Memory versus Search in Games

Memory versus Search in Games

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Maastricht,
op gezag van de Rector Magnificus,
Prof. dr. A.C. Nieuwenhuijzen Kruseman,
volgens het besluit van het College van Decanen,
in het openbaar te verdedigen
op vrijdag 16 oktober 1998 om 16.00 uur

door

Dennis Michel Breuker

Promotor: Prof. dr. H.J. van den Herik

Leden van de beoordelingscommissie:
Prof. dr. P.T.W. Hudson (voorzitter)
Prof. dr. A. de Bruin (Erasmus Universiteit Rotterdam)
Prof. dr. J. Schaeffer (University of Alberta, Edmonton)
Prof. dr. S.H. Tijs
Prof. dr. H. Visser

SIKS
Dissertation Series No. 98-4
ISBN 90-9012006-8
NUGI 855
Subject headings: artificial intelligence / games / search

Cover design and photography: Hans Hoornstra

Voor mijn moeder en
ter nagedachtenis aan mijn vader

Contents

Appendices

List of Tables

List of Figures

Preface

After receiving my M.Sc. degree in Leiden, I faced the pleasant choice between becoming a Ph.D. researcher in Leiden and becoming a Ph.D. researcher in Maastricht. After some thought I decided to go to Maastricht, a choice I have never regretted. I was inspired not only by the beauty of Maastricht, but also by many persons, whom I want to acknowledge here.

First of all, I would like to thank my supervisor Jaap van den Herik. His critical attitude towards my research always kept me going. Furthermore, he taught me valuable lessons on doing research and on writing scientific articles. Next, many thanks go to Jonathan Schaeffer. Talking to him always made me enthusiastic, and his experience in the domain of game playing was of great benefit. Also, his suggestion to shift my attention towards computer games, giving my research a new impulse, proved very valuable. I thank Jos Uiterwijk for the daily supervision. We had many fruitful discussions, and without his help the thesis would not have been what it is now.

Further, I want to thank Patrick Hudson for reading this thesis. His suggestions for correcting my English writings were helpful. Patrick, I promise never to fax you that many pages again. I thank the late Bob Herschberg for scrutinizing some parts of the thesis in an earlier stage, improving the text considerably.

I would like to thank my colleagues at the Department of Computer Science for making me feel comfortable. Of all the room-mates I have had in the past few years, I especially want to thank Victor Allis for his ever-present enthusiasm. I thank Patrick Schoo, Maarten van der Meulen and Jo "de la Hiti" Beerens for their technical support. Many thanks go to the administrative staff of the Department of Computer Science. Sabine, iech bedaank diech veur d'ne relativerende kiek op 't promovere en veur 't frisse winsje in de vakgróp. Joke, bedankt voor de kritische grapjes en voor het helpen met de crypto's.

I thank my chess pals Harold "Liebe" Bekkering and Gerrit "Big-Love" Verhagen for always being there when I needed a good friend to talk to (or to drink with on the OLVP). Further, I will never forget the good times I had with Hans "Hmpf" Hoornstra. Finally, the visits of Cees "Bebi" Beekhuis made my stay in Maastricht even more pleasant than it already was.

Ik wil in het bijzonder mijn ouders bedanken voor de mogelijkheden die ze me hebben geboden om me te ontwikkelen. Helaas is mijn vader te vroeg overleden om mijn promotie mee te maken, maar ik weet dat hij trots op me zou zijn geweest.

Acknowledgements

The research reported in this thesis is supported, in part, by the Foundation for Computer Science Research in the Netherlands (SION), with financial support from the Netherlands Organization for Scientific Research (NWO) (dossier code 612-22-306).

Part of the research has been performed in the framework of the SYRINX project (SYnthesis of Reliable Information using kNowledge of eXperts), a joint research effort of IBM and the Universiteit Maastricht (project code 561553). The research has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Dennis Breuker
Amsterdam, August 1998

Chapter 1

Introduction

In this thesis, research on the balance between memory and search is presented. As is well known, the trade-off between knowledge and search plays a key role in many domains, such as expert systems, theorem proving, and games. We explicitly note that our research has a focus which differs from the research on the trade-off between knowledge and search, where dealing with knowledge and knowledge representations in order to arrive at intelligent solutions is stressed. In our research we look at just one characteristic of knowledge, viz. the storage of knowledge. More specifically, the research presented in the thesis concentrates on *memory* versus search in games. The domain used is that of two-player zero-sum games, and in particular the games of chess and domineering.

1.1 Games

For as long as computers have existed, people have tried to let them play intelligent (non-trivial) games. The challenge of a computer playing an intelligent game is a classic problem within the field of Artificial Intelligence (AI). Two pioneers seriously considered a computer playing chess, when AI was still in its infancy. Shannon (1950) published a seminal research article in which he described mechanisms to be used in a program playing chess. Turing (1953) was the first to describe a chess-playing program. The first program that could play a reasonable game in a related domain was a checkers program (Samuel, 1959; Samuel, 1967). This program was able to learn from its mistakes, thereby improving its performance.

Since then, much research has been conducted in the domain of games, resulting in very strong programs for many intelligent games. Nowadays, the best checkers program is CHINOOK, written by Martin Bryant, Rob Lake, Paul Lu, Jonathan Schaeffer and Norman Treloar (Schaeffer *et al.*, 1992; Schaeffer, 1997). In 1992 it played the "Man versus Machine World Checkers Championship" against Dr. Marion Tinsley, who is considered the greatest human checkers player in the history of the game (after he became World Champion in 1955 he lost only five serious games out

of some thousands (Schaeffer, 1996a)). It was the first time in history that a machine challenged a human for an official title in a non-trivial game of skill. Tinsley won the match (4 wins, 33 draws, and 2 losses). In 1994 a rematch between Tinsley and CHINOOK was played. Tinsley withdrew due to illness after six games (all draws) and he passed the title to CHINOOK. Don Lafferty, the 1994 United States checkers Champion, then challenged CHINOOK and they tied the match (1 win, 18 draws, and 1 loss). Tinsley never recovered from his illness and died in 1995. The subsequent Man versus Machine World Championship against Don Lafferty (January 1995) ended in a victory for CHINOOK (1 win and 31 draws). Since then the gap between man and machine has widened considerably. Therefore, CHINOOK is considered to be the strongest checkers-player in the world.

The best chess program (or better: chess machine or chess computer) is DEEP BLUE, created by Jerry Brody, Murray Campbell, A. Joseph Hoane Jr., Feng-hsiung Hsu and Chung-Jen Tan (Hsu *et al.*, 1990). In February 1996 it played a six-game match under normal tournament conditions against the human chess World Champion of the PCA, Garry Kasparov. Kasparov won the match (3 wins, 2 draws, and 1 loss). However, DEEP BLUE won the first game, surprising many experts (Uiterwijk, 1996; Newborn, 1997). In May 1997 it played a second six-game match against Kasparov. This time DEEP BLUE was able to win the match (2 wins, 3 draws, and 1 loss) (Schaeffer and Plaat, 1997; Goodman and Keene, 1997; King, 1997). This was a milestone in AI history, finally realizing Shannon's dream of 50 years before.

One of today's strongest Othello programs is LOGISTELLO, written by Michael Buro (1994). In the 22 international Othello tournaments it has played so far, it ended first sixteen times, and second five times. In August 1997 it played a six-game match against the Othello World Champion Takeshi Murakami, and won the match with the perfect score of 6–0. This clearly shows that the best Othello programs have become stronger than any human player (Buro, 1997).

We offer four arguments why researchers are so interested in intelligent games.

First, games provide an exact, closed domain (the rules are well-defined), in contrast to real-world problems, which are often rather vague (Van den Herik, 1983). Often many pages are required to give a proper description of (the background of) a real-world problem. Some real-world problems (e.g., in law and legal knowledge-based systems) can be interpreted differently by different people (Van den Herik, 1991). In contrast, games can be defined with sufficient precision.

Second, intelligent games are not trivial. Although it takes only an hour or so to learn the rules of chess, so far it has been impossible, even for the best human, to play chess perfectly. Playing intelligent games is hard and the obstacles that have to be tackled reflect the complexities inherent in real-world problems. Minsky (1968) stated "It is not that the games and the mathematical problems are chosen because they are clear and simple; rather it is that *they give us, for the smallest initial structures, the greatest complexity*, so that one can engage some really formidable situations after a relatively minimal diversion into programming."

Third, the domain of games is well-suited for testing new ideas in problem solving. Therefore, Michie (1980) proposed computer chess as the *drosophila melanogaster*

(fruit fly) of machine intelligence. According to Fraenkel (1996) ideas from the domain of games have been used in mathematics, computer science and economics. Nilsson (1971) mentions several applications, including operations research (traveling-salesman problem) and chromosome matching.

Fourth, by creating a machine which plays an intelligent game, it may be possible to gain more insight into the way people reason. The Dutch psychologist De Groot (1946) has investigated the thinking process of chess-players during a game. The American psychologists Newell and Simon (1972) tried to build models of the human mind, based on the results of their research on computer chess (Newell *et al.*, 1958). Recently, a follow-up to De Groot's (1946) book has been published (De Groot and Gobet, 1996) concentrating on perception and memory of chess players.

All four arguments provide a legitimate reason to perform research on games. Developing computer programs or a new computer technique may help to model a domain adequately or may remove an obstacle. Our research is inspired by the arguments two and three. We have developed two new techniques (for transposition tables and for proof-number search) and solved two open problems (domineering and the graph-history-interaction problem).

1.2 Knowledge versus search

One of the best known trade-offs in Computer Science is the trade-off between space and time. In the domain of AI, especially game playing, this comes down to the trade-off between knowledge and search (Clarke, 1977). In theory, all problems with a finite state space are solvable (Allis *et al.*, 1991) in two distinct ways.

1. Solve the problem by knowledge, not using any search. This is possible if all information for the initial state and the subsequent states necessary for solving the problem is available. Moreover, there is sufficient *space* to store the information, and there is a way of discovering and representing the knowledge necessary for solving the problem. For instance, the game of nim can be solved by knowledge alone (Bouton, 1901).

2. Solve the problem by search, not using any knowledge. This is possible if there is sufficient *time* available to do a sufficiently large search of the state space. For instance, the game of tic-tac-toe can be solved by brute force alone (Berlekamp *et al.*, 1982a).

Most problems cannot be solved in practice by using knowledge only or search only. If the state space is too large to be searched in a reasonable time, knowledge is needed to guide the search and to reduce the state space. If the state space is too large to be stored in memory, search is needed to compensate for the loss of knowledge. Thus, for solving the majority of problems a combination of knowledge and search is needed. Some examples of games which have been solved by a combination of knowledge and search are qubic (Patashnik, 1980; Allis and Schoo, 1992), connect-

four (Allis, 1988; Uiterwijk *et al.*, 1989; Allen, 1989), go-moku (Allis *et al.*, 1993; Allis, 1994; Allis *et al.*, 1996) and nine men's morris (Gasser, 1995).

When opting for search to solve a problem, we basically distinguish two search categories.

1. *Full-width* search (henceforth called brute-force search uses minimal knowledge to guide the search. After expanding a node in the search, knowledge is used to sort the children. The choice of the node to expand next comes from a (small) subset of the nodes in the tree. For instance, in breadth-first search the next node to be expanded is one of the siblings of the last expanded node, and in depth-first search the next node to be expanded is one of the children of the last expanded node.

2. *Best-first* search uses more knowledge to guide the search. The knowledge is used for the choice on which node to expand next. The node selected can be any leaf in the tree.

Two basic types of knowledge interact with search (Berliner, 1984).

1. *Directing* knowledge. In brute-force, search directing knowledge affects the order in which the descendants of a node are examined. In best-first search, directing knowledge guides the search (i.e., selects which node to expand next).

2. *Terminal* knowledge. Terminal knowledge is applied to the leaves of the search tree. It produces either an exact value (win, draw, or loss), in case the leaf is a terminal node, or a measure of the goodness of the position the leaf represents. Terminal knowledge is used both in brute-force search and in best-first search[1].

In brute-force search many leaves will be evaluated during a search process. Terminal knowledge is applied to all leaves (millions of times during a single search process). Thus, each term in the terminal-knowledge function (the evaluation function) contributes to the cost of an evaluation, which affects the speed of the search process, and should be carefully weighed.

1.3 Memory versus search

The trade-off between knowledge and search mostly deals with knowledge and knowledge representations. We only look at one characteristic of knowledge, viz. the storage of knowledge in *memory*. The purpose of storing knowledge acquired during the search process is to re-use it at a later time. Here we introduce the trade-off between memory and search, by giving two points of view.

As a first observation we note that the size of computer memory is no longer an obstacle, making it easier to equip a computer with more memory. Now the question is: can we make use of the large amount of memory, by storing more knowledge into

[1] In best-first search terminal knowledge is closely related to directing knowledge, and may be identical (Berliner, 1984).

this memory, thereby decreasing the need for searching? Depth-first search needs little memory. Only the path from the root to the position under investigation needs to be stored in memory. To use the remaining memory, knowledge about positions encountered in the search process may be stored in a large table, the so-called *transposition table*. The knowledge stored in the table is used to relieve the search. Thus, searching is reduced at the cost of using more memory.

As a second observation we mention the noteworthy increase in computer speed. Can we make use of this increase by using speed to accelerate the search, thereby acquiring more knowledge, decreasing the need for memory? Most best-first search algorithms need a large amount of memory to store the entire search tree. The quality of a best-first search algorithm depends on the quality of the directing knowledge. The speed of a computer increases faster than the amount of internal memory in a computer. Thompson (1996b) states that from 1985 to 1996, the speed of a typical high-quality workstation has increased by a factor of 150 (from 1 MIP to 150 MIPs), whereas its internal memory only has increased by a factor of 16 (from 16 MB to 256 MB). At current computer speeds, memory is quickly filled. Therefore, ways have to be found to use the increase in speed to acquire more knowledge per node, improving the directing knowledge. Then, the search process will search the state space more efficiently, reducing the need for memory at the cost of more searching.

1.4 Problem statements

Three problem statements are considered. The first problem statement addresses decreasing the need for search by increasing the use of memory.

Problem statement 1: Which methods exist to improve the efficiency of a transposition table?

A transposition table is normally used in combination with a depth-first-search algorithm. The most commonly used depth-first algorithm for two-person games is the $\alpha\beta$ algorithm. In the thesis we present research on improving the efficiency of a transposition table used in the $\alpha\beta$ algorithm. The research is performed in two domains: chess and domineering.

The second problem statement addresses decreasing the need for memory by increasing the use of search.

Problem statement 2: Which methods exist for best-first search to reduce the need for memory by increasing the search, thereby gaining more knowledge per node?

In the thesis we present research on a relatively new best-first-search algorithm, proof-number search (pn search). Like many best-first search algorithms, pn search stores the complete search tree in memory. An attempt is made to reduce the need for memory for pn search, realized in a pn-search variant, called *pn^2 search*.

Summarizing, in the first problem statement the need for search is reduced by increasing the use of memory. Analogously, in the second problem statement the need for memory is reduced by increasing the use of search. An attempt to combine the advantages of both approaches (reducing the need for search *and* reducing the need for memory) is the following. In a search tree, it may happen that identical nodes are encountered at different places. If these so-called *transpositions* are not recognized, the search algorithm unnecessarily expands identical subtrees. Therefore, it is profitable to recognize transpositions and to ensure that for each set of identical nodes, only one subtree is expanded. If a best-first search algorithm (which stores the whole search tree in memory) is used, the search tree is converted into a search graph, by joining identical nodes into one node. This causes subtrees to be merged, decreasing the need for memory. Since the graph contains fewer nodes than the tree, less searching is needed as well. However, joining identical nodes into one node introduces the so-called *graph-history-interaction* (GHI) problem, since determining whether *nodes* are identical is not the same as determining whether the *search states* represented by the nodes are identical.

This is laid down in the third problem statement, addressing both the decrease in the need for memory *and* the decrease in the need for search.

Problem statement 3: Is it possible to give a solution for the GHI problem for best-first search?

1.5 Outline of the thesis

The contents of the thesis is as follows.

Chapter 1 contains an introduction, the three problem statements and an outline of the thesis.

Chapter 2 tries to answer the first problem statement by discussing enhancements to the implementation of a transposition table. First, some important notions and concepts, used throughout the thesis, are defined. Thereafter, it is explained why a transposition table is needed. Next, implementation details are given and an experimental set-up is presented. Three series of experiments are described: (1) when different positions compete for storage in the same entry of the transposition table, a *replacement scheme* has to be used for priority arguments; several replacement schemes are compared; (2) the merits of storing different characteristics of a position are quantified; (3) the use of additional memory is discussed, and it appears that there is still room for improvements. Finally, it is shown that a transposition table is a useful way of reducing the search at the cost of using more memory.

Chapter 3 presents the pn-search algorithm. Experiments with pn search have been performed to obtain more insight into the strengths and weaknesses of this algorithm when applied to a complex game such as chess (i.e., to positions of which it is possible to prove the game-theoretic value). The algorithm will be used as a test bed in the Chapters 4 and 5. First, an informal description of the algorithm is

given, followed by the pseudo-code. Experiments are reported, comparing the pn-search algorithm to the $\alpha\beta$-search algorithm. The strengths and weaknesses of the pn-search algorithm are discussed.

Chapter 4 tries to answer the second problem statement and presents the pn^2-search algorithm. This is a modification of the pn-search algorithm when only little memory is available, using less memory at the cost of more searching. Experiments are given, showing that this algorithm solves more positions in the test set than the standard pn-search algorithm, implying that pn^2 search is a useful algorithm when little memory is available.

Chapter 5 answers the third problem statement. A review of attempted solutions to the GHI problem is presented. Next, our practical solution for best-first search algorithms that keep the whole search tree in memory is presented. Thereafter, the pseudo-code for the implementation in pn search is shown. This algorithm is compared to the standard pn-search algorithm and its modifications. Experiments are reported, showing that this graph algorithm for pn search performs well.

The evaluation of the three problem statements, final conclusions, and future research are given in Chapter 6.

Appendix A lists the test set used for the chess middle-game transposition-table experiments. The test set used for the chess endgame transposition-table experiments is presented in Appendix B. Appendix C lists the results of all experiments described in Chapter 2. The test set used for the proof-number search experiments is given in Appendix D. Appendix E presents the results of the pn-search and $\alpha\beta$-search experiments described in Chapter 3. The results of the pn^2 experiments described in Chapter 4 are shown in Appendix F. Finally, Appendix G lists and compares the results of the experiments given in Chapter 5 with the results of the pn tree algorithm.

Chapter 2

The transposition table

This chapter is an updated and abridged version of[1]

1. Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1994a). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 17, No. 4, pp. 183–193,
2. Breuker D.M. and Uiterwijk J.W.H.M. (1995). Transposition Tables in Computer Chess. *New Approaches to Board Games Research: Asian Origins and Future Perspectives* (ed. A.J. de Voogt), pp. 135–143. International Institute for Asian Studies, Leiden, The Netherlands,
3. Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1996). Replacement Schemes and Two-Level Tables. *ICCA Journal*, Vol. 19, No. 3, pp. 175–180,
4. Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1997b). Information in Transposition Tables. *Advances in Computer Chess 8* (eds. H.J. van den Herik and J.W.H.M. Uiterwijk), pp. 199–211. Universiteit Maastricht, Maastricht, The Netherlands, and
5. Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1998b). Solving Domineering. Submitted as journal publication. Also published (1998) as Technical Report CS 98-05, Universiteit Maastricht, Maastricht, The Netherlands.

In this chapter we try to obtain more insight into the first problem statement: which methods exist to improve the efficiency of a transposition table?

In Section 2.1 some important notions and concepts, used throughout the thesis, are introduced. Section 2.2 explains what transpositions are and why it is important to recognize them. The concept behind the transposition table is given in Section 2.3. Section 2.4 lists several data structures suitable for a transposition table. The experimental set-up of our research is given in Section 2.5. Section 2.6 discusses the test domains. Three series of experiments to improve the efficiency of the transposition table are presented in Section 2.7. Section 2.8 provides conclusions.

[1] Thanks are due to the Editors of *Advances in Computer Chess 8*, the Editor of *New Approaches to Board Games Research*, and the Editorial Board of the *ICCA Journal* for giving permission to use the contents of the articles in this chapter.

2.1 Notions and concepts

In this section we define several notions and introduce various concepts which we will use throughout the thesis. The main notions are: game tree, search tree and search methods.

Game tree

A *game tree* is a representation of the state space of a game. In the case of a two-player zero-sum game, the game tree is an oriented AND/OR tree. A *node* in the tree represents a position in the game; an *edge* represents a move. A sequence of edges forms a *path* if each edge shares one node in common with the preceding edge, and the other node in common with the succeeding edge. The root of the tree is a representation of the initial position. A *terminal position* is a position where the rules of the game determine whether the result is a win, a draw, or a loss. A *terminal node* represents a terminal position. A node is *expanded* by generating all successors of the position represented by the node. A direct successor of a node is termed a *child* of the node. Analogously, the direct predecessor of a node is termed the *parent* of the node. Nodes having the same parent are termed *siblings*. A node with at least one successor is termed an *interior* node. We note that the root is the only interior node without a parent.

A game tree is generated by expanding all the interior nodes. This process is repeated until all unexpanded nodes are terminal nodes. It follows that the game tree for the initial position is an explicit representation of all possible paths of the game (Pearl, 1984). Zermelo (1912) was the first person stating that every position (not necessarily a terminal position) can be theoretically characterized as a win, a draw, or a loss in the game of chess. The *game-theoretic value* is the value of the initial position, given that both players play optimally. A *minimal game tree* is defined as a minimal part of the game tree necessary to determine the game-theoretic value. The game-theoretic value can, in principle, be determined by examining the complete game tree. Since for most games the game tree (and even a minimal game tree) is extremely large, this is not feasible in practice. For instance, in chess the game tree consists of roughly 10^{43} nodes (Shannon, 1950). Chinchalkar (1996) gives 1.77894×10^{46} as an upper bound and, according to Bonsdorff *et al.* (1978), N. Petrović assumes that the upper bound is approximately 2×10^{43}.

Search tree

When the game tree is too large to be generated completely, a *search tree* is generated instead. This tree is only a part of the game tree. The root represents the position under investigation, and all other nodes of the search tree are generated during the search process. The nodes which do not have children (yet) are termed *leaves*. Leaves include terminal nodes and nodes which are not yet expanded.

The *depth of a node* in a tree is zero for the root, and one plus the depth of its parent otherwise. A node P with a smaller depth than a node Q is an *ancestor* of

node Q if node P is on the path from the root to node Q. In this case, node Q is a *descendant*[2] of node P. A *subtree* of a tree is formed by a node together with all its descendants. The *depth of a tree* is equal to the largest depth of all leaves, often counted in *plies*. A ply can be viewed as a half move (a move by one of the two players). The term ply was introduced by Samuel (1959). A path from the root to a leaf is called a *variation*. Leaves are *evaluated* (given a value) with the aid of an evaluation function. A *principal variation* is a sequence of moves where both players play optimally, according to the evaluation function used.

The order in which the nodes of the search tree are generated is defined by the type of search method.

Search methods

Several search methods have been developed. They fall into three categories[3]: (1) depth-first search algorithms, (2) breadth-first search algorithms, and (3) best-first search algorithms.

In *depth-first* search algorithms the root is expanded and one of its *children* is chosen for further investigation. If the node chosen is not a terminal node, the node is expanded and again one of its children is chosen for further investigation. If the child chosen is a terminal node, one of the node's siblings is chosen for further investigation. If all children have been investigated, one of the siblings of the parent is chosen for further investigation, and so on. This process is repeated throughout the whole tree. In summary, the children are expanded *before* the sibling nodes. In Figure 2.1 a search tree of depth three is depicted. For all AND/OR trees/graphs in this thesis white squares represent OR nodes (positions with the first player to move), and black circles represent AND nodes (positions with the second player to move). As an aid to the reader we mention that OR nodes can be seen as playing positions with White to move (WTM), with one or two selected strategies in mind; AND nodes as playing positions with Black to move (BTM), in which case White has to be prepared for *all* countermoves. The numbers represent the order in which the nodes are generated with depth-first search.

An advantage of depth-first search is that it may find a solution rather quickly. However, a disadvantage is that this method often spends much time exploring unfruitful paths. An example of a depth-first search algorithm is $\alpha\beta$ search (Knuth and Moore, 1975).

In *breadth-first* search algorithms, first the node representing the initial state is expanded. Then one of the leaves of the next level is chosen for further investigation. If it is not a terminal node, it is expanded. Thereafter, the next leaf on this level is chosen for further investigation. If all the leaves on this level have been chosen, one of the leaves of the next level is chosen for further investigation. This process is repeated throughout the whole tree. In summary, the children are expanded *after*

[2] We note that a parent is a special case of an ancestor, and a child is a special case of a descendant.

[3] Here we split the brute-force search into two categories, effectively creating three categories instead of the two mentioned in Section 1.2.

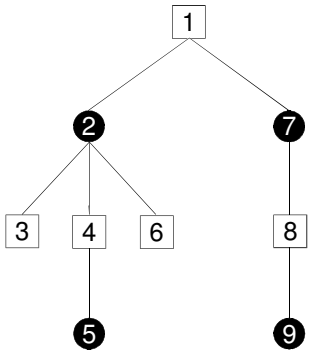


Figure 2.1: A depth-first traversal of a search tree.

the sibling nodes. This is illustrated in Figure 2.2. The numbers indicate the order in which the nodes are generated with breadth-first search.

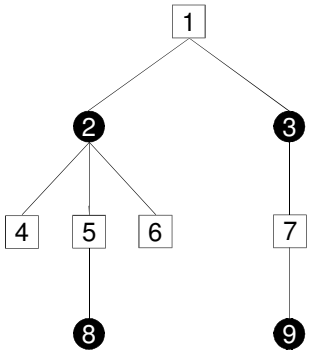


Figure 2.2: A breadth-first traversal of the search tree of Figure 2.1.

An advantage of breadth-first search is that the first solution found will be the solution with the shortest path. A major disadvantage is that it requires a large amount of memory to store all the nodes of the tree: a node is not needed (and does not have to be preserved in memory any more) *after* its subtree is expanded, but since breadth-first search expands the nodes one level after another, all nodes have to be kept in memory. Depth-first search first expands all the children of a node, and therefore a chosen node is not needed (and does not have to be preserved in memory any more) as soon as one of its siblings is chosen for expansion.

Finally, *best-first* search combines the advantages of both depth-first search and breadth-first search. At each step of the search process, the most promising path (according to some criterion) is expanded. Usually what happens is that some depth-

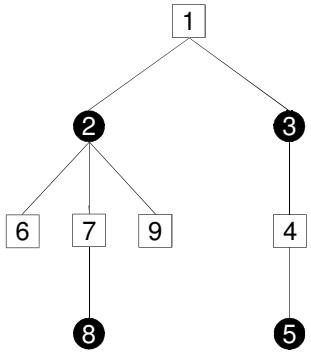


Figure 2.3: A best-first traversal of the search tree of Figure 2.1.

first searching occurs when the most promising branch is explored. Eventually, if the path looks less promising, one of the lower-level branches will be explored. However, search at the old branch is only suspended, and the search can return to it whenever it seems necessary. An example of a best-first search algorithm is proof-number search (Allis *et al.*, 1994). A best-first traversal is depicted in Figure 2.3. The numbers indicate a possible order in which the nodes might be generated.

Plaat (1996) states that the border between best-first search algorithms and depth-first search algorithms is not as clear as shown above. Plaat *et al.* (1996) give a new formulation of the SSS* algorithm (Stockman, 1979), based on the $\alpha\beta$ algorithm. Furthermore, they present a framework, termed MTD(f), that facilitates the construction of several best-first fixed-depth game-tree search algorithms, based on the depth-first minimal-window $\alpha\beta$ search, enhanced with storage.

2.2 Transpositions

When searching for a move, game programs build large search *trees*. Since a position can sometimes be arrived at by several distinct move sequences, the size of the search tree can be reduced considerably if the results of a position previously encountered remain available. The results can be stored in a large direct-access table, called a *transposition table* (Greenblatt *et al.*, 1967; Slate and Atkin, 1977). A closer inspection shows that the search tree then can be considered as a search *graph*, due to the *transpositions*. As an example we provide the chess position of Figure 2.4. It can be reached via the distinct move orders **1. e4** ♘**f6 2.** ♘**c3**, and **1.** ♘**c3** ♘**f6 2. e4**. To complicate matters, the following sequence of seven plies, **1.** ♘**f3** ♘**f6 2.** ♘**c3** ♘**g8 3. e4** ♘**f6 4.** ♘**g1**, also leads to the same position.

Assume that the position of Figure 2.4 appears somewhere in the search tree. After examining the position, a best move is found together with its score, based on a subtree of a certain depth. Since it is possible that this position exists elsewhere in

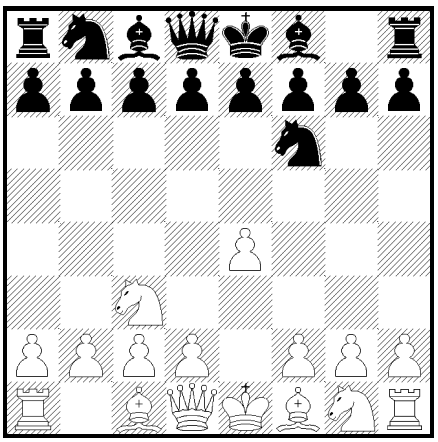


Figure 2.4: A BTM position that can be reached by distinct move orders.

the tree, the relevant information of the position is saved in the transposition table. The relevant information includes the score of the position, the best move, and the depth to which the subtree was searched. Adhering to $\alpha\beta$ search (Knuth and Moore, 1975), the score need not be an exact value, but may be a lower or an upper bound.

Slate and Atkin (1977) already remarked that, for chess, "Strictly speaking, positions reached via different branches are rarely truly identical, because the 50-move and three-time repetition draw rules make the identity of a position dependent on the history of moves leading to that position. This effect is small, and we decided to ignore it." However, ignoring the history of a position can give an incorrect result. This is known as the graph-history-interaction (GHI) problem, of which a solution is presented in Chapter 5. Up to Chapter 5 we concur in ignoring the history of a position.

2.3 A transposition table

2.3.1 Hashing

In the ideal case one would preserve every position encountered in a search process, together with its relevant information[4]. However, the memory required usually exceeds the available capacity of most present-day computers. Therefore, a transposition table is implemented as a finite *hash table* (Knuth, 1973). A position is converted into a sufficiently large number (the *hash value*) by using some hashing

[4] In chess, the side to move, castling rights and *en-passant* status are all part of the description of a position.

method. The most popular method used by game programmers is described by Zobrist (1970).

Hashing in chess

In chess there are twelve different pieces (Pawn, Knight, Bishop, Rook, Queen, King for both colours) and 64 squares. For any combination of a piece and a square a random number is generated. In addition, four unique random numbers are generated for castling rights, eight for *en-passant* rights, and one for changing the side to move. Thus, in total 781 ($12 \times 64 + 4 + 8 + 1$) unique numbers are available. The hash value for a position is calculated by doing an exclusive-or (XOR) of the numbers associated with the piece-square combinations of that position. If applicable, the castling and *en-passant* numbers are included too. This way of calculating a hash value has two advantages.

1. The XOR operation is a fast, bitwise operation.

2. The hash value can be updated incrementally. The hash value for a position resulting from some move can simply be obtained by doing an XOR between the hash value of the old position and the two numbers associated with the piece-fromSquare and the piece-toSquare of the move involved[5].

Warnock and Wendroff (1988) implemented in their program LACHEX a hashing-algorithm method used less frequently, based on the theory of error-correcting codes. Their hashing set is constructed from a Bose-Chaudhuri-Hocquenghem (BCH) code (MacWilliams and Sloane, 1977). The only other program we know which uses this method is ZUGZWANG (Feldmann, 1993). The method is not widely used; for details we refer to Warnock and Wendroff (1988).

Hashing in domineering

(For a description of the domineering game we refer to subsection 2.5.2.) For any occupied square on a board a unique random number is generated. (It is irrelevant whether a square is occupied by a vertically or horizontally placed domino.) So for the standard (8×8) board 64 unique numbers are sufficient. No random number for changing the side to move is needed, since it is impossible to have two equal positions with different players to move for the same starting player. The hash value of a position is calculated by doing an XOR of the numbers associated with the occupied squares. The hash value for a position resulting from some move is obtained by doing an XOR between the hash value of the old position and the two numbers associated with the squares of the move involved.

[5] One additional XOR is needed for changing the side to move. When capturing, castling or *en passant* is involved, one or a few additional XORs have to be applied.

Hash value and hash key

Figure 2.5 illustrates how the hash value is generally used. If the transposition table consists of 2^n entries, the n low-order bits of the hash value are used as a *hash index*. The remaining bits (the *hash key*) are used to distinguish among different positions mapping onto the same hash index (i.e., the same entry in the transposition table). Therefore, the total number of bits should be sufficiently large (Hyatt *et al.*, 1990). For instance, the chess program CRAY BLITZ uses a 64-bit hash value. For more details, we refer to subsection 2.4.2.

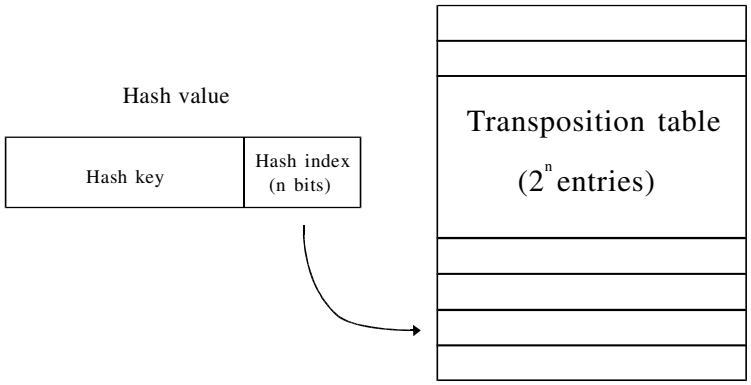


Figure 2.5: The hash value.

2.3.2 The traditional components

For an entry in a transposition table to be effective, it should at least contain the following information (Marsland, 1986; Hyatt *et al.*, 1990):

key [6]: contains the more significant bits of the hash value (see Figure 2.5). The key is used to distinguish among different positions having the same hash index.

move : contains the best move in the position obtained from a search. This is the move which either caused a cut-off, or obtained the highest score. The move is used for the directing knowledge (move ordering).

score : contains the value of the best move in the position obtained from a search. Since we adhere to $\alpha\beta$ search, the score can be an exact value, an upper bound or a lower bound. The score can be used to adjust the α and β bounds of the search.

flag : contains information on the score. The flag indicates whether the score is an exact value, an upper bound, or a lower bound.

[6] Marsland (1986) uses the term 'lock'.

depth : contains the relative depth of the subtree searched. When doing an n-ply search from the root and a position is stored at ply m of the tree, the search depth is $n-m$. The depth indicates how deep a previously encountered position has been investigated.

