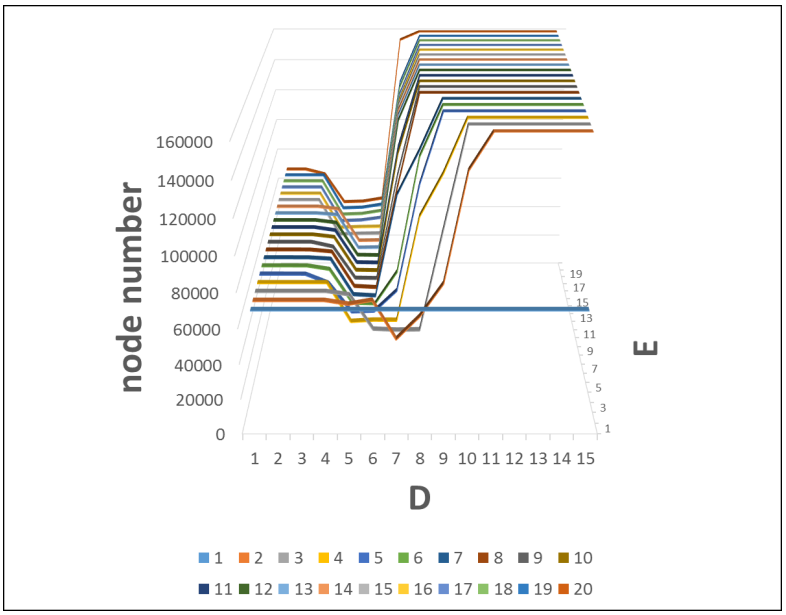


Figure 3-8: Deep df-pn and df-pn compared in node number (including repeatedly traversed nodes) with various values of parameter E and D for position 2 (Df-pn when $D = 1$)

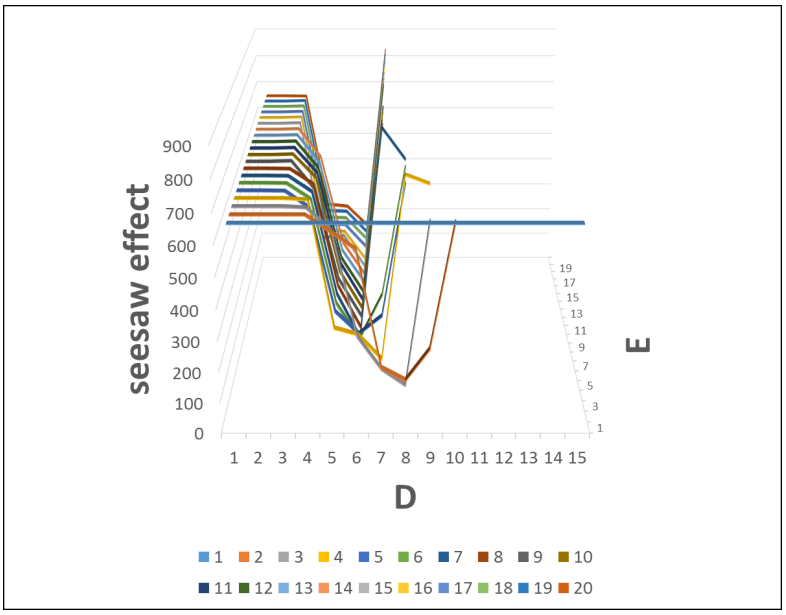


Figure 3-9: Deep df-pn and df-pn compared in seesaw effect number with various values of parameter E and D for position 2 (Df-pn when $D = 1$)

3.3.3 Experimental Design

To solve the positions of Connect6, we use relevance-zone-oriented Deep df-pn. Each time Deep df-pn generates the defender's moves, the VCDT solver generates relevance zones to indicate the necessary moves which the defender has to set on the board. Here, we remark that each time Deep df-pn generates the attacker's moves, it only generates the top 5 evaluated moves (according to some heuristic values) to reduce the complexity. Moreover, we did not recycle the child nodes after Deep df-pn has returned to its parent to reserve the winning path. Actually, these nodes *can* be recycled when Deep df-pn returns to its parent and are generated again for next time, if the cost of storage is considered. The VCDT solver is implemented with the techniques of iterative deepening and transposition table to control the time. It can search up to a depth of 25 where the longest winning path is 13 moves.

In this chapter, we first investigate 2 positions: position 1 (see Fig. 3-4) and position 2 (see Fig. 3-5). We use Deep df-pn to solve these positions with various values of parameter E and D (D is from 1 to 15 with a step length 1, and E is from 1 to 20 with a step length 1. Totally we can get 300 results). Then we get a series of changing curves of the node number (see Fig. 3-6 and Fig. 3-8) and the seesaw effect number (see Fig. 3-8 and Fig. 3-9) for parameter E and D. In this chapter, the node number equals VCDT node number + Deep df-pn node number. It includes repeatedly traversed nodes. And the seesaw effect number is initialized as 0 and increased by 1 when a node in Deep df-pn is traversed again. To obtain these curves efficiently, we set a threshold to the node number (500000 for position 1 and 160000 for position 2). When the node number of solving a position is already larger than the threshold, the solver will shut down to reduce the time cost, then we use the value of the threshold to replace the exact node numbers and use blank points to replace the exact seesaw effect numbers in the curves. The pattern of the search time is almost the same as the node number, so we do not show it in this chapter.

Moreover, we select other 6 positions (see Appendix) which can be solved by df-pn and apply Deep df-pn to them. Among all the positions (together 8 positions), 4 positions (see Fig. 3-4, Fig. 3-5, Fig. 6-1, and Fig. 6-4) are four-moves opening (Black has 2 moves and White has 2 moves), and Fig. 6-2 is a special opening, in which White sets two useless stones for its first move and Black is proved to win. We apply Deep df-pn with the best

selected E and D (E is selected out from 1 to 20, and D is selected out from 1 to 15) for each position, and present the experimental results of the 8 positions in Table 3.2 (column "Deep df-pn").

All the experiments are implemented on the computer with Windows 10 x64 system and Core i7-4790 CPU.

3.3.4 Results and Discussion

The first position of analysis is Fig. 3-4 (Black is to move and Black wins). If $E = 0$, Deep df-pn is a depth-first search which takes far more time than the original df-pn [1]. So we do not present it in this chapter. If $E > 0$ and $D > 0$, a series of changing curves for each value of parameter E and D can be obtained as shown in Fig. 3-6, and Fig. 3-7 with respect to the node number, and the seesaw effect, respectively. According to the curves, if $D = 1$ or $E = 1$, Deep df-pn is the same as df-pn. As E and D increase within a boundary, the node number and the seesaw effect number decrease, because Deep df-pn is forced to search deeper and obtains the solution faster. If E or D becomes too large, Deep df-pn is forced to search too deep. As a result, it takes more cost and causes more seesaw effect in the search process. When E and D are well chosen, Deep df-pn can get an optimal performance for a certain position. The second position of investigation is Fig. 3-5 (Black is to move and Black wins). It has a similar result as above. The changing curves obtained from Fig. 3-5 are presented in Fig. 3-8 and Fig. 3-9.

We also conduct experiments on other 6 positions (see Appendix) and present the experimental results of all the positions (together 8 positions) in Table 3.2 (column "Deep df-pn"). The experimental data is generated by Deep df-pn solver with the best selected parameter E and D (E is selected out from 1 to 20, and D is selected out from 1 to 15) for each position. According to the table, we can conclude that Deep df-pn with the best selected parameters is more efficient than original df-pn, because it reduces the node number and the seesaw effect number significantly.

[1] In this section, Deep df-pn is actually a relevance-zone-oriented Deep df-pn and the original df-pn is a relevance-zone-oriented df-pn for the application in Connect6.

3.3.5 Comparison

There is other techniques also trying to solve the seesaw effect, such as $1+\epsilon$ trick [36]. The algorithm of $1+\epsilon$ trick is almost the same as original df-pn. The only difference is the way of calculating the threshold $n_c.th_\delta$, which is presented below.

$$n_c.th_\phi = n.th_\delta + n_c.\delta - \sum n_{\text{child}}.\phi$$

$$n_c.th_\delta = \min(n.th_\phi, \lceil n_2.\delta(1+\epsilon)\rceil)$$

ϵ is a real number bigger than zero. If ϵ increases, $1+\epsilon$ trick searches deeper and usually has less seesaw effect. If ϵ equals a very small number, $1+\epsilon$ trick works the same as the df-pn.

To compare the performance, we implement a $1+\epsilon$ trick solver and conduct experiments on position 1 (see Fig. 3-4) and position 2 (see Fig. 3-5). The experimental results of position 1 are presented in Fig. 3-10 and Fig. 3-11. And the experimental results of position 2 are presented in Fig. 3-12 and Fig. 3-13. These figures show the changing curves of the node number and the seesaw effect number for various values of parameter ϵ. Here, ϵ is from 0.05 to 15 with a step length 0.05 (totally 300 items). To obtain these curves efficiently, we set a threshold to the node number (500000 for Fig. 3-4 and 160000 for Fig. 3-5). When the node number of solving a position is already larger than the threshold, the solver will shut down to reduce the time cost, and then we use the threshold value to replace the exact node numbers and use blank points to replace the exact seesaw effect numbers in the curves. According to the figures, the curves of $1+\epsilon$ trick are not so consistent as Deep df-pn's, so it is more likely affected by the *noise effect*. The noise effect can be concluded as hugely jumping up or down of solving time caused by slightly changing parameters of the modification which forces df-pn to stay longer in a subtree to avoid frequently switching to another branch. Considering that ϵ is a real number with an infinitesimal scale, it is more difficult for $1+\epsilon$ trick to find an optimal parameter ϵ in practice, while it is easy for Deep df-pn to find the optimal parameters by a hill-climbing method (see subsection 3.3.6).

To compare Deep df-pn with $1+\epsilon$ trick more precisely, we collect experimental data on all the 8 positions (see Appendix). For each position, we select the best case (case with the least node number) of both two methods by adjusting the parameters and present

them in Table 3.2. The results show that Deep df-pn has a better performance (less node number and less seesaw effect number) than $1 + \epsilon$ trick on average.

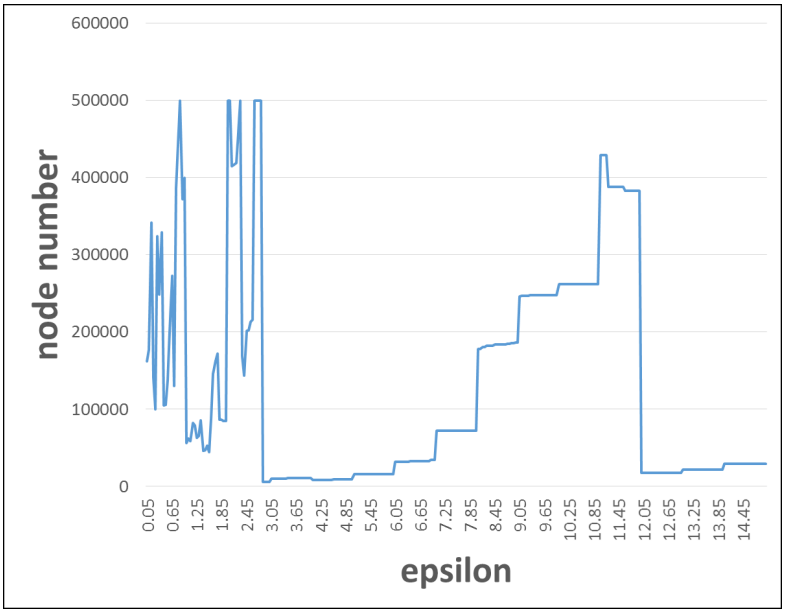


Figure 3-10: Node number (including repeatedly traversed nodes) of $1 + \epsilon$ trick with various values of parameter ϵ for position 1

3.3.6 Finding optimized parameters

For finding the optimized parameters E and D of Deep df-pn, the hill-climbing method, a kind of local search for finding optimal solutions, is used. Although hill-climbing does not necessarily guarantee to find the best possible solution, it is efficient and allows Deep df-pn to obtain a relatively better performance than the original df-pn. To avoid the noise effect which makes hill-climbing stop too early, this method is implemented to ignore some local optimums. Here, we set the $node(E, D)$ as the target function for minimizing the value of the function. The parameters E and D are input and the node number is output. The node number is computed in real time by the relevance-zone-oriented Deep df-pn solver. To control the time, we set a node number threshold N to the solver. When the node number is already larger than the threshold N, the solver will shut down and the target function will return ∞ representing that current values of parameters are not optimal and will not be considered. Relevant details of the method are presented in Algorithm 1. The procedure $isNotFlat()$ returns false, if the value of $node(E, D)$ does not change after several times iteration. And we call these continuous points (E, D) with a same value of

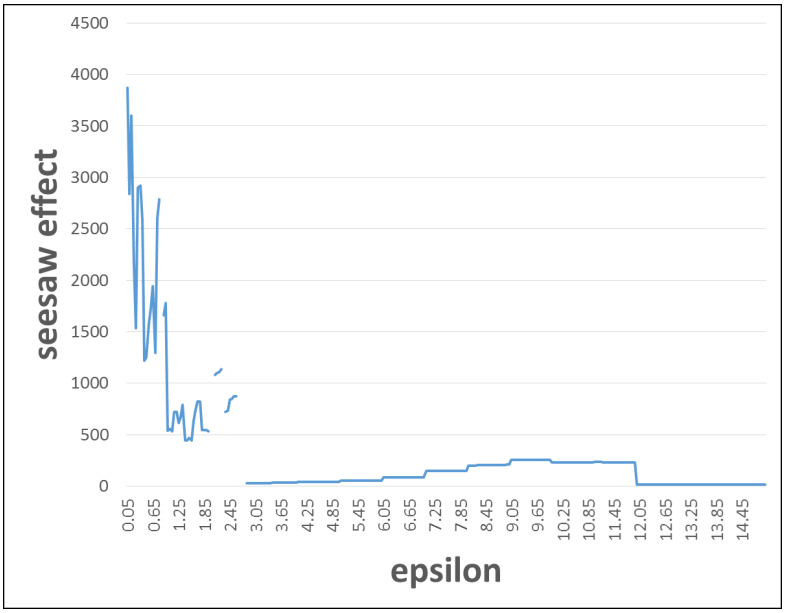


Figure 3-11: Seesaw effect number of $1 + \epsilon$ trick with various values of parameter ϵ for position 1