We call a transposition table with these five information fields a *traditional table*.

During the search, each position encountered is looked up in a table. If the position is found, the information stored can be used in three distinct ways, depending on the contents of *flag* and *depth*.

1. The depth still to be searched is less than or equal to the depth retrieved from the table *and* the retrieved value is an exact value. The position does not have to be searched: the search value is retrieved from the table[7]. Usually, the best move is also retrieved from the table, and used for determining the principal variation.

2. The depth still to be searched is less than or equal to the depth retrieved from the table *and* the retrieved value is *not* an exact value. The retrieved value can be used to adjust either the α value (if the retrieved value is a lower bound) or the β value (if the retrieved value is an upper bound). If this causes α to be greater than or equal to β, then a cut-off occurs and the position does not have to be searched. Otherwise, the retrieved move can be used as a first candidate, since it was considered best (or at least good enough to yield a cut-off) previously.

3. The depth still to be searched is greater than the depth retrieved from the table. In this case only the retrieved move is useful[8]. It can be investigated first, since it was considered best for a shallow search, the probability being high that it also will be best for deeper searches. Thus the move is used to improve the directing knowledge (move ordering).

When using iterative deepening (Gillogly, 1972; Slate and Atkin, 1977) and minimal-window search (Pearl, 1980; Marsland and Campbell, 1982; Reinefeld, 1983), transposition tables may significantly reduce the search effort, especially in chess endgame positions with only a few pieces on the board. Nelson (1985) states that "In normal situations the move generator is called only about 35% of the time, the other 65% being handled by the transposition-table move." Ebeling (1986) concludes that "not using the hash table for moves affects the search size by at least a factor of two." Hyatt *et al.* (1990) show that "these rules let Cray Blitz find about 30% of typical middle-game positions in the transposition table, and well beyond 90% in certain endgame positions." Berliner and Ebeling (1990) show that the use

[7] If the depth still to be searched is less than the depth retrieved, the search results may differ from the results when searching without a transposition table.

[8] Many heuristics, like aspiration search (Brudno, 1963; Berliner, 1974; Gillogly, 1978), ProbCut (Buro, 1995), and fail-high reductions (Feldmann, 1997) also use the retrieved value. It is used for setting the search window.

of transposition tables combined with good move-ordering heuristics may yield that "on average, the program searches only about 1.4 times the number of nodes that an $\alpha\beta$ search with perfect move ordering would search."

In chess, transposition tables are especially useful in positions without Pawns or with blocked Pawns. As an example, consider problem no. 70 from Fine (1941), shown in Figure 2.6. At first sight, this seems an easy position. However, White has only one winning move, which is the unexpected move **1. ♔b1!**. It is possible to find this move by using knowledge about *distant opposition* (Fine, 1941), or by doing a deep (at least 24 ply) search. Without a transposition table this is not possible in tournament time.

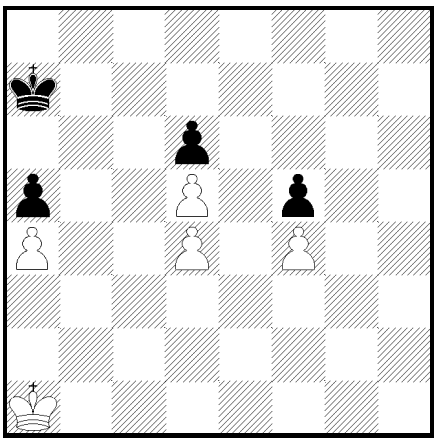


Figure 2.6: A WTM position with blocked Pawns.

For instance, assuming that both sides have five moves on average at their disposal for each position (an underestimation), the minimal game tree when searching 33 ply consists of some 9×10^{11} nodes ($\approx w^{\lfloor (d/2) \rfloor} + w^{\lceil (d/2) \rceil} - 1$ (Knuth and Moore, 1975), with $w = 5$, and $d = 33$). However, Hyatt *et al.* (1984) show that CRAY BLITZ searches only about 4×10^6 nodes when searching this position to a depth of 33 ply (reached in only 65 seconds on a Cray X-MP). The reduction in nodes searched by CRAY BLITZ (a factor of more than 200,000) is caused by the transposition table.

Sometimes transposition tables are used to store information about only a specific part of the position (e.g., the pawn structure, or king safety)[9]. Since this only replaces a part of the evaluation function, not reducing the number of nodes searched, it is outside the scope of our experiments.

[9] Warnock and Wendroff (1988) use the name *search tables* when talking about transposition tables in the broadest sense.

2.4 Implementing a transposition table

2.4.1 Data structures

Several data structures come to mind for implementing transposition tables (Pronk, 1987; Van Diepen and Van den Herik, 1987). Two main choices exist.

1. A table with a variable number of positions per entry (array of linked lists). Two advantages of this implementation are (1) the available memory can be divided flexibly among the entries, and (2) no memory is wasted on empty entries. Two major disadvantages are (1) the pointers of the linked list (needed to implement the variable number) take up much memory compared to the size of an entry position, and (2) more computation is needed to check for the existence of a position in a chain of the linked list.

2. A table with a fixed number of positions per entry (two-dimensional array). The advantage of this implementation is that no memory is wasted on extra pointers. The disadvantage is that memory will be wasted when the search is shallow and the table is not filled completely.

The disadvantages of a table with a variable number of positions per entry are more serious than the disadvantage of a table with a fixed number of positions per entry (Van Diepen and Van den Herik, 1987), leading to the logical choice of the latter implementation.

A position which needs to be stored in an entry where all positions are occupied is called an overflow. Overflows can be stored in an overflow area. Two choices for the overflow area exist.

1. The overflow area is implemented as another table or binary tree. Two disadvantages of this implementation are (1) in the overflow area the complete hash value has to be stored in memory, and (2) many comparisons may be needed to find a position in the table.

2. The overflow area is in the same table. The overflows are stored using *double hashing* (Knuth, 1973). An advantage of this implementation is that only one table needs to be used. A disadvantage is that again many comparisons may be needed to find a position in the table. For an extended review of this implementation, we refer to Beal and Smith (1996).

On the matter of implementation we distributed a questionnaire among readers of the *ICCA Journal* and the newsgroup `rec.games.chess.computer` (then called `rec.games.chess`). In the paragraph below we refer to their responses.

The implementation with double hashing is used by, amongst others, Hyatt (1994), Stanback (1994) and Weill (1994). Since using an overflow area may cause more computation to check whether a position exists in the table, a table with one position per entry, not using an overflow area, is used most frequently (Feldmann,

1994; Uiterwijk, 1994; Wendroff, 1994). Distinguishing between two identical positions with different side to move can be done in two ways: (1) use two different transposition tables (one for White and one for Black), or (2) use one transposition table, and use one additional random number for the player to move, which is XORed with the hash value. The latter method is used most frequently (Feldmann, 1994; Hyatt, 1994; Schaeffer, 1994; Uiterwijk, 1994; Weill, 1994; Wendroff, 1994).

2.4.2 Probability of errors

Implementing a transposition table as a hash table introduces two types of error, identified as early as 1970 by Zobrist. The first type of error (*type-1 error*) is the most important one. A type-1 error only occurs when the number of available hash values is much less than the total number of positions in a game, such as in chess. In this case, it can happen that two different positions yield the same hash value. This is a serious error, because when a type-1 error occurs, the information in this entry will be used for the wrong position and, if so, will introduce search errors. One way of detecting this error is to store the whole position in the transposition table. However, in many games this takes up too much space, and is therefore not feasible in practice. Another way of detecting this error is to test the move suggested by that transposition-table entry for legality in the position, effectively lowering the error rate. If the move is illegal, then the table entry must concern another position than the one being investigated. Note that if the move *is* legal, the positions still may differ. The probability of the occurrence of type-1 errors can be lowered by increasing the number of bits in the hash value.

The second type of error (*type-2 error*, or *clash*) occurs when two different positions map onto the same entry in the transposition table, i.e., the positions have equal hash indices, but different hash keys. This is known as a *collision* (Knuth, 1973). When a collision occurs, a choice has to be made which of the two positions involved should be preserved in the transposition table. Such a choice is based on a *replacement scheme*. Several replacement schemes are discussed in subsection 2.7.1. The probability of the occurrence of collisions can be lowered by increasing the number of bits in the hash index (thus increasing the number of entries in the transposition table).

The probability of a type-1 error and the probability of a collision are both calculated in the same way. The only difference is the number of distinguishable positions (for a type-1 error this is the number of possible hash values[10] and for a collision this is the number of possible hash indices, i.e., table entries[11]).

Let N be the number of distinguishable positions, and M be the number of different positions which have to be stored in the transposition table[12]. The probability that all M positions will have different hash values (i.e., the probability that no

[10] i.e., 2^k, where k is the number of bits of the hash value.
[11] i.e., 2^n, where n is the number of bits of the hash index.
[12] This number is equal to the number of non-empty positions in the transposition table after the search has been completed, augmented with the number of collisions during the search.

errors occur) is given by

$$P(\text{no errors}) = (1 - \frac{1}{N}) \times (1 - \frac{2}{N}) \times \cdots \times (1 - \frac{M-1}{N}).$$

If M is small compared to N, then all cross products can be neglected and we have the following approximation

$$P(\text{no errors}) \approx 1 - \frac{1 + 2 + \cdots + M - 1}{N} = 1 - \frac{M(M-1)}{2N}.$$

For small positive x we have $\log(1 - x) \approx -x$, and thus

$$\log P(\text{no errors}) \approx -\frac{M(M-1)}{2N}.$$

Thus, it follows that

$$P(\text{no errors}) \approx e^{-\frac{M(M-1)}{2N}}.$$

If M is sufficiently large, this formula yields

$$P(\text{no errors}) \approx e^{-\frac{M^2}{2N}}. \tag{2.1}$$

This result equals the formula given by Gillogly (1989)[13].

We note that the problem of calculating the probability that at least one error occurs (being $1 - P(\text{no errors})$), is analogous to the problem widely known as the *birthday paradox* (Feller, 1950), where the probability of at least two persons having the same birthday in a group of M persons (N being 365) has to be calculated.

The expected number of errors can be calculated as well. Feldmann (1993) derives the following formula for the expected number of errors (E):

$$E = M - N \times (1 - (\frac{N-1}{N})^M).$$

When N is sufficiently large (which is the case for a transposition table), this formula can be approximated by

$$E \approx M - N \times (1 - e^{-\frac{M}{N}}). \tag{2.2}$$

As an example we consider a program which searches 100,000 nodes per second. If it plays a game using a total of two hours of thinking time, the number of nodes searched is 7.2×10^8. Assume that for about 30% of the nodes, an attempt is made to store them in the transposition table. In the example, this is 216 million nodes. If the hash value consists of 32 bits, the probability of at least one type-1 error is

$$1 - e^{-\frac{216,000,000^2}{2 \times 2^{32}}}$$

which is very close to 1. So a hash value of 32 bits clearly is too small. If we want to reduce the probability of at least one error to less than 1 percent, Equation 2.1 says that at least 62 bits are required. When using a 64-bit hash value, the probability is reduced to about 1×10^{-3}. In this case, the expected number of type-1 errors for the example above is about 0.05.

[13] The article contains a typing error. The probability given here is correct (Gillogly, 1994).

2.5 Experimental set-up

The transposition-table experiments are performed in the domains of chess and domineering. The experimental set-ups for both domains are described in the next two subsections.

2.5.1 The game of chess

For the chess experiments we have developed a test program AliBaba, being a simple chess program, designed to be easily reproducible by other researchers[14]. This reproducibility serves to promote a uniform platform for research. The major components of AliBaba constitute the remainder of this section, viz. the search engine, the evaluation function, the move-ordering heuristics, and the transposition table.

The search engine

The search engine is based on a variant of $\alpha\beta$ search: iterative-deepening, minimal-window, principal-variation search[15] (Marsland, 1986). Furthermore, AliBaba uses *aspiration search* (Brudno, 1963; Berliner, 1974; Gillogly, 1978). At the start of each new iteration, the upper bound and lower bound of the window are set to the value resulting from the previous iteration *plus* and *minus* the value of a Pawn, respectively. If the search fails (the value does not lie within the $\alpha\beta$ window), the window is adjusted to either $(-\infty, \text{value})$ when failing low, or $(\text{value}, +\infty)$ when failing high.

Leaves in the search tree should be "relatively quiescent" when evaluated (Shannon, 1950). Not all leaves are quiescent, so they should be further investigated by a *quiescence search*. In this search only capturing moves and promotion moves are considered, except if the King is in check, when all moves must be searched. We note that in the former case a quiescence search may be terminated early, viz. as soon as it becomes clear that all moves to be generated will be disadvantageous (Schrüfer, 1989). No other search extensions are used in the experiments in order to avoid possible search anomalies.

Before executing the principal-variation search at a node in the search tree, it is checked whether the position represented by the node is a draw by stalemate, by three-fold repetition, or by the 50-move rule, or whether it is a win by checkmate.

[14] The full C source code is available by anonymous FTP.
The URL is `ftp://ftp.cs.unimaas.nl/pub/software/breuker/alibaba.tar.Z`

[15] We note that the version of principal-variation search as mentioned by Marsland (1986) is identical to the version of negascout as mentioned by Reinefeld (1989). We use the 1989 reference instead of 1983, which was the first source of this algorithm, since the algorithm described in Reinefeld (1983) contains minor errors.

The evaluation function

The evaluation function used is simple. It consists of a material part and a positional one. The material part counts the difference of material between sides. The positional part is restricted to summing piece-square-table values. During a game, for every type of piece a 64-square table is maintained. Each table contains positional values for that piece on every square on the board. Again, we tried to keep things as simple as possible for the reproducibility. Therefore the positional values are independent of the position at the root. The positional part of the evaluation function is updated incrementally: whenever a move is investigated during the search process, the positional value of the piece-fromSquare table entry is subtracted from it, and the value of the piece-toSquare table entry is added to it. Finally, the evaluation function also serves to detect draws by stalemate, by three-fold repetition and by the 50-move rule as well as checkmate.

The move-ordering heuristics

In any position, AliBaba generates only legal moves, excluding pseudo-legal moves, such as placing or leaving its own King in check. Since the move ordering is important for the efficiency of the $\alpha\beta$ algorithm the following ordering heuristics are implemented.

Refutation tables (Akl and Newborn, 1977). For every move in the root position, the main variation is stored. In the next iteration, moves out of these refutation lines are tried first.

History heuristic (Schaeffer, 1983; Schaeffer, 1989b). A score for every legal move encountered in the search tree is maintained. Every time a move is found to be best in a search, its score is adjusted by an amount proportional to the depth of the subtree investigated. When ordering moves using this heuristic, moves with a higher score are considered before moves with a lower score.

In AliBaba, the moves are ordered in the following way. The first move to be considered is the move from the refutation table (if present). Then, if the position is found in the transposition table (see page 24), the transposition-table move is the next move to be considered. These moves are followed by capture moves (the highest-valued piece to be captured first; if equal, then the lowest-valued capturing piece first). Thereafter follow the promotion moves (ordered by promotion piece; the highest-valued promotion piece first). The remaining moves are ordered according to their descending history-heuristic scores. In addition to the move-ordering heuristics mentioned above, applied immediately after move generation, the root moves are also ordered during the iterative-deepening search processes.

$\alpha\beta$ search combined with a transposition table

Whenever a move is investigated in the $\alpha\beta$ search, the resulting position is looked up in the transposition table. If the position is present, and the depth of the examined subtree is greater than or equal to the depth still to be searched, the information in the table is considered reliable. Therefore, if the score is an exact value, it can immediately be used; otherwise, it can be used to update the window bounds (possibly causing a cut-off). The transposition-table move is always used to order moves (see page 23).

After a position has been investigated to a certain depth, it is stored in the transposition table together with the best move (i.e., the move which caused a cut-off, or the move with the highest score), its score, a flag (denoting whether the score was an exact value, a lower bound, or an upper bound), and the search depth. During quiescence search, a position is never stored in the transposition table.

The results of a transposition-table look-up are used at *all* nodes in the tree. If a leaf position is present in the table, the transposition-table score is used for the evaluation. If the score was an exact value, this score is used as evaluation value. Otherwise, the position is evaluated using the evaluation function. If the evaluation value is higher than the transposition-table score and the bound is an upper bound, the evaluation value becomes equal to the transposition-table score (analogously for the lower-bound score). Since the evaluation function is also used in the quiescence search, the transposition table is used in the quiescence search as well. Note, however, that since positions are only retrieved and not stored during quiescence search, their usefulness is limited during that phase.

In our experiments the transposition table is implemented as a linear array with one or two table positions per entry. No overflow area is used (see also subsection 2.4.1). Furthermore, a 64-bit hash value is used[16]. More details of the implementation of a transposition table in plain $\alpha\beta$ search are given in Marsland (1986).

The pseudo-code (based on Marsland, 1986) for the implementation of a transposition table in plain $\alpha\beta$ search (in a negamax framework) is given in Figure 2.7. Details concerning enhancements, move-ordering techniques and quiescence search are omitted for clarity. The parameters of the function are the current position under investigation (position), the depth to be searched (depth), and the α and β bounds of the search window, respectively. We note that the function Evaluate needs as parameters the position and the transposition-table information. If a leaf position is present in the table, the transposition-table score is used for the evaluation (see above). Furthermore, the function TryToStore attempts to store the search information in the transposition table, using a *replacement scheme* (see Section 2.7.1) when encountering a collision. The function AlphaBeta returns the best value of the position under investigation.

[16] In the experiments the size of the transposition table ranges from 8K to 2048K entries. For these transposition-table sizes the hash index ranges from 13 to 21 bits.

```
function AlphaBeta( position, depth, α, β )
    oldα := α
    Retrieve( position, ttMove, ttScore, ttFlag, ttDepth )
    /* If the position is not found, ttDepth will be −1 and ttMove 0 */
    if ttDepth≥depth then begin
        if ttFlag=ExactValue then return ttScore
        elseif ttFlag=LowerBound then α := max( α, ttScore )
        elseif ttFlag=UpperBound then β := min( β, ttScore )
        if α≥β then return ttScore
    end
    if depth=0 then /* Leaf */
        return Evaluate( position, ttScore, ttFlag, ttDepth )
    if ttDepth≥0 then begin /* Examine tt-move first */
        newPos := DoMove( ttMove, position )
        bestValue := −AlphaBeta( newPos, depth−1, −β, −α )
        UndoMove( ttMove, newPos )
        bestMove := ttMove
        if bestValue≥β then goto Done
    end
    else bestValue := −∞
    GenerateMoves( moveList, nrMoves )
    if nrMoves=0 then
        return Evaluate( position, ttScore, ttFlag, ttDepth )
    for i:=1 to nrMoves do begin
        if moveList[ i ]≠ttMove then begin
            α := max( bestValue, α )
            newPos := DoMove( moveList[ i ], position )
            value := −AlphaBeta( newPos, depth−1, −β, −α )
            UndoMove( moveList[ i ], newPos )
            if value>bestValue then begin
                bestValue := value
                bestMove := moveList[ i ]
                if bestValue≥β then goto Done
            end
        end
    end
Done:
    if bestValue≤oldα then ttFlag := UpperBound
    elseif bestValue≥β then ttFlag := LowerBound
    else ttFlag := ExactValue
    TryToStore( position, bestMove, bestValue, ttFlag, depth )
    return bestValue
end /* AlphaBeta */
```

Figure 2.7: The $\alpha\beta$-search function with a transposition table.

2.5.2 The game of domineering

Like chess, domineering is a two-player zero-sum game with perfect information. The game is also known as crosscram, and as dominoes. It was proposed by Göran Andersson around 1973 (Gardner, 1974; Conway, 1976). In domineering the players alternately place a domino[17] (2×1 tile) on a board, i.e., on a finite subset of Cartesian boards of any size or shape. The game is usually played on rectangular boards. The two players are denoted by Vertical and Horizontal. In standard domineering the first player is Vertical, who is only allowed to place its dominoes vertically on the board. Horizontal may play only horizontally. Of course, dominoes are not allowed to overlap. As soon as a player is unable to move the player loses. Although domineering can be played on any board and with Vertical as well as Horizontal to move first, the original game is played on a (8×8) checker-board with Vertical to start, and this instance has generally been adopted as standard domineering. According to West (1996) this size is sufficiently large to be beyond the range of human analysis, and hence the size is fit for an interesting game.

For the domineering replacement-scheme experiments we have developed the program DOMI. The search engine is plain $\alpha\beta$ search. The evaluation function is a two-valued function, only returning the values *win* and *loss*.

The move-ordering heuristics

In DOMI a distinction is made between (1) the mobility, (2) the number of real moves, and (3) the number of safe moves. Mobility is defined as the number of distinct moves that a player can make in a position. The number of real moves is defined as the maximum number of moves that a player can make in a position, provided that the opponent does not make any move. The number of safe moves is defined as the maximum number of moves that a player can make from a given position in the remaining part of the game, irrespective of the moves that the opponent will make.

The mobility, the number of real moves, and the number of safe moves are updated incrementally. During the search, the decrements δ of the number of real moves and the number of safe moves are continuously updated for both players. The four values are instrumental for a move ordering within the $\alpha\beta$ search, the heuristic being: the higher the ordering value, the better the move likely is. The formula for the ordering value is

$$\begin{aligned} \textit{ordering value} = {} & \delta(\textit{real moves opponent}) - \delta(\textit{real moves player to move}) + \\ & \delta(\textit{safe moves opponent}) - \delta(\textit{safe moves player to move}). \end{aligned}$$

Forward cut-offs

The number of real moves indicates an upper bound of the search-tree depth, and the number of safe moves indicates a lower bound of the search-tree depth. If the number of safe moves of the player to move is greater than or equal to the number of real moves of the opponent after the player has made its move, the move is called

[17] The markings on the dominoes are irrelevant.

a *winning* move. In this case, no further moves are generated and the search at this position will be terminated, resulting in a win for the player to move. If the number of safe moves of the opponent is greater than the number of real moves of the player to move after the player to move has made its move, the move is called a *losing* move. In this case, the move is discarded and the next sibling, if any, will be generated.

$\alpha\beta$ search combined with a transposition table

The implementation of the transposition table is similar to the implementation given in Figure 2.7, with two exceptions: (1) ttFlag is always equal to ExactValue (since only the values *win* and *loss* are used and therefore no bound values are possible), and (2) only the best value and not the best move is stored in the table[18]. All symmetries of the rectangular board are used in DOMI. Whenever a node is investigated in the search, the resulting position is looked up in the transposition table. If it is not present, any of the three symmetrical positions (a horizontal, and/or a vertical reflection) is looked up. In the latter case, if present, the information of the symmetrical position is used[19].

2.6 The test domains

In this section we describe the test domains in which the experiments are performed.

2.6.1 Chess test sets in the literature

Several methods have been used to test the strength of a chess-playing program. In many cases a *test set* is used. Previous test sets mentioned in the literature are:

- the *Win-at-Chess* set of 300 tactical positions from Reinfeld (1958). These positions serve well to test the tactical ability of chess programs, although the strongest programs have outgrown the test (Anantharaman *et al.*, 1988);
- the Bratko-Kopec set of 24 positions (Kopec and Bratko, 1982). These positions are divided into two categories: twelve tactical and twelve positional positions. The positional positions all have a *pawn-lever* move (described by Kmoch, 1959) as their solution. This test suite has two disadvantages: (1) 24 positions are too few, and (2) the test is highly specialized in what it tests;
- a test set consisting of 86 positions, devised by Nielsen (1991). The main purpose of this test set is to estimate the ELO rating (Elo, 1978) of the program;

[18] If a position is present in the table, its game-theoretic value (*win* or *loss*) is known and no further search is needed at this point.

[19] We note that we do not make use of rotation symmetry, because that exchanges the concepts of horizontal and vertical.

- a large test set of 5551 positions, described by Lang and Smith (1993). It consists of roughly 2530 tactical positions, 800 positional positions, 2100 endgame positions and 110 opening positions. The test set seems very good, but it will take a long time to run a program on all the test positions. Even if the program is allowed to analyze each position for only three minutes (tournament speed), it will take more than 11 days of computing time. Considering that a programmer needs to test every modification of the program, we have not adopted this test set for our research.

Berliner *et al.* (1991) give a taxonomy of chess positions and have tried to devise a representative test set. Private communication between Lang and Berliner shows that great difficulties were encountered in creating such a set and only some twenty positions have been produced so far (Lang and Smith, 1993).

Finally, it is known (Lang and Smith, 1993) that many commercial companies, such as Fidelity Electronics and Heuristic Software, and many professional programmers, have created their own test sets, but they have rarely published these positions. Most of these tests are devised to test only one aspect of a chess program. Some of these tests are published in computer chess magazines, such as *Computerschaak*, *Modul*, and *ComputerSchach und Spiele*. With the popularity of the Internet nowadays, many more test sets (including the ones mentioned above) are available at several FTP sites. See, for instance, URL `ftp://external.nj.nec.com/pub/wds/`.

As already evident from above, test sets always have a disadvantage: either the number of positions is too small to be representative of positions in high-level chess games, or the number of positions is so large that it will take too much time to test a program on every position. Anantharaman (1991) mentions three other methods to test the strength of a chess program.

1. Play a large number of tournament games. The disadvantage is the time it will take to play a sufficient number of games to obtain a good impression of the strength of the program.

2. Play matches between two computers, starting from a set of chosen positions, playing both sides. This approach has been used by, amongst others, Gillogly (1978) and Schaeffer (1986). According to Anantharaman this will take much time too, because about 1,000 games are necessary to spot a rating difference of ten points[20].

3. Marsland and Rushton (1973) have taken 760 positions from a collection of games between human masters from several strong tournaments. They test the program using all these positions. Conclusions on the strength of a chess program are based on the average rank of the move the human master played. One of the disadvantages is that there is no distinction between minor mistakes

[20] This is only important if the versions tested do not differ much in strength. If one version is much stronger (say about 250 points) than the other version, it is not interesting to know whether it is 240, 250, or 260 points stronger.

and major blunders. Another disadvantage is that the possibility that the move played by the program is better than the human move is not taken into account.

Anantharaman (1991) describes another approach to test chess programs. The approach was designed to test search heuristics, but can equally well be applied to test other enhancements of a chess program. The method is used in testing DEEP THOUGHT and its successor DEEP BLUE. The quality of the test program is measured using a *deeper searching* reference program. This reference program is about 300 rating points stronger than the test program. Anantharaman used circa 3,600 positions to evaluate the test program. He concluded that comparing the move chosen by the test program with the move chosen by a human expert is not a reliable method for evaluating the test program. He showed further that comparing the move chosen by the test program with the move chosen by the reference program is a better way for evaluating the test program, correlating well with USCF ratings. Anantharaman reports that with the described technique the same reliability can be reached within only 6% to 16% of the time required when using matches between computers.

2.6.2 Our chess test set

Our testing method for chess differs from the methods discussed above. We have opted to use a sequence of positions derived from actual games as the test set. One advantage is that the chosen positions will not be biased towards tactical issues, but will automatically incorporate positional ones. Moreover, the choice also meets the requirement that successive positions should be related, which is essential when investigating the effects of clearing the transposition table between moves (see subsection 2.7.1). Finally, our goal is not to investigate the strength of the test program, but to investigate the sizes of the search trees involved.

The chess experiments have been divided into two parts. The first part concerns middle-game experiments, and the second part endgame experiments. The middle-game experiments and the endgame experiments are separated to see whether the results are different, since it is known that the benefits from the use of transposition tables are greater in endgame positions than in middle-game positions (Slate and Atkin, 1977).

For the middle-game experiments we have chosen positions from all six Kasparov games of the Euwe memorial VSB tournament 1994 as our test set. Clearly, Kasparov, being the World Champion, is a good player, so his games are of high quality. The opening phase is omitted. We shall only consider middle-game positions, defined as positions from move 15 onwards where both sides have at least 18 points of material[21]. We note that games 1, 2, and 6 terminate when they are still in the middle game according to this definition. Our final restriction is that only positions where Kasparov is to move are investigated[22], resulting in 94 positions as a middle-game test set. The positions are given in Appendix A.

[21] Pawn=1, Knight=3.25, Bishop=3.25, Rook=5, Queen=9. Kings do not contribute.

[22] This could be interpreted as a bias in the test positions.

For the endgame experiments we have chosen positions of five games, taken from four instructive endgame books (Fine, 1941; Bouwmeester, 1966; Levenfish and Smyslov, 1971; Averbakh, 1987). An endgame position is defined as a position where at least one side has less than 18 points of material. Only the WTM positions are considered. This results in an endgame test set, consisting of 112 positions. The positions are listed in Appendix B. The test set includes many different types of endgame, such as pawn endgames, bishop endgames, rook endgames and queen endgames. The number of blocked-pawn pairs ranges from zero to four.

2.6.3 The domineering test set

The domineering experiments have been divided into two parts. For the first series of experiments we have taken the empty standard (8×8) board as the test position. Next to the goal of finding the game-theoretic value of the test position, we have set as research goal: deciding which replacement scheme is best.

The second series of experiments concentrates on establishing the game-theoretic value of domineering, played on non-standard boards. We have investigated rectangular board sizes $m \times n$, with m ranging from 2 to 8, and n from m to 9. The variable m denotes the number of rows and the variable n denotes the number of columns of the rectangular board. Contrary to so-called *impartial* games, such as tic-tac-toe, were both players always have the same options, domineering is a game in which the options for both players are not alike. These games are called *partizan*. For partizan games it can matter which player starts the game. In the case of domineering, for square boards (including standard domineering) it is irrelevant whether Vertical or Horizontal starts, but for non-square boards it does matter. We explicitly refrain from the rule that Vertical always starts. Of course an $m \times n$ game started by Horizontal is equivalent to an $n \times m$ game started by Vertical. It thus makes sense to distinguish four possible outcomes for the various domineering games, denoted by '1', '2', 'V', and 'H'. The meanings are as follows:

1: a first-player win, independent of whether Vertical or Horizontal starts;

2: a second-player win, independent of whether Vertical or Horizontal starts;

V: a win for Vertical, independent of whether Vertical plays first or second;

H: a win for Horizontal, independent of whether Horizontal plays first or second.

2.7 Experiments and results

The literature on transposition tables is mainly tutorial in nature (e.g., Marsland, 1986), with only a few detailed discussions of performance (e.g., Ebeling, 1986; Schaeffer, 1989b). One frequently cited performance observation is that doubling the number of positions in the table reduces the size of the search tree. This is an obvious result, since the more information in the table, the greater the probability of

finding a transposition. Performance analyses of other aspects of transposition tables, such as which positions to replace, have not, as far as we know, been published in the literature. This section lists three of our experiments concerning transposition tables. In subsection 2.7.1 experiments on using replacement schemes are described. The results have been published before in Breuker *et al.* (1994a), Breuker *et al.* (1996), and Breuker *et al.* (1998b). Subsection 2.7.2 quantifies the merits of using the move information and the score information of the transposition table. In subsection 2.7.3 several ways of using the additional memory are examined. The results of the last two sections have been published before in Breuker and Uiterwijk (1995) and Breuker *et al.* (1997b).

2.7.1 Comparing replacement schemes

The most common implementation of a transposition table is a large hash table. Even though this table is usually made as large as possible, subject to memory constraints, and an overflow area is used, collisions (for which see subsection 2.4.2) are bound to occur. When a collision occurs, a choice has to be made whether to replace or to retain the position in the table. This choice is governed by a *replacement scheme*. From the literature and from discussions with computer-chess practitioners, it appears that the most common form of collision resolution is to prefer the results of deeper searches over shallower ones (Greenblatt *et al.*, 1967; Slate and Atkin, 1977; Marsland, 1986; Hyatt, 1994; Stanback, 1994). This has an intuitive appeal, but has not been supported empirically. This subsection compares the performance of seven collision-resolution schemes, the impact of clearing the transposition table between searches, and the effect of changing the number of positions in the table.

Replacement schemes

Whenever a collision is detected, a choice has to be made whether to replace the existing position in the transposition table. We examine seven different *replacement schemes*, viz. DEEP, NEW, OLD, BIG1, BIGALL, TWODEEP, TWOBIG1. They are based on five concepts, as numbered below.

1. Concept *Deep* (used in scheme DEEP).
   The concept *Deep* is traditional. It is based on the depths of the subtrees examined for the positions involved. In scheme DEEP at a collision, the position with the *deepest* subtree is preserved in the table (Marsland, 1986; Hyatt *et al.*, 1990). The rationale behind this scheme is that a subtree searched to a greater depth usually contains more nodes than a subtree searched to a shallower depth. Therefore, more time was invested in searching the larger tree. Hence, this value, if retrieved from the table, saves more work (i.e., eliminates a larger tree).

2. Concept *New* (used in scheme NEW).
   The concept *New* prefers the last examined position over earlier ones. The

replacement scheme NEW *always* replaces any position in the table when a collision occurs. This concept is based on the observation that most transpositions occur locally, within small subtrees of the global search tree (Ebeling, 1986).

3. Concept *Old* (used in scheme OLD).
   The concept *Old* prefers the earliest examined position over later ones. The replacement scheme OLD (the opposite of the scheme NEW) *never* replaces an existing position with a newer position. This scheme has only been included for the sake of completeness.

4. Concept *Big* (used in schemes BIG1 and BIGALL).
   The concept *Big* is based on the number of nodes of a subtree. Sometimes a subtree contains many forcing moves. It also may be potentially well-ordered (in which case many cut-offs have occurred). In such cases, the depth of the search tree fails to be a good indicator of the amount of search already performed and therefore potentially to be saved. It then may be attractive to select, for retention, the position with the *biggest* subtree rather than the one with the deepest subtree, going by number of nodes rather than by their depths. A drawback then is that the number of nodes must be retained as part of each transposition-table entry, reducing the effective number of positions possible for a given amount of storage.

   This concept is used in two schemes: BIG1 and BIGALL. The former counts a table position in a transposition table as a single node, the latter as N nodes, where N is the number of positions searched in order to obtain the information of the table position stored.

5. Concept *Two-level* (used in schemes TWODEEP and TWOBIG1).
   The concept *Two-level* uses a two-level transposition table (Ebeling, 1986; Schaeffer, 1994). Such a transposition table has two table positions per entry[23]. For the scheme TWODEEP the subtree of the first table position is larger than the subtree of the second table position. Upon a collision:

   - if the candidate position has been searched to a depth greater than or equal to the depth of the extant first table position, the first table position is shifted to the second table position, and the candidate position is stored in the first table position;
   - otherwise, the candidate position is stored in the second table position (possibly overwriting an existing position).

   Thus, the candidate position is always stored, and the less important of the remaining two positions (in terms of depth of search) is overwritten. We have also tested the analogous combination of the schemes NEW and BIG1 (further denoted as TWOBIG1).

[23] Ebeling (1986) implemented the two-level transposition table in a slightly different way.

We note that in all replacement schemes in our experiments the decision to overwrite an entry does not depend on the type of the score (exact value, lower bound, or upper bound) of the positions involved.