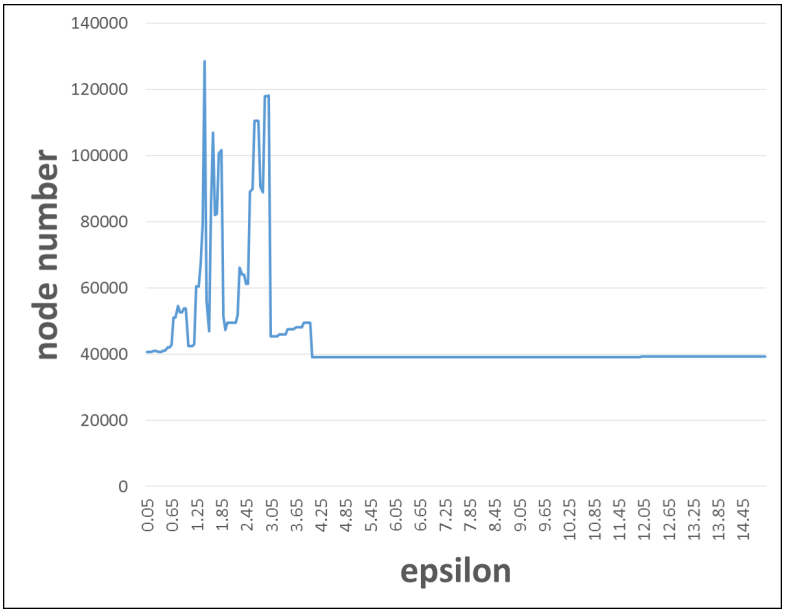


Figure 3-12: Node number (including repeatedly traversed nodes) of $1 + \epsilon$ trick with various values of parameter ϵ for position 2

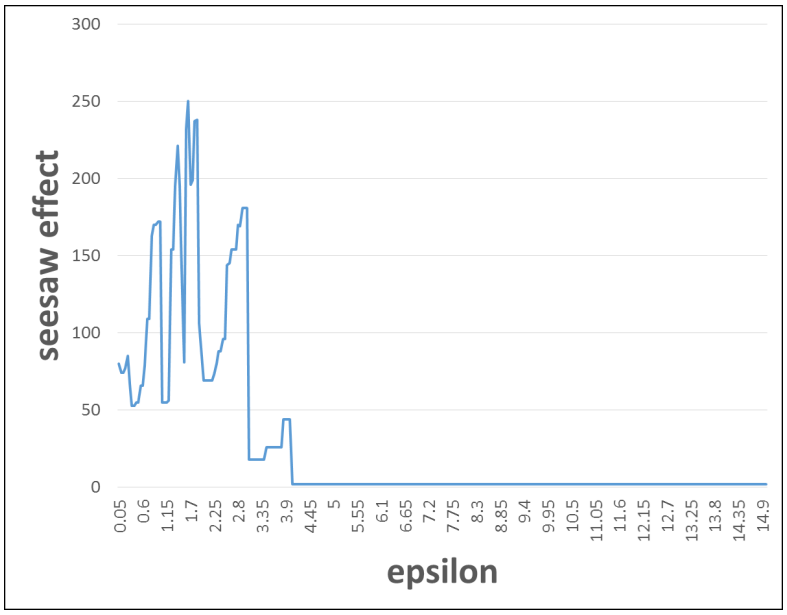


Figure 3-13: Seesaw effect number of $1 + \epsilon$ trick with various values of parameter ϵ for position 2

Table 3.2: Deep df-pn and $1 + \epsilon$ trick compared in the best case (The number in the bracket represents the reduction percentage compared with df-pn)

Position	Deep df-pn			$1 + \epsilon$ trick		
	Node number	Seesaw effect	E, D	Node number	Seesaw effect	ϵ
1	5568 (96.5%)	122 (97.3%)	17, 4	5633 (96.4%)	28 (99.4%)	2.85
2	45300 (33.7%)	101 (84.6%)	7, 6	38948 (43.0%)	2 (99.7%)	4.05
3	21157 (0.7%)	1 (95.2%)	5, 4	21309 (0%)	21 (0%)	0.05
4	99073 (17.1%)	128 (79.3%)	8, 6	95472 (20.2%)	372 (39.9%)	0.25
5	163 (99.8%)	0 (100%)	18, 2	82777 (5.7%)	936 (6.1%)	0.05
6	47213 (8.6%)	185 (35.8%)	14, 4	46255 (10.4%)	252 (12.5%)	0.15
7	74061 (45.9%)	582 (50.2%)	7, 4	143609 (-4.9%)	1158 (0.9%)	0.05
8	203188 (13.1%)	670 (38.1%)	5, 4	187198 (20.0%)	786 (27.4%)	0.25
average	61965.4 (43.5%)	223.6 (80.8%)		77650.1 (29.2%)	444.4 (61.9%)	

Table 3.3: Experimental data of Deep df-pn using hill-climbing method (The number in the bracket represents the difference between Deep df-pn using hill-climbing method and Deep df-pn in the best case)

Position	Node number	Seesaw effect	E, D	iteration time (s)
1	10529 (4961)	392 (270)	7, 4	159.7
2	45300 (0)	101 (0)	7, 6	208.6
3	21157 (0)	1 (0)	5, 4	66.4
4	107194 (8121)	912 (784)	6, 4	349.2
5	163 (0)	0 (0)	18, 2	346.5
6	50325 (3112)	268 (83)	3, 4	86.6
7	74061 (0)	582 (0)	7, 4	286.0
8	203188 (0)	670 (0)	5, 4	372.3
average	63989.6 (2024.3)	365.8 (142.1)		234.4

$node(E, D)$ as a "flat".

The experimental results of the 8 positions are presented in Table 3.3. According to the table, by using hill-climbing method, Deep df-pn can get the same performance (the difference is 0) as its best case for most of the positions. On average, the difference from the best case is small (about 3.3%: $2024.3/(63989.6 - 2024.3)$) and the iteration time is also acceptable.

3.4 Chapter Conclusion

In this chapter, we proposed a new proof-number algorithm called Deep Depth-First Proof-Number Search (Deep df-pn) to improve df-pn by reducing the seesaw effect. Deep df-pn is a natural extension of Deep Proof-Number Search (DeepPN) and df-pn. The relation between PN-search, df-pn, DeepPN and Deep df-pn was discussed. The main difference between Deep df-pn and df-pn is the proof number or disproof number of unsolved nodes. It is 1 in df-pn, while it is a function of depth with two parameters in Deep df-pn. By adjusting the values of the parameters, Deep df-pn changes its behavior from searching broadly to searching deeply which has been proved to be able to reduce the

Algorithm 1 Hill-climbing method

```
E = 2; D = 2;
while isNotFlat() do
    if node(E + 1, D) ≤ node(E, D + 1) then
        E′ = E + 1; D′ = D;
    else
        E′ = E; D′ = D + 1;
    end if
    if node(E′, D′) > node(E, D) && node(E′ + 1, D′) > node(E′, D′)
      && node(E′, D′ + 1) > node(E′, D′) then
        return E, D;
    end if
    N = node(E′, D′); E = E′; D = D′;
end while
return the minimum E and D on the flat;
```

seesaw effect. For evaluating the performance of Deep df-pn, we implemented a relevance-zone-oriented Deep df-pn to make it work efficiently in the domain of Connect6. The experimental results show a convincing effectiveness (see Table 3.2) in search efficiency, provided that the parameters E and D are well chosen.

In this chapter, Connect6 was chosen as a benchmark to evaluate the performance of Deep df-pn. Connect6 is a game with an unbalanced game tree (with a lot of sudden deaths). Our first recommendation is that further investigations will be made using other types of games with a balanced game tree (fix-depth tree or nearly fix-depth tree), such as Othello and Hex. Our second recommendation is that the procedure to find the optimal values of the parameters E and D is further analyzed and improved.

Chapter 4

Single Conspiracy Number

This chapter is an updated and abridged version of the following publications.

- Zhang Song, Hiroyuki Iida. (2017). Using Single Conspiracy Number to Analyze Game Progress Patterns. International Conference on Computer, Information and Telecommunication Systems (CITS).IEEE. Dalian, China. July. 2017.
- Zhang Song, Hiroyuki Iida. (2018). Using Single Conspiracy Number for Long Term Position Evaluation. ICGA Journal 40(3): 269-280.

4.1 Introduction

Conspiracy Number Search (CNS) is a MIN/MAX tree search algorithm to selectively expand nodes in the tree until a specified degree of confidence is achieved in the root [32]. The likelihood of the root taking on a particular value is reflected in that value's associated conspiracy number. The conspiracy number is the minimum number of leaf nodes in the tree that must change their score (by being searched deeper) to result in the root taking on that new value. CNS has been incorporated to create a strong program but suffers from low search efficiency because of its slow convergence and expensive cost of computing conspiracy numbers [41]. However, conspiracy number is still a promising indicator for measuring the "stability". Some variants inspired by CNS have been proposed to use such concept as a game-independent heuristic. The most successful one among them is Proof-Number Search.

Proof-Number Search (PNS) is one of the most efficient algorithms for solving games and complex endgame positions, inspired by the concept of conspiracy numbers [2]. PNS focuses on AND/OR tree and tries to establish the game theoretical value in a best-first manner. Each node in PNS has a proof number (pn) and disproof number (dn). A proof (disproof) number shows the scale of difficulty in proving (disproving) a node. PNS expands the most-proving node, which is the most efficient one for proving (disproving) the root. Compared with CNS, PNS is more successful in practical use, because it reduces the size of conspiracy numbers into two factors: proof number and disproof number, which improves search efficiency.

Recently, conspiracy number is gaining new grounds, such as using conspiracy number to identify critical positions for applying a speculative play [23] and using conspiracy number to improve move selection [53]. More recently a notion of Single Conspiracy Number (SCN) was proposed [57] [48], by which game progress patterns were analyzed. SCN was defined as an intermediate of conspiracy number and proof number, which indicates the difficulty of a root node changing its MIN/MAX value to a certain score decided by a threshold.

In this study, we propose a more rigorous approach to analyze game progress patterns by using SCN. AI engine 'Xiangqi Wizard' is modified to implement alpha-beta search to produce SCNs during the search process, and performance experiments are conducted on different positions of Chinese Chess. Experimental results show that the SCN is more consistent and accurate on long-term evaluation than conventional approach using evaluation function (MIN/MAX) values. One application of SCN is the Chess tutorial system. Besides the evaluation function, SCN provides another scalar axis showing the the changes of the game progress. It helps the users get more information about the game. Using SCNs together with evaluation function values enables us to better understand game progress patterns.

The structure of the chapter is as follows. In Section 4.2, we introduce the single conspiracy number and the method to analyze game progress patterns. We then conduct experiments on Chinese Chess in Section 4.3 and concluding remarks are given in Section 5.5

4.2 Basic Idea of Single Conspiracy Number

PNS is successful in practical use for reducing the size of conspiracy numbers into two factors: proof number and disproof number. Inspired by this idea, a notion of SCN is proposed as a factor combining the features of proof number and conspiracy number [57] [48].

SCN shows the difficulty of a node getting a value not less than T, where T is a threshold of legal MIN/MAX values. When T equals the maximum legal MIN/MAX value, the SCN is equivalent with the proof number. When T equals the minimum legal MIN/MAX value, the SCN is 0 because there is no difficulty for a node to get a value not less than the minimum. When T is between the maximum and the minimum legal MIN/MAX value, the SCN indicates the difficulty of a position getting a score not less than T.

Compared with the conventional evaluation way using MIN/MAX values, SCN is somehow more game-independent and expected to obtain more information on game progress patterns while showing the potential change of the MIN/MAX values. Therefore, SCN would be a good supplement to evaluation function values for analyzing game progress patterns. The formalism is presented below.

Let $n.scn$ be the SCN of a node n and m be the MIN/MAX value of n. T is a threshold of legal MIN/MAX values.

When n is a terminal node

(a) If $m \geq T$,

$$n.scn = 0.$$

(b) If $m < T$,

$$n.scn = \infty.$$

When n is a leaf node (not terminal)

(a) If $m \geq T$,

$$n.scn = 0.$$

(b) If $m < T$,

$$n.scn = 1.$$

When n is an internal node

(a) If n is a MAX node:

$$n.scn = \underset{n_c \in \text{children of n}}{Min} n_c.scn$$

(b) When n is a MIN node

$$n.scn = \sum_{n_c \in \text{children of n}} n_c.scn$$

To apply the SCN, we modified the AI engine implemented with alpha-beta search following above rules (regarding pruned nodes as leaf nodes). While the engine is running (playing Chinese Chess with itself), SCNs are produced and collected. Compared with conspiracy numbers, SCN has two advantages: (1) SCN has only one factor, not being a range of number, which saves the storage and computing time. So a computer can search deeply to keep the strength of the AI engine and get more accurate data. (2) To get SCNs with different thresholds, we can simply run the AI engine for many times with different thresholds or run in parallel, because SCNs with different thresholds are independent.

Usually, evaluation function values are used to analyze game progress patterns. One who gets a higher score on a position is supposed to have an advantage and is more likely to win in the end. Such evaluation function is designed based on the human experience of playing the game under consideration. For example, in Chinese Chess, rook has a higher score than knight, and pieces on critical positions have higher score than on normal positions. These experience sometimes can give an accurate analysis of game progress patterns, but sometime it is too static and does not work well, especially in some complex positions, which will be discussed in the next section. In this chapter, we propose a more rigorous method to analyze game progress patterns by using SCN. The formalism is presented below.

Let $P_0, P_1, ..., P_k$ be the position sequence in a game. Each P_{2i} ($i \in N$ and $i \leq k$) is a position where Red is to move and each P_{2i+1} is a position where Black is to move. For each P_i, run the engine with threshold T. Then we will get SCN of the root node of position P_i denoted as $r_i.scn$. We define Relative Single Conspiracy Number (say $RSCN$)

as below.