Time stamping

When playing a game, a choice must be made about what to do with the positions stored in the transposition table during the search from a previous position in the game. Successive positions in a game are related to one another, and it therefore may seem best to retain *all* positions in the transposition table[24]. However, these positions are subject to aging, and will be of little use after a few moves in the game. Consequently, clearing the transposition table between searches may also seem attractive, e.g., when the evaluation function between searches is changed.

Instead of physically clearing positions in the transposition table, it may be preferable to time-stamp them after the completion of each search. A time-stamped position remains stored in the table until a collision occurs, when it is unconditionally overwritten. While time-stamped but not overwritten, it will still be used for retrieving information. A position not time-stamped holds information more recent than any previous search.

Table sizes

Undoubtedly, many experiments have been conducted to test the effect of the transposition-table size on the number of nodes investigated. In spite of this, there are few reports in the literature. Ebeling (1986) states: "each doubling in the hash table size yields only a 7% decrease in the search size."

Schaeffer (1994) reported a 5% decrease in the number of nodes searched when doubling the number of positions in the transposition table. It is remarkable that both authors arrive at effects of the same order of magnitude in spite of employing different move-ordering techniques.

We have tested the effect of doubling the number of positions in the transposition tables by conducting the experiments for chess with eight different table sizes, ranging from 8K to 1024K positions and for domineering with four different table sizes, ranging from 256K to 2048K positions, each time doubling the number of positions[25].

The chess experiments

To test the ideas mentioned, the following chess experiments were conducted. The first series of experiments concerned middle-game positions only. It observed the performance of every combination of the seven replacement schemes (with and without

[24] We note that if the evaluation function depends on the position at the root of the search tree, search anomalies can occur if the values of positions from a previous search are retrieved from the transposition table. In our chess experiments we did not encounter that problem, since in ALIBABA the evaluation values are independent of the position at the root (cf. page 23).

[25] We use K as an abbreviation for 1024.

time stamping) and the eight table sizes. The middle-game tests have been conducted on 94 middle-game positions, taken from six games between chess experts (see Section 2.6). Each position was searched for 3 to 7 ply. For table sizes of 16K, 64K, 256K and 1024K positions 8-ply searches were performed on 44 middle-game positions taken from the first three games given in Appendix A.

The second series of experiments concerned endgame positions only. It observed the performance of every combination of the seven replacement schemes (only with time stamping) and eight table sizes (ranging from 8K to 1024K positions). These tests have been conducted on 112 endgame positions, taken from five games between chess experts (see Section 2.6). Each position was searched to a depth of 10 ply.

The domineering experiments

In the first series of experiments five replacement schemes have been compared. From the chess experiments it will be evident (as expected) that the scheme OLD is not a good candidate for practical use (cf. page 36), since it uses by far more nodes than all other replacement schemes considered. Therefore, scheme OLD is not considered for the domineering experiments. Further, it will be shown that the differences between schemes BIG1 and BIGALL are marginal in chess (cf. page 36). Therefore, for the domineering experiments we decided to drop scheme BIGALL. Thus, the following five replacement schemes are considered: TWOBIG1, TWODEEP, BIG1, DEEP and NEW. As mentioned before, the experiments are performed with four different transposition-table sizes, ranging from 256K to 2048K positions.

For the second series of experiments we have used the best replacement scheme (TWOBIG1) found from the first series of experiments together with a transposition table of 2048K positions. All boards with $m \neq n$ are investigated twice: (1) with the first player moving vertically, and (2) with the first player moving horizontally.

The performance metric

As the measure for quantifying the search effort in the chess and domineering experiments we use the number of *all* nodes investigated, i.e., the sum of the interior nodes and the leaves. The complete results of all experiments are listed in Appendix C. A few typical results are graphically illustrated in this section. When comparing the replacement schemes for each table size the number of *positions* has been kept constant. This implies that the three BIG schemes (BIG1, BIGALL, TWOBIG1) use slightly more memory than the other schemes because each table position has one additional field (to store the information about the size of the subtree searched). It is claimed that these minor differences do not affect the interpretation of the results. Further, we note that the two-level schemes (TWOBIG1, TWODEEP) have half the number of entries compared to the other five schemes.

The chess middle-game experiments without time stamping

Figure 2.8 shows the middle-game results for the seven replacement schemes using 7-ply searches without time stamping. The graph plots the number of nodes investigated (in millions) as a function of transposition-table size. The number of nodes is the sum of the nodes investigated for the 94 test positions.

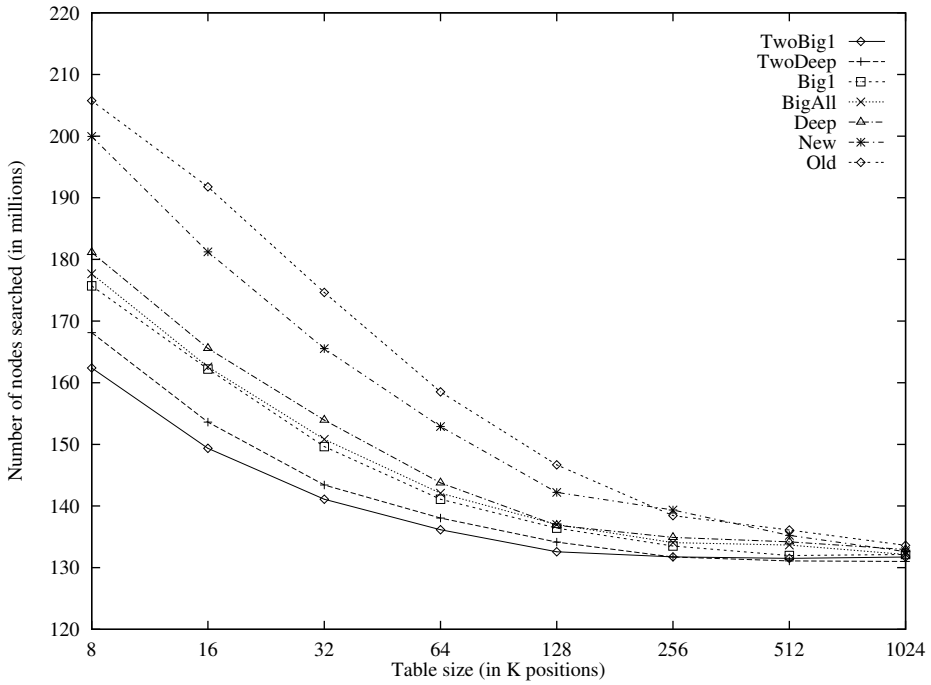


Figure 2.8: Comparing replacement schemes in the chess middle game (without time stamping, 7-ply searches).

The following trends seem to be evident.

- As the table size increases, the number of nodes searched tends to level out to a constant. In other words, at some point, possibly before 1024K in our case, no significant gains may be hoped for by increasing the table size. This is caused by the larger percentage of tree nodes that can be retained in the transposition table: the probability of harmful collisions (i.e., collisions that cost many nodes) then greatly decreases. At a certain point the transposition table is sufficiently big to hold the entire search tree.
- As the table size increases, the spread between replacement schemes shrinks. For table sizes from 512K upwards, the spread is only around 3%, whereas the smallest practicable size, 8K, suggests a spread of no less than 21% between

the best (TwoBig1) and worst (Old) scheme. This is a consequence of the argument above.

- The two-level-table schemes outperform those with one level only. For most data points, TwoBig1 is better than TwoDeep.
- The schemes Old and New are worse than the other three one-level-table schemes. This can be explained by observing that Old and New do not take into account the amount of work done to investigate a position.
- There is hardly any difference between the schemes Big1 and BigAll.
- Our data for small table sizes (8K to 64K) confirm Ebeling's (1986) statement, based on 10 positions, that TwoDeep "reduces search times by 5 to 10% for middle game positions" when compared with Deep.

It is important to observe that the deeper the search performed, the larger the transposition table should be. Beyond 256K positions for a 7-ply search, performance levels off; there is little further to gain. However, some programs can search considerably deeper than 7 ply. They may not have sufficient memory to allow a transposition-table size large enough to reach the point where doubling the number of positions in the table has a limited benefit. The shape of the lines in Figure 2.8 may provide some insight into the effect of transposition-table performance for deeper searches. For example, assuming that searching one ply deeper increases the tree size by a factor of about 4 (Thompson, 1982; Junghanns *et al.*, 1997) a 9-ply search might build a 16 times larger tree than a 7-ply search. The 9-ply results for 256K positions can be approximated by using the 16K ($\frac{256K}{16}$) data point of the 7-ply results. This shows TwoBig1 to be a clear winner.

If we use a 1% reduction in node counts as a criterion for the usefulness of doubling the number of positions in the transposition table, then we obtain from Figure 2.8 for 3, 4, 5, 6, and 7-ply searches in the middle game the following suggested table sizes: $\leq$8K, 16K, 32K, 32K, and 256K positions, respectively.

The chess middle-game experiments with time stamping

The same experiments as above were performed, the only difference being time stamping. This means that each time after a search was completed, the table positions were given a time-stamp, as opposed to clearing the table positions. Thereafter, the next position in the game was searched. Thus the results of a previous search could still be used. Figure 2.9 shows the results of these experiments.

Comparing this figure to Figure 2.8, the following trends seem to be evident.

- The shapes of all graphs are similar in the Figures 2.8 and 2.9.
- The relative order of merit of the replacement schemes seems to be invariant for time stamping; whether one time-stamps or clears the transposition tables between moves, TwoBig1 appears to have a persistent edge.

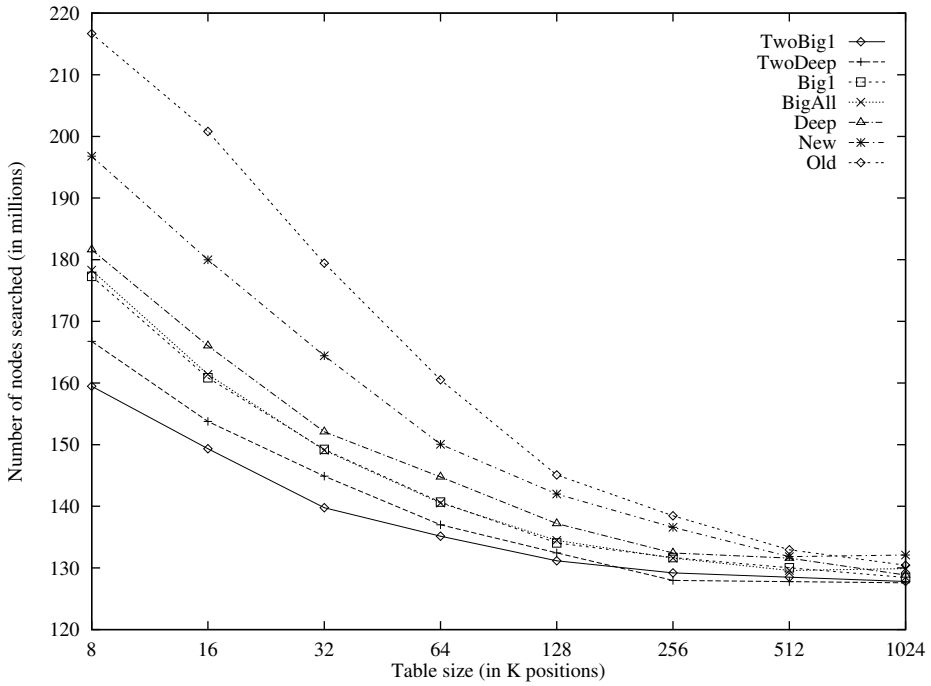


Figure 2.9: Comparing replacement schemes in the chess middle game (with time stamping, 7-ply searches).

Time stamping has a slight performance benefit. The savings with time stamping are some 2%. Therefore, it can be recommended since it only requires one additional bit per table position and requires little additional computation.

As mentioned on page 33, 8-ply searches have been performed on middle-game positions for table sizes of 16K, 64K, 256K and 1024K positions, again with and without time stamping. The results are given in Appendix C. Assuming a ratio of four in search size between subsequent ply depths, the 7-ply results for table sizes of 16K, 64K and 256K positions should be scalable to the 8-ply results for table sizes of 64K, 256K and 1024K positions, respectively. Inspection of the results verifies this. In other words, the 7-ply search conclusions given above are confirmed by the 8-ply search results, in particular the conclusion that the two-level schemes outperform those with one level.

The benefit of a transposition table in chess middle games

Figure 2.10 shows the relation between the benefit of using a transposition table and the search depth for all 94 middle-game positions. The data are shown for a transposition table of 1024K positions and replacement scheme TwoBig1, using

time stamping. The search size without a transposition table is 1.

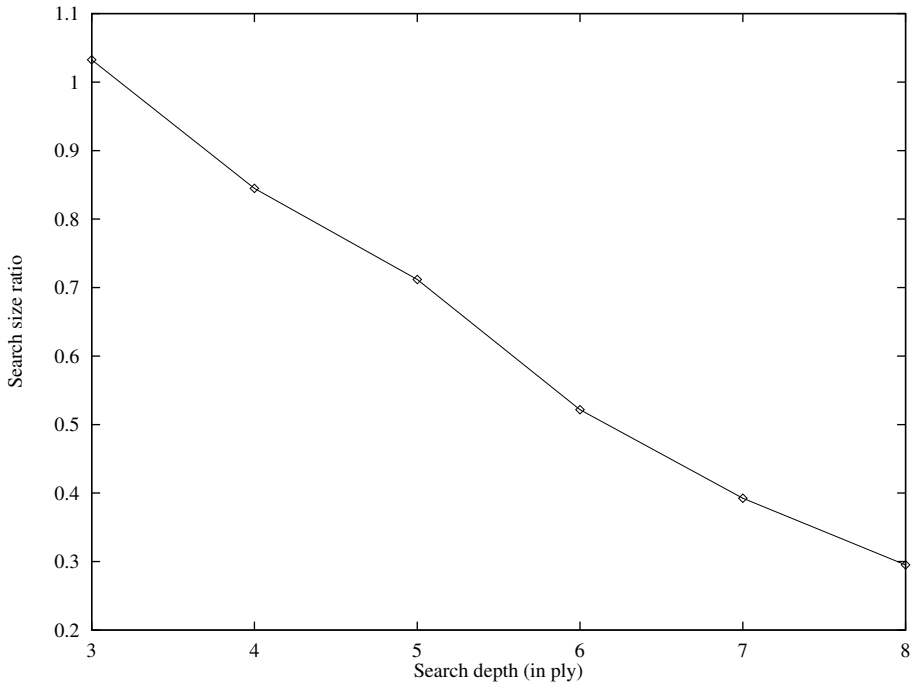


Figure 2.10: Using a transposition table in the chess middle game (with time stamping, scheme TwoBig1, 1024K positions).

For this example we see that, limiting ourselves to a 3-ply search in middle-game positions, the use of a transposition table with time stamping is even counterproductive in that it prolongs the search. The probable cause is an unfavourable move ordering, caused by a poor best-move suggestion from the transposition table. However, it is reassuring that the use of transposition tables is definitely advantageous at more realistic search depths of over 3 ply.

Ebeling (1986) concludes that "not using the hash table for moves affects the search size by at least a factor of two." The graph confirms this factor for searches of 6 ply and deeper. It is noted that transposition tables reduce the search considerably in many other domains, such as domineering (cf. page 40) and also in single-agent-search problems, such as sokoban (Junghanns and Schaeffer, 1997).

The chess endgame experiments with time stamping

Figure 2.11 shows the endgame results for the seven replacement schemes using 10-ply searches with time stamping. The graph plots the number of nodes investigated

(in millions) as a function of transposition-table size. The number of nodes is the sum of the nodes investigated for the 112 test positions.

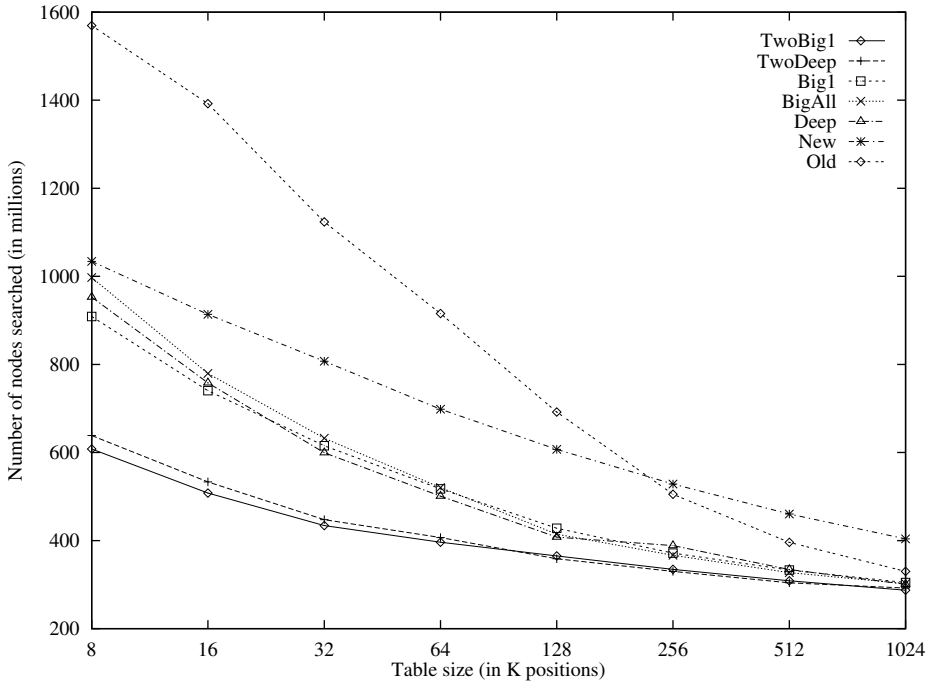


Figure 2.11: Comparing replacement schemes in the chess endgame (with time stamping, 10-ply searches).

From this graph it follows that the conclusions given for the middle-game experiment also hold for the endgame, with one exception. In middle-game positions it is clear that the concept BIG works better than the concept DEEP: schemes BIG1 and BIGALL search fewer nodes than scheme DEEP. and scheme TWOBIG1 fewer nodes than scheme TWODEEP. The difference between the two concepts has disappeared in the endgame. This is explained as follows. If a subtree contains many forcing moves or is well-ordered, cut-offs occur. Since in the middle game the mobility of each player is higher than in the endgame, such pruning will on average cause larger savings in middle-game positions than in endgame positions. Therefore, the size of search trees of equal depth will vary more in middle-game positions than in endgame positions. The concept DEEP does not have a preference for any of two such subtrees, whereas the concept BIG has a preference for the largest subtree. Thus, in the middle game the size (as compared to the depth) of the search tree investigated will be a better characteristic measuring the work performed than it is in the endgame.

If we again use a 1% reduction in node counts as a criterion for the usefulness of

doubling the number of positions in the transposition table, then we obtain for 3, 4, 5, 6, 7, 8, 9, and 10-ply searches in the endgame the following suggested table sizes: $\leq$8K, $\leq$8K, $\leq$8K, 32K, 64K, 512K, $\geq$1024K, and $\geq$1024K positions, respectively.

Solving domineering

From preliminary experiments it was obvious that standard 8×8 domineering could not be solved in a reasonable amount of time without using a transposition table (Fotland, 1997). Using a transposition table, we solved the game. It appeared to be a first-player win. Later on, we were informed that this result was independently found by Morita (1997).

In Figure 2.12 the results for the five replacement schemes in domineering are given. Detailed results are listed in Appendix C. The graph plots the number of nodes investigated (in millions) to solve the standard game as a function of transposition-table size.

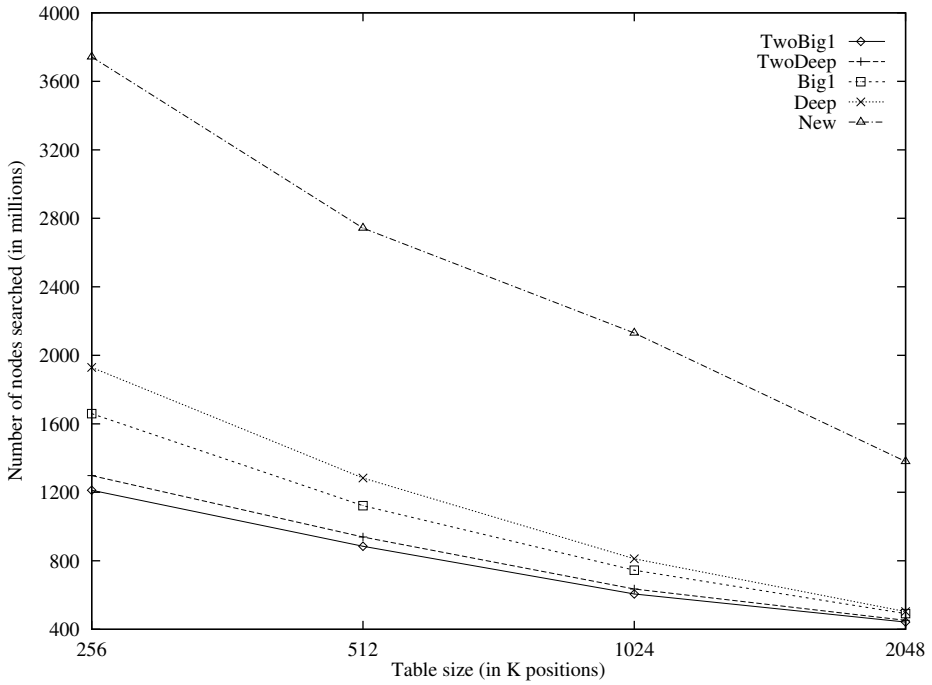


Figure 2.12: Comparing replacement schemes in domineering.

It is noted that the conclusions from the domain of chess also hold in the domain of domineering and are even more pronounced: two-level replacement schemes work much better than one-level schemes. Furthermore, the concept *Big* shows more improvement over the concept *Deep* than in chess.

Solving domineering for non-standard boards

Table 2.1 gives the results for the second series of experiments. The numbers indicate the real number of nodes investigated. The scheme used is TwoBig1 with 2048K positions. In the first column the board size is depicted. The second column gives the game-theoretic value, with '1', '2', 'V', and 'H' as defined in subsection 2.6.3.

Size	Res	Nodes	Size	Res	Nodes	Size	Res	Nodes
2×2	1	1	3×7	H	77	5×8	H	30,348
2×3	1	2	3×8	H	74	5×9	H	177,324
2×4	H	13	3×9	H	99	6×6	1	17,232
2×5	V	15	4×4	1	40	6×7	V	302,259
2×6	1	14	4×5	V	87	6×8	H	3,362,436
2×7	1	17	4×6	1	1,327	6×9	V	18,421,911
2×8	H	67	4×7	V	1,984	7×7	1	408,260
2×9	V	126	4×8	H	12,024	7×8	H	12,339,876
3×3	H	1	4×9	V	45,314	7×9	H	320,589,295
3×4	H	10	5×5	2	604	8×8	1	441,990,070
3×5	H	19	5×6	H	1,500	8×9	V	70,918,073,509
3×6	H	40	5×7	H	13,584			

Table 2.1: Game-theoretic results of domineering for various board sizes.

Our results fully agree with the results published earlier by Berlekamp and coworkers as far as investigated by them (see Berlekamp *et al.*, 1982b; Berlekamp, 1988; Guy, 1991). They provide complete analyses for boards with a size of $2\times n$ ($2\leq n\leq 7$), $3\times n$ ($3\leq n\leq 5$), and 5×5. We remark that games with a game-theoretic value '1', '2', 'H' and 'V' match their characterizations of fuzzy, zero, positive and negative games, respectively. By using a straightforward $\alpha\beta$ algorithm, returning only whether a position is a win or a loss, we did not keep track by what difference a position is won or lost. Hence, it is impossible to provide a detailed comparison with their analyses.

Another subset of our results coincide with the results obtained previously by Fotland (1997), who did his investigations several years ago. Fotland also used a straightforward $\alpha\beta$ algorithm plus a large transposition table. He did not solve the 8×8, and the $m\times 9$ ($5\leq m\leq 8$) boards. Our program Domi never investigated more nodes than Fotland's program; Domi has a more efficient node investigation than Fotland's program by a ratio of up to 10 for the larger boards.

In Table 2.1 we may discern several patterns of exponential growth with the board size, e.g., the $n\times n$ series, the $m\times n$ series with fixed m, *etc.* The results suggest that the ratio always grows exponentially with the board size. Since the 8×9 board took more than 600 hours to be solved, we did not investigate the 9×9 board. It is interesting to note that of all boards considered the 5×5 board is the only one in which the second player wins.

2.7.2 Quantifying the merits of move and score

Although it is evident that the use of a transposition table reduces the search effort, two open questions still exist. First, how big is the overall reduction? And second, which information has the largest impact on the reduction? This subsection consists of two parts. The first part compares storing the best *move* with storing the *value* of the best move. The second part compares storing the *bound* values for minimal-window search with storing the *exact* values[26].

From the components mentioned in subsection 2.3.2 it follows that a transposition table is used for two reasons: (1) the score is used for establishing the value of the position, and (2) the retrieved move is used for move ordering. In the first case the value is either an exact value, and this position does not have to be re-searched, or a bound value, in which case either the α value or the β value might be adjusted[27].

We have investigated the merits of these individual components in order to obtain more insight into the way a transposition table helps to reduce the search effort. This information may help in devising more efficient transposition-table schemes and may deliver guidelines about what additional information can be useful. For investigating the merits of *move* and *score* we have performed six experiments.

1. Search without a transposition table.
2. Search with a traditional transposition table, without *score*.
3. Search with a traditional transposition table, without *move*.
4. Search with a traditional transposition table, without *move*, only storing and using the score information if the score is an *exact value*.
5. Search with a traditional transposition table, without *move*, only storing and using the score information if the score is a *bound value*.
6. Search with a traditional transposition table, with *move* and *score*, storing and using the score information both if the score is an *exact value* or a *bound value* (i.e., use the transposition table fully).

The experiments 1 and 6 are performed to obtain upper and lower bounds.

Results of the merits of move and score

As the measure for quantifying the search effort we use the number of *all* nodes investigated, i.e., the sum of interior nodes and leaves. The test set used for the experiments consists of 18 consecutive WTM middle-game positions taken from the game Kasparov-Short, Amsterdam 1994, and 21 consecutive WTM endgame positions taken from the game Rabinovich-Romanovsky, Leningrad 1934 (see Appendix A and

[26] We note that the experiments are only performed in the chess domain, since in the domineering experiments no moves and no bound values are stored.

[27] Obviously, when the depth still to be searched is greater than the depth in the transposition table, the score from the transposition table is not used.

B). Both games were played by human experts. The 18 middle-game positions have been searched to a depth of 8 ply, and the 21 endgame positions to a depth of 10 ply. The replacement scheme used for all experiments is TwoBig1, the scheme which performs best (see subsection 2.7.1). All experiments have been performed with a series of transposition tables, ranging from 8K positions to 256K positions, since beyond 256K positions there is little further to gain, as is shown in subsection 2.7.1. Time stamping (see page 33) is used. The complete results can be found in Appendix C. The number of nodes are the cumulative results of all 18 and 21 positions, respectively. The merits of the best move and its score stored in a transposition-table entry have been examined separately.

Middle-game experiments

In Figure 2.13 the results of the use of a traditional transposition table for the middle-game positions are depicted. The figure shows the number of nodes investigated as a function of the transposition-table size. The numbers in the legend refer to the experiments mentioned on page 42.

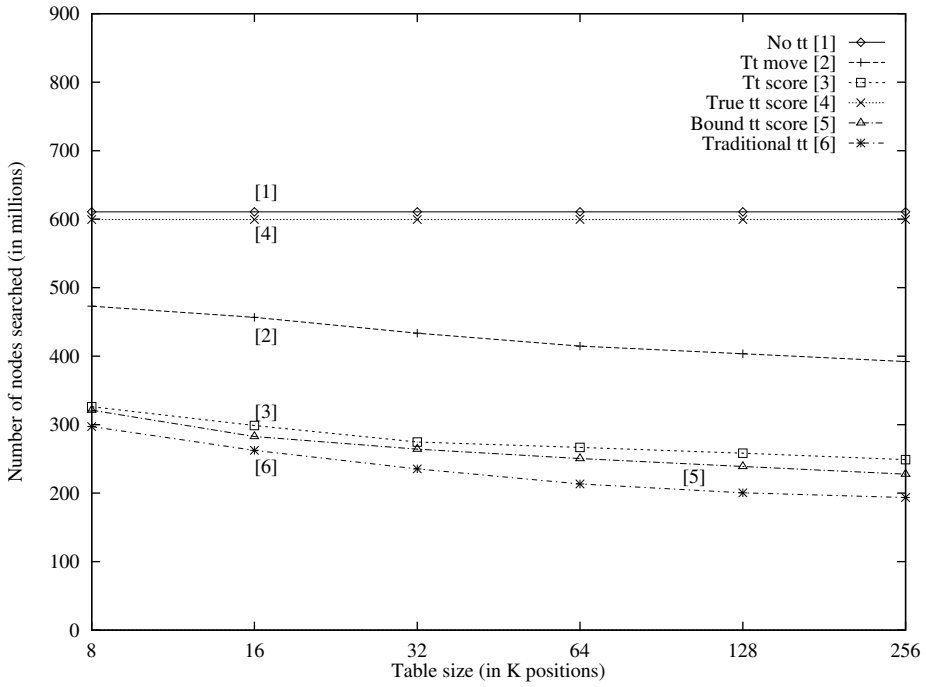


Figure 2.13: Comparing move and score in the chess middle game (8-ply searches).

Figure 2.13 clearly shows that the use of a transposition table (experiment 6) is

very profitable in terms of number of nodes searched compared to searching without a transposition table (experiment 1); a result which was already evident from the results of subsection 2.7.1. Further, using the field *score* of a transposition table (experiment 3) is more important than using the field *move* (experiment 2)[28]. This is caused by the minimal-window search: whenever one of the bounds of the minimal window is updated, its lower bound will be greater than its upper bound, thereby causing a cut-off. Experiments show that whenever a position is found in the transposition table, the retrieved value causes a cut-off in about 50% of the cases[29]. However, this effect stems fully from bound values (experiment 5). Exact values (experiment 4) hardly have any effect in this respect. Upon closer investigation it becomes clear that exact values are used only a few times. Typically, an exact value is encountered tens of times in the transposition table, while a bound value is encountered tens of thousands of times[30].

Endgame experiments

The results of the experiments on the endgame positions are analogous to the results of the experiments on the middle-game positions, but they are more pronounced, as can be seen in Figure 2.14. Moreover, the use of a transposition table is more profitable in endgames than in middle games. We see that the largest (256K positions) transposition table used in middle games with only the field *move* (experiment 2) results in about a 36% node decrease, whereas in the endgame the decrease is about 65%. If in addition *score* is used (experiment 6), a total decrease of about 68% in the middle game and about 89% in the endgame is obtained.

2.7.3 Using additional memory

A *collision* (Knuth, 1973) occurs when two different board positions map onto the same entry in the transposition table (i.e., they have an equal hash index, but a different hash value). Regardless of whether the old entry is replaced by the new one, collisions will have a negative effect on the efficiency of a transposition table, since one of the two positions will not be present in the table. The probability of the occurrence of collisions can be lowered by increasing (doubling) the number of positions in the transposition table. However, at a certain point the doubling is not profitable any more (cf. subsection 2.7.1). This subsection looks at other ways to use additional memory, by comparing the use of more information per entry position with the use of more positions in the table.

[28] We note that the results of the experiments depend on the move-ordering mechanism used (for which see page 23).

[29] The minimal window causes the retrieved value to be either a fail low, or a fail high.

[30] All nodes (except nodes on the principal variation and fail-high nodes) are searched with a minimal window. Therefore, no exact value is known for these nodes.

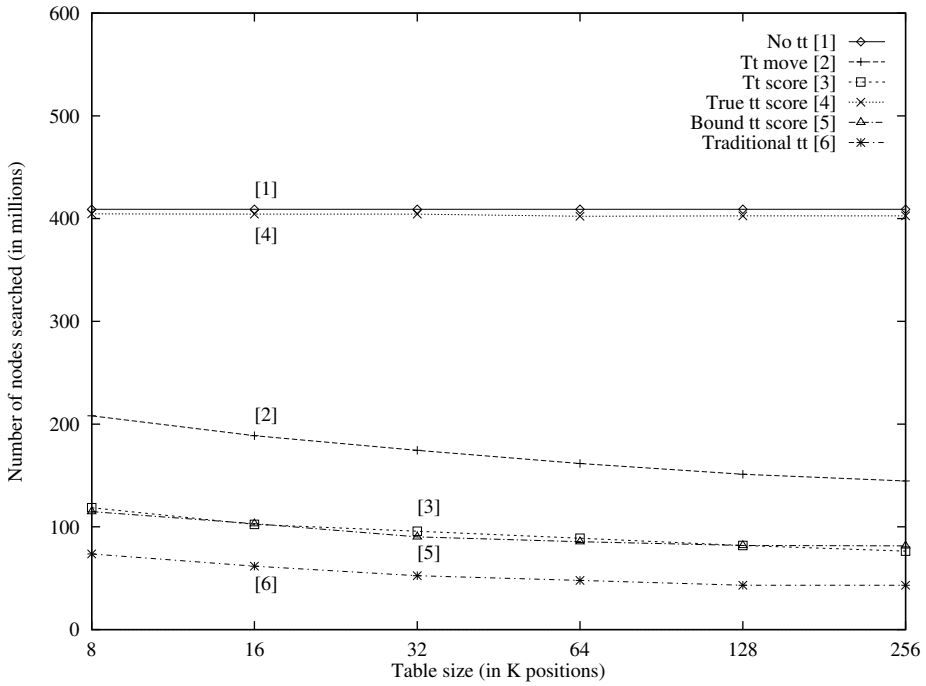


Figure 2.14: Comparing move and score in the chess endgame (10-ply searches).

Additional components

In our search for storing additional information in a transposition-table entry we have found several suggestions, amongst others made by Schaeffer (1994), Stanback (1994), and Thompson (1996a). From their suggestions, we mention six additional components.

date : contains the root's ply number in the game at the time when the position was stored[31]. Sometimes only a 1-bit date flag is used, stating whether the position is from an 'old' search or not. The date is used for time stamping. A position will be overwritten by a position with a newer date.

depth : contains the number of ply seen from the root. A position is more important if it is nearer to the root, since there it has a higher probability of being researched; possible savings are then most likely larger than savings for positions deeper in the tree.

[31] Feldmann (1996) defines *date* as the number of conversion moves (irreversible moves) made in the game.

extension : contains a Boolean value, denoting if a search extension was done at this position. The extension criteria of a node may vary (e.g., because the extension is dependent on the $\alpha\beta$ window), resulting in an extension one time and *not* in an extension the other time. The Boolean extension helps to overcome this problem (which is especially important when doing a re-search).

principal : contains a Boolean value, denoting if this position is part of the principal variation of a child of the root[32]. Positions which are part of the principal variation of a root's child are important positions, and may not be overwritten by other positions.

draw : contains a Boolean value, denoting if the backed-up score of this position is a proved draw. This is useful for distinguishing between variations resulting in positions which are real draws, and variations resulting in balanced positions (which obtain a draw value).

additional bound : instead of storing only a lower bound or an upper bound of the score, both bounds can be stored in an entry, with separate search depths for each. This is done by Truscott (1981) in the program DUCHESS.