If P_i is a position where Red is to move,

$$RSCN_i = \begin{cases} r_i.scn - r_{i+1}.scn & i < k \\ \\ \infty & i = k \end{cases}$$

If P_i is a position where Black is to move,

$$RSCN_i = r_i.scn - r_{i-1}.scn$$

In this chapter, $RSCN_i$ is used to scale the possibility of a player to get a MIN/MAX value not less than the threshold T in advance of the opponent in position P_i. Notice that smaller $RSCN_i$ corresponds to a higher possibility. Assume that P_i is a position where Red is to move. If $i = k$, then $RSCN_i$ is ∞, which means that the possibility is the lowest because Red loses (does not need to make a move) in position P_k. In the next section, performance experiments are conducted on Chinese Chess to examine if $RSCN$ gives more consistent and accurate information on game progress pattern than evaluation function values.

4.3 Experiments and Discussion

In this section, self-play experiments using a Chinese Chess program are performed to evaluate our proposed idea with SCN, and the results are discussed. This study is an initial work of applying SCN to analyze game progress patterns. So we start the work from simple positions with clear outcome to find the potential relations between SCN and game progress. Based on these work, we propose a hypothesis and show some evidence. Future work will focus on more complex and general situations.

4.3.1 Experimental Design

To examine the effectiveness of using SCNs, self-play experiments are conducted on Xiangqi Wizard (Light version) which is a famous AI engine of Chinese Chess where alpha-beta search and iterative deepening are mainly incorporated. The score distribution is from -10000 to 10000. It is sparser from 600 to 1000 than from 0 to 600. And it is symmetric for the score smaller than 0. For the experiments, the original engine is modified to produce SCNs. Particularly, the principal variation search [37] and transportation table are removed because they prune too many branches which make SCNs always equal to 1. We run the modified AI engine for self-play experiments on three different types of test positions: (1) tactical positions, (2) drawn positions, and (3) opening positions. All of these positions are the positions where Red is to move. Below we show the detail of these experiments and its results are discussed.

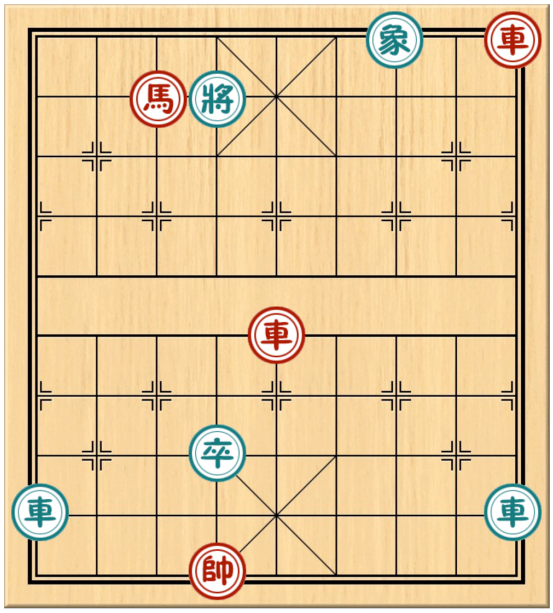


Figure 4-1: An example of tactical position where Red is to move (Red wins)

4.3.2 Tactical Positions

In Chinese Chess, there are endgame problems called tactical positions. These positions are usually not taken from real games, but are specially composed as puzzles. In tactical positions, Red is threatened by Black. To survive, Red has to checkmate the opponent until the winning, or it loses the game. There are two reasons to select tactical positions

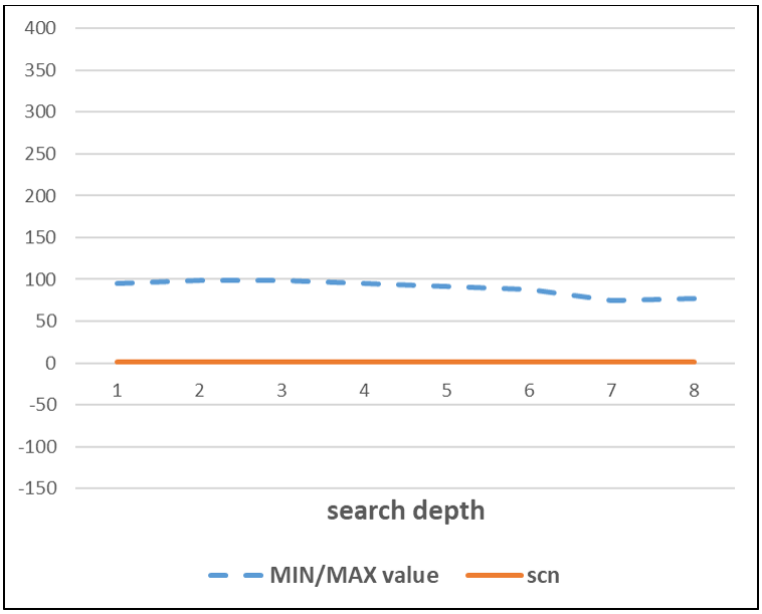


Figure 4-2: Red's MIN/MAX value and SCN in position P_0 with different search depth ($T = 600$)

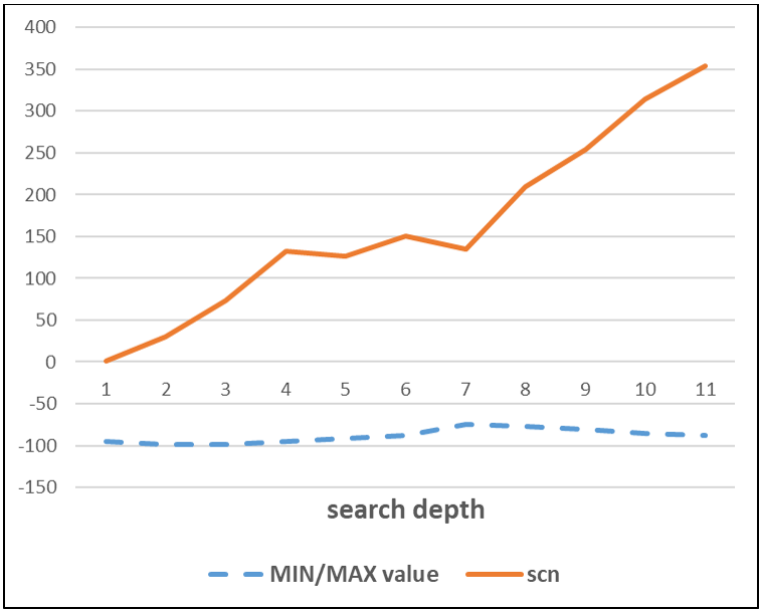


Figure 4-3: Black's MIN/MAX value and SCN in position P_1 with different search depth ($T = 600$)

as a test position: (1) in these positions, Red absolutely has more advantages than Black as it can make continuous checkmates until Red wins; (2) starting from a tactical position, each subsequent position where Red is to move is still a tactical position, so $RSCN$s and MIN/MAX values of all the subsequent positions are almost symmetric. Therefore, it is possible to obtain the average $RSCN$ denoted as $RSCN_{red}$ to reduce the noise. The same way is applied to Black, so we denote the average $RSCN$ as $RSCN_{black}$.

Fig. 4-1 shows an example of tactical position. Suppose that this position is P_0 and the subsequent positions starting from P_0 are $P_1, P_2..., P_k$. After running the AI engine with iterative deepening for one move for each player (set $T = 600$ which is relatively closed to the maximum of legal evaluation values), Fig. 4-2 and Fig. 4-3 come out. Fig. 4-2 shows the curves of Red's MIN/MAX values and SCNs in position P_0 with different search depth. Fig. 4-3 shows the curves of Black's MIN/MAX values and SCNs in position P_1 with different search depth. After continuously running the engine until the end of game, two series of curves corresponding with positions P_{2i} and positions P_{2i+1} respectively are obtained.

To compute $RSCN_i$ in position P_i where Red is to move, we first compute the linear regression function of the SCN curves of position P_i and position P_{i+1}, denoted as $\hat{f}_i(d)$ and $\hat{f}_{i+1}(d)$ where variable d stands for the search depth (if the SCN in the curves is infinity, it is abandoned to keep the consistency. If the total search depth is smaller than 4, the curve is abandoned). Then let $r_i.scn = \hat{f}_i(10)$ and $r_{i+1}.scn = \hat{f}_{i+1}(10)$. According to the definition of $RSCN$, we have

$$RSCN_i = \begin{cases} \hat{f}_i(10) - \hat{f}_{i+1}(10) & i < k \\ \\ \infty & i = k \end{cases}$$

Similarly, for $RSCN_i$ in position P_i where Black is to move, we have

$$RSCN_i = \hat{f}_i(10) - \hat{f}_{i-1}(10).$$

Finally, following above methods, sequence $RSCN_0, RSCN_2, ..., RSCN_{2i}$ for Red and sequence $RSCN_1, RSCN_3, ..., RSCN_{2i+1}$ for Black are obtained. As discussed above, $RSCN$ in each sequence are almost symmetric. Therefore, it is possible to get the average of $RSCN$ denoted as $RSCN_{red}$ and $RSCN_{black}$ to reduce the noise. The formalism is as below.

$$RSCN_{red} = \frac{\sum_{i=0}^{k_r} RSCN_{2i}}{k_r},$$

$$RSCN_{black} = \frac{\sum_{i=1}^{k_b} RSCN_{2i+1}}{k_b},$$

where k_r is the number of $RSCN_{2i}$ and k_b is the number of $RSCN_{2i+1}$ in the sequence.

To compare the result with MIN/MAX values, we also compute the linear regression function $\hat{g}_i(d)$ of the MIN/MAX value curves (if the MIN/MAX value in the curves is infinity, it is abandoned to keep the consistency. If the total search depth is smaller than 4, the curve is abandoned). The formalism is as below.

For position P_i,

$$Vl_i = \hat{g}_i(10).$$

Then compute Vl_{red} and Vl_{black} as below.

$$Vl_{red} = \frac{\sum_{i=0}^{k_r} Vl_{2i}}{k_r},$$

$$Vl_{black} = \frac{\sum_{i=1}^{k_b} Vl_{2i+1}}{k_b},$$

where k_r is the number of Vl_{2i} and k_b is the number of Vl_{2i+1} in the sequence.

Totally, 96 tactical positions are tested. Finally, we get 192 $RSCN$s and Vls. The histograms of $RSCN$ and Vl are presented in Fig. 4-4 and Fig. 4-5. According to the figures, Red's $RSCN$ is smaller than Black's $RSCN$, which means that Red has higher possibility to get a MIN/MAX value not less than $T = 600$ in advance of Black. Here, 600 is relatively a large number which is closed to the maximum among all legal MIN/MAX values. In such case, the SCN is similar to the proof number which is supposed to have better performance in endgame positions. As a result, $RSCN$ in Fig. 4-4 shows that Red

has much bigger advantages than Black, which is in accordance with what we discussed above: in tactical positions, Red has absolutely more advantages than Black, because it can make continuous checkmate until Red's wins. Moreover, there is a clear boundary between Red and Black in the histogram of $RSCN$, while there is not in the histogram of Vl. Above all, we can conclude that SCN can distinguish winning positions from losing positions better than evaluation function values.

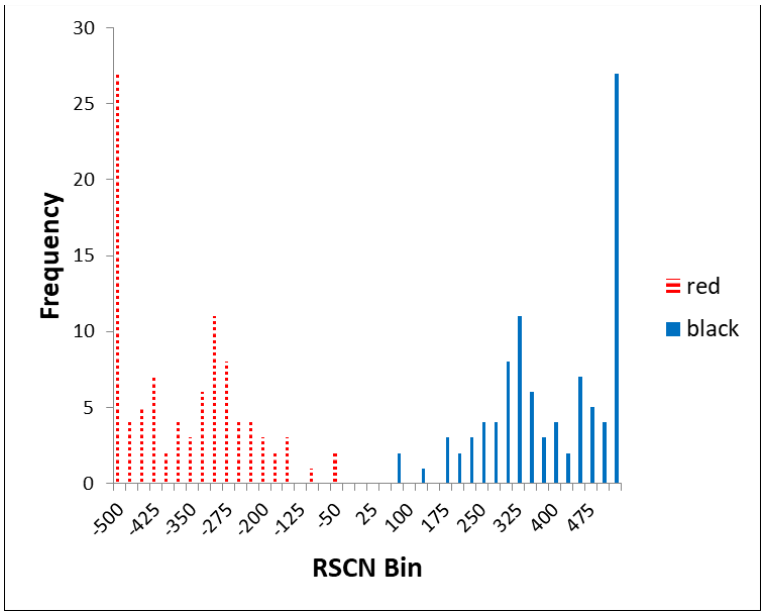


Figure 4-4: Histogram of $RSCN_{red}$ and $RSCN_{black}$ ($T = 600$)

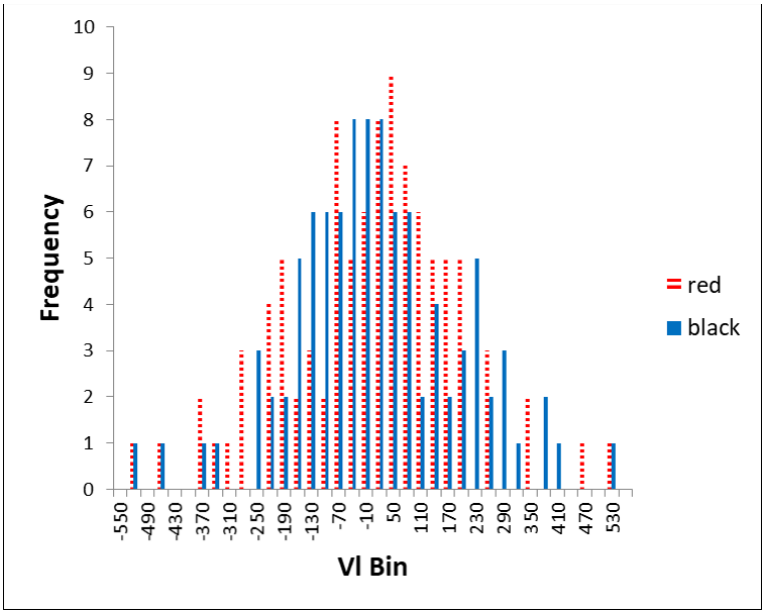


Figure 4-5: Histogram of Vl_{red} and Vl_{black}

4.3.3 Drawn Positions

To investigate the performance of $RSCN$ on analyzing drawn positions, we compose 96 drawn positions by removing all the rooks, knights, cannons and pawns from the board. Note that other pieces on the board cannot make checkmate. It is clear that both players have approximate advantages and game progress patterns are very stable in these positions. Moreover, starting from a drawn position, each subsequent position is still a drawn position, so $RSCN$s and MIN/MAX values of all the subsequent positions are almost symmetric. Therefore, we apply the same method introduced in the last subsection into drawn positions. We collect $RSCN$s and MIN/MAX values and compare with those in tactical positions. The results are presented in Fig. 4-6 and Fig. 4-7.

According to the figures, $RSCN$s from draw positions [-36.2, 36.2] are ranged between $RSCN$s from winning positions [-1021.3, -53.7] and losing positions [53.7, 1021.3]. Moreover, there are clear boundaries between winning positions, losing positions and drawn positions in the histogram of $RSCN$, whereas there is not in the histogram of Vl. Thus, we observe that SCN can distinguish winning positions, losing positions and drawn positions better than evaluation function values.

Based on the observation we made, we propose a hypothesis in the following way. For a given threshold T,

- if $RSCN \in [-1021.3, -53.7]$, then the MIN/MAX value of the position has a high possibility to change its score not less than T in advance of the opponent;
- if $RSCN \in [53.7, 1021.3]$, then the MIN/MAX value of the position has a low possibility to change its score not less than T in advance of the opponent;
- if $RSCN \in [-36.2, 36.2]$, then the MIN/MAX value of the position has a normal possibility to change its score not less than T in advance of the opponent.

Fig. 4-8 shows the relationships between high possibility, low possibility and normal possibility of changing the MIN/MAX values of positions not less than the threshold T in advance of the opponent scaled by $RSCN$. We show a preliminary verification of this hypothesis in the next subsection and further investigations will be made in the future.

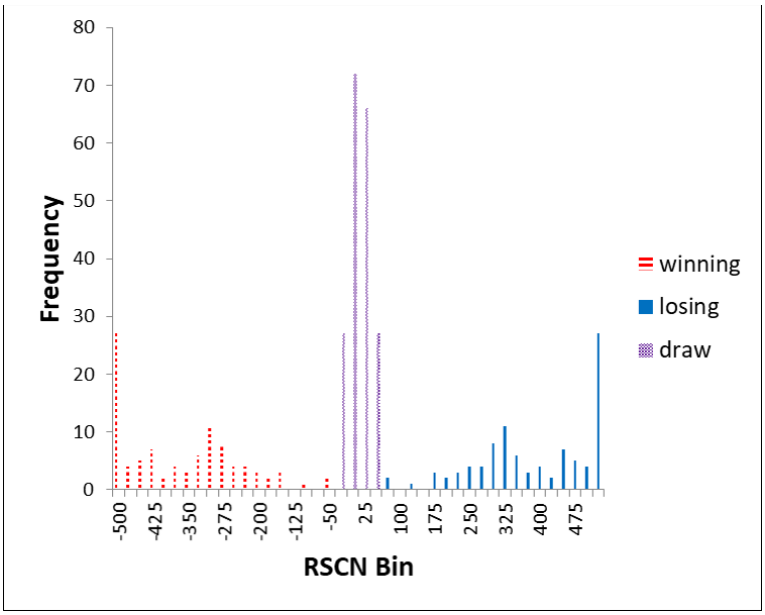


Figure 4-6: Histogram of $RSCN$ in winning positions, losing positions and drawn positions ($T = 600$)

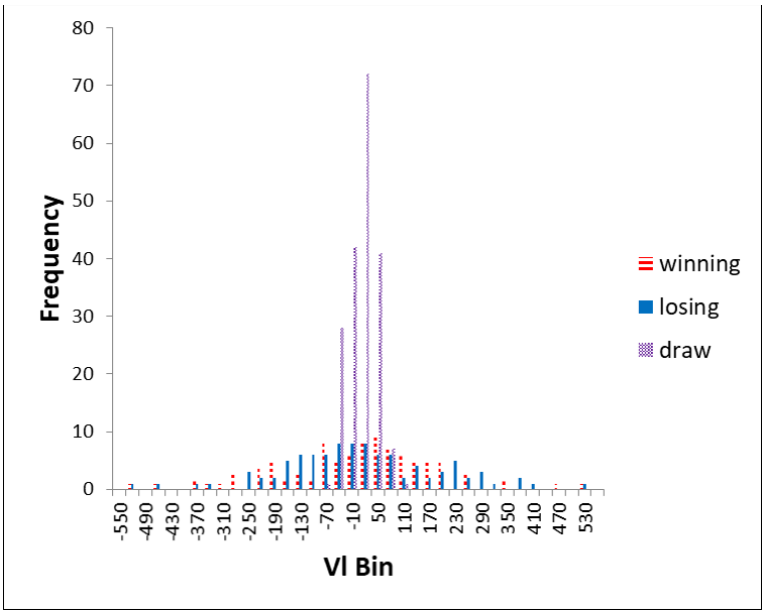


Figure 4-7: Histogram of Vl in winning positions, losing positions and drawn positions

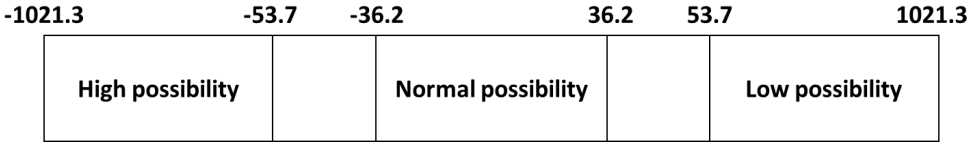


Figure 4-8: Relationship between high possibility, low possibility and normal possibility of changing the MIN/MAX values of positions not less than the threshold T in advance of the opponent scaled by $RSCN$

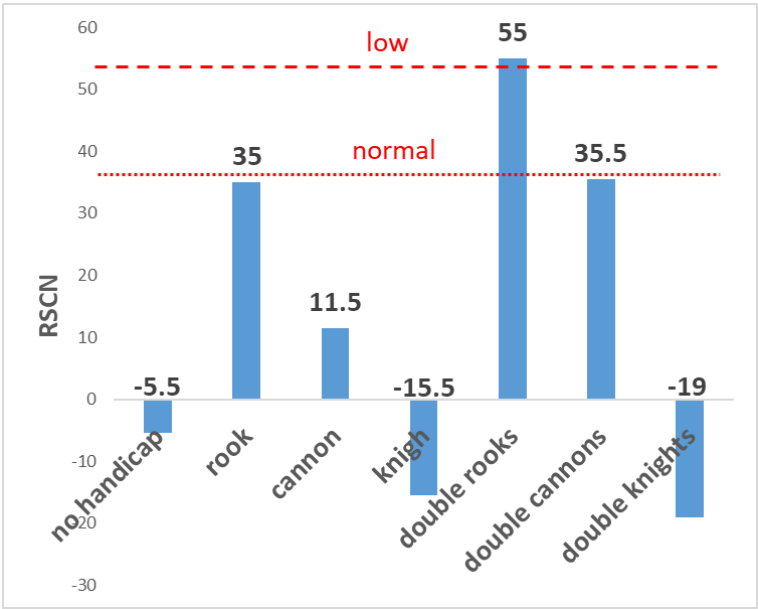


Figure 4-9: $RSCN$s for different handicap openings with $T = 200$

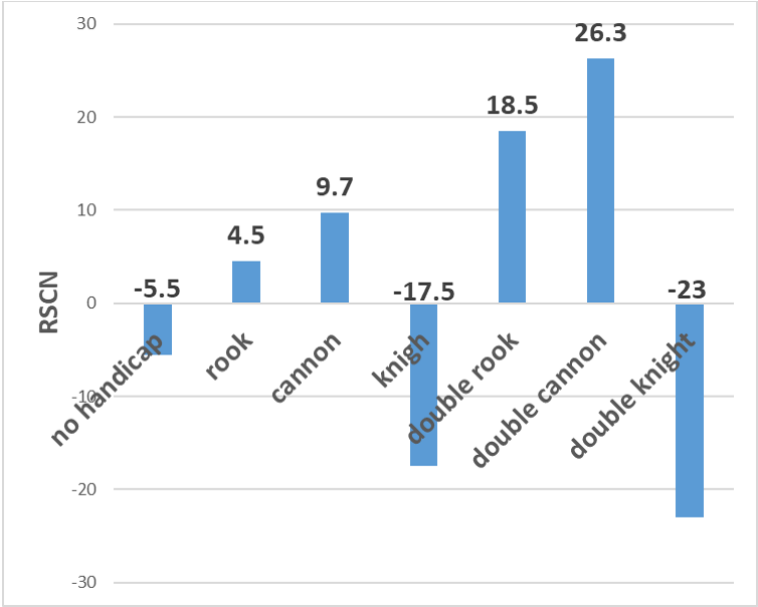


Figure 4-10: $RSCN$s for different handicap openings with $T = 600$

4.3.4 Opening Positions

In this study, opening positions are taken from handicap games. The reason why we select handicap opening positions is that the handicap openings are simpler and its outcome is clearer than normal openings and middle games. In Chinese Chess, handicap game is played in such a way that some pieces are removed from the stronger's side as handicap(s) before the game starts. The handicap pieces are usually rooks, cannons and knights. To

keep the balance, there are some protections for the handicap player. For example, if a player gives a knight handicap, its middle pawn cannot be captured unless the pawn has been moved. The MIN/MAX values of positions in a handicap game depend on the value of the handicapped pieces. Commonly, the MIN/MAX values of rook-handicap positions are much lower than knight-handicap positions because rook is much more important than knight in Chinese Chess. To analyze these positions, we apply $RSCN$ with threshold $T = 200$ corresponding to the short term advantage and $T = 600$ corresponding to the long term advantage respectively in 6 handicap openings and a normal opening. At each position of the opening considered, AI engine is run for two moves for each player to compute $RSCN$s because the SCNs and MIN/MAX values of the root nodes are almost symmetric in the first two moves for both players. The results are presented in Fig. 4-10 and Fig. 4-9.

According to the figures and the hypothesis proposed in the previous subsection, when the threshold is 200, only $RSCN$ of the double-rook-handicap opening is in the range of low possibility, which means that the handicap player is hard to get a MIN/MAX value not less than 200 in advance of the opponent. $RSCN$s of the one-rook-handicap opening and the double-cannon-handicap openings are in the range of normal possibility, but closed to the boundary, still showing a considerable disadvantage. Other handicap opening positions are all in the range of [-36.2, 36.2], which means that these opening positions are all not so disadvantageous for the handicap player, compared with other handicap opening positions on 200 score level.

It is important to notice that $RSCN$ of the no-handicap opening is smaller than zero, which is in accordance with the fact that first player takes some advantages. Moreover, $RSCN$s of knight-handicap opening positions are also smaller than zero, which means that knight-handicap takes advantages for the handicap player reversely. It seems strange, but understandable in Chinese Chess. This is because in Chinese Chess knight-handicap openings are very special. Without the knight, the rook of the handicap player has more freedom and can start an immediate attack. There are many discussions about the tactics of knight-handicap opening in Chinese Chess.

When the threshold is 600, $RSCN$s of all the openings are in the range of normal possibility, which means that these handicap openings are not so disadvantageous for the

handicap player to get a MIN/MAX value not less than 600 in advance of the opponent. So the handicap player still has chances to accumulate long-term benefits. By setting different thresholds, SCN can evaluate the positions on different levels in such a way that small thresholds correspond to the short term advantage and big thresholds corresponding to the long term advantage. This process seems like scanning the game tree repeatedly to obtain more information than MIN/MAX values which mainly concern about static evaluation such as piece material and king safety. As a result, SCN is more consistent and accurate on long-term position evaluation than conventional evaluation functions (MIN/MAX values), and using single conspiracy number together with evaluation function enables us to better understand game progress patterns.

4.4 Chapter Conclusion

Single Conspiracy Number (SCN) is a variant concept of conspiracy number and proof number which indicates the difficulty of a root node changing its MIN/MAX value to a certain score. It makes up the drawbacks of conspiracy number on computing complexity, and can be easily applied into different search frameworks.

In this chapter, we used Relative Single Conspiracy Number $RSCN$, a product of SCN as a kind of position score to analyze game progress patterns in Chinese Chess. We modified AI engine 'Xiangqi Wizard' implemented with alpha-beta search to produce SCNs during the search process, and self-play experiments using Chinese Chess AI are conducted on tactical positions, drawn positions and opening positions. The experimental results show that SCN is more consistent and accurate on long-term position evaluation than conventional evaluation functions (MIN/MAX values), and using SCN together with evaluation function enables us to better understand game progress patterns. Thus, we proposed a hypothesis about the relationships between the high possibility, the low possibility, and the normal possibility scaled by $RSCN$. We have given a preliminary verification of this hypothesis in Section 4.3.4. In future works, further experiments on more complex positions in Chinese Chess and other domains will be conducted to verify this hypothesis. Then such concept can be applied into some Chess tutorial system to help users get more information about the game.

Chapter 5

Probability-based Proof Number Search

This chapter is an updated and abridged version of the following publication.

- Z. Song, J. van den Herik, H. Iida. (2019). Probability based Proof Number Search. 11th International Conference on Agent and Artificial Intelligence (ICAART). Prague, February. 2019.

5.1 Introduction

Proof number search (PN-search) [2] is a search algorithm and is one of the most successful approaches for solving games or endgame positions. In PN-search, each node in a search tree incorporates two indicators called proof number and disproof number, respectively, indicating the "difficulty" of proving and disproving a game position corresponding with this node. For all unsolved nodes (leaf nodes), the proof number and disproof number are 1. For a winning node, the proof number and disproof number are 0 and infinity, respectively. For a non-winning node, it is the reverse. For internal nodes, the proof number and disproof number are back propagated from its children following MIN/SUM rules: at OR nodes, the proof number equals the minimum proof number of its children, and the disproof number equals the summation of the disproof numbers of its children. It is the reverse for AND nodes. For each iteration, going from the root to a leaf node, PN-search selects either the child with the minimum proof number at OR nodes, or the

child with the minimum disproof number at AND nodes. Finally, it regards the leaf node arrived at as the most proving node to expand.