Presumably, the information contained in these six components will have an impact on the number of nodes searched. However, only very few researchers have published even provisional results about experiments on these additional components. In subsection 2.7.1 we mentioned an experiment testing the use of a 1-bit date flag (time stamping), concluding that time stamping has a slight edge. In general it seems that adding these new components to an entry is not very profitable (Schaeffer, 1996b).

Storing the additional information described above does not take up much memory. Most fields need one bit of storage only, since they are Booleans. The choice for small additional components is made on purpose, since a larger entry results in a transposition table with fewer entries (assuming the same amount of memory is available). However, once a critical transposition-table size has been reached not much is to be gained from doubling the number of positions. Moreover, if the available memory is less than the memory needed for doubling the number of positions in the table, it still can be used for storing more information in an entry.

The above considerations have led to the question of how to use additional fields, taking up more memory than only one bit. Instead of storing the best move (which can be seen as a 1-ply principal variation) in a transposition-table entry, it may be interesting to investigate the effects of storing a deeper *principal variation* in an entry (Schaeffer, 1996b). This principal variation (PV) can be used to guide the search. If a position is not present in the transposition table, a good move may still be available from the n-ply PV information of an ancestor position.

[32] Note that this is a way to implement the refutation table using the transposition table.

Additional memory

Below we describe a limited set of experiments investigating the effects of storing an n-ply PV in a transposition-table entry[33]. The PV information is used as follows. If a position is found in the transposition table, the corresponding PV is retrieved from the table. The first move in the PV is used for move ordering and the remainder of the PV is used in further search. If a position is not found in the transposition table, and a good move is available from the PV of an ancestor position, then this move is used for move ordering.

The conditions for the experiments are the same as the conditions mentioned in subsection 2.7.2. Again, the number of nodes in the Figures 2.15 and 2.16 are the cumulative results of all 18 and 21 positions, respectively. We have tested the results of storing an n-ply PV ($n = 2...5$) in an entry versus storing only the best move (a 1-ply PV). The complete results of the experiments are presented in tabular form in Appendix C.

Middle-game experiments

In Figure 2.15 the results of the PV experiments on middle-game positions are depicted. The number of nodes investigated are shown as a function of the transposition-table size.

Our first observation is that storing an n-ply PV seems hardly worthwhile: the effects are small and severely dependent on the size of the transposition table. The explanation for this is that for less than 0.1% of the nodes investigated a position appears to be absent in the transposition table, whereas a PV from an ancestor still is available. To give some quantification, it can be seen that with the largest transposition table (256K positions), storing a 5-ply PV instead of a 1-ply PV wins roughly 5%, outperforming the 1% gain by simply doubling the number of positions in the table to 512K (see subsection 2.7.1).

Endgame experiments

The results of the experiments on the endgame positions are analogous to the results of the experiments on the middle-game positions, as can be seen in Figure 2.16. Here again, for the largest transposition-table size, the 5-ply PV outperforms the 1-ply PV, this time by some 12%.

2.8 Chapter conclusions

This chapter has shown that a transposition table (memorizing the outcome of positions previously analyzed in games, such as chess and domineering) is a useful technique. The technique has enabled us to solve a large number of different-sized

[33] We note that these experiments are solely performed in the chess domain, since no moves are stored in the domineering experiments.

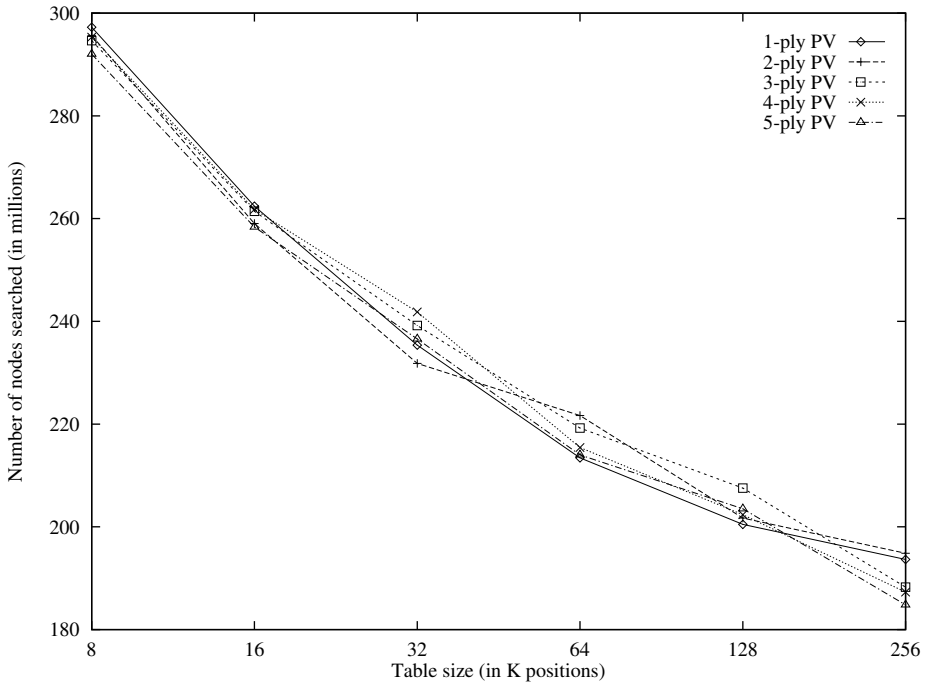


Figure 2.15: Storing an n-ply PV in the chess middle game (8-ply searches).

domineering games, including the standard 8×8 game. Without a transposition table this takes a much longer time and can therefore be considered practically impossible. We have described three series of experiments on the use of a transposition table. The goal of these experiments was to obtain more insight into the first problem statement: which methods exist to improve the efficiency of a transposition table?

First, we have tested which replacement scheme performs best. On logical grounds, one is tempted to conclude that the *number of nodes* of a subtree (used in schemes BIG1 and BIGALL) is a better estimate of the work performed (and therefore potentially to be saved) than the *depth* of that subtree (used in scheme DEEP), especially in positions with a large mobility. The experiments support this logic. In chess middle-game positions and in domineering the schemes based on the concept BIG perform better than the schemes based on the concept DEEP. In chess endgame positions this difference disappears, since the lower mobility then diminishes the differences in effects of the two measures. Based on the 7-ply and 8-ply results in chess middle games, the 10-ply results in chess endgames and the domineering results, we conclude that a two-level scheme is better than any one-level scheme. Thus it follows that the most widely used scheme, DEEP, is not best. Based on the conclusions

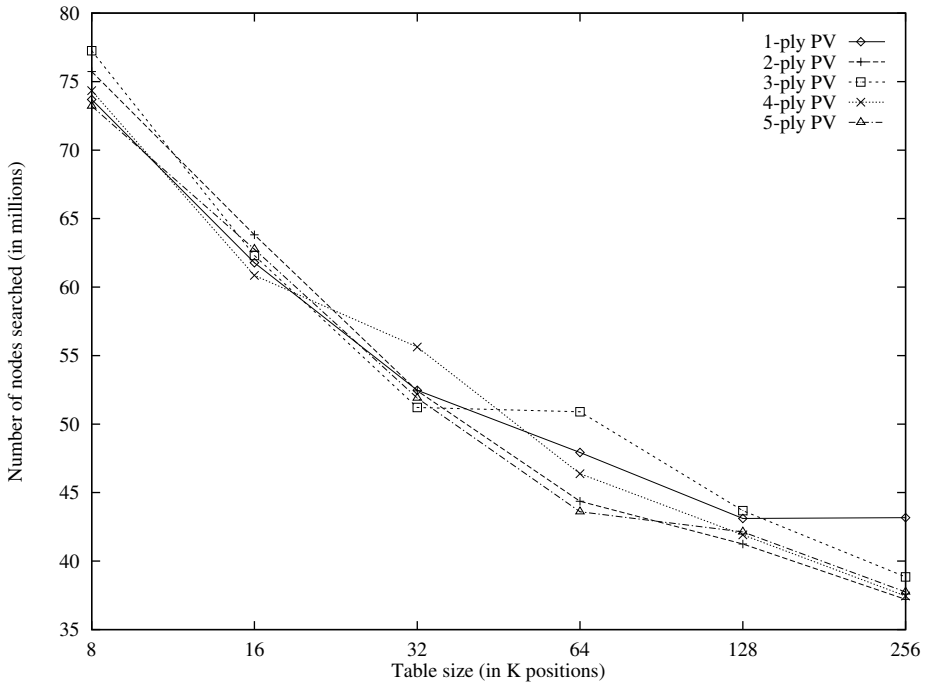


Figure 2.16: Storing an n-ply PV in the chess endgame(10-ply searches).

we recommend using the scheme TwoBig1 as the best replacement scheme for a transposition table.

Second, it is examined which information is more important to store in a transposition-table entry: the best move in a position, or the score of that move. It follows that storing the score of a position is more profitable than storing the best move. This result holds for chess middle-game positions as well as endgame positions. It was also found that for minimal-window search bound values have a much larger effect than exact values. This effect, although nowadays expected, contrasts with the idea for which transposition tables originally were devised, i.e., avoiding the re-search of positions searched before.

Third, we have tested the effect of storing an n-ply PV ($n = 2...5$) in an entry, instead of only the best move (a 1-ply PV). Preliminary results show that a 5-ply PV may win roughly 5% for the chess middle game, and 12% for the endgame, though more experiments are necessary to validate the conjecture that it really is profitable to use additional memory by storing a 5-ply PV instead of increasing the number of positions in the transposition table.

From the experiments it follows that it is important to choose a good replacement scheme. Further, the available memory can be used to make the transposition

table as large as possible. However, once a critical transposition-table size has been reached not much is to be gained from doubling the number of positions in the table. In that case, better ways exist for using the available memory. Instead of doubling the number of positions in the transposition table, it is better to use the additional memory by storing more information in an entry, thereby enlarging the entry size. Based on the above experiments it is recommended to concentrate on storing additional information which affects the number of cut-offs generated by bound values.

Chapter 3

The proof-number search algorithm

This chapter is a slightly adapted version of Breuker D.M., Allis L.V., and Herik H.J. van den (1994b). How to Mate: Applying Proof-Number Search. *Advances in Computer Chess 7* (eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 251–272. University of Limburg, Maastricht, The Netherlands[1].

The second and third problem statement deal with best-first search. In this chapter we therefore present a relatively new best-first search algorithm, called *proof-number search* (pn search), which will be used in the experiments addressing the second and third problem statement.

The basic ideas behind the pn-search algorithm are presented in Section 3.1. Section 3.2 lists the pseudo-code of the pn-search algorithm for trees. The experimental set-up is given in Section 3.3, and the test set is described in Section 3.4. Section 3.5 provides the experiments, of which the results are discussed in Section 3.6. Finally, Section 3.7 evaluates the experiments.

3.1 An informal description

In this section we present a short overview of pn search, based on Allis (1994). A detailed description of pn search can be found in Allis *et al.* (1994).

Proof-number search is a best-first AND/OR tree-search algorithm, and is inspired by the conspiracy-number algorithm (McAllester, 1988; Schaeffer, 1990). Before starting the search, a search goal is defined (e.g., try to reach at least a draw). The evaluation of a node returns one of three values: *true*, *false*, or *unknown*. The evaluation is seen from the point of view of the player to move in the root position. The value *true* indicates that the player to move in the root position can achieve the

[1]Thanks are due to the Editors of *Advances in Computer Chess 7* for giving permission to use the contents of the article in this chapter.

goal, while *false* indicates that the goal is unreachable. A node is *proved* if its value has been established to be *true*, whereas the node is *disproved* if its value has been determined to be *false*. A node is *solved* as soon as it has been proved or disproved. A tree is solved (proved or disproved) if its root is solved. The goal of pn search is to solve a tree.

Two variants of creating a search tree exist (cf. Allis, 1994).

1. *Immediate evaluation*. Each node in the tree is immediately evaluated after it is generated. The tree is built by first generating (and evaluating) the root. Then at each step a leaf is selected, expanded and all its children are immediately evaluated.

2. *Delayed evaluation*. Each node is only evaluated when it is selected, and not immediately after it is generated. The tree is built by first generating the root (without evaluation). Then, at each step a leaf is selected and evaluated. If the evaluation value is *unknown*, the node is expanded (without evaluating its children).

The advantage of immediate over delayed evaluation is that in the former variant more information is available. However, if the evaluation takes much time, it is better to use the delayed variant, avoiding the evaluation of many nodes that will not be used for solving the tree. Since, in the standard pn-search experiments described in this chapter, the evaluation is fast (only checking whether the position is a win, a loss, or a draw) we use the immediate variant in our further description of pn search.

Like other best-first search algorithms, pn search repeatedly selects a leaf, expands it, evaluates all its children, and updates the tree with the information obtained from the expansions and evaluations. Unlike most other best-first search algorithms, pn search does not use a heuristic evaluation function in order to determine a most-promising node. Instead, the shape of the search tree (the number of children of every internal node) and the values of the leaves determine which node to select next.

In general, to solve a tree, a number of leaves of the current search tree needs to be proved or disproved. A set of leaves, which, if all proved, would prove the tree, is called a *proof set*. Likewise, a set of leaves, which, if all disproved, would disprove the tree, is called a *disproof set*. The size of the smallest proof set of the tree is a lower bound for the number of node expansions necessary to prove the tree, while the size of the smallest disproof set of the tree is a lower bound for the number of node expansions necessary to disprove the tree.

In Figure 3.1 an AND/OR tree has been depicted. The numbers to the left of a node denote proof numbers, while the numbers to the right of a node denote disproof numbers. A *proof number* of a node represents the minimum number of leaves which have to be proved in order to prove that node. Analogously, a *disproof number* of a node represents the minimum number of leaves which have to be disproved in order to disprove that node.

Proved nodes (e.g., node K in Figure 3.1) have proof number 0 and disproof number ∞. This follows from the fact that no expansions are needed to prove the

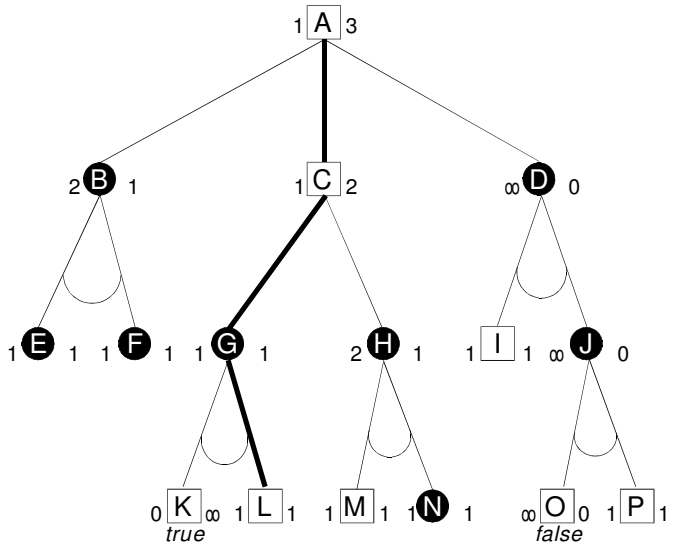


Figure 3.1: An AND/OR tree with proof and disproof numbers.

node, since it is already proved, and that no number of expansions could ever disprove the node. Analogously, disproved nodes (e.g., node O in Figure 3.1) have proof number ∞ and disproof number 0. Unsolved leaves (e.g., nodes E, F, L, M, N, I, and P) have a proof and disproof number of unity, as expanding the node itself may be sufficient to solve the node.

Internal AND nodes have as proof numbers the sum of the proof numbers of their children, since to prove an AND node, all children must be proved. The disproof number of an AND node equals the minimum of its childrens' disproof numbers, since only one child needs to be disproved to disprove the AND node. For instance, the proof number of node H is equal to the sum of the proof numbers of its children M and N ($2 = 1+1$). The disproof number of node H is equal to the minimum of the disproof numbers of its children (1). Analogously, the proof number of an internal OR node equals the minimum of the proof numbers of its children, whereas its disproof number equals the sum of the disproof numbers of its children. For instance, the proof number of node A is equal to the minimum of the proof numbers of its children B, C and D (1). The disproof number of node A is equal to the sum of the disproof numbers of its children ($3 = 1+2+0$).

The root (A) has proof number 1. This means that at least one leaf (in this case node L) should be proved to prove the root. The disproof number of the root is equal to 3. This means that at least three nodes (node E or node F, node L, and node M or node N) have to be disproved to disprove the root.

The main assumption underlying pn search is that it is generally better to expand

those nodes which are in the smallest proof and/or disproof sets. In other words, pn search concentrates at each step on the potentially least amount of work necessary to solve the tree.

The only remaining question is: when to select a node from the smallest proof set of the root and when to select a node from its smallest disproof set? Surprisingly, we can always do both at the same time. Allis *et al.* (1994) prove that the intersection of any smallest proof set and any smallest disproof set of the same node is always non-empty. The nodes which are elements of both a smallest proof set and a smallest disproof set of the root are called *most-proving nodes*. Thus, if after expansion of a most-proving node P, it obtains the value *true*, the proof number of the root is decremented by unity, while if P obtains the value *false*, the disproof number of the root is decremented by unity. If the value of P remains *unknown*, the newly generated children may have their impact on the proof and/or disproof numbers of P and its ancestors. A most-proving node is determined in the tree by selecting, at AND nodes, a child with disproof number equal to its parent's, and at OR nodes a child with proof number equal to its parent's. By thus traversing the tree from its root to a leaf (e.g., the bold path from A to L in Figure 3.1), it is shown that a most-proving node is found (Allis *et al.*, 1994).

3.2 The pseudo-code of the algorithm

All algorithms given in this section are based on the algorithms given by Allis (1994). The main proof-number search algorithm is given in Figure 3.2. The only parameter of the procedure is root, being the root of the search tree. After execution of the procedure, the root's value can have one of three values: *true*, *false* or *unknown*. First, the root is evaluated and its proof and disproof numbers are initialized. Then, in the main loop, repeatedly a most-proving node is selected, expanded, and all its children are evaluated. Thereafter, traversing the tree backwards to the root, the proof and disproof numbers are adjusted.

The function Evaluate evaluates a position, and returns one of the following three values: *true*, *false*, or *unknown*. The function SetProofAndDisproofNumbers initializes the proof and disproof numbers of a node. The algorithm is given in Figure 3.3. The only parameter of the function is node, being the node to be initialized. Two cases are distinguished. In the first case the node is an internal node (since it is expanded), and the proof and disproof numbers are initialized according to the proof and disproof numbers of its children. In the second case the node is not expanded, but it is evaluated, since immediate evaluation is used. The proof and disproof numbers are initialized according to the evaluation.

The function ResourcesAvailable returns a Boolean value indicating whether sufficient resources are available to continue searching. This is usually dependent on the available memory, but can also depend on a limited amount of time available. The function SelectMostProvingNode finds a most-proving node. The algorithm is given in Figure 3.4. The only parameter of the function is node, being the root of the

```
procedure ProofNumberSearch( root )
    Evaluate( root )
    SetProofAndDisproofNumbers( root )
    root.expanded := false

    while root.proof≠0 and root.disproof≠0 and
            ResourcesAvailable() do begin
        mostProvingNode := SelectMostProvingNode( root )
        ExpandNode( mostProvingNode )
        UpdateAncestors( mostProvingNode, root )
    end

    if root.proof=0 then root.value := true
    elseif root.disproof=0 then root.value := false
    else root.value := unknown /* resources exhausted */
end /* ProofNumberSearch */
```

Figure 3.2: The pn-search algorithm for trees.

(sub)tree where the most-proving node is located. As long as the node is expanded, a child is chosen with proof or disproof number (dependent on the type of node) equal to that of the parent. If a leaf is reached, the algorithm stops, and that node is returned.

The most-proving node found is expanded. This is done by the procedure ExpandNode. The only parameter of this procedure is node, being the node to be expanded. In Figure 3.5 its algorithm is depicted. First, all children are generated. Next, every child is evaluated and its proof and disproof numbers are set according to this evaluation.

After the expansion of the most-proving node, the new information has to be backed up throughout the whole tree. This is done by the procedure UpdateAncestors. The procedure has two parameters. The first parameter (node) is the node to be updated, while the second parameter (root) is the root of the search tree. Its algorithm is shown in Figure 3.6.

3.3 Experimental set-up

Pn search first examines the most forcing variations where the mobility of the opponent is as small as possible. This is explained as follows. The OR player chooses a child with the lowest proof number. By definition, the proof number of this child (an AND node) is equal to the sum of the proof numbers of its children. It follows that the AND child with the lowest proof number has the lowest mobility. Because pn search first examines forcing variations, it is expected that it will work extremely

```
procedure SetProofAndDisproofNumbers( node )
    if node.expanded then /* internal node */
        if node.type=AND then begin /* AND node */
            node.proof := 0
            node.disproof := ∞
            for i:=1 to node.numberOfChildren do begin
                /* Add up proof numbers and minimize disproof numbers */
                node.proof := node.proof + node.children[ i ].proof
                if node.children[ i ].disproof<node.disproof then
                    node.disproof := node.children[ i ].disproof
            end
        end else begin /* OR node */
            node.proof := ∞
            node.disproof := 0
            for i:=1 to node.numberOfChildren do begin
                /* Minimize proof numbers and add up disproof numbers */
                if node.children[ i ].proof<node.proof then
                    node.proof := node.children[ i ].proof
                node.disproof := node.disproof + node.children[ i ].disproof
            end
        end
    else /* leaf */
        case node.value of begin
            false:
                node.proof := ∞
                node.disproof := 0
            true:
                node.proof := 0
                node.disproof := ∞
            unknown:
                node.proof := 1
                node.disproof := 1
        end
end /* SetProofAndDisproofNumbers */
```

Figure 3.3: The proof-and-disproof-numbers-calculation algorithm.

```
function SelectMostProvingNode( node )
    while node.expanded do begin
        i := 1
        if node.type=OR then /* OR node */
            while node.children[ i ].proof≠node.proof do i := i+1
        else /* AND node */
            while node.children[ i ].disproof≠node.disproof do i := i+1
        node := node.children[ i ]
    end

    return node
end /* SelectMostProvingNode */
```

Figure 3.4: The most-proving-node-selection algorithm.

```
procedure ExpandNode( node )
    GenerateAllChildren( node )
    for i:=1 to node.numberOfChildren do begin
        Evaluate( node.children[ i ] )
        SetProofAndDisproofNumbers( node.children[ i ] )
        node.children[ i ].expanded := false
    end
    node.expanded := true
end /* ExpandNode */
```

Figure 3.5: The node-expansion algorithm.

well in cases where the goal can be reached by forcing variations. Therefore, we have chosen to investigate this in the domain of finding checkmates.

3.3.1 The search engine

The proof-number search engine is implemented according to the description in Section 3.1. The most important enhancement of the pn-search implementation, relative to a naïve implementation is in the initialization of proof and disproof numbers at the leaves. In the standard algorithm, proof and disproof numbers are each initialized to unity. Assume that *after* expansion all the n children evaluate to the value unknown. Then the proof and disproof numbers of the most-proving node are set to 1 and n for an OR node, and to n and 1 for an AND node. In our implementation, to distinguish between leaves, *before* expansion, we set the proof and disproof number of node P to 1 and n (or n and 1, depending on the node type), where n is the

```
procedure UpdateAncestors( node, root )
    SetProofAndDisproofNumbers( node )
    while node≠root do begin
        node := node.parent
        SetProofAndDisproofNumbers( node )
    end
end /* UpdateAncestors */
```

Figure 3.6: The ancestor-updating algorithm.

number of legal moves in the position represented by P. Experiments show that the extra overhead introduced by counting the number of legal moves at each node is more than compensated for by the value of the extra information thus revealed to the node-selection process (Allis, 1994).

3.3.2 The move ordering

For pn search the move ordering is of less importance than for $\alpha\beta$ search. For the reproducibility of the experiments we have chosen to order the moves in descending square order (h8, g8, ..., a8, h7, ..., a7, ... , h1 ... a1). The moves are sorted according to their *from* squares. If two moves have identical *from* squares they are sorted according to their *to* squares. If these are also identical, then the moves must be promotion moves, and the moves are sorted according to their promotion pieces (in the order Queen, Rook, Bishop, Knight).

As an example we provide the starting position at the game of chess. The White moves are sorted thus: **h2–h4**, **h2–h3**, **g2–g4**, **g2–g3**, **f2–f4**, **f2–f3**, **e2–e4**, **e2–e3**, **d2–d4**, **d2–d3**, **c2–c4**, **c2–c3**, **b2–b4**, **b2–b3**, **a2–a4**, **a2–a3**, ♘**g1–h3**, ♘**g1–f3**, ♘**b1–c3**, ♘**b1–a3**.

3.4 The test set

For our experiments we used a diverse set of mating problems. They are taken from Krabbé's (1985) *Chess Curiosities* and Reinfeld's (1958) *Win at Chess*. The 35 positions taken from Krabbé (1985) are mating problems in six moves or more. They are indicated by the name Kx, in which x refers to the diagram number in the source and takes the values 8, 35, 37, 38, 40, 44, 60, 61, 78, 192, 194, 195, 196, 197, 198, 199, 206, 207, 208, 209, 210, 211, 212, 214, 215, 216, 217, 218, 219, 220, 261, 284, 317, 333 and 334. The 82 positions taken from Reinfeld (1958) are problems where we know that a forced mate is possible. They are indicated by the name Rx, x again referring to the problem number in the source, this time running over 1, 4, 5, 6, 9, 12, 14, 27, 35, 49, 50, 51, 54, 55, 57, 60, 61, 64, 79, 84, 88, 96, 97, 99, 102, 103, 104, 105, 132, 134, 136, 138, 139, 143, 154, 156, 158, 159, 160, 161, 167,

168, 172, 173, 177, 179, 182, 184, 186, 188, 191, 197, 201, 203, 211, 212, 215, 217, 218, 219, 222, 225, 241, 244, 246, 250, 251, 252, 253, 260, 263, 266, 267, 278, 281, 282, 283, 285, 290, 293, 295 and 298. This results in a test set of 117 positions (see Appendix D).

3.5 Experiments

Pn search always aims at proving or disproving a certain goal. In our experiments the only goal is searching for mate. In our description, we distinguish between the attacker and the defender. The attacker is the player to move in the root position, while the defender is the opponent. A position is proved if the attacker can mate, while draws (by stalemate, by repetition of positions and by the 50-move rule) and mates by the defender are defined to be disproved positions for the attacker. If a position is neither proved nor disproved, it is said not to be solved.

We have compared the the pn-search algorithm to the $\alpha\beta$-search algorithm, implemented in DUCK[2] on the test set described in Section 3.4. We note that it is possible to create a special mate searcher using $\alpha\beta$ search, which will perform better than DUCK. However, pn search does not use any chess-specific knowledge other than recognizing mates, stalemates, and drawn positions. Therefore, we decided to choose DUCK as the $\alpha\beta$ searcher. The search was terminated as soon as any mate was found. The experiments are conducted to investigate how a best-first search algorithm (using much memory, since it stores the whole search tree) compares to the widely used depth-first $\alpha\beta$-search algorithm (using little memory). Furthermore, in the next chapters we concentrate on best-first search, using proof-number search as example and using the same test set.

We have performed experiments in which both programs had to solve each position within 1,000,000 nodes. This limit was selected for two reasons:

1. The calculation time (up to 5 minutes on the hardware used) corresponds roughly to tournament conditions.

2. The search tree for pn search must be kept in memory during the calculations: a tree of 1,000,000 nodes is close to the maximum achievable on the hardware used.

3.6 Results

This section contains the results of the experiments described in Section 3.5. The complete results for every test position individually are presented in Appendix E. As

[2] In contrast to ALIBABA, of which the $\alpha\beta$ search engine was designed specifically for the transposition-table experiments, DUCK is a full-blown tournament program, incorporating a detailed evaluation function and several $\alpha\beta$ enhancements such as extension heuristics. For more details, see Breuker *et al.* (1994b).

the measures of performance we use the number of positions solved and the number of nodes investigated.

From the 117 test positions, 106 positions were solved by at least one algorithm: 73 were solved by both algorithms, 30 by pn search only and 3 by $\alpha\beta$ search only. We have stated for each test position the algorithm by which it could be solved within 1,000,000 nodes.

Both algorithms: K35, K38, K197, K211, K212, K261, K317, R1, R4, R5, R9, R12, R14, R27, R35, R49, R50, R54, R55, R57, R60, R61, R64, R79, R84, R88, R97, R99, R102, R103, R104, R132, R134, R136, R139, R143, R154, R156, R158, R160, R161, R167, R172, R173, R177, R179, R184, R186, R188, R191, R197, R203, R211, R212, R215, R217, R219, R225, R244, R246, R251, R253, R260, R263, R266, R267, R278, R282, R283, R285, R290, R295, R298.

Pn search only: K37, K61, K192, K194, K196, K198, K199, K206, K207, K208, K214, K215, K216, K218, K219, K333, K334, R6, R51, R138, R159, R168, R182, R218, R222, R241, R250, R252, R281, R293.

$\alpha\beta$ search only: K60, K284, R105.

Neither algorithm: K8, K40, K44, K78, K195, K209, K210, K217, K220, R96, R201.

In Table 3.1 the results are summarized. In the first row of the table the total number of nodes searched on the 73 positions solved by both algorithms is listed. The second row contains the average number of nodes searched per position. The third row lists the number of times the stated algorithm outperformed the other (by the criterion of the number of nodes searched). In the fourth and fifth row a position is selected where the ratio of nodes visited was lowest for pn search and $\alpha\beta$ search, respectively. The sixth row shows the average number of nodes searched by pn search on the 30 positions not solved within a million nodes by $\alpha\beta$ search. The last row shows the average number of nodes searched by $\alpha\beta$ search on K60, K284 and R105, the only three positions solved by $\alpha\beta$ search but not by pn search.

Comparing the performance of pn search with $\alpha\beta$ search creates a consistent impression of a general superiority of pn search as a mate searcher.

- The total number of nodes investigated by pn search is about 20% of the number of nodes investigated by $\alpha\beta$ search.
- Pn search outperformed $\alpha\beta$ search in some 84% of the cases.
- The ratio of nodes visited from the point of view of pn search was lowest in the case of position R217. The number of investigated nodes by pn search is only a fraction (0.08%) of the number investigated by $\alpha\beta$ search. From the point of view of $\alpha\beta$ search the best position was R253. Here, the number of nodes investigated by $\alpha\beta$ search is about 13% of the number investigated by pn search.

	Pn search	$\alpha\beta$ search
Total number of nodes searched	953,762	5,198,074
Average number of nodes per position	13,065	71,206
Best performer	61	12
Best instance (nodes) of pn search (R217)	271	331,404
Best instance (nodes) of $\alpha\beta$ search (R253)	2,355	311
Pn search (nodes) where $\alpha\beta$ search failed	230,166	>1,000,000
$\alpha\beta$ search (nodes) where pn search failed	>1,000,000	644,058

Table 3.1: Comparing pn search and $\alpha\beta$ search.

- The average number of nodes investigated by pn search on the 30 positions that $\alpha\beta$ search did not solve within 1,000,000 nodes is 230,166. In contrast, the number of nodes investigated by $\alpha\beta$ search on the three positions that pn search did not solve within 1,000,000 nodes is much higher (644,058).

In the next subsection the particular strengths and weaknesses of pn search are discussed.

3.6.1 Strengths of pn search

This subsection discusses two strengths of pn search: (1) the algorithm does not need specific chess knowledge, and (2) the algorithm finds deep, forced mates.

No specific chess knowledge

The pn-search algorithm does not use any specific chess knowledge. All that is needed is a move generator, and an evaluation function able to recognize mate, stalemate and draws by repetition or by the 50-move rule. We would like to stress that, quite unlike $\alpha\beta$ search, move ordering has not much influence on the performance of pn search. This phenomenon is explained by the way pn search builds its tree. At each step, the child with the smallest proof or disproof number (depending on the node type) is selected. Only if two children tie is the selection of a node based on the move ordering. Experiments with changing the move ordering showed that this ordering has little influence on the number of nodes grown[3]. Not using any chess knowledge has the advantage that pn search can be incorporated into any chess program, regardless of the evaluation function and of the heuristics applied.

[3] This is contrary to the results found for conspiracy-number search, as given by Klingbeil and Schaeffer (1990). They show that move ordering does have influence when searching in tactical chess positions.

Finding deep, forced mates

The strategy of pn search may be described as investigating first those variations in which the opponent has the least mobility. Instead of examining the mobility for a single position, pn search examines the mobility of the search tree as a whole. The proof number of the root indicates, at any point in time during the computation, the mobility left to the defender for escaping mate. The achievements seem to indicate that, during a mate search, mobility is the most important factor. Clearly, chess characteristics, such as material balance and positional advantage, lose most of their meaning when trying to force a mate is the unique goal aimed at. Moreover, the distance-to-mate is no longer a dominant factor in the size of the search tree grown. As long as the mobility of the defender is restricted, pn search will continue to explore a variation, regardless of the depth of the subtree explored. We present two sample positions where this characteristic leads to the discovery of a deep mate, which would not be found if the depth of the subtree explored was an important factor (as it is in $\alpha\beta$ search).

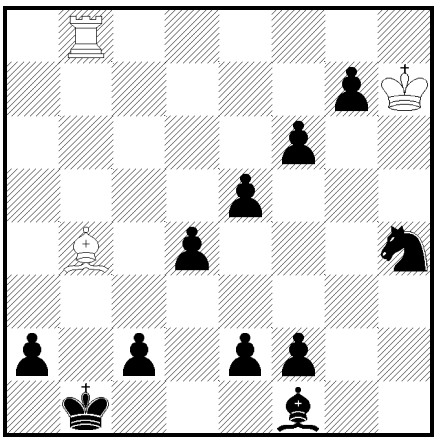


Figure 3.7: Mate in 38 (WTM); (L. Ugren).