PN-search is an advanced approach for proving the game-theoretic value, especially for sudden-death games that may abruptly end by the creation of one of a prespecified set of patterns such as they occur in Gomoku and Chess. PN-search works so well because the two games mentioned usually have an unbalanced game tree with various branching factors. Obviously, the proof number and disproof number are highly instrumental when the branching factor varies. As a result, the proof number and disproof number can give distinguishable information to indicate the shortest path of proving or disproving a node. For games with a balanced tree and with an almost fixed depth and an almost fixed branching factor such as Hex and Go, the PN-search is quite weak, because the proof numbers and disproof numbers are too similar to give distinguishable information.

To solve this obstacle, some PN-search variants were proposed with the idea of using a parameter to enforce a deeper search, such as Deep PN-search [20] and Deep df-pn [57]. Another possible solution is to utilize the heuristic information of the leaf node, such as df-pn* [34]. In the last few years, the Monte-Carlo tree search (MCTS) [13] has become quite successful on balanced tree games such as Go. Hence, using Monte-Carlo evaluations to obtain the proof number and disproof number of the leaf nodes is a new promising method to improve PN-search. One pointer in this direction is the Monte-Carlo proof number search (MCPN-search) [38]. MCPN-search has exactly the same rules as PN-search except that the proof number and the disproof number of unsolved nodes are derived from Monte-Carlo simulations. This method makes MCPN-search more efficient than PN-search especially in balanced tree games. However, there still is a new obstacle in MCPN-search: using the same backpropagation rules (MIN/SUM rules) as PN-search does not work well with Monte-Carlo evaluations, as the Monte-Carlo evaluation leads to a convergent real number while the MIN/SUM rules are proposed for discrete integer numbers. Hence, the Monte-Carlo evaluations will cause an information loss in the backpropagation step.

In this chapter, we propose a new application of Monte-Carlo proof number search named probability-based proof number search (PPN-search). The idea originates from the concept "searching with probabilities" [35]. The core idea is that the probability of proving

a node is computed from the probabilities of proving its children while following the AND/OR rules of probability events. The combined operation is based on the hypothesis that proving each of the children of a node is an independent event. In Palay (1983), this idea is applied on B* search [6] without using Monte-Carlo evaluations. In this chapter, we adopt the idea together with Monte-Carlo evaluations on PN-search and propose a new search algorithm called probability based proof number search (PPN-search). To show the efficiency of PPN-search, we conduct experiments on P-game trees [29] which are randomly constructed game trees with a fixed depth and a fixed branching factor, normally used to simulate balanced game trees, such as the ones occurring in Hex and Go. We compare the performance of PPN-search, MCPN-search, PN-search and other Monte-Carlo based game solvers such as the UCT solver and the pure MCTS solver [54]. The experimental results show that PPN-search outperforms other existing solvers by taking less time and fewer iterations to converge to the correct solution on average. Moreover, the error rate of the selected moves decreases faster and more smoothly as the number of iterations increases.

The rest of the chapter is organized as follows. In Section 5.2, the formalism and the algorithm of PPN-search are presented. In Section 5.3, two benchmarks about Monte-Carlo based game solvers are introduced and the relations between PPN-search and these game solvers are discussed. We conduct experiments and discuss the results in Section 5.4. Finally, we conclude in Section 5.5.

5.2 Probability-based Proof Number Search

In this section, we present the formalism (Subsection 5.2.2) and the algorithm (Subsection 5.2.3) of probability based proof number (PPN-search).

5.2.1 Main Concept

In PPN-search, only one indicator is incorporated in each node. The indicator is the probability. It indicates the probability of proving a node. The PPN of a leaf node is derived from Monte-Carlo simulations. For each iteration, for all nodes from the root to a leaf node, PPN-search selects the child with the maximum PPN at OR nodes and the

child with the minimum PPN at AND nodes. The resultant node is regarded as the most proving node for expansion. When new nodes are available, PPNs are back propagated to the root while following the AND/OR probability rules.

Similar to the proof number, the PPN is highly relevant with the branching factor because of the probability rules. So the PPN contains the information of the tree structure above the leaf nodes. Moreover, the Monte-Carlo simulations give the PPN more information beneath the leaf nodes. As a result, PPN becomes such a domain independent heuristic combining the information above and beneath the leaf nodes.

5.2.2 Probability-based Proof Number

Let $n.ppn$ be the PPN of a node n. There are three types of nodes to be discussed below.

(1) Assume n is a terminal node

(a) If n is a winning node,

$$n.ppn = 1.$$

(b) If n is not a winning node,

$$n.ppn = 0.$$

(2) Assume n is a leaf node (not terminal), and R is the winning rate computed by applying several playouts from this node. Take θ as a small positive number smaller than 1 and closed to 0 to avoid overestimating the node.

(a) If $R \in (0, 1)$,

$$n.ppn = R.$$

(b) If $R = 1$,

$$n.ppn = R - \theta.$$

(c) If $R = 0$,

$$n.ppn = R + \theta.$$

(3) Assume n is an internal node, using AND/OR probability rules of independent events.

(a) If n is an OR node,

$$n.ppn = 1 - \prod_{n_c \in \text{children of n}} 1 - n_c.ppn \tag{5.1}$$

(b) If n is an AND node,

$$n.ppn = \prod_{n_c \in \text{children of n}} n_c.ppn \tag{5.2}$$

5.2.3 Algorithm

The PPN-search includes the following four steps.

(1) Selection: for all nodes from the root to a leaf node, do select the child with the maximum PPN at OR nodes and the child with the minimum PPN at AND nodes, while regarding it as the most proving node for expansion.

(2) Expansion: expanding the most proving node.

(3) Play-out: The play-out step begins when we enter a position that is not a part of the tree yet. Moves are selected in a randomly self-play mode until the end of the game. After several play-outs, the PPNs of the expanded nodes are derived from Monte-Carlo evaluations.

(4) Backpropagation: updating the PPNs from the extended nodes to the root, while following the AND/OR probability rules given above.

5.3 Benchmarks

In this section, two benchmarks of the type Monte-Carlo based game solver are introduced (see Subsection 5.3.1 and Subsection 5.3.2). Moreover, the relations between PPN-search and these two benchmarks are discussed.

5.3.1 Monte-Carlo Proof Number Search

Monte-Carlo proof number search (MCPN-search) [38] is an enhanced proof number search by adding the flexible Monte-Carlo evaluation to the leaf nodes. We discuss three differences between MCPN-search and PPN-search.

(1) MCPN-search uses two indicators: proof number (PN) and disproof number (DN). The PN (DN) of a leaf node equals the non-winning (winning) rate derived from Monte-Carlo simulations. In contrast, PPN-search uses only one indicator PPN. The PPN of a leaf node equals the winning rate derived from Monte-Carlo simulations.

(2) MCPN-search when going from the root to a leaf node, selects the child with the minimum PN (DN) taking into consideration whether the current node is an OR (AND) node, just as the original PN-search. In contrast, PPN-search when going from the root to a leaf node, solely selects the child with the maximum PPN at OR nodes and the child with the minimum PPN at AND nodes.

(3) MCPN-search backpropagates PN and DN by following MIN/SUM rules as the original PN-search. In contrast, PPN-search backpropagates PPN by following the AND/OR probability rules of independent events.

Compared with PPN-search, an important obstacle of MCPN-search is that using the same updating rules (MIN/SUM rules) as the PN-search does not go along well with Monte-Carlo evaluations, as the Monte-Carlo evaluation leads to a convergent real number whereas the MIN/SUM rules are proposed for discrete integer numbers. It will cause an information loss in the backpropagation step. For example, Figure 5-1 shows two trees ((a) and (b)) in MCPN-search where the root is an OR node. According to the MIN rule of MCPN-search, the PN of the root equals the minimum PN of its children (being 0.2). However, tree (a) and tree (b) obtain the same PN for the root, which means that both trees have the same "difficulty" to be proved. Yet, if we investigate the PN distribution of the leaves, tree (a) is more promising to be proved because all leaves have relatively small PNs (0.2, 0.2 and 0.3). This is especially true if the PN of the leaf node is derived from Monte-Carlo evaluations which are usually slightly different. In other words, actually all branches have an influence on the root in a game tree even though the root is an OR node, especially when the proof number or disproof number indicators of leaf nodes are derived from Monte-Carlo evaluations. For PPN-search, such an obstacle will not occur.

In comparison with MCPN-search, we simply change the PNs of the leaves in Figure 5-1 to PPNs by the following operation: let PPN = 1 - PN, which corresponds to the definitions of PPN and PN. Then use the OR rule (Eq.(5.1)) to update the PPN. As is shown in Figure 5-2, tree (a) has larger PPN than tree (b), which implies that tree (a) is more promising to be proved than tree (b). This conclusion is fitting to our intuition.

Figure 5-3 and Figure 5-4 show such phenomenon for the SUM rule of MCPN-search. Here, the root is an AND node and the PN of the root equals the summation of the PNs of its children. As a result, tree (a) and tree (b) obtain the same PN which implies that both trees have the same "difficulty" to be proved. However, there is one leaf in tree (b) with a very big PN 0.8 which means that this leaf is very likely to be disproved, whereas all the leaves in tree (a) have relatively smaller and more similar PNs. As is known, for an AND node, if there exists one child that is disproved, the node will be disproved. Therefore, the SUM rule loses some information during the backpropagation process. PPN-search is able to solve this obstacle by changing the SUM rule to the AND rule (Eq.(5.2)). As is shown in Figure 5-4, tree (a) obtains a larger PPN than tree (b), which implies that tree (a) is more promising to be proved, which corresponds to our intuition.

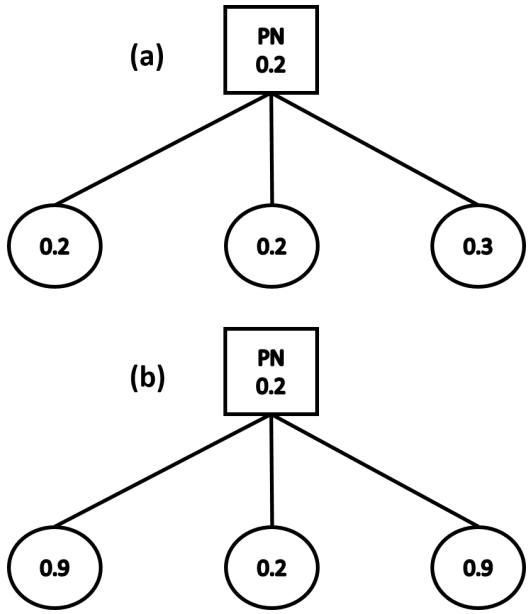


Figure 5-1: Two examples of updating PN by MIN rule in MCPN-search (the square represents the OR node).

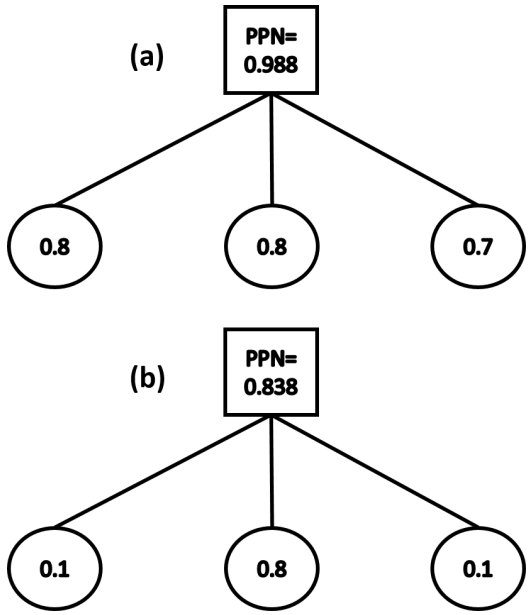


Figure 5-2: Two examples of updating PPN by OR rule in PPN-search (the square represents the OR node). Notice that PPN = 1 − PN.

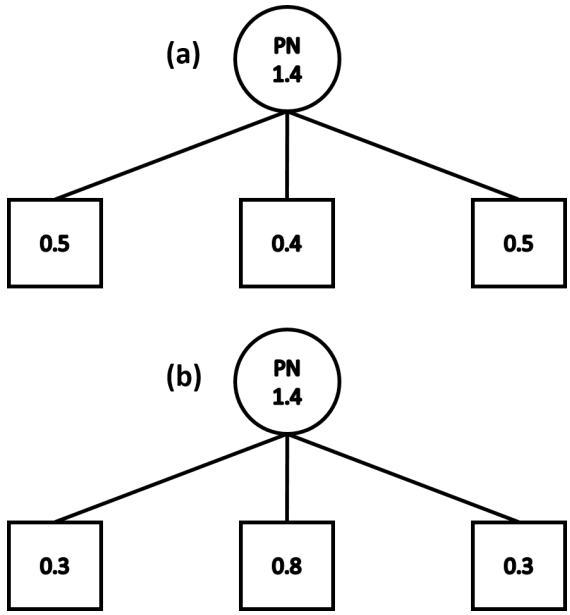


Figure 5-3: Two examples of updating PN by SUM rule in MCPN-search (the circle represents the AND node).