The position in Figure 3.7 is taken from Diagram 194 in Krabbé (1985). We note that the square a1 is the left-bottom square, so Black has 4 Pawns ready to promote. For the chess-playing reader we cite the solution as stated by Krabbé: " **1. ♗a3+ ♔a1 2. ♗b2+ ♔b1 3. ♗xd4+ ♔c1** If White could now play 4. ♗b2+ ♔b1 5. ♗xe5+ *etc.*, that would shorten the procedure enormously, but of course Black would escape: 4. ..., ♔d2. This necessitates the repetition of a seven-move operation to bring the zwickmühle around: **4. ♗e3+ ♔d1 5. ♖d8+ ♔e1 6. ♗d2+ ♔d1 7. ♗b4+ ♔c1 8. ♗a3+ ♔b1 9. ♖b8+ ♔a1 10. ♗b2+ ♔b1 11. ♗xe5+ ♔c1 12. ♗f4+ ♔d1 13. ♖d8+ ♔e1 14. ♗d2+ ♔d1 15. ♗b4+**

♔**c1 16.** ♗**a3+** ♔**b1 17.** ♖**b8+** ♔**a1 18.** ♗**b2+** ♔**b1 19.** ♗**xf6+** ♔**c1 20.** ♗**g5+** ♔**d1 21.** ♖**d8+** ♔**e1 22.** ♗**d2+** ♔**d1 23.** ♗**b4+** ♔**c1 24.** ♗**a3+** ♔**b1 25.** ♖**b8+** ♔**a1 26.** ♗**b2+** ♔**b1 27.** ♗**xg7+** ♔**c1 28.** ♗**h6+** ♔**d1 29.** ♖**d8+** ♔**e1 30.** ♗**d2+** ♔**d1 31.** ♗**b4+** ♔**c1 32.** ♗**a3+** ♔**b1 33.** ♖**b8+** ♔**a1 34.** ♗**e7!** and finally the idea is clear: f6 is the only safe square to threaten mate; on other squares the ♘h4 or one of the Pawns could have thwarted that mate. ♙d4 and ♙e5 had to go to open the diagonal, ♙f6 to gain access to g7, and ♙g7 to gain access to f6. After 34. ♗e7, mate cannot be staved off for more than a few moves." The remaining moves are: **34.** ..., **c1=**♕ **35.** ♗**f6+** ♕**b2 36.** ♖**xb2 e1=**♕ **37.** ♖**b8+** ♕**e5 38.** ♗**xe5** mate.

Proof-number search solves this mate in 229,423 nodes, whereas our implementation of $\alpha\beta$ search fails to solve it within 50,000,000 nodes[4].

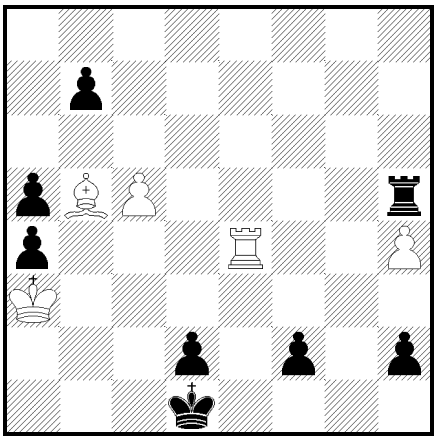


Figure 3.8: Mate in 25 (WTM); (J.-L. Seret).

The position in Figure 3.8 is taken from Diagram 199 in Krabbé (1985). Again, for the chess-playing reader we give the solution as stated by Krabbé: "Here, there are also two troublemakers and White disposes of an extended zwickmühle like the one in diagram 194 [our Figure 3.7] to silence them. 1. ♔b2 would mean mate in 2 if Black didn't have 1. ..., a3+. That Pawn can be immediately removed with 1. ♗xa4+, but after ♔c1 2. ♖c4+ ♔b1 3. ♗c2+ ♔a1! (♔c1 4. ♗f5+ allows White to enter the solution at move 14) 4. ♔b3 Black has the nasty 4. ..., b5 5. cxb6 ♖b5+ *etc.* Therefore, in order to remove the ♙a4, White must first remove the ♙b7. Hence **1. ♗e2+** ♔**e1** (♔c2 2. ♖c4+ ♔b1 3. ♗d3+ ♔a1 4. ♖c2 and 5. ♖a2 mate) **2.** ♗**g4+** ♔**f1 3.** ♗**h3+** ♔**g1 4.** ♖**g4+** ♔**h1 5.** ♗**g2+** ♔**g1**

[4] This result is heavily dependent on the search extensions used. The $\alpha\beta$ program THETURK solves this mate in 3,325,715 nodes when choosing the right extensions (Schaeffer, 1998)

6. ♗xb7+! ♔f1 7. ♗a6+ ♔e1 8. ♖e4+ ♔d1 9. ♗e2+ ♔e1 10. ♗b5+! ♔d1 and we are back in the diagram, but without the ♙b7 which means the ♙a4 meets its end too. **11. ♗xa4+ ♔c1 12. ♖c4+ ♔d1 13. ♗c2+ ♔c1!** Because if now 13. ..., ♔a1 14. ♔b3! **14. ♗f5+ ♔d1 15. ♗g4+ ♔e1 16. ♖e4+ ♔f1 17. ♗h3+ ♔g1 18. ♖g4+ ♔h1 19. ♗g2+ ♔g1 20. ♗c6+! ♔f1 21. ♗b5+ ♔e1 22. ♖e4+ ♔d1** and there we are: back in the diagram, but without those inconvenient Pawns. **23. ♔b2! ♖xc5 24. ♗a4+ ♖c2+ 25. ♗xc2 mate**."

Proof-number search solves this mate in 370,016 nodes, whereas our implementation of $\alpha\beta$ search fails to solve it within 50,000,000 nodes.

In Figures 3.7 and 3.8, the mate found by pn search is also the intended solution to the problem. Since the solutions contained many forcing moves (leaving the defender few moves), pn search performed very well. $\alpha\beta$ search performed very poorly because the solutions were very deep (75 and 49 ply, respectively). As we will see in subsection 3.6.2, in some cases, the duty of playing the most-forcing moves imposed by pn search may lead to excessive departures from the optimal solution.

3.6.2 Weaknesses of pn search

This subsection discusses three weaknesses of pn search: (1) the inability to find good, non-forcing moves, (2) the inability to find the shortest mate, and (3) the inability to deal with transpositions.

Non-forcing moves

In many mating problems, the attacker delivers check on most moves, thus restricting the options of the defender. In some cases, however, the attacker plays a non-forcing move, after which almost any move by the defender leads to the same decisive attack. Since the mobility of the opponent is increased by such a non-forcing move, pn search prefers first to investigate those variations in which the defender is most confined.

Hence, if the only solution requires one or more non-forcing moves, pn search will not perform as well as it will when a mate exists with forcing moves only. We note here that its preference is not merely for checking moves (which are forcing moves in human parlance), but it must, by its algorithm, prefer the most-forcing checks. A similar problem is recognized by Schaeffer (1989a, 1990) when using conspiracy-number search as a tactical analyzer.

As a measure of the difficulty of a position for pn search caused by non-forcing moves, we propose considering the number of different variations within the solution. We present a sample position where pn search performed worse than $\alpha\beta$ search. The existence of non-forcing moves proved a significant factor in degrading its performance. Problem 14 of Reinfeld (1958) (Figure 3.9) is a mate in four moves consisting of 49 variations. After **1. ♕xh7+ ♔f8**, the best move is the non-forcing move **2. ♗f6**, threatening the unavoidable **2. ..., ♕g7 mate**. Proof-number search solves this mate in 324,542 nodes, whereas $\alpha\beta$ search only needs 127,519 nodes.

We conclude that in positions where the solution requires non-forcing moves, pn search is at a disadvantage. The three positions not solved by pn search (K60,

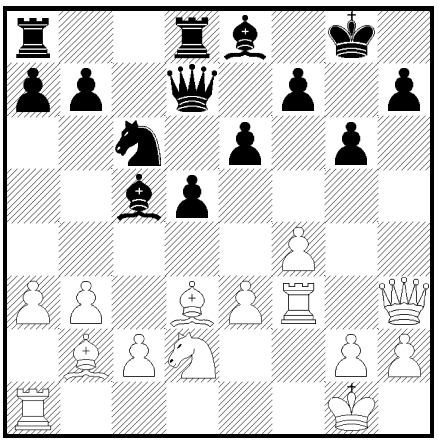


Figure 3.9: Problem 14 of *Win at Chess* (WTM).

K284 and R105) have solutions with non-forcing moves. It is even worse, since the first moves of both solutions are non-forcing, making it impossible for pn search to find the solutions within 1,000,000 nodes.

Mate length

As stated before, pn search is indifferent to the depth of the search, being governed only by the defender's number of options. As a consequence, pn search finds mates in over 100 moves, while optimal ones exist in fewer than ten moves. The position shown in Figure 3.10 is problem 150 of Howard (1961). It shows an example of pn search finding a mate in 114 moves while an optimal mate of four moves exists. The intended solution reads **1.** ♔**e4** and now either **1.** ..., **fxe6 2. f7 e5 3. f8=**♗ ♔**g8 4.** ♘**f6** mate, or **1.** ..., ♔**g8 2. exf7+** ♔**h7 3. f8=**♘**+** ♔**g8 4. f7** mate.

As a solution we suggest initializations of the proof and disproof numbers different from the ones proposed above, specifically with the initial values depending on the depths in the search tree. This may solve the problems of the apparently aimless and certainly long paths to mate.

Transpositions

A third weakness encountered when using pn search is the inability of dealing with transpositions. Assume that an identical subvariation occurs as six separate subtrees within one variation tree. Then, the number of variations to be solved increases by a factor of six. The amount of search to be performed, however, increases by a factor of far more than six. Since, by the rules of combining proof and disproof numbers in

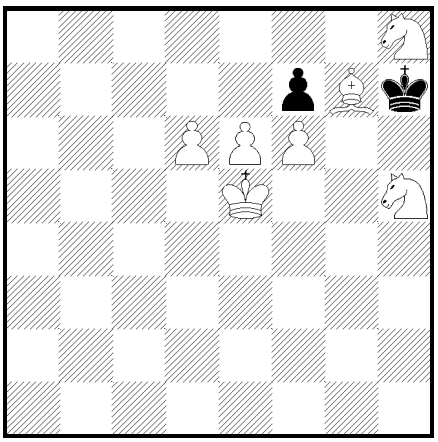


Figure 3.10: Problem 150 of *The Enjoyment of Chess Problems* (WTM).

AND/OR trees, the difficulty of each subtree is propagated upwards sixfold, pn search may well be led to the investigation of other subtrees. If these subtrees fail to deliver a mate pn search will, at long last, arrive at the correct branch in another subtree leading to mate.

As an example we provide problem 213 of Reinfeld (1958) (Figure 3.11). The intended solution starts with the moves **1. ♖xh7+ ♔xh7 2. ♕h5+ ♔g8 3. ♖xg7+ ♔xg7 4. ♗h6+ ♔h7 5. ♗g5+ ♔g7 6. ♕h6+ ♔f7 7. ♕f6+ ♔g8 8. ♕g6+ ♔h8**, reaching the position of Figure 3.12. In the solution tree this position occurs six times, depending on Black's defence at moves 4, 5 and 6. The proof number of this position will be high because the distance-to-mate from that position is still considerable. Upward propagation will expand the proof number six-fold. The resultant high proof number provides an obstacle which pn-search was unable to overcome.

3.7 Chapter conclusions

In this chapter we have described experiments comparing pn search with $\alpha\beta$ search. Pn search has been presented as a best-first search technique easy to implement and uniquely attuned to finding mates in chess. Beyond recognizing mates, stalemates, and drawn positions, no chess-specific knowledge is required. When a mate exists within its horizon, this technique consistently outperforms conventional techniques in terms of nodes visited, except when the solution relies on the presence of non-forcing moves, transpositions, or on providing the shortest mate.

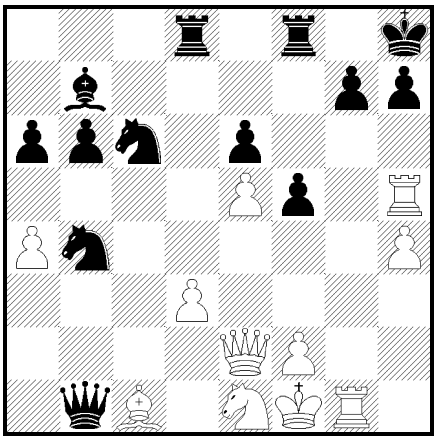


Figure 3.11: Problem 213 of *Win at Chess* (WTM).

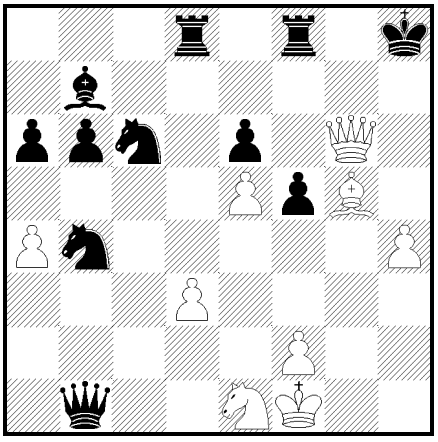


Figure 3.12: Six-fold transposition in problem 213 of *Win at Chess* (WTM).

Chapter 4

The pn^2-search algorithm

One of the drawbacks of proof-number search (pn search) is that the whole search tree has to be stored in memory. Since computers are fast, the search tree grows quickly, causing the memory to be filled up completely. When the memory is full, the search process has to be terminated prematurely. Consequently, no solution will be found. For reducing memory usage, Allis *et al.* (1994) suggest two techniques which reduce the size of a generated search tree: the DeleteSolvedSubtrees technique and the DeleteLeastProving technique. The first technique removes all nodes which are solved. The technique is not very successful in searches which fail to determine the root value. The second technique removes parts of the tree least likely needed in the search. In this chapter we introduce a third technique which increases the information of the nodes in the tree in order to guide the search in a better way, thereby finding a solution more quickly. By this method we attempt to obtain more insight into the second problem statement of this thesis: which methods exist for best-first search to reduce the need for memory by increasing the search, thereby gaining more knowledge per node?

Section 4.1 introduces the pn^2-search algorithm. Details concerning this algorithm are discussed in Section 4.2. Section 4.3 presents the experiments. The results of the experiments are listed in Section 4.4. Section 4.5 states the conclusions.

4.1 Pn search with small memory: pn^2 search

Gaining more knowledge per node searched can be realized by using a better evaluation function at the leaves. One way of doing this is to use a *search process* at the leaves to obtain a more accurate evaluation. This method is used by other researchers as well. Berliner (1979) already used this idea in the B* algorithm, in which a shallow $\alpha\beta$ search evaluates the leaves. Pijls and De Bruin (1994) described the RSEARCH algorithm, in which certain leaves (the so-called *pseudo-terminals*) are evaluated by doing another RSEARCH. Recently, Baum and Smith (1997) reported on their Bayesian model of searching game trees: a two-stage Bayesian search is

performed in which an outer search is called by the inner search as its evaluation function. For the *pn^2-search* algorithm, introduced here, we also use this idea. It is briefly mentioned by Allis (1994), but so far no thorough research has been done on pn^2 search.

Pn^2 search is a search process consisting of two levels of pn search. The first-level search builds a tree in the same way as the standard pn-search algorithm for trees, as described in Section 3.1. However, the evaluation of the most-proving node is not performed by an evaluation function, but by a second-level pn search. The most-proving node of the first-level search tree acts as the root of the second-level search tree. The leaves in the second-level search are evaluated in the standard way, i.e., by an evaluation function returning one of the values *true*, *false*, and *unknown*. The leaf values are backed-up as usual leading to an evaluation of the most-proving node of the first-level pn search with more knowledge (acquired by using the second-level pn search) than in the standard way. After termination of the second-level pn search, the second-level tree is disposed of and the first-level search tree is updated using the new proof and disproof numbers of the most-proving node.

For pn^2 search the same tree-creation variants exist as in standard pn search: *immediate evaluation* and *delayed evaluation* (see Section 3.1). For the first-level search, the evaluation of a leaf takes much time, since it is a (pn-)search process itself. Therefore, it is efficient to use the *delayed-evaluation* variant for the first-level pn search. A limited set of experiments has shown that the delayed-evaluation variant indeed performs better for the first-level pn search than the immediate-evaluation variant. For the second-level search it is efficient to use the *immediate-evaluation* variant of the pn-search algorithm for trees, because the second-level pn search uses a fast evaluation function.

If the evaluation by the second-level pn search yields *unknown*, the most-proving node of the first-level search should be expanded, because delayed evaluation is used. However, this node has just been expanded by the second-level pn search. Hence, after completion of the second-level pn search, the children of the root of the second-level search tree (the most-proving node of the first-level search tree) are preserved, but the subtrees of the children are removed. In this way, whenever a most-proving node evaluates to *unknown*, it has already been expanded by the second-level search. If the evaluation by the second-level is *true* or *false* (solving the most-proving node of the first-level search) the second-level search tree is removed completely.

The following important question arises: how many nodes should the second-level pn search use for the evaluation of the most-proving node of the first-level pn search? An attempt to answer this question is made in the next sections.

4.2 The size of the second-level pn search

In this section we investigate how many nodes the second-level pn search should use for the evaluation of the most-proving node of the first-level search. It is not advisable for this number to be large, when the first-level pn search is still small,

because the evaluation of the most-proving nodes then proportionally consumes too much time. Hence, the size of the second-level search tree should be in relation to the size of the first-level search tree.

Allis suggested making this size *equal* to the size of the first-level search tree. From this position he made two statements which are paraphrased below.

1. A search resulting in a first-level tree of size N has searched approximately $\frac{1}{2} \times N^2$ nodes in the second-level search. This can be shown easily by investigating successive steps in the search process. First, the root is evaluated using a second-level search of one node. Then, the root is expanded, and the first new node is evaluated using a second-level search of at least two nodes (depending on the number of children of the root), *etc.* A first-level tree of size N has therefore searched at most $\sum_{i=1}^{N} i = \frac{N \times (N+1)}{2}$ nodes in the second-level search, which, for big N, is approximately equal to $\frac{1}{2} \times N^2$.

2. The memory requirements during the creation of a first-level tree of size N are $2 \times N$ nodes. This is trivial, since the size of the second-level search is set equal to the size of the first-level tree (being N). Therefore, at most $N + N$ nodes are needed to search a first-level tree with N nodes.

A new idea

Allis' suggestion has the disadvantage that relatively easy problems will take much longer to be solved than with standard pn search (see the first statement above). Therefore, we introduce the following idea: start searching with the standard pn-search algorithm; only when it appears that the solution will not be found, start using a second-level search with *growing* size. In this case, solutions of easy problems will still be found fast, and solutions of more difficult problems may also be found because of the increase in directing knowledge since the second-level search tree grows. In conclusion, we suggest that the size of the second-level search tree is some fraction (between 0 and 1) of the size of the first-level search tree. This fraction should preferably start small, and grow larger as the size of the first-level search tree increases.

Let $f(x)$ be a function that determines the fraction, x being the size (i.e., the number of nodes) of the first-level search tree. A standard model for the desired type of growth of the second-level search tree is the logistic-growth model (Berkey, 1988). From this model we adopt the following function

$$f(x) = \frac{1}{1 + e^{(a-x)/b}} \tag{4.1}$$

with two parameters a and b, both strictly positive. The parameter a determines the transition point of the function: as soon as the size of the first-level search tree reaches a nodes, the second-level search uses half the size of the first-level search tree (the larger a, the later this occurs). Parameter b determines the S-shape in the

function (the larger b, the more stretched the S-shape is). We note that the pn^2-search algorithm as suggested by Allis (1994) is a special case of this function: the size of the second-level search tree is equal to the size of the first-level search tree when both parameter a and parameter b become small, because then the fraction function approaches $f(x) = 1$. When a becomes large and b becomes small the fraction function approaches $f(x) = 0$, which means that standard pn search is used[1].

Preliminary experiments revealed that it is advisable to choose the value of parameter a in the order of magnitude of the maximum number of nodes. If parameter a is chosen too small, the transition point moves too far to the left side of the graph (see Figure 4.1), meaning that easy problems will not be found fast any more. If parameter a is chosen too large, the transition point moves too far to the right side of the graph, meaning that too few nodes are used for the second-level pn search, which does not improve the directing knowledge. In this case, the resulting pn^2-search algorithm will have the same drawback as the standard pn-search algorithm, viz. the memory will be filled before a solution is found. Parameter b may have any positive value.

The fraction function exemplified

Figure 4.1 presents four sample functions (with different parameters a and b), illustrating the functions given by Equation 4.1, together with the function $f(x) = 1$. The x-axis shows the number of nodes in the first-level search tree (in thousands). The y-axis shows the corresponding values of the function $f(x)$. Since the pn^2 algorithm will be used when the amount of memory available is low, we assume that no more than 300,000 (300K) nodes fit in memory. Therefore, the range of the x-axis is chosen from 0 to 300K nodes.

The figure shows that when parameter a increases (in this case from 100K to 150K), the transition point moves to the right (compare [4] and [1]). Further, when parameter b increases, the S-shape becomes more stretched (compare [3] and [1]). If b is relatively large, the S-shape may even disappear (compare [2] and [1]).

The theoretical size of the second-level tree

The sizes of the corresponding second-level searches for the five functions of Figure 4.1 are shown in Figure 4.2. The x-axis again shows the number of nodes in the first-level search tree (in thousands). The y-axis shows the size of the corresponding second-level search tree, given by $x \times f(x)$.

From the figure it follows that the size of the second-level search tree grows with increasing size of the first-level search tree. When parameter a increases, the growth of the second-level search tree starts at a later point (compare [4] and [1]). Further, when parameter b increases, the growth of the second-level search tree starts at an earlier point, but the increase becomes slower (compare [3], [1] and [2]).

[1] We note that in our implementation it results in the delayed-evaluation variant of standard pn search.

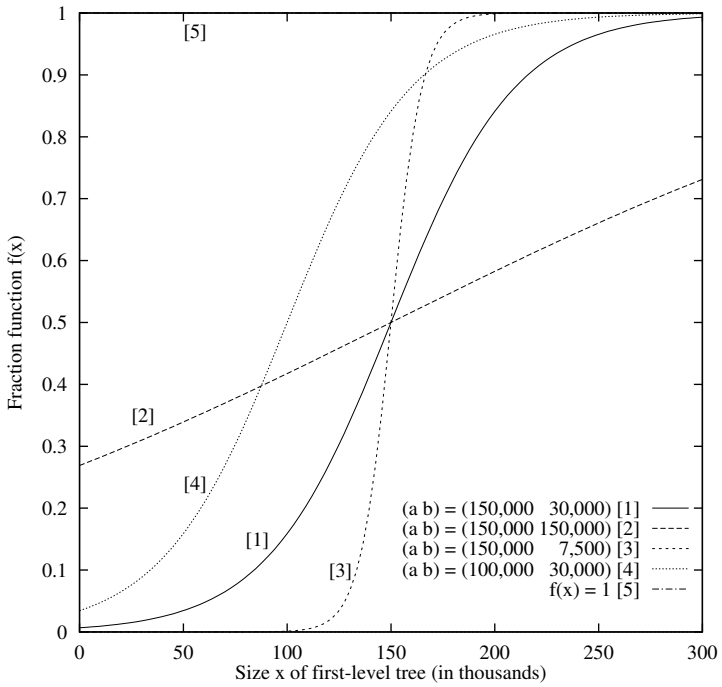


Figure 4.1: The fraction function $f(x)$.

The practical size of the second-level tree

In practice, the sizes of the second-level searches are bounded by the maximum number of nodes that fit into memory (in our case 300K). For instance, if the first-level search tree contains 225K nodes and function $f(x) = 1$ is used, then the second-level search tree should search 225K nodes as well. However, the maximum number of nodes that fit into memory is 300K. Therefore, in this case the second-level search tree has a maximum size of 75K. As soon as this size is reached, the second-level search is terminated, the second-level tree is disposed of, and the first-level search continues. When the first-level tree has reached a size of 300K nodes, no memory is left for the second-level search. In this case, the complete search is terminated and it is indicated that no solution is found.

If these memory bounds are taken into account, the five functions given in Figure 4.2 transform into the functions illustrated in Figure 4.3. The axes are equal to the axes in Figure 4.2

The figure shows that the sizes of the second-level searches are bounded by 300K$-x$.

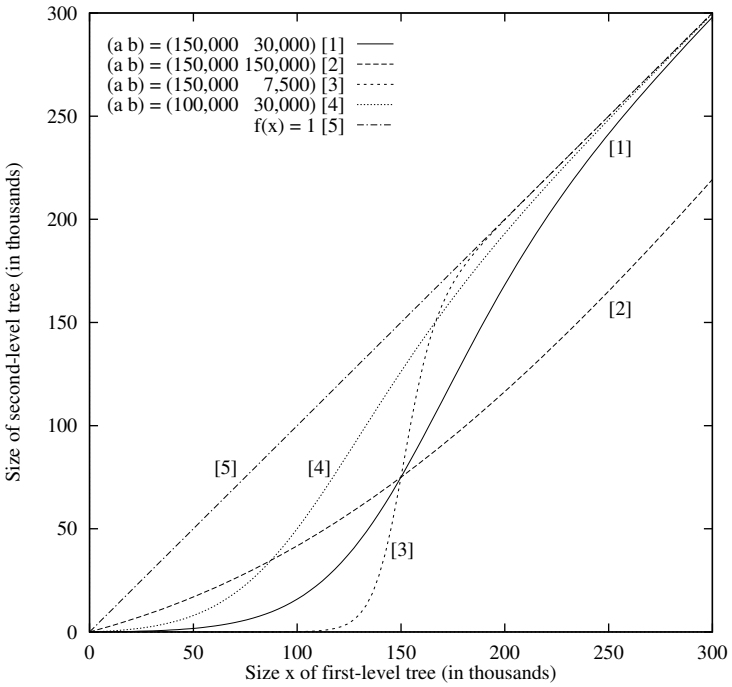


Figure 4.2: The theoretical size of the second-level search.

4.3 Experiments

In this section we describe three series of experiments. They are performed to investigate the behaviour of the pn^2-search algorithm using function $f(x)$ with different parameters a and b. In the first series of experiments we examine the effect of varying parameter b, while keeping parameter a constant. Then, in the second series of experiments we examine the effect of varying parameter a, while keeping parameter b constant. Finally, in the third series of experiments we examine the effect of varying the ratio $\frac{b}{a}$. The idea of examining this ratio stems from the following observations. If a increases, the transition point of the fraction function $f(x)$ shifts to the right. This means that a large first-level tree in memory does not have much information per leaf, because small second-level searches are used for the evaluations. If the first-level tree contains many nodes, the problem to be solved may be a difficult problem, and more directing knowledge may be needed to solve the problem. Therefore, it is then advisable to increase the directing knowledge by also increasing parameter b. Analogously, if a decreases, the transition point of the fraction function $f(x)$ moves to the left in the graph. This means that a small first-level tree in memory already contains much information per node, because relatively large second-level searches are used for the evaluations. In order to limit this overhead it can be wise to reduce

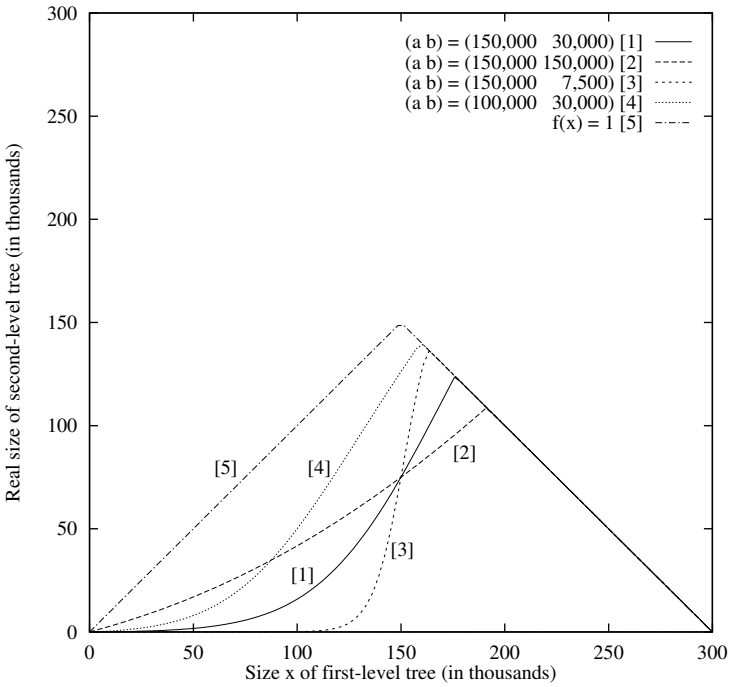


Figure 4.3: The practical size of the second-level search.

the initial size of the second-level search tree, which is taken care of by decreasing parameter b. In both cases the ratio roughly remains equal.

The experiments are performed with second-level searches of size $x \times f(x)$, with $f(x)$ given in Equation 4.1, and with the special case that the second-level search size is x. Further, the maximum number of nodes to be held in memory is set to 300K. The value of parameter a ranges from 75K ($\frac{1}{4}$ of the maximum number of nodes) to 750K ($2\frac{1}{2}$ times the maximum number of nodes). The value of parameter b ranges from 3,750 to 750K. These values were found by trial and error.

The test set for the experiments is a large subset (108 positions) of the set used in the pn-search experiments (see Section 3.4). The positions not tested are K8, K40, K44, K78, K195, K209, K210, K217, and K220[2].

4.4 Results

In this section we discuss the most important results of the experiments mentioned in the previous section. The complete results can be found in Appendix F. We mention

[2] These positions were not solved within 1,000,000 nodes in the previous chapter, and we did not expect that they could be solved within the experimental bounds of this chapter.

that in almost all cases pn^2 search solves all 108 test positions, contrary to pn search for trees (cf. Chapter 3). Therefore, we use as a measure the total amount of nodes searched (i.e., including both first-level and second-level searches) over all 108 test positions.

Results of the first series

Figure 4.4 shows the results of the first series of experiments (varying parameter b, while keeping parameter a constant). Parameter a takes values of 75K, 150K, 300K, 450K, 600K, and 750K, and for each value of a parameter b takes values of 15K, 30K, 60K, 90K, 120K, 150K, 180K, 210K, and 240K. The results of the experiments with function $f(x) = 1$ are shown for comparison. The x-axis shows the value of parameter b (in thousands), and the y-axis shows the total number of nodes searched (in millions).

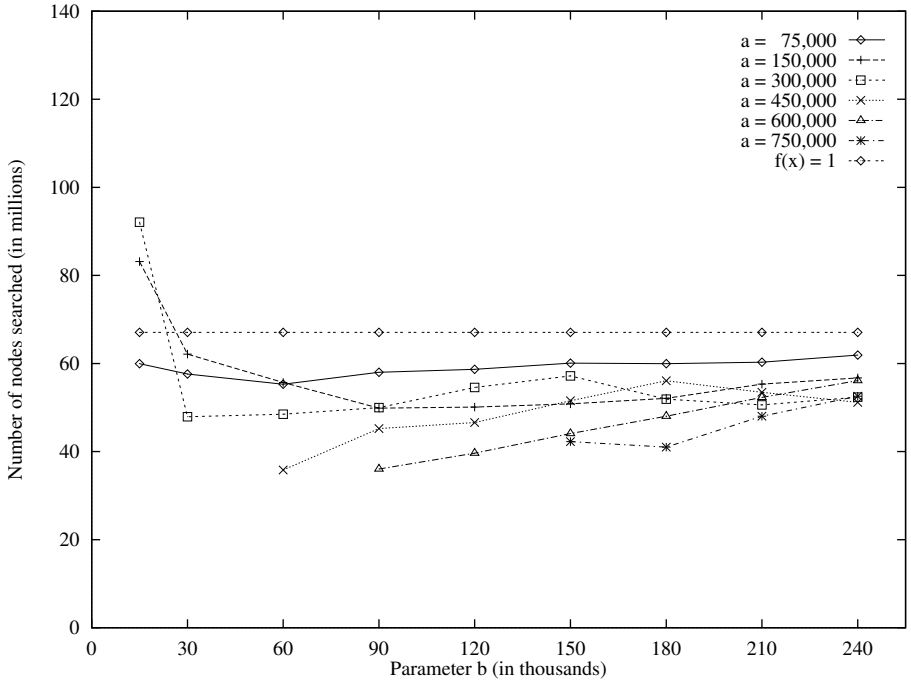


Figure 4.4: The pn^2 results with fixed parameter a.

We note that in the experiments with (a b) equal to (450K 15K), (450K 30K), (600K 15K), (600K 30K), (600K 60K), (750K 15K), (750K 30K), (750K 60K), and (750K 90K) not all 108 test positions are solved. Therefore, these points are not shown in the figure. The number of positions solved in these cases is 88, 104, 87, 91, 106, 87, 87, 99, and 107, respectively. In the positions not solved, the first-level

search tree contained 300K nodes without finding a solution. In these cases the nodes in the first-level search tree do not have sufficient information to direct the search, because the transition point lies too far to the right in the graph (large parameter a).

Results of the second series

The results of the second series of experiments (varying parameter a, while keeping parameter b constant) are shown in Figure 4.5. The x-axis shows the value of parameter a (in thousands). Further explanation for Figure 4.5 is analogous to that for Figure 4.4.

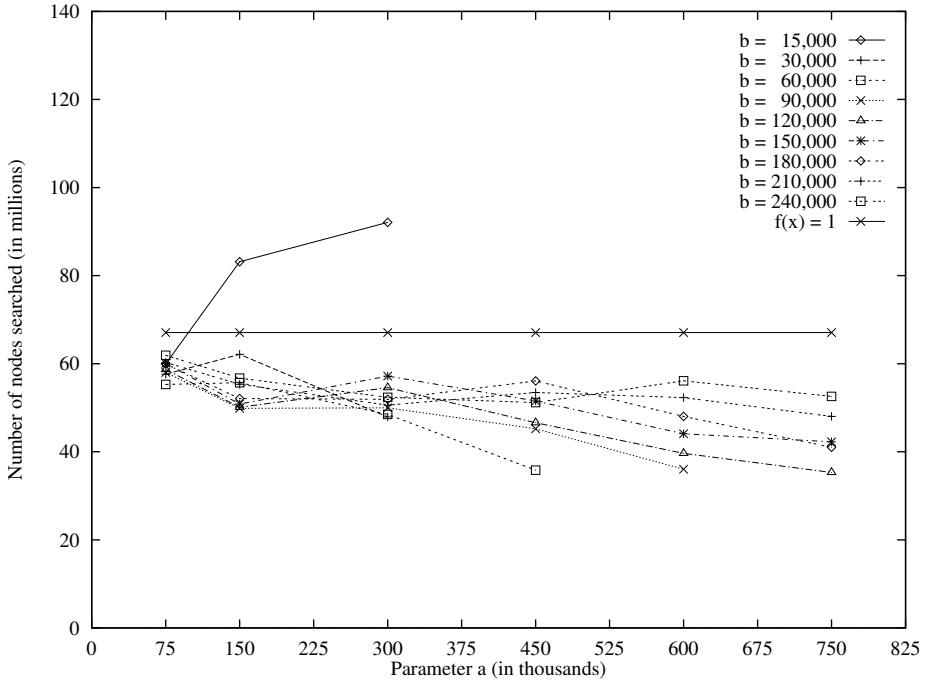


Figure 4.5: The pn^2 results with fixed parameter b.

From these two series of experiments we provisionally conclude that a small parameter b is to be preferred in terms of the number of nodes searched. However, we note that there is a risk of choosing b too small, in which case pn^2 search will not always find a solution. Also, it shows that a large parameter a is to be preferred. Further, the proper use of fraction function $f(x)$ given by the logistic-growth model is significantly better than the function $f(x) = 1$.

Results of the third series

In figure 4.6 the results of the third series of experiments are given (fixed ratio $\frac{b}{a}$). The ratio takes the values 0.05, 0.1, 0.2, 0.4, 0.6, 0.8 and 1.0. Again, the results of the experiments with function $f(x) = 1$ are shown for comparison. We note that in the experiments with (a b) equal to (450K 22.5K), (600K 30K), (600K 60K), (750K 37.5K), and (750K 75K) not all test positions are solved. Therefore, these points are not shown in the figure. The number of positions solved in these cases is 98, 91, 106, 87, and 104, respectively.

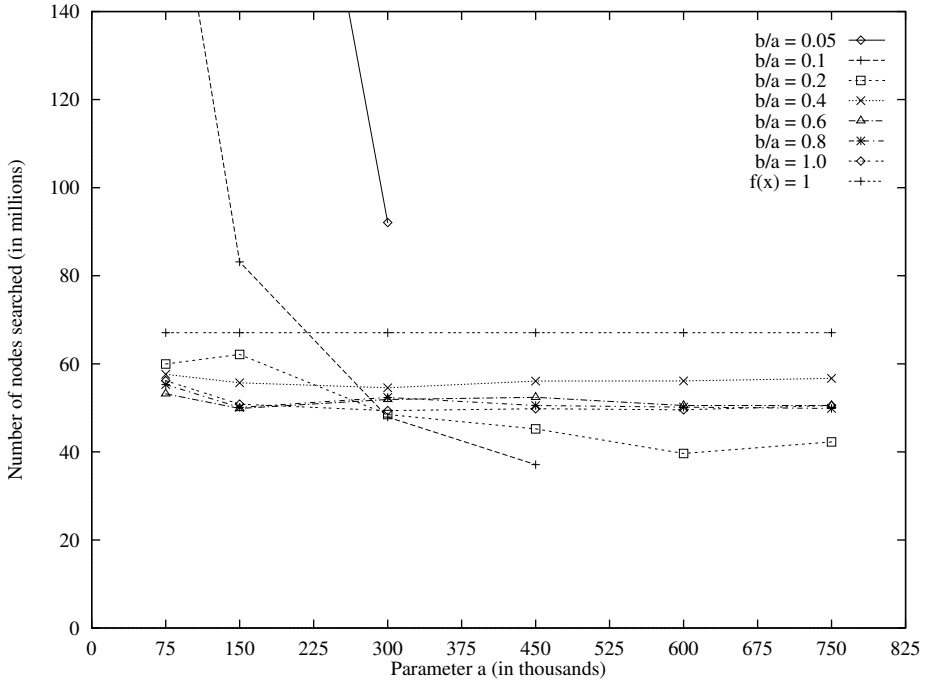


Figure 4.6: The pn^2 results with fixed ratio $\frac{b}{a}$.

From this graph we conclude that, with a ratio of more than 0.1, the results seem to be fairly independent of the choice of the parameters.

4.5 Chapter conclusions

For the pn^2 algorithm we conclude that the use of the function $f(x) = 1$ works well, since it solves all test positions unlike standard pn search. However, the use of fraction function $f(x)$ as given by Equation 4.1, gives significantly better results.

If the goal of the search is to find a solution as quickly as possible, it is recommended to take the fraction function with large parameter a and small parameter b. The disadvantage is that sometimes solvable positions will not be solved because the standard pn search takes too long, filling the memory with nodes not containing sufficient directing knowledge.

If the goal of the search is to solve any solvable position, it is wiser to choose parameters a and b, such that $\frac{b}{a}$ is sufficiently large, i.e., at least more than 0.1. The best performance on the test set is obtained by choosing (a b) equal to (600K 80K) (cf. Appendix F). All test positions are then solved within about 35 million nodes.

Of the 108 test positions, 92 were solved by both the immediate-evaluation variant of standard pn search and the best version of pn^2 search (a = 600K and b = 80K). For these 92 positions the number of nodes searched by pn^2 search (5,856,337) is about twice the number of nodes searched by standard pn search (2,974,602). In our view this is an affordable price for the advantage of the larger probability of finding mates (in this case amounting to an additional 18% solved positions). Further, for these 92 positions pn^2 search used at maximum about 240K (first-level and second-level) nodes in memory.

The results of the experiments in this chapter show that the pn^2-search algorithm is an adequate method for reducing the need for memory in the standard pn-search algorithm. This is accomplished by gaining more knowledge per node through increasing the search: leaves are evaluated using a second-level pn search. The use of the growth-function $f(x)$ proposed here gives significantly better results than the naïve implementation of pn^2 search (effectively using $f(x) = 1$).

Chapter 5

The graph-history-interaction problem

This chapter is an updated and abridged version of

1. Breuker D.M., Herik H.J. van den, Allis L.V., and Uiterwijk J.W.H.M. (1997a). A Solution to the GHI Problem for Best-First Search. *Proceedings of the Ninth Dutch Conference on Artificial Intelligence* (eds. K. van Marcke and W. Daelemans), pp. 457–468. University of Antwerp, Antwerp, Belgium, and
2. Breuker D.M., Herik H.J. van den, Allis L.V., and Uiterwijk J.W.H.M. (1998a). A Solution to the GHI Problem for Best-First Search. Submitted as journal publication. Also published (1997) as Technical Report CS 97-02, Universiteit Maastricht, Maastricht, The Netherlands.

5.1 The history of a position

In a search tree, it may happen that identical nodes are encountered at different places. If these so-called *transpositions* are not recognized, the search algorithm unnecessarily expands identical subtrees. Therefore, it is profitable to recognize transpositions and to ensure that for each set of identical nodes, only one subtree is expanded.

In computer-chess programs using a *depth-first* search algorithm, this idea is realized by storing the result of a node's investigation in a transposition table. For details, see Section 2.3. If an identical node is encountered in the search process, the result is retrieved from the transposition table and used without further investigation.

If a (selective) *best-first* search algorithm (which usually stores the whole search tree in memory) is used, the search tree is converted into a search graph, by joining identical nodes into one node, thereby merging the subtrees.

These common ways of dealing with transpositions contain an important flaw: determining whether *nodes* are identical is not the same as determining whether the *search states* represented by the nodes are identical (cf. Section 2.1). For two reasons, the path leading to a node cannot be ignored. First, the history of a node may partly determine the *legitimacy of a move*. For instance, in chess, castling rights are not only determined by the position of the pieces on the board, but also by the knowledge that in the position under investigation the King and Rook have not moved previously. Second, the history of a node may play a role in determining the *value of a node*. For instance, a position may be declared a draw by its three-fold repetition or by the so-called k-move rule (Kažić *et al.*, 1985).

We refer to the first problem as the *move-generation problem*, and to the second problem as the *evaluation problem*. The combination of these two problems is referred to as the *graph-history-interaction* (GHI) problem (cf. Palay, 1985; Campbell, 1985).

The GHI problem is a noteworthy problem not only in chess but in the field of game playing in general. Its applicability extends though to all domains where the history of states is important. To mention just one example: in job-shop scheduling problems the costs of a task may be dependent on the tasks done so far, e.g., the cost of preparing a machine for performing some process depends on the state left after the previous process.

A possible solution to the GHI problem is to include in all nodes the status of the relevant properties of the history of the node, i.e., the properties which may influence either the move generation or the evaluation of the node. A disadvantage of such a solution is that too many properties may be relevant, resulting in the need for storing large amounts of extra information in each node. For chess, we can distinguish four relevant properties of the history of a position (the first two being relevant for the move-generation problem, and the last two for the evaluation problem):

1. the castling rights (Kingside and Queenside for both players),
2. the *en-passant* capturing rights,
3. the number of moves played without a capture or a pawn move, and
4. the set of all positions played on the path leading to this node.

The first two properties can be included in each node, without much overhead. The third property can be included in each node, but will reduce the frequency of transpositions drastically. The inclusion of the fourth property is necessary to determine whether a draw by three-fold repetition has been encountered. Unfortunately, it would require too much overhead. As a result, in most chess programs, the first two properties are included in a node, while the last two are not.

Depending on which properties are included in a node, the probability of two nodes being identical will be reduced. If not all relevant properties are included and transpositions are used, it is possible that incorrect conclusions are drawn from the transpositions (cf. Section 2.2). Campbell (1985) mentioned that, contrary to best-first search (which he calls selective search), in depth-first search the GHI problem occurs less frequently.

In this chapter we deal with the third problem statement: is it possible to give a solution for the GHI problem for best-first search? A solution to the GHI problem for best-first search is presented with only a few relevant properties included in a node. In Section 5.2 an example of the GHI problem is given. Previous work on the GHI problem is discussed in Section 5.3. In Section 5.4 the general solution to the GHI problem for best-first search is described. A formalized description and the pseudo-code for the implementation in pn search is given in Section 5.5. Section 5.6 lists experiments with the new algorithm. It is compared to three other pn-search variants. The results are presented in Section 5.7. Finally, Section 5.8 provides conclusions.

5.2 An example of the GHI problem

Figure 5.1 shows a pawn endgame position, taken from Campbell (1985), where the GHI problem can occur. White (to move) has achieved a potentially won position. However, we show that it is possible to evaluate this position incorrectly as a draw. In this chapter we assume that a single repetition of positions evaluates to a draw, in contrast with the FIDE ruling which stipulates that the same position must occur three times.

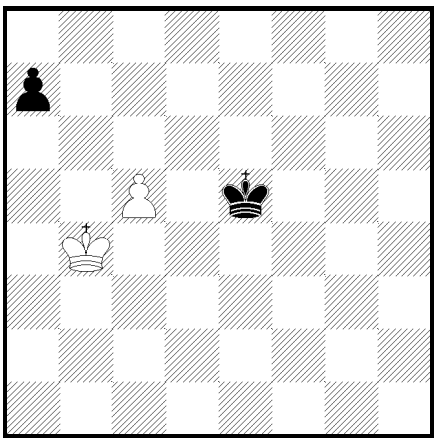


Figure 5.1: A pawn endgame (WTM).

In Figure 5.2 a relevant part of the search tree is depicted. After the move sequence **1.** ♔**b5? ♔e6? 2.** ♔**a6? ♔d5 3.** ♔**b5 ♔e6** the position after move 1 is repeated (node E), and evaluated as a draw. Since White does not have any better alternative on the third move, the position after **2.** ♔**a6** (node H) is evaluated as a draw. Backing up this draw leads to the incorrect conclusion that node A evaluates to a draw. However, after the winning move sequence **1.** ♔**a5! ♔e6 2.** ♔**a6!** the

same position (node H) is reached, which is now evaluated as a win after **2.** ..., ♔**d5 3.** ♔**b5** ♔**e6 4.** ♔**c6!** (node G). Backing up this win leads to the correct conclusion that node A evaluates to a win.

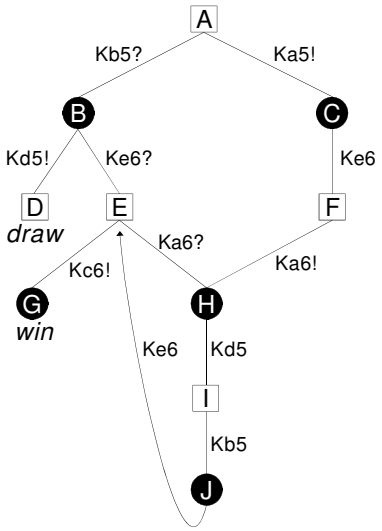


Figure 5.2: The GHI problem in the pawn endgame.

An example of the general case is given in Figure 5.3. It shows an AND/OR search tree with identical positions[1]. The values of the leaves (given in italics) are seen from the OR player's point of view. The values given next to the nodes are back-up values. We note that the GHI problem can occur in any type of AND/OR tree. However, to keep the example as clear as possible we have chosen to show the example for a minimax game tree.

The terminal nodes E and G are a win for the OR player, and the terminal nodes C and F are evaluated as a draw because of the repetition of positions. Propagating the evaluation values of the terminal nodes through the search tree results in a win at the root. When making use of transpositions, every node should occur only once in the tree. Assume that a parent generates its children and that one of its children already exists in the tree. Then a connecting edge from the parent to the existing node is made. This transforms the search tree into a Directed Cyclic Graph (DCG) (Figure 5.4).

In this DCG it is difficult to determine unambiguously the value of node F due to the GHI problem. The value of this node is dependent on the path leading to it. Following the path A–B–C–F, child C of node F is a repetition and hence F is

[1] In games such as chess, a repetition of positions is impossible after only two ply (node C in the left subtree of node B and node F in the subtree of node D). Our example disregards this characteristic for simplicity's sake.

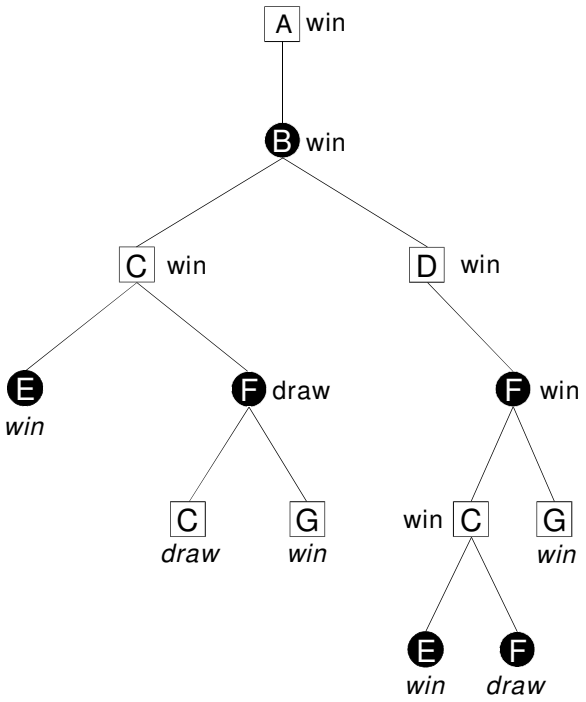


Figure 5.3: A search tree with repetitions.

evaluated as a draw, but following the path A–B–D–F, child C is not a repetition and is not evaluated as a draw. Thus, in the DCG, node F has two different values. Hence, in this example it is not possible to determine the value of root A, since in the first mentioned case it is a draw, and in the second case it is a win, due to the values of E and G.

5.3 A review of previous work

Although several authors have mentioned the GHI problem, so far no solution to this problem has been described. Only provisional ideas have been given. Below, we review the five most important ideas[2].

Palay (1985) first identified the GHI problem. He suggested two "solutions": (1) refrain from using graphs, and (2) recognize when the GHI problem occurs and handle accordingly. The first "solution" (apart from not being a real solution, it merely ignores the problem) had as a drawback that large portions of the graph now

[2] Berliner and McConnell (1996) suggested the use of conditional values as an idea to solve the GHI problem. They promised details in a forthcoming paper.

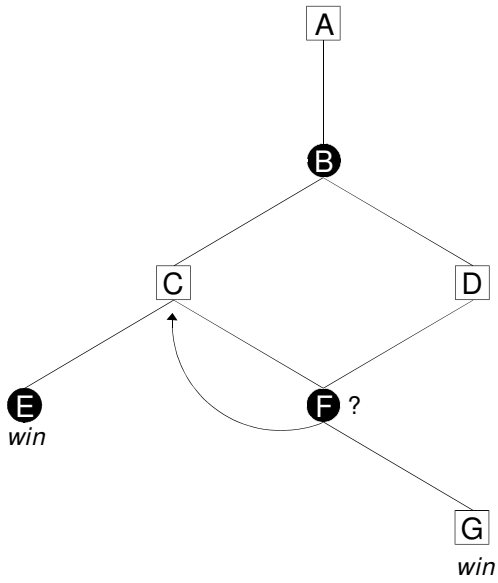


Figure 5.4: The DCG corresponding with the tree of Figure 5.3.

would be duplicated every time a duplicate node occurred, wasting a large amount of time and memory. The second solution worked as follows. When the positions suffering from the GHI problem were recognized, the path from the repetition node upwards to the ancestor with multiple parents was split into separate paths. He did not implement this strategy, since he conjectured that such positions only occurred occasionally (the GHI problem occurred in three out of 300 test positions). A disadvantage of this solution is that the recognition of positions suffering from the GHI problem is not straightforward.

Another idea for a solution originates from Thompson (Campbell, 1985). While building a tactical analyzer, Thompson (1995) used a DCG representation. He saw it suffering from the GHI problem. He cured the problem by taking into account the history of the node to be expanded. The value of this node was then, if necessary, corrected for its history. The newly-generated children were evaluated by doing $\alpha\beta$ searches, yet neglecting their history. As a consequence, the only history errors could occur at the leaves. These errors were corrected as soon as such a leaf was expanded, but it could happen that the expansion of a node was suppressed due to the error.

Campbell (1985) discussed the GHI problem thoroughly, applying it to depth-first search only. The key in avoiding most occurrences of the GHI problem appears to be iterative deepening. Some problems (called "draw-first") can be overcome[3]. However,

[3] In the draw-first case node F in Figure 5.4 is first reached through path A–B–C–F (and the

other problems, which he called "draw-last" could not be solved by his approach[4]. Finally, he remarked that "the GHI problems occur much more frequently in selective search programs, and require some solution in order to achieve reasonably general performance. Both Palay's and Thompson's approaches seem to be acceptable." We conclude that Campbell gave a partial solution for depth-first search, and no solution for best-first search.

Baum and Smith (1995) stumbled on the GHI problem when implementing their best-first search algorithm BPIP (Best Play for Imperfect Players). Baum and Smith completely store the DCG in memory and grow it by using "gulps". In each gulp a fraction of the most interesting leaves is expanded. For each parent-child edge e a subset S(e) was defined as the intersection of *all* ancestor nodes and *all* descendant nodes of edge e. A DCG was claimed to be legitimate (i.e., no nodes have to be split) if and only if, for all children C with more than one parent P, S(e_{PC}) is independent of P. Their solution was as follows. Each time a new leaf was created three possibilities were distinguished: (1) if the leaf was a repetition it was evaluated as a draw, else (2) if a duplicate node existed in the graph, these two nodes were merged on the condition that the resultant DCG was legitimate, else (3) the node was evaluated normally. After leaf expansion it was exhaustively investigated whether every node C with multiple parents passed the S(e) test. If not, such a node C was split into several nodes C', C'', ..., with distinct subsets S(e_{PC}). Then, the subtrees of the newly-created nodes had to be rebuilt and re-evaluated. Baum and Smith gave this idea as a solution to the GHI problem without the support of an implementation. Moreover they remarked that "Implementation in a low storage algorithm would probably be too costly". We believe that the overhead introduced by our idea, described in the next section, is much less than the overhead introduced by the idea of Baum and Smith.

Schijf *et al.* (1994) investigated the problem in the context of pn search (Allis *et al.*, 1994). They examined the problem in Directed Acyclic Graphs (DAGs) and DCGs separately. They noted that, when the pn-search algorithm for trees is used in DAGs, the proof and disproof numbers are not necessarily correctly computed, and the most-proving node is not always found. Schijf (1993) proved that the most-proving node always exists in a DAG. Furthermore, he formulated an algorithm for DAGs that correctly determines the most-proving node. However, this algorithm is only of theoretical importance, since it has an unfavourable time-and-memory complexity. Therefore, a practical algorithm was developed. Surprisingly, only two minor modifications to the pn-search algorithm for trees are needed for a practical algorithm for DAGs. The first modification is that instead of updating only *one* parent, *all* parents of a node have to be updated. The second modification is that when a child is generated, it has to be checked whether this node is a

value of node F is based on child C being a repetition) and later in the search node F is reached through path A–B–D–F and the previous value of node F is used.

[4] In the draw-last case node F in Figure 5.4 is first reached through path A–B–D–F (and the value of node F is based on child C being *no* repetition) and later in the search node F is reached through path A–B–C–F and the previous value of node F is used.

transposition (i.e., if it was generated earlier). If this is the case, the parent has to be connected to this node that has already been generated. Schijf *et al.* (1994) note that this algorithm contains two flaws. First, the proof and disproof numbers do not represent the cardinality in the smallest proof and disproof set, but these numbers are upper bounds to the real proof and disproof numbers. Second, the node selected by the function SelectMostProvingNode is not always equal to a most-proving node. However, it still holds that if the node chosen is proved, the proof number of the root decreases, whereas if this node is disproved, the disproof number of the root decreases. In either case the proof or disproof number may decrease by more than unity, as a result of the transpositions present. This algorithm has been tested on tic-tac-toe (Schijf, 1993). The DAG algorithm uses considerably fewer nodes (viz. a factor of five) to prove the game-theoretic value of tic-tac-toe. For the problem of applying pn search to a DCG, Schijf *et al.* (1994) give a time-and-memory-efficient algorithm, which, however, sometimes inaccurately evaluates nodes as a draw by repetition. They remark that, as a consequence, their algorithm is sometimes unable to find the goal, even though it should have found it.

5.4 BTA: an enhanced DCG algorithm

In this section we describe a new algorithm (denoted BTA: Base-Twin Algorithm) for solving the GHI problem for best-first search. The algorithm had been developed in a joint effort with Victor Allis. Its correctness has been proven experimentally. A formal proof is beyond the scope of this research. The description given below provides a clarity of reasoning, which in our opinion, is sufficiently convincing in its own.

The BTA algorithm is based on the distinction of two types of nodes, termed *base nodes* and *twin nodes*. The purpose of these types is to distinguish between identical positions with different history. Although it was known in the DCG algorithm described by Schijf *et al.* (1994) that nodes sometimes *may* be incorrectly evaluated as a draw, their algorithm was unable to note *when* this occurs. We have devised an alternative in which a sufficient set of relevant properties for correct evaluation is recorded. We have chosen to include in a node only a small number of relevant properties. The reasons for not including *all* relevant properties are:

- some properties are only relevant for a *small* number of nodes,
- the more properties that are included, the lower the frequency of transpositions, and
- some properties require too much overhead and/or take up too much space when included in a node.

The move-generation problem (cf. Section 5.1) can easily be solved by including the relevant properties (in chess these are the castling rights and the *en-passant* capturing rights) into each node. Hence, only the evaluation problem (cf. Section 5.1)

needs to be solved. We have chosen to describe the solution of repetition of positions, since repetition of positions occurs in many search problems, and the k-move rule is a special rule which seldomly shows up in practice. As mentioned before, we assume that a single repetition of positions results in a draw.

Our representation of a DCG

Basically the GHI problem occurs because the search tree is transformed into a DCG by merging nodes representing the same position, but having a different history. To avoid such an undesired coalescence, we propose an enhanced representation of a DCG. In the graph we distinguish two types of nodes: *base* nodes and *twin* nodes. After a node is generated, it is looked up in the graph by using a pointer-based table. If it does not exist, it is marked as a *base node*. If it exists, it is marked as a *twin node*, and a pointer to its base node is created. Thus, any twin node points to its base node, but a base node does not point to any of its twin nodes. Only base nodes can be expanded. The difference with the "standard implementation" of a DCG is that if two or more nodes are represented by the same position (ignoring history) they are not merged into one node. However, their subtree is generated only once. In general, a twin node may have a value different from its base node, although they represent the same position.

Figure 5.5 exhibits our implementation of the DCG given in Figure 5.4 (assuming that the position corresponding with node F is first generated as child of node C and only later as child of node D). Nodes in upper-case are base nodes, nodes in lower-case are twin nodes. The dashed arrows are pointers from twin nodes to base nodes. The problem mentioned in Figure 5.4 can now be handled by assigning separate values to nodes F and f, and to C and c, depending on the paths leading to the corresponding positions.

The BTA algorithm as solution

As stated before, encountering a repetition of positions in node p does not mean that the repetition signals a *real* draw (defined as the inevitability of a repetition of positions under optimal play). To handle the distinction, we introduce the new concept of *possible-draw*. Node p is marked as a *possible-draw* if a node is a repetition of a node P in the search path. (Whether a possible draw also is a real draw depends on the history.) Then the depth of node P in the search path (termed the *possible-draw depth*) is stored in node p.

The BTA algorithm for best-first search consists of three phases. Phase 1 deals with the selection of a node. Phase 2 evaluates the selected node. Phase 3 backs up the new information through the search path. The three phases are repeatedly executed until the search process is terminated.

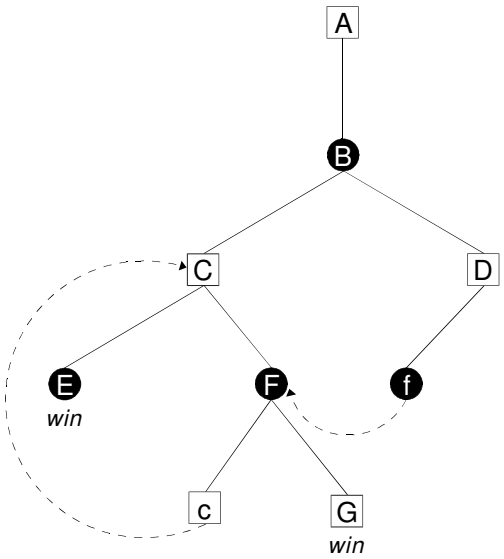


Figure 5.5: Our DCG with base nodes and twin nodes corresponding with the DCG of Figure 5.4.

5.4.1 Phase 1: select the best node

In phase 1 a node is selected for evaluation[5]. This is accomplished in a way similar to the best-first tree algorithm (see Section 2.1). For comparison, a short outline of the tree algorithm is given. First, the root is selected. Next, a best child from the selected node is selected according to the best-first-search criteria. The last step is repeated until (1) a repetition has been encountered (evaluating to a draw), or (2) a leaf has been found.

The selection of a node in the BTA algorithm is as follows. First, the root is selected (for further selection, see below). Then, for each selected node, two cases exist:

1. if a child of the selected node is marked as a *possible-draw*, and the remaining children are either real draws, or marked as *possible-draw*s, then the selected node is marked as a *possible-draw* and the corresponding possible-draw depth is set to the minimum of the possible-draw depths of the children. Subsequently, all possible-draw markings from the children are removed and the parent of the selected node is re-selected for investigation;

2. otherwise, a best child is selected for investigation, ignoring the children which are either real draws, or marked as a *possible-draw*.

[5] We assume that the selection of a node proceeds in a top-down fashion.

Assume that a node at depth d in the search path is marked as a *possible-draw* and the corresponding possible-draw depth is equal to d. This implies that the possible-draw marking of this node is based solely on repetitions of positions *in the subtree of the node* and on real draws. Therefore, the node is a real draw by repetition, independent of the history of the node. Hence, the node is evaluated accordingly.

The selection of a node is repeated until (1) a real draw by repetition has been encountered, or (2) (a twin node of) a base node with known game-theoretic value has been found[6], or (3) a leaf has been found.

The selection of a node in the BTA algorithm is illustrated below. In Figure 5.6 part of a search graph is depicted. The selection starts at the root (node A). Assume the traversal is in a left-to-right order. Then, at a certain point, node c is selected, and marked as a *possible-draw* because it is a repetition of node C at depth two in the search path. See Figure 5.6 (the equal sign represents the possible-draw marking and the subscript two represents the possible-draw depth).

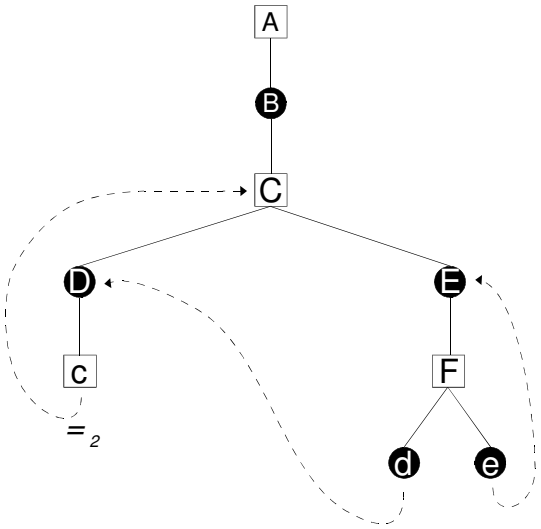


Figure 5.6: Encountering the first repetition c.

After marking node c as a *possible-draw*, the parent of this node (node D) is re-selected and marked as a *possible-draw*, with the same possible-draw depth as node c. Further, the possible-draw marking of node c is removed. After marking node D as a *possible-draw*, its parent C is re-selected. The next best child (not marked as a *possible-draw*) E is selected. Continuing this procedure, at a certain point child d of node F is selected. The child c of twin node d is found by directing the search to

[6] This is possible, because a base node does not point to its twin nodes. If the game-theoretic value of a twin node becomes known, its corresponding base node is evaluated accordingly, but other twin nodes remain unchanged.

the base node D of node d. Node c is (again) marked as a *possible-draw* because it is a repetition of node C at depth two in the search path. See Figure 5.7.

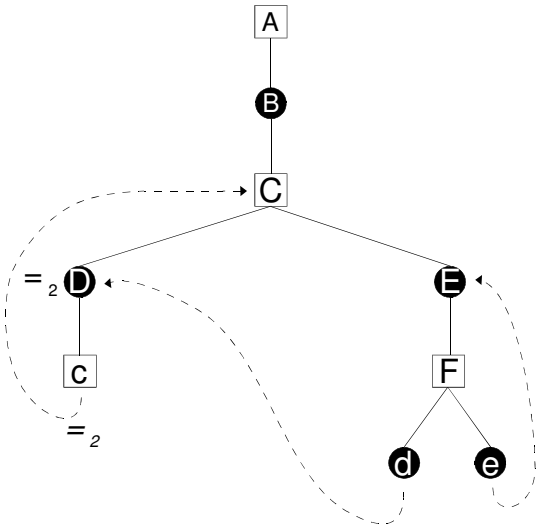


Figure 5.7: Encountering the second repetition c.

After the re-marking of node c as a *possible-draw*, the parent of this node (twin node d) is re-selected and marked as a *possible-draw*, with the same possible-draw depth as node c. Thereafter, the possible-draw marking of node c is removed (for the second time). After marking node d as a *possible-draw*, its parent F is re-selected. The next best child (not marked as a *possible-draw*) e is selected. This node is a repetition of node E at depth three in the search path, and is marked as a *possible-draw*. See Figure 5.8.

After marking node e as a *possible-draw*, the parent of this node (node F) is re-selected. All its children are marked as a *possible-draw*. Therefore, node F is also marked as a *possible-draw*, with a possible-draw depth of two (the minimum of the possible-draw depths of the children). Further, the possible-draw markings of all children are removed. See Figure 5.9.

After marking node F as a *possible-draw*, the parent of this node (node E) is re-selected and marked as a *possible-draw*, with the same possible-draw depth as node F. Subsequently, the possible-draw marking of node F is removed. After marking node E as a *possible-draw*, its parent (node C) is re-selected. However, all its children are marked as a *possible-draw*. Therefore, node C is also marked as a *possible-draw*, with a possible-draw depth of two (the minimum of the possible-draw depths of the children). Again, the possible-draw markings of all children are removed. See Figure 5.10.

Now the selection process finishes, since node C at depth two in the search path

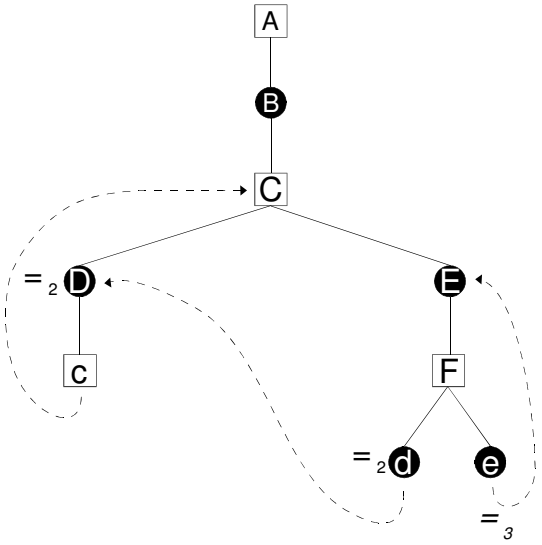


Figure 5.8: Encountering the repetition e.

is marked as a *possible-draw*, **and** its corresponding possible-draw depth is equal to the depth of the node in the search path. This means that *all* continuations from C lead, in one or another way, to repetitions occurring in the subtree of node C. Therefore, node C is evaluated as a real draw by repetition, independent of the history of the node, but on the basis of its potential continuations.

5.4.2 Phase 2: evaluate the best node

In phase 2 the selected node (say node P) is evaluated. For comparison, again a short outline of the tree algorithm is given. The evaluation of node P is dependent on the condition under which phase 1 has terminated.

1. If node P is a repetition, it is evaluated as a draw.

2. If node P is a leaf, it is expanded, the children are evaluated and node P is evaluated using the evaluation values of the children.

For the evaluation of node P in the BTA algorithm three cases are distinguished.

1. If node P is a real draw by repetition, it is evaluated as a draw. The corresponding base node (if existing) is also evaluated as a draw.

2. If node P is a twin node and its corresponding base node is a terminal node, node P becomes a terminal node as well and is evaluated as such.

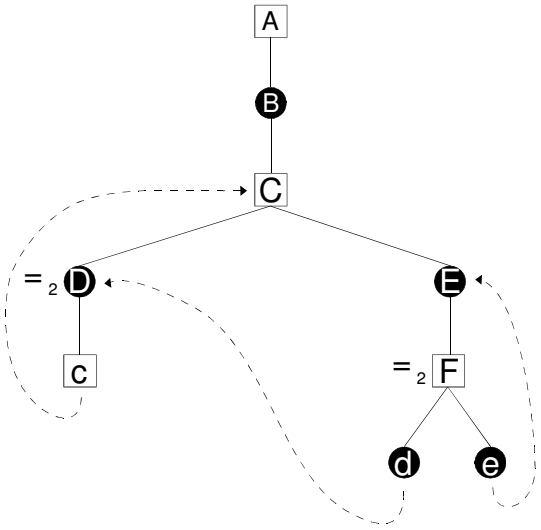


Figure 5.9: Marking node F as a *possible-draw*.

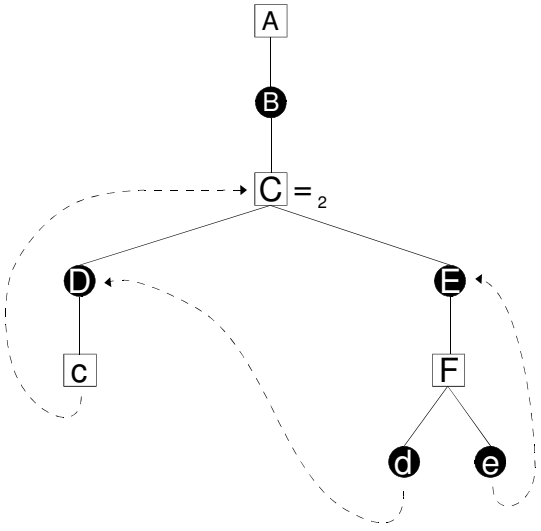


Figure 5.10: Marking node C as a *possible-draw*.

3. If node P is a leaf, it is expanded, the children are evaluated, and node P is evaluated using the evaluation values of the children.