5.3.2 Monte-Carlo Tree Search Solver

Monte-Carlo Tree Search (MCTS) [13] is a best-first search guided by the results of Monte-Carlo simulations. In the last few years, MCTS has advanced the field of computer Go substantially. Although MCTS equipped with the UCT (Upper Confidence Bounds applied to Trees) formula which enables the evaluations to converge to the game-theoretic

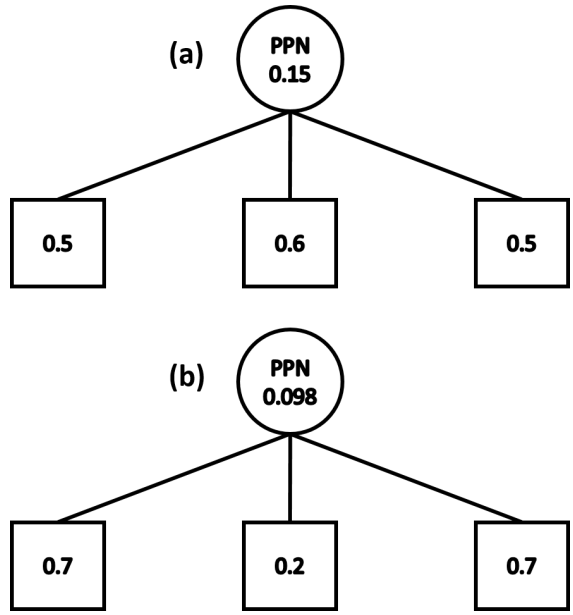


Figure 5-4: Two examples of updating PPN by AND rule in PPN-search (the circle represents the AND node). Notice that PPN = 1 − PN.

value, it is still not able to prove the game theoretic value of the search tree. This is even more true for sudden-death games, such as Chess. In this case, some endgame solvers (i.e., PN-search) are traditionally preferred above MCTS. To transform MCTS to a good game solver, Winands et al. introduced an MCTS variant called MCTS solver [54], which has been designed to prove the game-theoretical value of a node in a search tree. The MCTS solver includes the following four strategic steps.

(1) Selection: Selection picks a child to be searched based on the previously gained information. For pure MCTS when going from the root to a leaf node, the child with the largest simulation value will be selected. For UCT, an enhanced version of MCTS, it controls the balance between exploitation and exploration by selecting the child with the largest UCT value:

$$v_i + \sqrt{\frac{C \times \ln n_p}{n_i}},$$

where v_i is the simulation value of the node i, n_i is the visit count of child i, and n_p is the visit count of current node p. C is a coefficient parameter, which has to be tuned experimentally. Winands et al. also consider other strategies to optimize the selection based on UCT, such as progressive bias (PB). But in this chapter, to make it easy to follow, we only apply the UCT strategy. To transform UCT and pure MCTS to a solver,

a node is assumed to have the game-theoretical value ∞ or $-\infty$ that corresponds to a proved win or not win, respectively. In this chapter, we consider all the drawn games as proved to be not win games to make the experimental results more easy to interpret. When a child is a proven win, the node itself is a proven win, and no selection has to take place. But when one or more children are proven to be not a win, it is tempted to discard them in the selection phase. In this chapter, to make it easy to compare, i.e., we do not consider the proved win or the proved not win node in the play-out step, because such technique can similarly be applied into PPN-search and MCPN-search. Moreover, for the final selection of the winning move at the root, often, it is the child with the highest visit count, or with the highest value, or a combination of the two. In the UCT solver or in the pure MCTS solver, the strategy is to select the child of the root with maximum quantity $v + \frac{A}{\sqrt{n}}$, where A is a parameter (here, set to 1), v is the node's simulation value, and n is the node's visit count.

(2) Play-out: The play-out step begins when we enter a position that is not a part of the tree yet. Moves are selected in self-play until the end of the game. This task might consist of playing plain random moves.

(3) Expansion: Expansion is the strategic task that decides whether nodes will be added to the tree. In this chapter, we expand one node for each iteration.

(4) Backpropagation: Backpropagation is the procedure that propagates the result of a simulated game back from the leaf node, through the previously traversed node, all the way up to the root. A usual strategy of UCT or pure MCTS is taking the average of the results of all simulated games made through this node. For the UCT solver and the pure MCTS solver (in addition to back propagating the values 1,0,-1) the search also propagates the game-theoretical values ∞ or $-\infty$. The search assigns ∞ or $-\infty$ to a won or lost terminal position for the player to move in the tree, respectively. Propagating the values back in the tree is performed similar to negamax in the context of MIN/MAX searching in such a way that we do not need to distinguish between MIN and MAX nodes.

Compared with PPN-search, the main difference between the pure MCTS solver and PPN-search is the backpropagation strategy. For a pure MCTS solver, the backpropagation strategy of a node is taking the average of the simulation results of its children. In contrast, PPN-search follows the AND/OR probability rules presented in Eq.(5.1) and

Eq.(5.2). Actually, both backpropagation strategies have been discussed in an early paper of MCTS [15] that points out the weakness of AND/OR probability backpropagation rules for MCTS. Compared with taking the average, it is noted that they have to assume some degree of independence between probability distributions. This assumption of independence is wrong in the case of Monte-Carlo evaluation because the move with the highest value is more likely to be overestimated than other moves. Moreover, a refutation of a move is likely to refute simultaneously other moves of a node. Such statement [15] is true for MCTS when it is used to find an approximate best move in a game AI, but is not appropriate when MCTS is used to solve a game or a game position. There are two reasons: (1) To solve a game or a game position, the search algorithm has to go deeply until to the terminal nodes to completely prove the game-theoretic value. So it is not necessary for a search algorithm to avoid overestimating the move with the highest value. In contrast, what really matters for a search algorithm is the speed to approach the terminal nodes. (2) To solve a game or a game position, we need to search on an AND/OR tree to find the solution. Therefore, the AND/OR probability backpropagation rules are more suitable than taking the average. For example, Figure 5-5 shows two trees in an UCT solver or pure MCTS solver where the root is an OR node. Assuming that all the children have the same visit count, for updating the simulation value of the root, we take the average of the simulation value of its children. Then both trees obtain the same simulation value, which implies that both trees have the same possibility to win. However, to prove a game, things are different. As the root is an OR node, it will be proved as long as there exists one child that can be proved. In Figure 5-5, tree (b) has a child with a very large winning rate 0.9, while all children in tree (a) have relatively small winning rate, so tree (b) is absolutely more likely to be proved than tree (a). If we use AND/OR probability rules to update these simulation values, it is clear that as is shown in Figure 5-6 tree (b) obtains larger PPN values than tree (a), which means that tree (b) is more likely to be proved. And this is surely more fitting to our intuition. It is similar for the AND rule of PPN-search. Therefore, it is difficult to prove a game-theoretic value of search tree. In summary, PPN-search with AND/OR backpropagation rules is more suitable than the UCT solver and the pure MCTS solver.

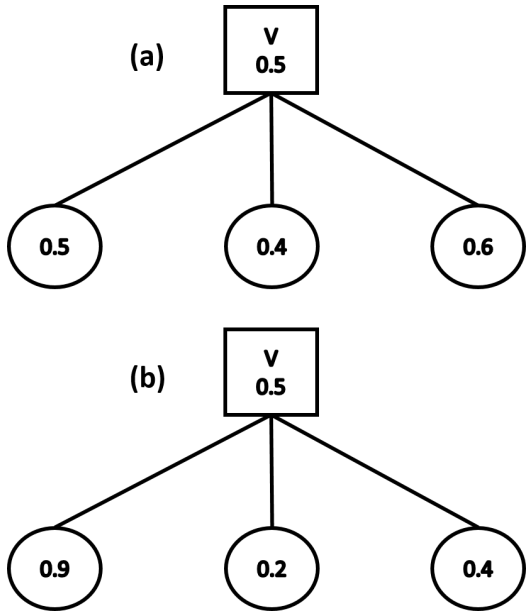


Figure 5-5: Two examples of updating simulation values by taking the average in the UCT solver or the pure MCTS solver (the square represents the OR node).

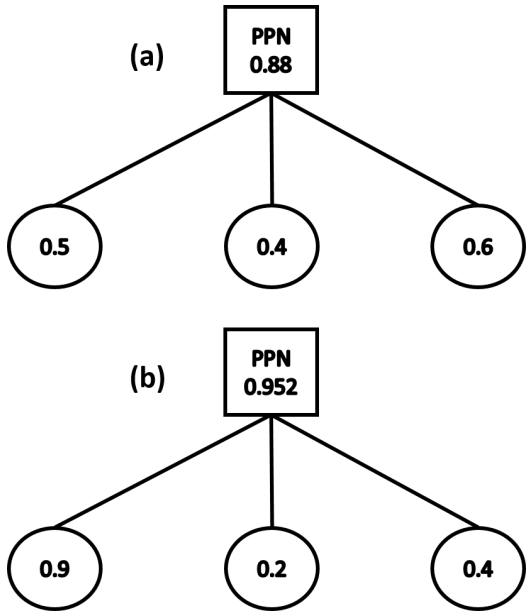


Figure 5-6: Two examples of updating PPN by OR rule in PPN-search (the square represents the OR node).

5.4 Experiments

To examine the effectiveness of PPN-Search, we conducted two series of experiments on P-game trees. The P-game tree [29] is a MIN/MAX tree where a randomly chosen value is assigned to each move. The value of a leaf node is given by the sum of the move values along the path. If the sum is positive, the result is a win for MAX, if negative it is a

win for MIN, and it is draw if the sum is 0. In all experiments, for the moves of MAX the value was chosen uniformly from the interval [0,127] and for MIN from the interval [−127,0].

In series 1, we construct 200 P-game trees randomly, with 2 branches and 20 layers, and apply five distinct types of search: PPN-search, PN-search, MCPN-search, the UCT solver, and the pure MCTS solver to prove (winning) or disprove (non-winning) these game trees. For each expanded leaf node, we set 10 playouts to compute the winning rate for the following four types of search PPN-search, MCPN-search, the UCT solver, and the pure MCTS solver (further investigation shows that the number of playouts does not influence the experimental results). Here we report experimental results as shown in Figure 5-7, Figure 5-8, and Figure 5-9. Figure 5-7 shows the average search time for proving or disproving a P-game tree with 2 branches and 20 layers for all five types of search PPN-search, PN-search, MCPN-search, the UCT solver, and the pure MCTS solver, respectively. Figure 5-8 shows the average number of iterations for proving or disproving a P-game tree with 2 branches and 20 layers for all five types of search. Figure 5-9 shows the error rate of selecting a correct solution by PPN-search, PN-search, MCPN-search, the UCT solver, and the pure MCTS solver for each iteration on P-game trees with 2 branches and 20 layers. More concretely, the error rate equals the number of wrong moves selected by the search among 200 tests divided by the testing times 200. Notice that the UCT solver or the MCTS solver expands 1 node per iteration while others expand 2 nodes. So, we regard 2 iterations of the UCT solver or the MCTS solver as 1 iteration, and present it in the figures.

In series 2, we construct 200 P-game trees randomly, with 200 trees with 8 branches and 8 layers, and apply five distinct types of search (the same ones as in series 1). Figure 5-10, Figure 5-11, and Figure 5-12 show the analogous experimental results on P-game trees with 8 branches and 8 layers. Notice that the UCT solver or the MCTS solver expands 1 node per iteration while others expand 8 nodes. So, we regard 8 iterations of the UCT solver or the MCTS solver as 1 iteration, and present it in the figures. According to the figures, compared with the four types (PN-search, MCPN-search, the UCT solver, and the pure MCTS solver), our PPN-search outperforms the others while averagely taking less time and fewer iterations to prove or disprove a game tree. Furthermore,

compared with the three types (PN-search, MCPN-search and the pure MCTS solver), our PPN-search converges faster to the correct solution, and the error rate of selected moves decreases more smoothly as the number of iterations increases. For MCPN-search, it takes more time and more iterations than PPN-search to converge to the correct solution on average, and the error rate waves as the number of iterations increases, because of its inconsistent backpropagation rules. As for the pure MCTS solver according to the figures, the performance is better than the PN-search, and competitive with MCPN-search, but worse than PPN-search. The UCT solver converges faster to the correct solution than the other types of search, but averagely takes more time and more iterations to solve a P-game tree than PPN-search, MCPN-search, and the pure MCTS solver. Here, we tested the UCT solver with different parameters but only show one of them with parameter $\sqrt{2}$ in the figures. Actually, all these UCT solvers averagely take more time and more iterations to solve a P-game tree than PPN-search, MCPN-search, and the pure MCTS solver. One possible reason is that in P-games trees, the game trees are so well balanced that the exploration strategy of UCT may not be advantageous to enforce a deep search to solve a game tree fast. In other words, UCT is a good search algorithm to find the approximate best move for a game AI, but the UCT solver is not a good search algorithm to solve a game.

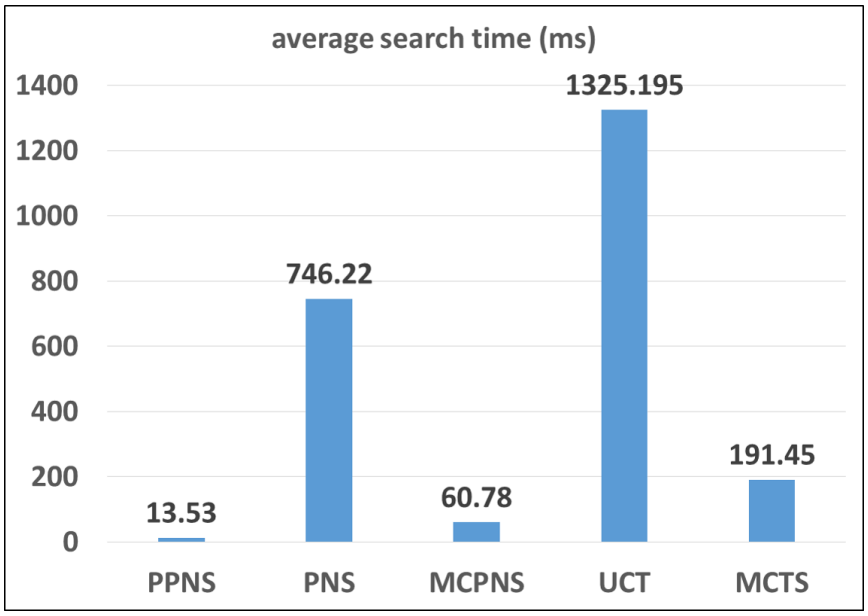


Figure 5-7: Comparison of average search time for a P-game tree with 2 branches and 20 layers.