5.4.3 Phase 3: back up the new information

In phase 3 the value of the selected node is updated to the root[7] and all *possible-draw* markings are removed. In contrast to the tree algorithm, in the BTA updating process nodes marked as a *possible-draw* may occur. The back-up value of a node is determined by using only the evaluation values of children not marked as a *possible-draw*. Thus, the children marked as a *possible-draw* are ignored, because in the next iteration the search could be mistakenly directed to one of these children, whereas this child was a repetition in the current path, not giving any new information. After establishing the back-up value of a node, the *possible-draw* markings of the children are removed.

5.5 The pseudo-code of the BTA algorithm

In this section an implementation of the BTA algorithm in pn search (see Chapter 3) is given. An explanation following the three phases of Section 5.4 provides details on the seven relevant pn-search procedures and functions. We will make use of several properties of pn search, in order to simplify and accelerate the general BTA algorithm. For chess, The goal of pn search is finding a mate. A loss and a real draw are in this respect equivalent (i.e., they are no win). Hence, two types of nodes with a known game-theoretic value exist: proved nodes (*win*) and disproved nodes (*no win possible*). A proved or disproved node is called a solved node.

5.5.1 Phase 1: select the most-proving node

Phase 1 of the algorithm deals with the selection of a (best) node for evaluation. This node is termed the most-proving node. In Figure 5.11 the main BTA pn-search algorithm is shown. The only parameter of the procedure is root, being the root of the search tree. The BTA algorithm resembles the tree algorithm described in Section 3.2, a difference being that procedure UpdateAncestors is called with the *parent* of the most-proving node as the parameter instead of the most-proving node itself, since the most-proving node already has been evaluated in procedure ExpandNode.

The procedures Evaluate and SetProofAndDisproofNumbers and the function ResourcesAvailable are identical to the same procedures and function in the standard tree algorithm (see Figure 3.2), and not detailed here. The function SelectMostProvingNode finds a most-proving node, according to certain conditions. The function is given in Figure 5.12. The only parameter of the function is node, being the root of the (sub)tree where the most-proving node is located.

The function starts to examine whether the node under investigation (say node P) is a twin node. If so, then the investigation proceeds with the associated base node.

[7] In a DCG there can exist more than one path from a node to the root. However, only the path along which the node was selected is taken into account. Other paths, if any, may be updated after other selection processes.

```
procedure BTAProofNumberSearch( root )
    Evaluate( root )
    SetProofAndDisproofNumbers( root )
    root.expanded := false
    root.depth := 0

    while root.proof≠0 and root.disproof≠0 and
            ResourcesAvailable() do begin
        mostProvingNode := SelectMostProvingNode( root )
        ExpandNode( mostProvingNode )
        UpdateAncestors( mostProvingNode.parent, root )
    end

    if root.proof=0 then root.value := true
    elseif root.disproof=0 then root.value := false
    else root.value := unknown /* resources exhausted */
end /* BTAProofNumberSearch */
```

Figure 5.11: The BTA pn-search algorithm for DCGs.

If node P has been solved (case 1), node P is returned, because the graph has to be backed up using this new information.

If node P has not been solved, it is examined whether node P is a repetition in the current path (case 2). If so, it is marked as a *possible-draw*. Its ancestor transposition node in the current path is looked up, and the pdDepth (possible-draw depth) of the node becomes equal to the depth in the search path of the ancestor node[8]. Since it is not useful to examine a repetition node further, the selection of the most-proving node is directed to the parent of node P.

If node P has not been solved and is not a repetition in the current path, it is checked whether node P is a leaf (case 3). If so, node P is the most-proving node which has to be expanded, and node P is returned.

Otherwise (case 4), a best child is selected by the function SelectBestChild, to be discussed later. If no best child was found, it means that every child is either solved (proved in case of an AND node, and disproved in case of an OR node) or is marked as a *possible-draw*. If any of the children is marked as a *possible-draw*, the node P is marked as a *possible-draw* as well. The pdDepth of the node is set to the minimum of the children's pdDepths and the markings of *all* children are removed, *etc.* See Section 5.4.

In Figure 5.13 the function SelectBestChild is listed. The function has three parameters. The first parameter (node) is the parent from which a best child will be

[8] The variable pdDepth will act as an indicator of the lowest level in the tree at which there are nodes having repetition nodes in their subtrees.

```
function SelectMostProvingNode( node )
    if NodeHasBaseNode( node ) then baseNode := BaseNode( node )
    else baseNode := node
    /* 1: Base node has been solved */
    if baseNode.proof=0 or baseNode.disproof=0 then return node
    elseif Repetition( node ) then begin /* 2: Repetition of position */
        MarkAsPossibleDraw( node )
        ancestorNode := FindEqualAncestorNode( node )
        node.pdDepth := ancestorNode.depth
        return SelectMostProvingNode( node.parent )
    end elseif not baseNode.expanded then /* 3: Leaf */
        return node
    else begin /* 4: Internal node; look for child */
        bestChild := SelectBestChild( node, baseNode, pdPresent )
        if bestChild=NULL then begin
            if pdPresent then begin
                MarkAsPossibleDraw( node )
                node.pdDepth := ∞
                for i:=1 to baseNode.numberOfChildren do begin
                    if PossibleDrawSet( baseNode.child[ i ] ) then
                        if baseNode.child[ i ].pdDepth<node.pdDepth then
                            node.pdDepth := baseNode.child[ i ].pdDepth
                        UnMarkAsPossibleDraw( baseNode.child[ i ] )
                end
                if node.depth=node.pdDepth then return node
                else return SelectMostProvingNode( node.parent )
            end else begin /* All children are solved, so choose any one */
                baseNode.proof := baseNode.child[ 1 ].proof
                baseNode.disproof := baseNode.child[ 1 ].disproof
                return node
            end
        end else begin
            bestChild.depth := node.depth+1
            return SelectMostProvingNode( bestChild )
        end
    end
end /* SelectMostProvingNode */
```

Figure 5.12: The function SelectMostProvingNode.

```
function SelectBestChild( node, baseNode, pdPresent )
    bestChild := NULL
    bestValue := ∞
    pdPresent := false
    if node.type=OR then begin /* OR node */
        for i:=1 to baseNode.numberOfChildren do begin
            if PossibleDrawSet( baseNode.child[ i ] ) then
                pdPresent := true
            elseif baseNode.child[ i ].proof<bestValue then begin
                bestChild := baseNode.child[ i ]
                bestValue := bestChild.proof
            end
        end
    end else begin /* AND node */
        for i:=1 to baseNode.numberOfChildren do begin
            if PossibleDrawSet( baseNode.child[ i ] ) then begin
                pdPresent := true
                break
            end
            if baseNode.child[ i ].disproof<bestValue then begin
                bestChild := baseNode.child[ i ]
                bestValue := bestChild.disproof
            end
        end
    end

    return bestChild
end /* SelectBestChild */
```

Figure 5.13: The function SelectBestChild.

selected. The second parameter (baseNode) is the base node of that parent[9]. Finally, the third parameter (pdPresent, meaning possible draw present) indicates whether one of the children is marked as a *possible-draw*. The parameter pdPresent is initialized by the function SelectBestChild. If the node is an OR node, a child marked as a *possible-draw* will not be selected as best child, since it gains nothing and the goal (win) cannot be reached. A best child (of an OR node) is a child with the lowest proof number. If the node is an AND node, a child marked as a *possible-draw* is a best child, since the player to move in the AND node is satisfied with a repetition (thereby making it impossible for the opponent to reach the goal). Otherwise, a best child (of an AND node) is a child with the lowest disproof number. This best child

[9] We note that if the parent is a base node itself, then the base node is equal to the parent.

is returned. If the best child is either solved or marked as a *possible-draw*, NULL is returned.

5.5.2 Phase 2: evaluate the most-proving node

After the most-proving node has been found, it has to be expanded and evaluated. Phase 2 of the algorithm performs this task. Figure 5.14 provides the procedure ExpandNode. The only parameter is node, being the node to be expanded.

```
procedure ExpandNode( node )
    if NodeHasBaseNode( node ) then baseNode := BaseNode( node )
    else baseNode := node

    if baseNode.proof=0 or baseNode.disproof=0 then begin
        /* 1: base node already solved */
        node.proof := baseNode.proof
        node.disproof := baseNode.disproof
    end elseif PossibleDrawSet( node ) then begin
        /* 2: node has become a real draw */
        node.proof := ∞
        node.disproof := 0
        baseNode.proof := ∞
        baseNode.disproof := 0
    end else begin
        /* 3: node has to be expanded */
        GenerateAllChildren( baseNode )
        for i:=1 to baseNode.numberOfChildren do begin
            Evaluate( baseNode.child[ i ] )
            SetProofAndDisproofNumbers( baseNode.child[ i ] )
            if not NodeHasBaseNode( baseNode.child[ i ] ) then
                baseNode.child[ i ].expanded := false
        end
        SetProofAndDisproofNumbers( baseNode )
        baseNode.expanded := true
        node.proof := baseNode.proof
        node.disproof := baseNode.disproof
    end
end /* ExpandNode */
```

Figure 5.14: The procedure ExpandNode.

The procedure starts establishing the base node of the node[10]. If the base node

[10] We note that if the node is a base node itself, then the base node is equal to the node.

is solved (case 1), the node is evaluated accordingly.

Otherwise, if the node is marked as a *possible-draw* (case 2) (and since it was chosen by function SelectMostProvingNode), it is evaluated as a real draw.

In case 3 the node has to be expanded. All children are generated, and evaluated. If a generated child has no corresponding base node, the attribute expanded is initialized to false; if it has a corresponding base node, the attribute expanded has been initialized before. Then the node itself is initialized by procedure SetProofAndDisproofNumbers.

5.5.3 Phase 3: back up the new information

Phase 3 of the algorithm has as task to back up the evaluation value of the most-proving node. The procedure for updating the values of the nodes in the path is listed in Figure 5.15. The procedure has two parameters. The first parameter (node) is the node to be updated, while the second parameter (root) is the root of the search tree. Depending on the node type, UpdateOrNode (Figure 5.16) or UpdateAndNode (Figure 5.17) is performed.

```
procedure UpdateAncestors( node, root )
    while node≠nil do begin
        if NodeHasBaseNode( node ) then baseNode := BaseNode( node )
        else baseNode := node

        if node.type=OR then UpdateOrNode( node, baseNode )
        else UpdateAndNode( node, baseNode )

        node := node.parent /* parent in current path */
    end
    if PossibleDrawSet( root ) then
        UnMarkAsPossibleDraw( root )
end /* UpdateAncestors */
```

Figure 5.15: The procedure UpdateAncestors.

The parameters of UpdateOrNode are node and baseNode. The algorithm basically is the same as the OR part of the procedure SetProofAndDisproofNumbers. It only differs when a child is marked as a *possible-draw*. In that case, the child is discarded so its value is not used when calculating the back-up value of the node. Then, the *possible-draw* marking of the child is removed. If the node appears to be disproved (since all children are either disproved or marked as a *possible-draw*) and a repetition child exists, the value of the node is calculated by procedure SetProofAndDisproofNumbers. Otherwise, the value has been calculated correctly. If the node has been solved, its base node is initialized accordingly.

```
procedure UpdateOrNode( node, baseNode )
    min := ∞
    sum := 0
    pdPresent := false
    for i:=1 to baseNode.numberOfChildren do begin
        if PossibleDrawSet( baseNode.child[ i ] ) then begin
            pdPresent := true
            proof := ∞
            disproof := 0
            UnMarkAsPossibleDraw( baseNode.child[ i ] )
        end else begin
            proof := baseNode.child[ i ].proof
            disproof := baseNode.child[ i ].disproof
        end
        if proof<min then min := proof
        sum := sum + disproof
    end

    if min=∞ and pdPresent then
        SetProofAndDisproofNumbers( node )
    else begin
        node.proof := min
        node.disproof := sum
    end
    if node.proof=0 or node.disproof=0 then begin /* node solved */
        baseNode.proof := node.proof
        baseNode.disproof := node.disproof
    end
end /* UpdateOrNode */
```

Figure 5.16: The procedure UpdateOrNode.

The two parameters of UpdateAndNode are equal to the parameters of procedure UpdateOrNode. The procedure differs from the AND part of the procedure SetProofAndDisproofNumbers when the node is solved, and hence the value of its base node is evaluated accordingly[11].

```
procedure UpdateAndNode( node, baseNode )
    min := ∞
    sum := 0
    for i:=1 to baseNode.numberOfChildren do begin
        proof := baseNode.child[ i ].proof
        disproof := baseNode.child[ i ].disproof
        sum := sum + proof
        if disproof<min then min := disproof
    end

    node.proof := min
    node.disproof := sum
    if node.proof=0 or node.disproof=0 then begin /* node solved */
        baseNode.proof := node.proof
        baseNode.disproof := node.disproof
    end
end /* UpdateAndNode */
```

Figure 5.17: The procedure UpdateAndNode.

5.6 Experimental set-up

In this section give the experimental set-up for evaluating the BTA pn-search algorithm presented in Section 5.5. The game of chess is used as the test domain. Our BTA algorithm, denoted by *BTA*, is compared with the following three pn-search variants:

1. the standard tree algorithm (see Section 3.2), denoted by *Tree*,
2. a DAG algorithm, developed by Schijf (1993), denoted by *DAG*, and
3. an (incorrect) DCG algorithm, developed by Schijf *et al.* (1994), denoted by *DCG*.

[11] We note that it is impossible for a child of an AND node to be marked as a *possible-draw*, since in that case the search for a most-proving node would have been terminated in an earlier phase, and the parent already would have been marked as a *possible-draw*.

In all implementations, the move ordering is identical. The test set of 117 positions is given in Section 3.4 (see Appendix D). All four algorithms searched for a maximum of 500,000 nodes per test position. After 500,000 nodes the search was terminated and if no solution had been found the problem was marked as not solved[12].

5.7 Results

To verify our solution we have first tested the position given in Figure 5.1[13]. *Tree* finds a solution within 482,306 nodes. *DCG*, ignoring the history of a position, incorrectly states that White cannot win (due to the GHI problem). Our *BTA* does find a solution within 10,694 nodes. This provides evidence that the occurrence of the GHI problem has been correctly handled. *BTA* shows the benefit of being a DCG algorithm, as evidenced by the decrease in number of nodes investigated by a factor of roughly 40 as compared to *Tree*.

	# of pos. solved (out of 117)	Total nodes (96 positions)
Tree	99	4,903,374
DAG	102	3,222,234
DCG	103	2,482,829
BTA	107	2,844,024

Table 5.1: Comparing four pn-search variants.

Thereafter, we have performed the experiments as described in Section 5.6. The outcomes are summarized in Table 5.1. The complete results are listed in Appendix G. The first column shows the four pn-search variants. The number of positions solved by each algorithm is given in the second column. Exactly 96 positions were solved by all four algorithms. *BTA* solves each position which was solved by at least one of the other three algorithms. In the third column the total number of nodes evaluated for the 96 positions are listed. The additional positions solved per algorithm are as follows.

Tree: K208, K215, R281.

DAG: K208, K215, K216, R168, R182, R281.

DCG: K44, K60, K217, K284, R168, R182, R252.

[12] The maximum number of nodes in these pn-search experiments is lower than the corresponding number given in Chapter 3 due to implementation details.

[13] We note that for this problem the goal for White was set to promotion to Queen (without Black being able to capture it on the next ply) instead of mate. Further, the search was restricted to the 5×5 a4–e8 board. This helps to find the solution faster, but does not influence the occurrence of the GHI problem.

BTA: K44, K60, K208, K215, K216, K217, K284, R168, R182, R252, R281.

Neither algorithm: K8, K40, K78, K195, K209, K210, K220, R96, R105, R201.

Obviously, *Tree* investigates the largest number of nodes. The explanation is easy: the algorithm does not recognize transpositions. Further, *DCG* examines the smallest number of nodes: this algorithm sometimes prematurely disproves positions; hence, on the average fewer nodes have to be examined. However, if such a prematurely disproved position does lead to a win and the node is important to the principal variation of the tree, the win can be missed, as happens in the positions K208, K215, K216 and R281. This is already remarked by Schijf *et al.* (1994).

From Table 5.1 it further follows that *BTA* performs best. The four positions which were incorrectly disproved by *DCG* were proved by *BTA*. Compared to the tree algorithm, *BTA* solves eight additional positions and uses only 58% of the number of nodes: a clear improvement. The reduction in nodes compared to *DAG* is still 11.7%. The increase in nodes searched relative to *DCG* (12.7%) is already explained by the unreliability of the latter. We feel that the advantage of the larger number of solutions found heavily outweighs the drawback of the increase in nodes searched. We note that the selection of the most-proving node in *BTA* can be costly in positions with many possible transpositions. However, in these types of positions the reduction in the number of nodes searched is even larger than in "normal" positions.

As a case in point we present Figure 5.18 corresponding with Diagram 216 in Krabbé (1985). It is solved by our BTA algorithm (in 247,686 nodes) and by the DAG algorithm (in 366,336 nodes) and not by the two other algorithms (within 500,000 nodes). Many transpositions (and many repetitions of positions) exist, since after **1. ♖a5+ ♔b8** White has a so-called *zwickmühle* and can position the Bishop anywhere along the a7–g1 diagonal for free. For instance, after **2. ♗a7+ ♔a8 3. ♗b6+ ♔b8** almost the same position with the same player to move has been reached: the Bishop has moved from d4 to b6. At any time White can choose such a manoeuvre. For the chess-playing reader, the solution is **1. ♖a5+! ♔b8 2. ♗a7+ ♔a8 3. ♗c5+! ♔b8 4. ♖b5+ ♔a8 5. ♖e7! ♗f7 6. ♖a5+ ♔b8 7. ♗a7+ ♔a8 8. ♗d4+! ♔b8 9. ♖b5+ ♔a8 10. ♖d7! ♕g5 11. ♖a5+ ♔b8 12. ♗a7+ ♔a8 13. ♗b6+ ♔b8 14. ♗xc7 mate**.

5.8 Chapter conclusions

In this chapter we have given a solution to the GHI problem, resulting in an improved DCG algorithm for pn search, called BTA (Base-Twin Algorithm). It is shown that the restricted version (a4–e8 board) of a well-known position, in which the GHI problem occurs when a naïve DCG algorithm is used, our BTA algorithm finds the correct solution. The results on a test set of 117 positions do not falsify our claim. Despite the additional overhead for recognizing positions suffering from the GHI problem, our BTA algorithm is hardly less efficient than other, not entirely correct DCG algorithms, and finds more solutions.

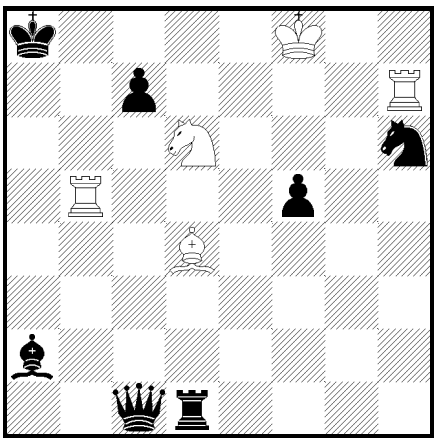


Figure 5.18: Mate in 14 (WTM); (J. Kriheli).

Although our algorithms are confined to pn search, the strategy used is generally applicable to any best-first search algorithm. The only important criterion for application is that a DCG is being built according to the best-first principle (choose some leaf node, expand that node, evaluate the children, and back up the result). We consider the GHI problem in best-first search to be solved. The importance of this statement is that with the increasing availability of computer memory a growing tendency exists to use best-first search algorithms and variants thereof, or best-first fixed-depth algorithms (Plaat *et al.*, 1996), which no longer suffer from the GHI problem.

Our solution to the GHI problem gives an affirmative answer to the third problem statement: is it possible to give a solution for the GHI problem for best-first search? By transforming the search tree into our DCG representation, less memory is needed, since only the roots of equal subtrees are duplicated. Moreover, less search is needed, since the DCG contains fewer nodes than the tree. One disadvantage is the cost of finding a most-proving node. If many transpositions exist in the tree, many *possible draw*s will occur, prolonging the search for a most-proving node. We are convinced that the advantage of solving the GHI problem outweighs this disadvantage. What remains is solving the GHI problem for depth-first search. This will need a different approach, storing additional information in transposition tables rather than in the search tree/graph in memory. However, Campbell (1985) already noted that in depth-first search the frequency of GHI problems is considerably smaller than in best-first search. The solution of the GHI problem for depth-first search therefore seems to be of minor importance for practical use.

Chapter 6

Evaluations and conclusions

In this thesis we have presented research on the balance between memory and search in two-player zero-sum games. As example domains we have used the games of chess and domineering. The trade-off between memory and search led to the formulation of three problem statements. In this chapter the problem statements are re-addressed and evaluated.

6.1 More memory and less search

We investigated whether we can exploit the large amount of memory currently available. The underlying idea is that storing more knowledge into memory may result in a decreasing need for search. A depth-first search algorithm only stores the path from the root to the node under investigation, and hence it uses little memory. However, many depth-first search algorithms use the available memory for keeping a transposition table. The transposition table eliminates the need for search at identical nodes, because the results of previous search processes have been saved in the table. The first problem statement addresses decreasing the need for search by increasing the use of memory.

Problem statement 1: Which methods exist to improve the efficiency of a transposition table?

In Chapter 2 we investigated three methods of improving the efficiency of a transposition table. Irrespective of the size of the transposition table, collisions (cf. subsection 2.4.2) are bound to occur. When a collision occurs, a choice has to be made which of the two positions involved should be preserved in the transposition table. Such a choice is governed by a replacement scheme.

The first method to improve the efficiency of the transposition table is *to improve the replacement scheme*. Experiments have shown that a two-level scheme works significantly better than a traditional one-level scheme. Further, the concept *Big*

(based on the number of nodes of the subtree) works better than the most widely used concept *Deep* (based on the depth of the subtree).

The second (obvious) method of improving the efficiency of a transposition table is *to increase the number of positions in the table*. In most implementations the number of positions usually is a power of two. Hence, increasing the number of positions means doubling the number of positions. However, after a certain transposition-table size has been reached it turns out that doubling the number of positions in the table has a limited benefit. Moreover, doubling the number of positions in the table can cause the table to take up too much memory.

When doubling the number of positions has a limited benefit the memory can be used *to store additional information in an entry*. This is the third method for improving the efficiency of the transposition table. We first have performed experiments to investigate which information is more important to store in a transposition-table entry: the best move in a position, or the score of that move. Experiments show that the score is more important than the move. Further, we have investigated the effect of storing an n-ply PV ($n = 2..5$) in an entry, instead of only the best move (a 1-ply PV). Our results show that storing additional information in an entry is a profitable way of using the available memory, which outperforms the benefit of doubling the number of positions in the table. We believe that this is a fruitful domain for future research (cf. Section 6.4).

6.2 Less memory and more search

We investigated whether we can exploit the increase in computer speed. The underlying idea is that more speed enables more search, thereby acquiring more knowledge, and hence decreasing the need for memory. Best-first search needs a large amount of memory to store the entire search tree. At present computer speeds, the memory available is quickly filled. Since the quality of a best-first search algorithm depends on the quality of the directing knowledge, ways have to be found to use the increase in speed to acquire more knowledge per node, hence also improving the directing knowledge. Consequently, the search process will search the state space more efficiently, reducing the need for memory at the cost of more search. The second problem statement addresses decreasing the need for memory by increasing the use of search.

Problem statement 2: Which methods exist for best-first search to reduce the need for memory by increasing the search, thereby gaining more knowledge per node?

In Chapter 4 we introduced the pn^2-search algorithm. This is a best-first search algorithm (pn search), using a second search (also pn search) as evaluation of a leaf, thereby adding more (directing) knowledge to every node in the search tree. Experiments with this algorithm (listed in section 4.4) show that pn^2 search is a good method of reducing the need for memory by increasing the search. The pn^2-search algorithm uses roughly twice as much search time compared to the traditional

pn-search algorithm, leading to a decrease in the need for memory. A further advantage of the pn^2-search algorithm is that it solves test positions not solvable (due to memory constraints) by a standard pn-search framework.

6.3 Less memory and less search

In the first problem statement we tried to reduce the need for search by increasing the use of memory. Analogously, in the second problem statement we tried to reduce the need for memory by increasing the use of search. An attempt to combine the advantages of both approaches (reducing the need for search *and* reducing the need for memory) is the following. In a search tree it is profitable to recognize transpositions and to ensure that for each set of identical nodes, only one subtree is expanded. If a best-first search algorithm (which stores the whole search tree in memory) is used, the search tree is converted into a search graph, by joining identical nodes into one node. This causes subtrees to be merged, decreasing the need for memory. Since the graph contains fewer nodes than the tree, less searching is needed as well. However, joining identical nodes into one node introduces the so-called *graph-history-interaction* (GHI) problem, since determining whether *nodes* are identical is not the same as determining whether the *search states* represented by the nodes are identical. The third problem statement addresses decreasing the need for memory *and* decreasing the need for search.

Problem statement 3: Is it possible to give a solution for the GHI problem for best-first search?

In Chapter 5 we have given a solution to the GHI problem for best-first search, resulting in a Directed-Cyclic-Graph (DCG) algorithm for pn search, called the BTA (Base-Twin Algorithm) algorithm. This algorithm is based on the distinction of two types of nodes, termed *base nodes* and *twin nodes*. The purpose of these types is to distinguish between equal positions with different history. By transferring the search tree into our implementation of a search DCG, less memory is needed, since only the roots of equal subtrees are duplicated. Furthermore, less search is needed, since the DCG contains fewer nodes than the tree. It is shown that our algorithm is hardly less efficient than other, not entirely correct DCG algorithms in terms of numbers of nodes searched. One drawback of our solution is the cost of finding the node to be expanded, in the case that many transpositions occur. We are convinced that the advantage of solving the GHI problem outweighs this drawback.

6.4 Future research

In this section several recommendations for future research on the trade-off between memory and search are given.

6.4.1 More memory and less search

This subsection provides some ideas for future research on $\alpha\beta$ search in combination with a transposition table. The idea of using an n-ply principal variation in an entry, instead of only the best move (cf. subsection 2.7.3), seems worthy of further investigation. Based on the experiments concerning $\alpha\beta$ search with a transposition table (cf. Chapter 2) it is advised to concentrate on using additional information affecting the number of cut-offs generated by bound values.

A second recommendation is to store the best n moves with their respective values (exact values, upper bounds, or lower bounds) in an entry, instead of only storing the best move.

As a third recommendation it may be worthwhile investigating whether an entry is still effective in the table. To this end we store in a transposition-table entry the last time the position from this entry has been *read* in the search[1], and we use this stamp for the decision what to do when a collision occurs.

The transposition table can also be used to store results of partial game boards. when using partition search (Ginsberg, 1996). After a certain number of moves played in the game of domineering, the board is usually divided into separate (and smaller) regions. The search time will decrease considerably if the results of these regions can be found in the transposition table. In this case it is not sufficient to store only the values of *win* and *loss* in the table, since it has to be known by what margin a player can win a region (Conway, 1976; Berlekamp, 1988).

6.4.2 Less memory and more search

This subsection lists some ideas for future research on modifications of the pn-search algorithm (or other best-first search algorithms), decreasing the need for memory. The fraction function used in Section 4.2 works well. Still, it would be interesting to investigate whether other fraction functions perform even better. After every initialization of a most-proving node in the first-level tree, pn^2 search deletes the second-level tree. If the next most-proving node is one of the children of the previously expanded node, then the second-level tree is recreated. Therefore, it could be advantageous to store the last N second-level trees in a cache to reduce this overhead, a proposal already suggested by Schaeffer (mentioned by Allis, 1994).

The pn^2-search algorithm can be seen as a pn-search algorithm with another pn search for evaluation. Other combinations are worthwhile to be investigated, such as the combination of pn search and $\alpha\beta$ search, leading to two variants: (1) use the pn-search algorithm with $\alpha\beta$ search for evaluation, and (2) use the $\alpha\beta$ algorithm with pn search for evaluation. The first variant can be used e.g., in a chess tactical analyzer: pn search uses $\alpha\beta$ search at the leaves to get a more accurate evaluation. The second variant can be used e.g., in a chess program: a (positional) $\alpha\beta$ search uses pn search at the leaves to check for forced mates.

[1] We note that this is a method different from time stamping, where the last time a position has been *written* into an entry is stored.

6.4.3 Proof-number search

In this subsection we give two recommendations for improving pn search as it is used in the game of chess.

First, pn search often finds a longer mate than the optimal shortest one. If it is desired to urge pn search to find a shorter mate than it does at present, the following two solutions are suggested: (1) after a mate has been found, try searching for a shorter mate by only examining nodes in the search tree at a lower depth than the depth of the shortest mate found so far, or (2) the proof and disproof numbers in the leaves are initialized to values over unity, say at the depth of the node in question in the tree; this deters deep searches and hence long mates.

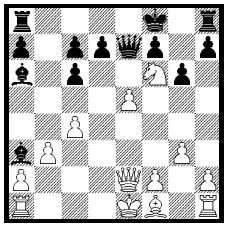
Second, in Chapter 3 it is shown that pn search is a good searcher for mates, especially when the winning variation contains forcing moves. When considering extending pn search to other tactical problems, say as a tactical analyzer for gaining material, a difficulty arises: the condition for suspending search (recognizing the proved or disproved nature of a node) is not easy to formulate. Temporary gains should be discarded, and proved or disproved should hold only when the material gain is permanent. Then and only then the goal is reached and the node should be evaluated to *true*, as is a win in standard pn search. A possible definition, worthwhile testing, is: the gain value of a node is stable if the attacker is to move and has gained at least the material expected. Since this definition of a stable gain is a heuristic, it may be incorrect. To prevent unwanted effects, the variation found by pn search can be checked by an $\alpha\beta$ search. The variation can also be used to sort the moves in the $\alpha\beta$ search, resulting in deeper searches than a standard full-width search, because of the additional cut-offs of pn search.

Appendix A

The chess middle-game test set

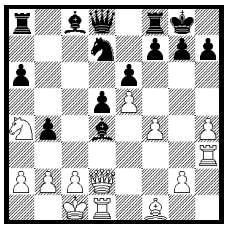
For the transposition-table experiments on chess middle-game positions described in Chapter 2 the following test set has been used.

- The 6 WTM positions from move 15 onwards of the following game:
  Kasparov–Ivanchuk, Amsterdam (round 1) 1994
  1. e4 e5 2. ♘f3 ♘c6 3. d4 exd4 4. ♘xd4 ♘f6 5. ♘xc6 bxc6 6. e5 ♕e7 7. ♕e2 ♘d5 8. c4 ♗a6 9. b3 g6 10. ♗a3 ♕g5 11. g3 ♘c3 12. ♘xc3 ♗xa3 13. ♘e4 ♕e7 14. ♘f6+ ♔f8 15. ♗g2 ♗b4+ 16. ♔f1 ♖d8 17. ♕b2 ♗a3 18. ♕c3 ♗b4 19. ♕b2 ♗a3 20. ♕c3 ♗b4 $\frac{1}{2}$–$\frac{1}{2}$



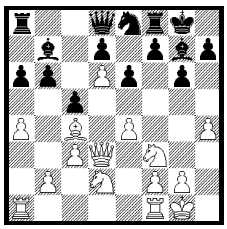
Move 15 (WTM)

- The 18 WTM positions from move 15 onwards of the following game:
  Kasparov–Short, Amsterdam (round 2) 1994
  1. e4 e6 2. d4 d5 3. ♘c3 ♘f6 4. e5 ♘fd7 5. f4 c5 6. ♘f3 ♘c6 7. ♗e3 cxd4 8. ♘xd4 ♗c5 9. ♕d2 0–0 10. 0–0–0 a6 11. h4 ♘xd4 12. ♗xd4 b5 13. ♖h3 b4 14. ♘a4 ♗xd4 15. ♕xd4 f6 16. ♕xb4 fxe5 17. ♕d6 ♕f6 18. f5 ♕h6+ 19. ♔b1 ♖xf5 20. ♖f3 ♖xf3 21. gxf3 ♕f6 22. ♗h3 ♔f7 23. c4 dxc4 24. ♘c3 ♕e7 25. ♕c6 ♖b8 26. ♘e4 ♘b6 27. ♘g5+ ♔g8 28. ♕e4 g6 29. ♕xe5 ♖b7 30. ♖d6 c3 31. ♗xe6+ ♗xe6 32. ♖xe6 1–0



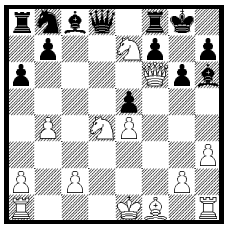
Move 15 (WTM)

- The 20 BTM positions from move 15 up to and including move 34 of the following game:
  Timman–Kasparov, Amsterdam (round 3) 1994
  1. d4 ♘f6 2. ♘f3 g6 3. ♗g5 ♗g7 4. c3 b6 5. ♗xf6 ♗xf6 6. e4 ♗b7 7. ♗d3 c5 8. d5 e6 9. ♗c4 0–0 10. 0–0 ♘a6 11. ♕d3 ♘c7 12. d6 ♘e8 13. ♘bd2 ♗g7 14. h4 a6 15. a4 ♕b8 16. e5 f6 17. h5 fxe5 18. hxg6 h6 19. ♖fe1 ♕xd6 20. ♕xd6 ♘xd6 21. ♘xe5 ♗xe5 22. ♖xe5 ♖f4 23. ♗d3 ♖af8 24. f3 a5 25. ♔f2 ♔g7 26. ♖h5 ♘e8 27. ♔g3 ♘f6 28. ♖e5 ♘d5 29. ♗e4 ♖4f6 30. ♘c4 ♘f4 31. ♗xb7 ♖xg6+ 32. ♔h2 ♖xg2+ 33. ♔h1 d5 34. ♘xb6 ♖b8 35. ♖xe6 ♖xb7 36. ♖d6 ♖g5 37. ♖d1 d4 38. ♘c4 ♔h7 39. ♖e1 ♖h5+ 40. ♔g1 ♖g7+ 0–1



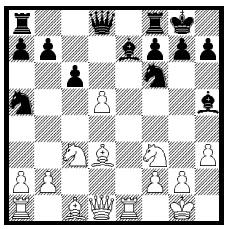
Move 15 (BTM)

- The 24 BTM positions from move 15 onwards of the following game:
  Ivanchuk–Kasparov, Amsterdam (round 4) 1994
  1. e4 c5 2. ♘f3 d6 3. d4 cxd4 4. ♘xd4 ♘f6 5. ♘c3 a6 6. f4 ♕c7 7. ♕f3 g6 8. ♗e3 ♗g7 9. h3 e5 10. fxe5 dxe5 11. ♗h6 ♗xh6 12. ♕xf6 0–0 13. ♘d5 ♕a5 14. b4 ♕d8 15. ♘e7+ ♕xe7 16. ♕xe7 exd4 17. ♗c4 ♘c6 18. ♕c5 ♗e3 19. ♖f1 ♘d8 20. ♖f3 ♗e6 21. ♖xe3 dxe3 22. ♗xe6 ♘xe6 23. ♕xe3 a5 24. b5 ♖ac8 25. 0–0–0 ♖c5 26. ♖d5 b6 27. ♕g3 ♖c7 28. ♕d6 ♖fc8 29. ♖d2 ♖b7 30. g4 ♘c5 31. ♕f6 h6 32. e5 ♖e8 33. h4 ♔h7 34. h5 g5 35. ♖d6 ♖e6 36. ♕d8 ♔g7 37. a3 a4 38. ♔b2 ♖be7 39. ♖xb6 1–0



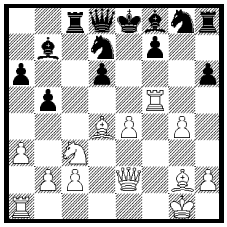
Move 15 (BTM)

- The 15 WTM positions from move 15 up to and including move 29 of the following game:
  Kasparov–Timman, Amsterdam (round 5) 1994
  1. e4 e5 2. ♘f3 ♘f6 3. ♘xe5 d6 4. ♘f3 ♘xe4 5. d4 d5 6. ♗d3 ♘c6 7. 0–0 ♗e7 8. ♖e1 ♗g4 9. c4 ♘f6 10. ♘c3 dxc4 11. ♗xc4 0–0 12. d5 ♘a5 13. ♗d3 c6 14. h3 ♗h5 15. ♖e5 ♗g6 16. ♗g5 ♗d6 17. ♖e2 ♗b4 18. ♗xf6 gxf6 19. ♖c1 ♖c8 20. ♘e4 f5 21. ♘g3 ♕xd5 22. a3 ♗d6 23. ♘xf5 ♖cd8 24. ♖e5 ♗xe5 25. ♘e7+ ♔g7 26. ♘xd5 ♗xb2 27. ♘f4 ♗xd3 28. ♘xd3 ♗xc1 29. ♕xc1 ♖xd3 30. ♕g5+ 1–0



Move 15 (WTM)

- The 11 BTM positions from move 15 onwards of the following game:
  Short–Kasparov, Amsterdam (round 6) 1994
  1. e4 c5 2. ♘c3 e6 3. ♘f3 a6 4. d4 cxd4 5. ♘xd4 d6 6. g4 b5 7. a3 h6 8. ♗g2 ♗b7 9. 0–0 ♘d7 10. f4 ♖c8 11. ♗e3 g5 12. ♕e2 gxf4 13. ♖xf4 e5 14. ♖f5 exd4 15. ♗xd4 ♘e5 16. ♘d5 ♗g7 17. ♖af1 ♖h7 18. ♔h1 ♗h8 19. c3 ♘e7 20. ♗xe5 dxe5 21. ♕f3 ♘xf5 22. ♕xf5 ♖g7 23. ♘f6+ ♔f8 24. ♘d7+ ♔g8 25. ♘f6+ ♔f8 $\frac{1}{2}$–$\frac{1}{2}$



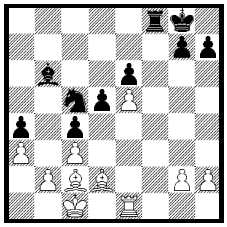
Move 15 (BTM)

Appendix B

The chess endgame test set

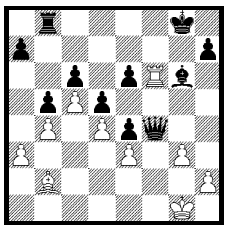
For the transposition-table experiments on chess endgame positions described in Chapter 2 the following test set has been used.