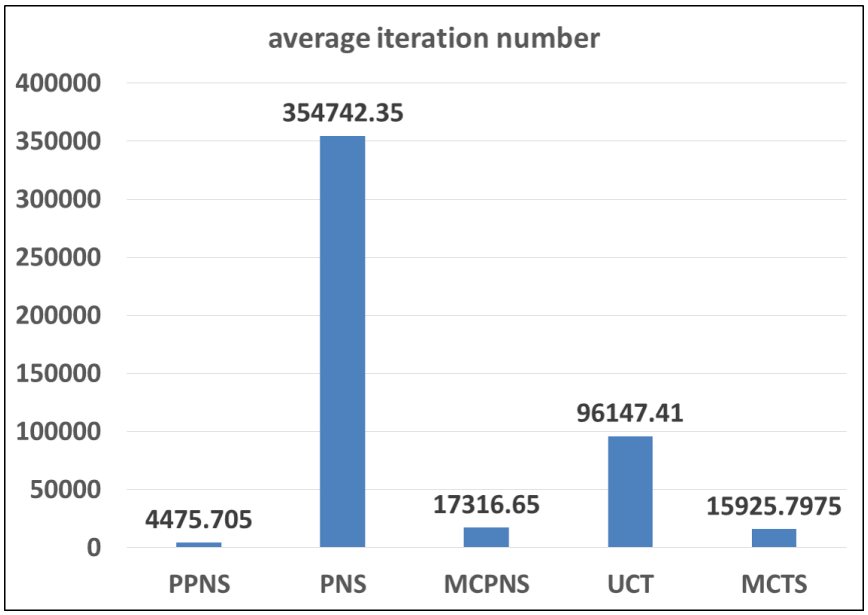


Figure 5-8: Comparison of average numbers of iterations for a P-game tree with 2 branches and 20 layers.

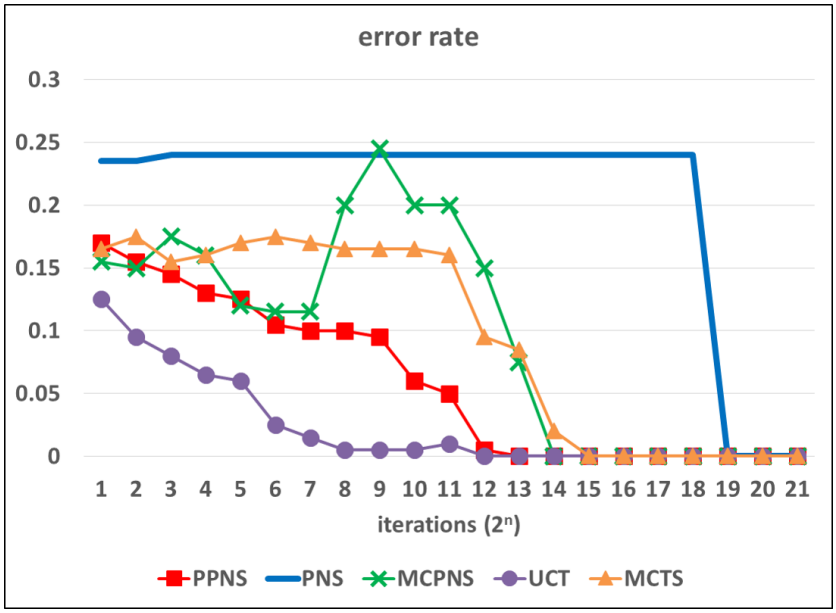


Figure 5-9: Comparison of the error rate of selected moves for each iteration on P-game trees with 2 branches and 20 layers.

5.5 Chapter Conclusion

PPN-search is a promising variant of proof number search based on Monte-Carlo simulations and probability backpropagation rules. It only uses one indicator PPN to indicate the "probability" of proving a game position, and back propagates PPNs by AND/OR probability rules of independent events. Compared with PN-search, MCPN-search, the

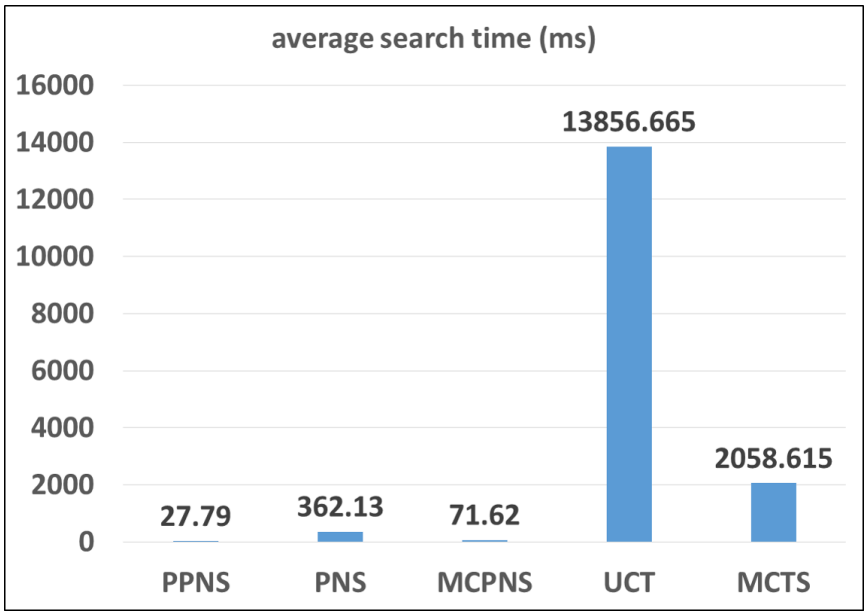


Figure 5-10: Comparison of average search time for a P-game tree with 8 branches and 8 layers.

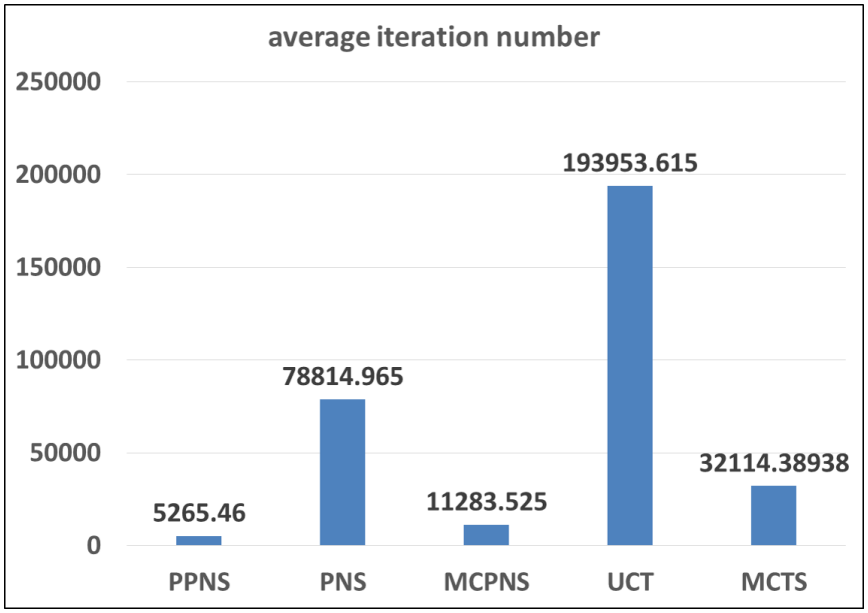


Figure 5-11: Comparison of average numbers of iterations for a P-game tree with 8 branches and 8 layers.

UCT solver, and the pure MCTS solver, PPN-search outperforms them while taking less time and fewer iterations to prove or disprove a game tree on average. Moreover, the error rate of the selected moves decreases faster and more smoothly as the number of iterations increases.

Further works may include (1) applying PPN-search into real games with large-size

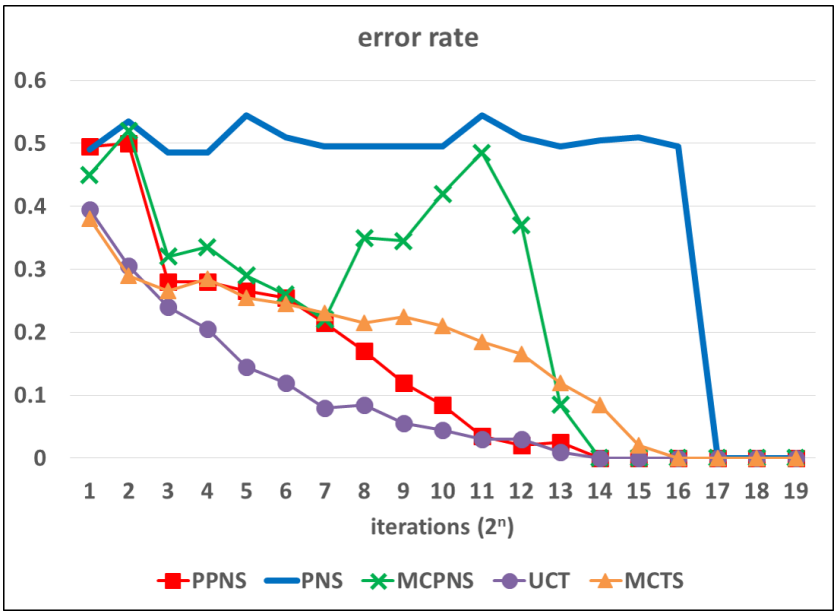


Figure 5-12: Comparison of the error rate of selected moves for each iteration on P-game trees with 8 branches and 8 layers.

balanced game trees and unbalanced game trees, respectively, to further investigate its performance; (2) proposing probability-based conspiracy number search (PCN-search) by incorporating the notion of the single conspiracy number [49].

Chapter 6

Conclusion

Conspiracy number and proof number are two game-independent heuristics in a game-tree search. The conspiracy number is proposed in Conspiracy Number Search (CNS) which is a MIN/MAX tree search algorithm, trying to guarantee the accuracy of the MIN/MAX value of a root node. It shows the scale of "stability" of the root value. The proof number is inspired by the concept of conspiracy number, and applied in an AND/OR tree to show the scale of "difficulty" for proving a node. It is first proposed in Proof-Number Search (PN-search) which is one of the most powerful algorithms for solving games and complex endgame positions. The Monte Carlo evaluation is another promising domain-independent heuristic which focuses on the analysis based on random sampling of the search space. The Monte Carlo evaluation does not reply on any prior knowledge of human and has made significant achievements in complex games such as Go.

In this thesis, we selected the conspiracy number search, the proof number search and the Monte-Carlo tree search as three example search algorithms with domain-independent heuristics to study its relations and differences, and finally gave a new perspective of the game tree search with domain-independent heuristics. The relations and differences of the three search algorithms mentioned can be summarized as follows. The Monte-Carlo tree search uses Monte-Carlo evaluations for the leaf nodes to indicate the most promising node for expansion. In other words, the Monte-Carlo evaluation can be regarded as a detector to obtain the information beneath the leaf nodes to forecast the promising search direction in advance. In contrast, the conspiracy number search and the proof number search tend to use the indicators corresponding to the structure or the shape of the part of the

search tree that has already been expanded. Therefore, it can be regarded as forecasting the promising search direction according to the information above the leaf nodes. As a natural induction of such understanding of the game tree search using domain-independent heuristics, we may get some improvements by combining the conspiracy number or the proof number idea with the Monte-Carlo evaluation into a search algorithm, which can be considered as a combination of "the information above leaf nodes" and "the information beneath the leaf nodes". To prove such hypothesis, we proposed a new search algorithm named probability-based proof number search (PPN-search) using both the proof number idea and Monte-Carlo evaluation. Experimental results showed that probability-based proof number search outperforms other famous approaches.

In this thesis, we applied and refined domain-independent heuristic such as the conspiracy number, the proof number and the Monte-Carlo evaluation to achieve such three goals: (1) enhancing current search algorithm. For this purpose, we proposed the Deep df-pn search algorithm to improve df-pn which is a depth-first version of PN-search by forcing a deep search with a parameter. The experiments performed in Connect6 show a good performance of Deep df-pn. (2) Analyzing and visualizing game progress patterns for better understanding games and master thinking way. For this purpose, We proposed the single conspiracy number method for long term position evaluation in Chinese Chess and got good results. (3) Studying the relations and differences between the conspiracy number, proof number and the Monte-Carlo evaluation and combining "the information above leaf nodes" and "the information beneath the leaf nodes" to propose a new search algorithm with domain-independent heuristics. For this purpose, we proposed the probability-based proof number search. A series of experiments showed that probability-based proof number search outperforms other famous search algorithms for solving games and endgame positions.

Appendix

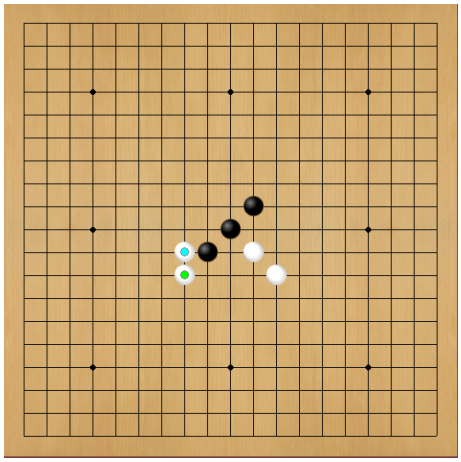


Figure 6-1: Example position 3 of Connect6 (Black is to move and Black wins)

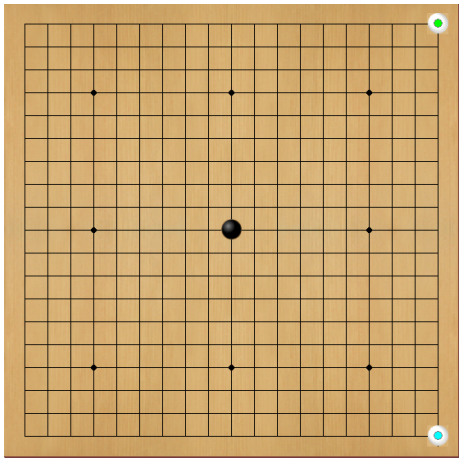


Figure 6-2: Example position 4 of Connect6 (Black is to move and Black wins)

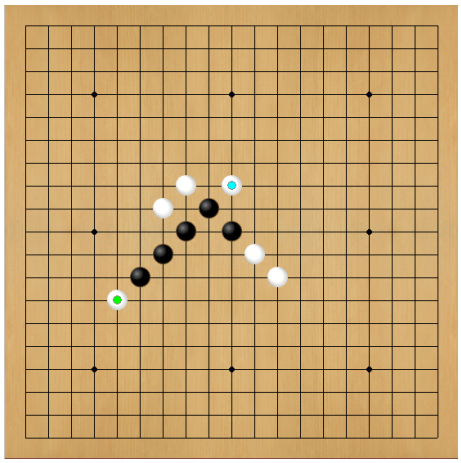


Figure 6-3: Example position 5 of Connect6 (Black is to move and Black wins)

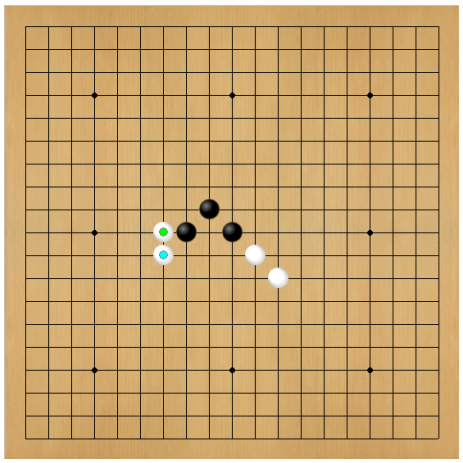


Figure 6-4: Example position 6 of Connect6 (Black is to move and Black wins)

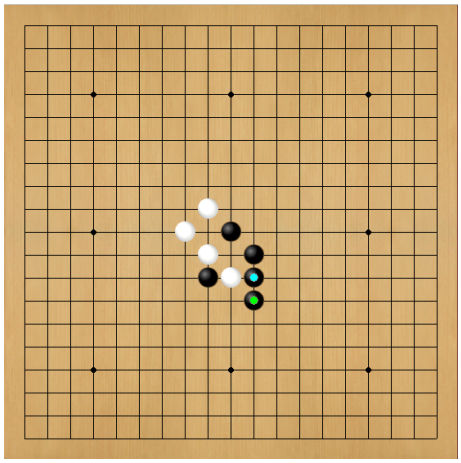


Figure 6-5: Example position 7 of Connect6 (White is to move and White wins)

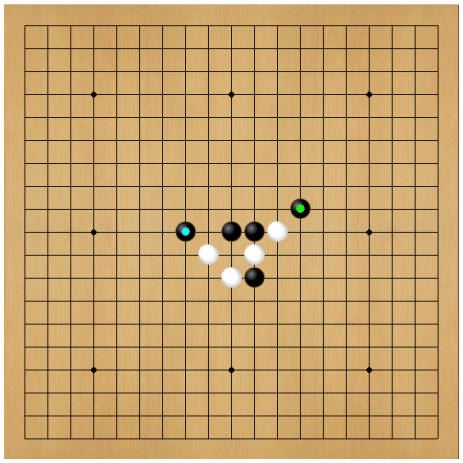


Figure 6-6: Example position 8 of Connect6 (White is to move and White wins)

Algorithm 2 Deep df-pn (part I)

```
// At the root
procedure DEEPDFPN(r)
    r.φ = ∞; r.δ = ∞;
    MID(r);
end procedure
// Exploring node n
procedure MID(n)
    // 1. Look up transposition table
    LookUpTranspositionTable(n,φ,δ);
    if n.φ ≤ φ || n.δ ≤ δ then
        n.φ = φ; n.δ = δ;
        return ;
    end if
    // 2. Generation of legal moves
    if n is a terminal node then
        if (n is an AND node && Eval(n) = true) ||
        (n is an OR node && Eval(n) = false) then
            n.φ = ∞; n.δ = 0;
        else
            n.φ = 0; n.δ = ∞;
        end if
        PutInTranspositonTable(n,n.φ,n.δ);
        return ;
    end if
    GenerateLegalMoves();
    // 3. Avoidance of cycle by using transposition table
    PutInTranspositonTable(n,φ,δ);
    // 4. Multiple Iterative Deepening
    while 1 do
        // Stop searching if φ or δ is above or equal to
        its threshold
        if n.φ ≤ ΔMin(n) || n.δ ≤ ΦSum(n) then
            n.φ = ΔMin(n); n.δ = ΦSum(n);
            PutInTranspositonTable(n,φ,δ);
            return ;
        end if
        n_c = SelectChild(n,φ_c,δ_c,δ_2);
        n_φ = n_δ + φ_c − ΦSum(n);
        n_δ = min(n.φ, δ_2 + 1);
        MID(n_c);
    end while
end procedure
// Record into the transposition table
procedure PUTINTRANSPOSTIONTABLE(n,φ,δ)
    Table[n].φ = φ; Table[n].δ = δ;
end procedure
```

Algorithm 3 Deep df-pn (part II)

```
// Look up the transposition table
procedure LookupTranspostionTable(n,&φ,&δ)
    if n is already recorded then
        φ =Table[n].φ; δ =Table[n].δ;
    else
        // In df-pn φ = 1, δ = 1
        if E = 0 then
            φ = 0; δ = 0;
        else if D ≤ n.depth then
            φ = 1; δ = 1;
        else
            φ = E^(D−n.depth); δ = E^(D−n.depth);
        end if
    end if
end procedure
// Selection of the child
procedure SelectChild(n,&φ_c,&δ_c,&δ_2)
    δ_c = ∞; δ_2 = ∞;
    for each child node n_child do
        LookUpTranspositionTable(n_child,φ,δ);
        if δ < δ_c then
            n_best = n_child;
            δ_2 = δ_c; φ_c = φ; δ_c = δ;
        else if δ < δ_2 then
            δ_2 = δ;
        end if
        if φ = ∞ then
            return n_best;
        end if
    end for
    return n_best;
end procedure
// Calculate the minimum δ among all the children
procedure ΔMin(n)
    min = ∞
    for each child node n_child do
        LookUpTranspositionTable(n_child,φ,δ);
        min = min(min, δ);
    end for
end procedure
```

Algorithm 4 Deep df-pn (part III)

```
// Calculate the summation of φ among all the children
procedure ΦSum(n)
    sum = 0
    for each child node n_child do
        LookUpTranspositionTable(n_child,φ,δ);
        sum = sum + φ;
    end for
    return sum;
end procedure
```

Bibliography

[1] Bruce Abramson. Expected-outcome: A general model of static evaluation. *IEEE transactions on pattern analysis and machine intelligence*, 12(2):182–193, 1990.

[2] L.Victor Allis, Maarten van der Meulen, and H.Jaap van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91 – 124, 1994.

[3] Vadim V Anshelevich. A hierarchical approach to computer Hex. *Artificial Intelligence*, 134(1-2):101–120, 2002.

[4] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[5] József Beck and József Beck. *Combinatorial games: tic-tac-toe theory*, volume 114. Cambridge University Press, 2008.

[6] Hans Berliner. The B* tree search algorithm: A best-first proof procedure. In *Readings in Artificial Intelligence*, pages 79–87. Elsevier, 1981.

[7] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[8] Michael Buro. The evolution of strong Othello programs. In *Entertainment Computing*, pages 81–88. Springer, 2003.

[9] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[10] Hyeong Soo Chang, Michael C Fu, Jiaqiao Hu, and Steven I Marcus. An adaptive sampling algorithm for solving Markov decision processes. *Operations Research*, 53(1):126–139, 2005.

[11] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In *AIIDE*, 2008.

[12] Guillaume M JB Chaslot, Mark HM Winands, H JAAP VAN DEN HERIK, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[13] Guillaume Maurice Jean-Bernard Chaslot Chaslot. *Monte-Carlo Tree Search*. PhD thesis, Maastricht University, 2010.

[14] Gabriella Cortellessa, Alfonso Emilio Gerevini, Daniele Magazzeni, and Ivan Serina. Automated planning and scheduling. *Intelligenza Artificiale*, 8(1):55–56, 2014.

[15] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[16] Martin Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[17] Zong Woo Geem. Harmony search algorithm for solving sudoku. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 371–378. Springer, 2007.

[18] Junichi Hashimoto. *A study on domain-independent heuristics in game-tree search*. PhD thesis, JAIST, 2011.

[19] Taichi Ishitobi. *Deep Proof-Number Search and Aesthetics of Mating Problems*. JAIST Press PhD thesis, 2016.

[20] Taichi Ishitobi, Aske Plaat, Hiroyuki Iida, and Jaap van den Herik. Reducing the Seesaw Effect with Deep Proof-Number Search. In Aske Plaat, Jaap van den Herik, and Walter Kosters, editors, *Advances in Computer Games*, pages 185–197, Cham, 2015. Springer International Publishing.

[21] Tomoyuki Kaneko. Parallel Depth First Proof Number Search. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[22] Garry Kasparov. The chess master and the computer. *The New York Review of Books*, 57(2):16–19, 2010.

[23] Mohd Nor Akmal Khalid, E. Mei Ang, Umi Kalsom Yusof, Hiroyuki Iida, and Taichi Ishitobi. Identifying Critical Positions Based on Conspiracy Numbers. In Béatrice Duval, Jaap van den Herik, Stephane Loiseau, and Joaquim Filipe, editors, *Agents and Artificial Intelligence*, pages 100–127, Cham, 2015. Springer International Publishing.

[24] Akihiro Kishimoto. *Correct and Efficient Search Algorithms in the Presence of Repetitions*. University of Alberta, 2011.

[25] Akihiro Kishimoto, Mark H. Minands, Martin Müller, and Jahn-Takeshi Saito. Game-Tree Search Using Proof Numbers: THE FIRST TWENTY YEARS, 2012.

[26] Akihiro Kishimoto and Martin Müller. Search Versus Knowledge for Solving Life and Death Problems in Go. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, AAAI'05, pages 1374–1379. AAAI Press, 2005.

[27] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[28] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[29] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[30] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[31] Inês Lynce and Joël Ouaknine. Sudoku as a SAT Problem. In *ISAIM*, 2006.

[32] David Allen McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(3):287 – 310, 1988.

[33] Martin Müller. Computer go. *Artificial Intelligence*, 134(1-2):145–179, 2002.

[34] Ayumu Nagai. *Df-pn algorithm for searching AND/OR trees and its applications*. PhD thesis, University of Tokyo, 2002.

[35] Andrew J Palay. Searching with probabilities. Technical report, Carnegie-mellon Univ Pittsburgh Pa Dept of Computer Science, 1983.

[36] Jakub Pawlewicz and Łukasz Lew. Improving Depth-First PN-Search: $1+\epsilon$ Trick. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers, editors, *Computers and Games*, pages 160–171, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[37] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1):255 – 293, 1996.

[38] Jahn-Takeshi Saito, Guillaume Chaslot, Jos W. H. M. Uiterwijk, and H. Jaap van den Herik. Monte-Carlo Proof-Number Search for Computer Go. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers, editors, *Computers and Games*, pages 50–61, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[39] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE transactions on pattern analysis and machine intelligence*, 11(11):1203–1212, 1989.

[40] Jonathan Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, 1990.

[41] Jonathan Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67 – 84, 1990.

[42] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[43] Jonathan Schaeffer and Aske Plaat. New advances in alpha-beta searching. In *Proceedings of the 1996 ACM 24th annual conference on Computer science*, pages 124–130. ACM, 1996.

[44] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[45] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[46] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017.

[47] Stephen JJ Smith and Dana S Nau. An analysis of forward pruning. In *AAAI*, pages 1386–1391, 1994.

[48] Zhang Song and Hiroyuki Iida. Using single Conspiracy Number to analyze game progress patterns. In *Computer, Information and Telecommunication Systems (CITS), 2017 International Conference on*, pages 219–222. IEEE, 2017.

[49] Zhang Song and Hiroyuki Iida. Using Single Conspiracy Number for Long Term Position Evaluation (forthcoming). 2019.

[50] H. Jaap van den Herik and Mark H. M. Winands. *Proof-Number Search and Its Variants*, pages 91–118. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[51] H.Jaap van den Herik, Jos W.H.M. Uiterwijk, and Jack van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1):277 – 311, 2002.

[52] Jaap Van den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1):277–311, 2002.

[53] Quang Vu, Taichi Ishitobi, Jean-Christophe Terrillon, and Hiroyuki Iida. Using Conspiracy Numbers for Improving Move Selectionin Minimax Game-Tree Search. pages 400–406. SCITEPRESS, 2016.

[54] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo Tree Search Solver. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, pages 25–36, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[55] I. Wu and P. Lin. Relevance-Zone-Oriented Proof Search for Connect6. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(3):191–207, Sept 2010.

[56] I-Chen Wu and Dei-Yen Huang. A New Family of k-in-a-Row Games. In H. Jaap van den Herik, Shun-Chin Hsu, Tsan-sheng Hsu, and H. H. L. M. (Jeroen) Donkers, editors, *Advances in Computer Games*, pages 180–194, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[57] Song Zhang, Hiroyuki Iida, and H. Jaap van den Herik. Deep df-pn and Its Efficient Implementations. In Mark H.M. Winands, H. Jaap van den Herik, and Walter A. Kosters, editors, *Advances in Computer Games*, pages 73–89, Cham, 2017. Springer International Publishing.

Publications

International Conference (refereed)

[1] Z. Song, H. Iida. (2017). Using Single Conspiracy Number to Analyze Game Progress Patterns. International Conference on Computer, Information and Telecommunication Systems (CITS). IEEE. Dalian, China. July. 2017

[2] Z. Song, J. van den Herik, H. Iida. (2019). Probability based Proof Number Search. 11th International Conference on Agent and Artificial Intelligence (ICAART). Prague, February. 2019.

[3] S. Busala, Z. Song, H. Iida, M. N. A. Khalid, U. K. Yusof. (2019). Single Conspiracy Number Analysis in Checkers. 11th International Conference on Agent and Artificial Intelligence (ICAART). Prague, February. 2019.

Book Chapter (refereed)

[4] Z. Song, H. Iida, and H. J. van den Herik. (2017). Deep df-pn and Its Efficient Implementations. Advances in Computer Games. Springer, Cham, 2017.

Journal (refereed)

[5] Z. Song, H. Iida. (2018). Using Single Conspiracy Number for Long Term Position Evaluation. ICGA Journal 40(3): 269-280.

Others

[6] Z. Song, H. Iida, and J. van den Herik. (2016). Deep df-pn and its Application to Connect6. Game Programming Workshop (GPW). Hakone, Japan. November. 2016