- The 28 WTM positions from move 31 onwards of the following game:
  Gossip–Mason, New York (round 20) 1889
  1. e4 e6 2. d4 d5 3. ♘c3 ♘f6 4. e5 ♘fd7 5. f4 c5 6. dxc5 ♘c6 7. ♘f3 ♗xc5 8. ♘e2 ♕b6 9. c3 ♗f2+ 10. ♔d2 ♕e3+ 11. ♔c2 ♕e4+ 12. ♕d3 ♘c5 13. ♕xe4 ♘xe4 14. ♘ed4 ♗d7 15. ♘xc6 bxc6 16. ♗d3 ♘c5 17. ♗e2 ♘b7 18. ♖f1 ♗b6 19. ♗d2 c5 20. ♗a6 ♖b8 21. ♖ae1 ♘d8 22. ♗d3 a5 23. ♔c1 a4 24. a3 ♘b7 25. f5 c4 26. fxe6 ♗xe6 27. ♗c2 ♘c5 28. ♘d4 0–0 29. ♘xe6 fxe6 30. ♖xf8+ ♖xf8 31. ♗e3 ♘b3+ 32. ♗xb3 ♗xe3 33. ♖xe3 cxb3 34. ♖e2 ♖f4 35. ♔d2 ♔f7 36. ♖e3 ♖f2+ 37. ♖e2 ♖xe2+ 38. ♔xe2 ♔g6 39. ♔e3 ♔f5 40. ♔d4 h5 41. g3 g5 42. h3 h4 43. g4+ ♔f4 44. ♔c5 ♔xe5 45. ♔b4 d4 46. ♔xa4 d3 47. ♔xb3 ♔e4 48. a4 ♔e3 49. a5 d2 50. ♔b4 d1♕ 51. b3 ♕a1 52. c4 ♔d4 53. ♔b5 ♕c3 54. ♔c6 ♕xb3 55. a6 ♕xc4+ 56. ♔b7 ♕b5+ 57. ♔a7 ♔c5 58. ♔a8 ♔c6 0–1



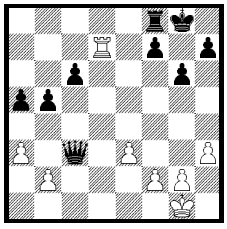
Move 31 (WTM)

- The 21 WTM positions from move 34 onwards of the following game:
  Rabinovich–Romanovsky, Leningrad 1934
  1. c4 ♘f6 2. ♘c3 c6 3. d4 d5 4. ♘f3 ♘e4 5. e3 e6 6. ♗d3 f5 7. ♕c2 ♘d7 8. b3 ♗b4 9. ♗b2 ♕a5 10. ♖c1 0–0 11. 0–0 ♗d6 12. ♘e2 ♕d8 13. ♘e5 ♕h4 14. f3 ♘ec5 15. g3 ♕h6 16. ♘f4 ♘xd3 17. ♘exd3 g5 18. ♘g2 ♘f6 19. ♖ce1 g4 20. fxg4 ♘xg4 21. ♘gf4 ♘f6 22. ♖e2 ♖f7 23. b4 ♘e4 24. ♘c5 ♖b8 25. a3 b6 26. ♘xe4 fxe4 27. ♖ef2 ♗d7 28. c5 ♗xf4 29. ♖xf4 ♖xf4 30. ♖xf4 b5 31. ♕f2 ♗e8 32. ♖f6 ♗g6 33. ♕f4 ♕xf4 34. ♖xf4 h5 35. h3 ♔g7 36. ♗c3 ♗f5 37. g4 hxg4 38. hxg4 ♗g6 39. ♔g2 ♔h6 40. ♖f6 ♖e8 41. ♗e1 ♔g7 42. ♖f1 ♖a8 43. ♔h3 a6 44. ♗g3 ♖h8 45. ♗h4 ♖f8 46. ♖xf8 ♔xf8 47. ♗g3 e5 48. ♗xe5 ♔f7 49. ♔h4 ♔e6 50. ♔g5 ♗e8 51. ♔h6 ♗f7 52. ♔g7 ♗e8 53. g5 ♔f5 54. ♔f8 1–0



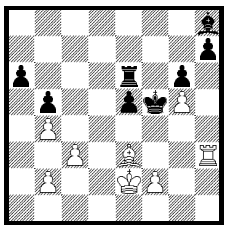
Move 34 (WTM)

- The 17 WTM positions from move 26 onwards of the following game:
  Capablanca–Alekhine, Buenos Aires World Championship (game 5) 1927
  1. d4 d5 2. c4 e6 3. ♘c3 ♘f6 4. ♗g5 ♘bd7 5. e3 c6 6. a3 ♗e7 7. ♘f3 0–0 8. ♗d3 dxc4 9. ♗xc4 ♘d5 10. ♗xe7 ♕xe7 11. ♖c1 ♘xc3 12. ♖xc3 e5 13. dxe5 ♘xe5 14. ♘xe5 ♕xe5 15. 0–0 ♗e6 16. ♗xe6 ♕xe6 17. ♖d3 ♕f6 18. ♕b3 ♕e7 19. ♖fd1 ♖ad8 20. h3 ♖xd3 21. ♖xd3 g6 22. ♕d1 ♕e5 23. ♕d2 a5 24. ♖d7 b5 25. ♕c3 ♕xc3 26. bxc3 ♖c8 27. ♔f1 ♔g7 28. ♖a7 a4 29. c4 ♔f6 30. ♖a5 ♔e6 31. ♔e2 bxc4 32. ♖c5 ♔d6 33. ♖xc4 ♖a8 34. ♖d4+ ♔e6 35. ♔d3 c5 36. ♖h4 h5 37. g4 hxg4 38. ♖xg4 ♔d6 39. ♖f4 f5 40. ♖h4 ♔d5 41. ♔c2 ♖a6 42. ♔c3 $\frac{1}{2}$–$\frac{1}{2}$



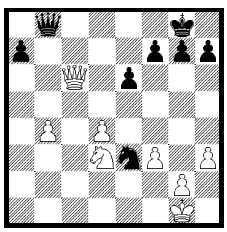
Move 26 (WTM)

- The 17 WTM positions from move 38 onwards of the following game:
  Fischer–Reshevsky, New York US Championship (round 5) 1962
  1. e4 c5 2. ♘f3 d6 3. d4 cxd4 4. ♘xd4 ♘f6 5. ♘c3 a6 6. h3 g6 7. g4 ♗g7 8. g5 ♘h5 9. ♗e2 e5 10. ♘b3 ♘f4 11. ♘d5 ♘xd5 12. ♕xd5 ♘c6 13. ♗g4 ♗xg4 14. hxg4 ♕c8 15. ♕d1 ♘d4 16. c3 ♘xb3 17. axb3 ♕e6 18. ♖a5 f6 19. ♕d5 ♕xd5 20. ♖xd5 ♔d7 21. gxf6 ♗xf6 22. g5 ♗e7 23. ♔e2 ♖af8 24. ♗e3 ♖c8 25. b4 b5 26. ♖dd1 ♔e6 27. ♖a1 ♖c6 28. ♖h3 ♗f8 29. ♖h1 ♖c7 30. ♖h4 d5 31. ♖a1 ♖c6 32. exd5+ ♔xd5 33. ♖d1+ ♔e6 34. ♖d8 ♔f5 35. ♖a8 ♖e6 36. ♖h3 ♗g7 37. ♖xh8 ♗xh8 38. ♖xh7 ♖e8 39. ♖f7+ ♔g4 40. f3+ ♔g3 41. ♔d3 e4+ 42. fxe4 ♖d8+ 43. ♗d4 ♔g4 44. ♖f1 ♗e5 45. ♔e3 ♗c7 46. ♖g1+ ♔h5 47. ♔f3 ♖d7 48. e5 ♖f7+ 49. ♔e4 ♖f5 50. e6 ♗d8 51. ♗f6 ♗xf6 52. gxf6 ♖xf6 53. ♔e5 ♖f2 54. ♖e1 1–0



Move 38 (WTM)

- The 29 WTM positions from move 38 onwards of the following game:
  Lisitsin–Capablanca, Moscow (round 5) 1935
  1. ♘f3 d5 2. c4 c6 3. e3 ♘f6 4. ♘c3 ♗g4 5. cxd5 ♘xd5 6. ♗e2 e6 7. d4 ♘d7 8. 0–0 ♕c7 9. ♗d2 ♗d6 10. ♘e4 ♘7f6 11. ♘xd6+ ♕xd6 12. ♘e5 ♗xe2 13. ♕xe2 0–0 14. ♖c1 ♘b6 15. ♘d3 ♖e8 16. ♖fe1 ♘bd7 17. h3 ♕d5 18. b3 ♕b5 19. ♗c3 ♘d5 20. ♕d2 ♘xc3 21. ♕xc3 ♖ad8 22. a4 ♕b6 23. b4 ♘f6 24. ♕c4 ♘e4 25. a5 ♕c7 26. a6 ♖c8 27. axb7 ♕xb7 28. ♖a1 ♖c7 29. ♖ec1 ♖b8 30. ♕c2 ♕c8 31. ♖a5 ♖b6 32. ♕a4 ♕b8 33. f3 ♘f6 34. ♖c5 ♘d5 35. ♖xc6 ♖cxc6 36. ♖xc6 ♖xc6 37. ♕xc6 ♘xe3 38. ♘c5 ♘d5 39. b5 ♘b6 40. ♘d7 ♕d8 41. ♘xb6 axb6 42. ♕c4 h5 43. ♔f1 g6 44. ♔g1 ♔g7 45. ♔f1 ♕d6 46. ♔g1 ♕f4 47. ♕c3 ♔h7 48. ♔f1 ♕f5 49. ♕c4 ♔g7 50. ♔f2 ♕g5 51. ♕e2 ♔f6 52. ♕b2 ♕d5 53. ♔e3 e5 54. f4 exf4+ 55. ♔xf4 ♔e6 56. h4 f6 57. ♔e3 ♕c4 58. g3 g5 59. hxg5 fxg5 60. ♕h2 ♕b3+ 61. ♔e4 g4 62. ♕e2 ♕xg3 63. ♕c4+ ♔e7 64. ♕c8 ♕f3+ 65. ♔e5 ♕f6+ 66. ♔d5 ♕d6+ 0–1



Move 38 (WTM)

Appendix C

The transposition-table results

This appendix presents the results of the three series of experiments given in Chapter 2. The first series of experiments investigates which replacement scheme performs best. The second series of experiments examines which information is more important to store in a transposition-table entry: the best move in a position, or the score of that move. Finally, the third series of experiments investigates the effect of storing an n-ply PV ($n = 2..5$) in an entry, instead of only the best move (a 1-ply PV).

Comparing replacement schemes

The first series of experiments consists of three parts. First, the 3-ply to 8-ply transposition-table results for the seven replacement schemes (TwoBig1, TwoDeep, Big1, BigAll, Deep, New and Old) on chess middle-game positions are listed in Tables C.1 to C.12. The middle-game figures reported are number of nodes searched in thousands. The 3-ply to 7-ply results are listed with eight table sizes (8K, 16K, 32K, 64K, 128K, 256K, 512K and 1024K). These results are the cumulative results of all six games (94 middle-game positions) given in Appendix A. The 8-ply results are listed with four table sizes (16K, 64K, 256K and 1024K). These results are the cumulative results of the first three games (44 middle-game positions) given in Appendix A. For every ply depth two tables are given: one without time stamping and one with it. In the former case the transposition tables are cleared between moves.

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	684	684	686	686	687	687	688
16K	684	684	683	683	684	684	685
32K	684	684	684	684	684	684	685
64K	684	684	684	684	684	684	684
128K	684	684	684	684	684	684	684
256K	684	684	684	684	684	684	684
512K	684	684	684	684	684	684	684
1024K	684	684	684	684	684	684	684

Table C.1: Replacement-scheme results for the chess middle game (without time stamping, 3-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	660	660	660	660	665	665	661
16K	660	660	660	660	660	660	660
32K	660	660	660	660	660	660	660
64K	660	660	660	660	660	660	660
128K	660	660	660	660	660	660	660
256K	660	660	660	660	660	660	660
512K	660	660	660	660	660	660	660
1024K	660	660	660	660	660	660	660

Table C.2: Replacement-scheme results for the chess middle game (with time stamping, 3-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	2,737	2,774	2,790	2,800	2,816	2,789	2,814
16K	2,757	2,751	2,808	2,807	2,780	2,784	2,788
32K	2,758	2,756	2,754	2,753	2,791	2,778	2,765
64K	2,758	2,758	2,735	2,735	2,763	2,755	2,776
128K	2,757	2,757	2,763	2,763	2,749	2,766	2,756
256K	2,757	2,757	2,745	2,745	2,734	2,730	2,776
512K	2,757	2,757	2,766	2,766	2,761	2,761	2,782
1024K	2,757	2,757	2,767	2,767	2,759	2,759	2,791

Table C.3: Replacement-scheme results for the chess middle game (without time stamping, 4-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	2,706	2,673	2,648	2,705	2,727	2,735	2,739
16K	2,671	2,697	2,698	2,698	2,696	2,644	2,688
32K	2,687	2,703	2,683	2,701	2,661	2,692	2,704
64K	2,682	2,682	2,651	2,659	2,719	2,670	2,674
128K	2,684	2,684	2,633	2,633	2,705	2,714	2,647
256K	2,684	2,684	2,679	2,679	2,681	2,700	2,700
512K	2,684	2,684	2,680	2,680	2,672	2,674	2,693
1024K	2,684	2,684	2,680	2,680	2,673	2,674	2,701

Table C.4: Replacement-scheme results for the chess middle game (with time stamping, 4-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	8,967	9,186	9,185	9,206	9,341	9,312	9,435
16K	8,932	9,004	9,101	9,109	9,221	9,141	9,235
32K	8,988	8,999	9,020	9,079	9,061	9,046	9,169
64K	9,014	9,016	9,000	9,006	9,037	9,089	9,091
128K	8,985	8,975	8,945	8,946	9,024	9,089	8,965
256K	8,993	8,982	9,036	9,038	9,010	8,955	9,052
512K	8,964	8,964	9,003	9,003	8,967	8,968	9,066
1024K	8,964	8,964	8,939	8,939	8,983	8,987	8,986

Table C.5: Replacement-scheme results for the chess middle game (without time stamping, 5-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	8,892	8,815	8,912	8,854	9,093	9,065	9,160
16K	8,755	8,806	8,798	8,833	8,841	8,842	9,005
32K	8,741	8,796	8,844	8,835	8,922	8,829	8,744
64K	8,724	8,718	8,721	8,767	8,779	8,750	8,893
128K	8,755	8,744	8,793	8,855	8,763	8,776	8,801
256K	8,736	8,745	8,802	8,801	8,765	8,723	8,739
512K	8,732	8,732	8,743	8,798	8,703	8,690	8,836
1024K	8,732	8,732	8,736	8,764	8,744	8,746	8,769

Table C.6: Replacement-scheme results for the chess middle game (with time stamping, 5-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	40,619	42,120	42,280	42,694	42,903	45,932	46,581
16K	39,210	39,896	40,305	40,112	40,831	42,864	43,693
32K	38,146	38,930	39,165	39,321	39,172	40,506	41,828
64K	38,088	38,061	38,901	38,919	38,695	39,265	39,751
128K	38,247	37,884	38,798	38,287	38,313	38,635	38,781
256K	37,892	38,059	37,867	38,210	37,983	38,042	38,607
512K	38,036	37,871	37,757	38,178	38,009	38,342	38,269
1024K	37,959	37,790	37,901	37,734	38,454	37,810	38,243

Table C.7: Replacement-scheme results for the chess middle game (without time stamping, 6-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	40,148	41,390	41,846	41,421	42,747	45,249	46,631
16K	38,190	39,367	39,680	39,380	40,780	42,457	43,294
32K	37,187	37,695	38,583	38,344	38,638	40,169	40,462
64K	37,008	36,898	37,764	37,555	37,813	38,940	38,501
128K	36,754	36,959	37,077	37,131	37,004	37,437	37,781
256K	36,854	36,705	36,697	37,053	36,689	38,015	37,529
512K	36,865	36,658	36,446	36,613	36,756	36,901	36,894
1024K	36,914	36,425	36,983	36,265	36,462	36,690	36,447

Table C.8: Replacement-scheme results for the chess middle game (with time stamping, 6-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	162,393	168,136	175,713	177,681	181,147	199,937	205,747
16K	149,363	153,613	162,211	162,578	165,565	181,221	191,766
32K	141,105	143,399	149,651	150,810	153,938	165,531	174,633
64K	136,147	138,046	141,101	142,081	143,767	152,896	158,519
128K	132,571	134,148	136,419	136,996	136,879	142,206	146,668
256K	131,739	131,720	133,513	134,047	134,900	139,362	138,438
512K	131,511	131,092	131,961	133,667	134,183	135,244	136,112
1024K	131,722	130,997	132,152	132,130	132,938	132,597	133,623

Table C.9: Replacement-scheme results for the chess middle game (without time stamping, 7-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	159,463	166,755	177,310	178,308	181,614	196,798	216,668
16K	149,330	153,771	160,847	161,390	166,035	179,992	200,805
32K	139,768	144,890	149,229	149,105	152,088	164,413	179,437
64K	135,154	137,008	140,681	140,523	144,751	150,047	160,516
128K	131,154	132,456	134,055	134,473	137,187	141,991	145,095
256K	129,206	127,970	131,685	131,598	132,414	136,596	138,460
512K	128,502	127,783	130,016	129,589	131,591	131,839	132,956
1024K	127,797	127,592	128,450	129,906	128,856	132,099	130,440

Table C.10: Replacement-scheme results for the chess middle game (with time stamping, 7-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
16K	433,734	446,535	487,496	501,011	513,109	588,646	632,460
64K	363,502	370,538	397,446	395,755	411,775	494,620	501,081
256K	319,972	330,656	357,510	341,741	343,183	382,383	397,500
1024K	316,183	309,192	327,089	324,889	323,494	344,971	339,173

Table C.11: Replacement-scheme results for the chess middle game (without time stamping, 8-ply searches).

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
16K	421,898	441,366	495,898	493,721	527,186	590,553	706,031
64K	354,250	366,945	400,034	393,498	415,413	493,801	545,271
256K	321,320	321,361	342,764	337,188	349,470	397,421	407,303
1024K	300,430	312,459	320,724	316,789	317,324	329,933	333,470

Table C.12: Replacement-scheme results for the chess middle game (with time stamping, 8-ply searches).

Second, the 10-ply results for the seven replacement schemes (TwoBig1, TwoDeep, Big1, BigAll, Deep, New and Old) on chess endgame positions are listed in Table C.13 with eight table sizes (8K, 16K, 32K, 64K, 128K, 256K, 512K and 1024K). All figures reported are number of nodes searched in millions. These results are the cumulative results of all five games (112 endgame positions) given in Appendix B. For every endgame experiment time stamping was used.

	TwoBig1	TwoDeep	Big1	BigAll	Deep	New	Old
8K	608	638	909	997	953	1,034	1,569
16K	508	533	740	779	758	914	1,392
32K	434	447	615	632	599	807	1,124
64K	396	407	517	520	501	698	916
128K	365	359	428	415	408	607	692
256K	334	330	372	367	388	528	505
512K	309	304	333	327	334	460	396
1024K	287	292	305	302	301	404	330

Table C.13: Replacement-scheme results for the chess endgame (with time stamping, 10-ply searches).

Third, the results in the domain of domineering for five replacement schemes (TwoBig1, TwoDeep, Big1, Deep and New) are listed with four table sizes (256K, 512K, 1024K and 2048K) in Table C.14. All figures reported are number of nodes searched in millions. The results are given for the empty standard (8×8) board. Obviously, no time stamping was used, since the test set consists of only one position.

Quantifying the merits of move and score

For the second series of experiments the following six experimental searches have been performed.

1. Search without a transposition table.
2. Search with a traditional transposition table, without *score*.
3. Search with a traditional transposition table, without *move*.
4. Search with a traditional transposition table, without *move*, only storing and using the score information if the score is a *true value*.
5. Search with a traditional transposition table, without *move*, only storing and using the score information if the score is a *bound value*.
6. Search with a traditional transposition table, with *move* and *score*, storing and using the score information both if the score is a *true value* or a *bound value* (i.e., use the transposition table fully).

	TwoBig1	TwoDeep	Big1	Deep	New
256K	1,212	1,298	1,659	1,930	3,742
512K	885	939	1,122	1,283	2,743
1024K	607	635	745	812	2,130
2048K	442	452	492	504	1,380

Table C.14: Replacement-scheme results for domineering.

First, the 8-ply transposition-table results for the replacement scheme TwoBig1 on 18 consecutive WTM middle-game positions taken from the game Kasparov-Short, Amsterdam (round 2) 1994 (cf. Appendix A) are listed in Table C.15.

Second, the 10-ply results for the replacement scheme TwoBig1 on 21 consecutive WTM endgame positions taken from the game Rabinovich-Romanovsky, Leningrad 1934 (cf. Appendix B) are listed in Table C.16. The experiments have been performed with six table sizes (8K, 16K, 32K, 64K, 128K and 256K). For every experiment time stamping was used. All figures reported are number of nodes searched in thousands.

	No tt	Tt-move	Tt-score	True tt-score	Bound tt-score	Traditional tt
8K	610,696	473,041	326,139	599,491	321,141	297,244
16K	610,696	456,754	298,797	599,491	282,651	262,409
32K	610,696	433,439	274,639	599,491	264,205	235,358
64K	610,696	414,718	266,698	599,491	250,568	213,422
128K	610,696	403,509	258,279	599,491	239,020	200,483
256K	610,696	392,076	248,917	599,491	227,815	193,644

Table C.15: Transposition-table results for the chess middle game (with time stamping, 8-ply searches).

Using additional memory

For the third series of experiments we have tested the results of storing an n-ply PV ($n = 2..5$) in an entry versus storing only the best move (a 1-ply PV).

First, the 8-ply transposition-table results for the replacement scheme TwoBig1 on 18 consecutive WTM middle-game positions taken from the game Kasparov-Short, Amsterdam (round 2) 1994 (cf. Appendix A) are listed in Table C.17.

Second, the 10-ply results for the replacement scheme TwoBig1 on 21 consecutive WTM endgame positions taken from the game Rabinovich-Romanovsky, Leningrad 1934 (cf. Appendix B) are listed in Table C.18. The experiments have

	No tt	Tt-move	Tt-score	True tt-score	Bound tt-score	Traditional tt
8K	409,119	208,056	118,658	404,658	114,804	73,690
16K	409,119	188,754	102,370	404,394	102,988	61,779
32K	409,119	174,397	95,666	404,415	90,221	52,473
64K	409,119	161,670	88,890	402,261	85,470	47,928
128K	409,119	151,189	81,801	402,644	81,811	43,107
256K	409,119	144,681	76,388	402,644	81,501	43,168

Table C.16: Transposition-table results for the chess endgame (with time stamping, 10-ply searches).

been performed with six table sizes (8K, 16K, 32K, 64K, 128K and 256K). For every experiment time stamping was used. All figures reported are number of nodes searched in thousands.

	1-ply PV	2-ply PV	3-ply PV	4-ply PV	5-ply PV
8K	297,244	295,662	294,646	295,330	292,025
16K	262,409	259,041	261,433	261,788	258,357
32K	235,358	231,825	239,193	241,828	236,579
64K	213,422	221,658	219,221	215,449	213,998
128K	200,483	201,704	207,532	202,358	203,506
256K	193,644	194,858	188,259	187,287	184,820

Table C.17: PV results for the chess middle game (with time stamping, 8-ply searches).

	1-ply PV	2-ply PV	3-ply PV	4-ply PV	5-ply PV
8K	73,690	75,732	77,254	74,340	73,209
16K	61,779	63,815	62,303	60,851	62,768
32K	52,473	52,432	51,213	55,629	51,898
64K	47,928	44,365	50,903	46,377	43,603
128K	43,107	41,250	43,680	41,919	42,128
256K	43,168	37,208	38,843	37,451	37,744

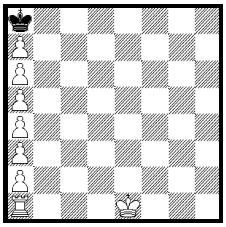
Table C.18: PV results for the chess endgame (with time stamping, 10-ply searches).

Appendix D

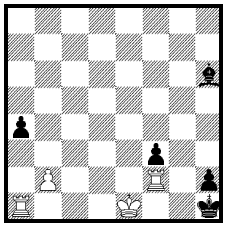
The pn-search and pn^2-search test set

This appendix lists the test set of 117 positions used for the proof-number-search experiments described in Chapters 3, 4, and 5.

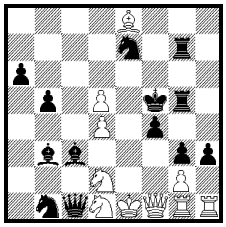
The following WTM positions from Reinfeld (1958) and Krabbé (1985) have been used:



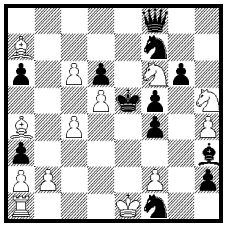
Krabbé #35



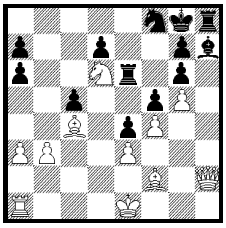
Krabbé #37



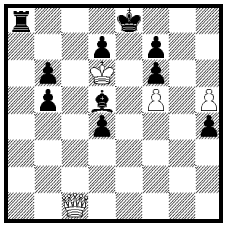
Krabbé #38



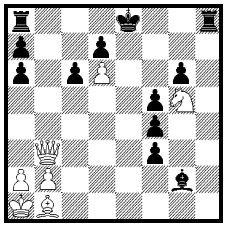
Krabbé #40



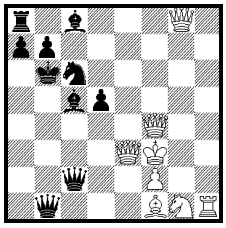
Krabbé #44



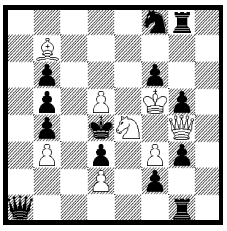
Krabbé #60

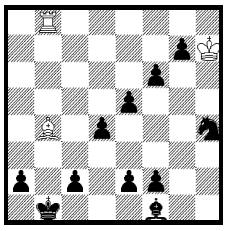


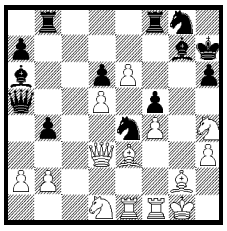
Krabbé #61

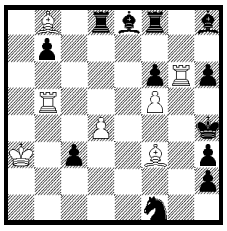


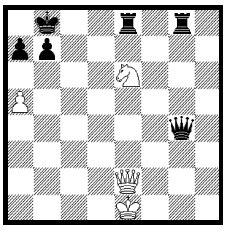
Krabbé #78

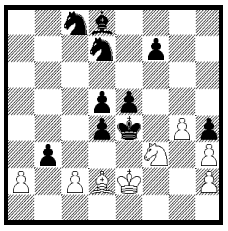

Krabbé #192

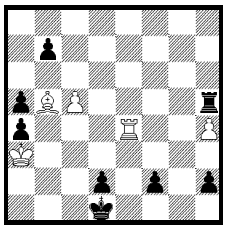

Krabbé #194

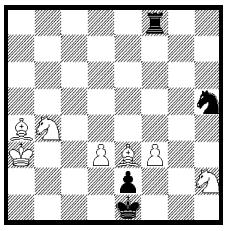

Krabbé #195

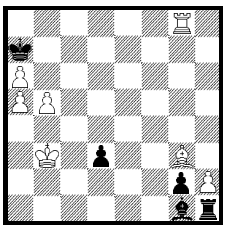

Krabbé #196

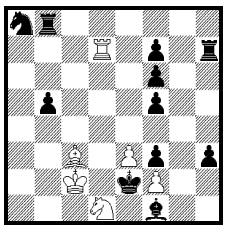

Krabbé #197

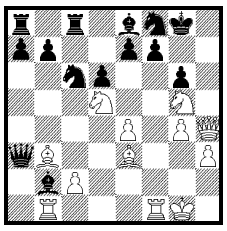

Krabbé #198

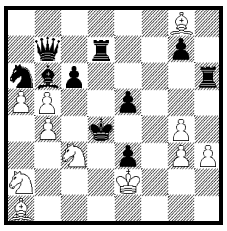

Krabbé #199

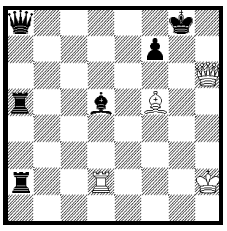

Krabbé #206

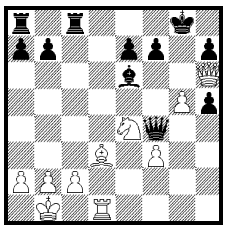

Krabbé #207

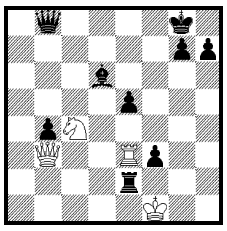

Krabbé #208

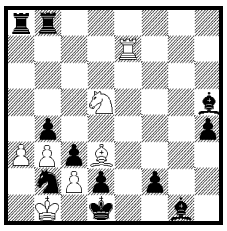

Krabbé #209

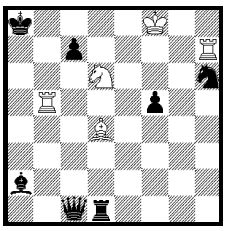

Krabbé #210

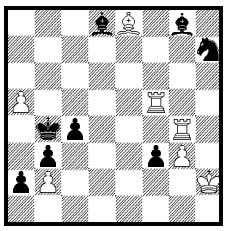

Krabbé #211

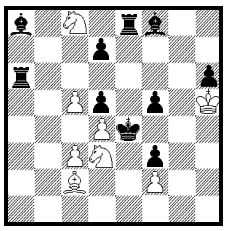

Krabbé #212

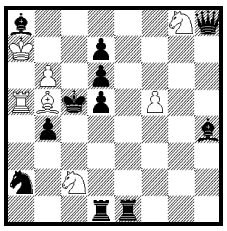

Krabbé #214

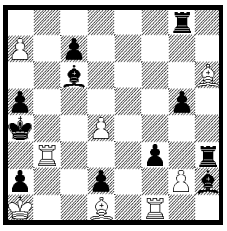

Krabbé #215

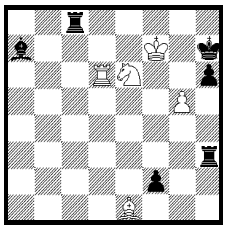

Krabbé #216

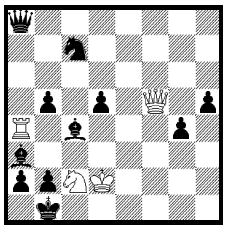

Krabbé #217

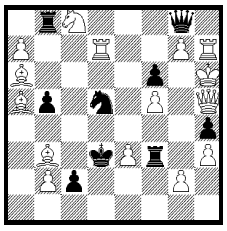

Krabbé #218

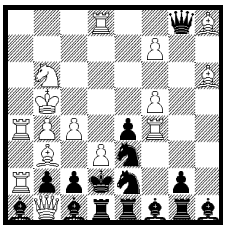

Krabbé #219

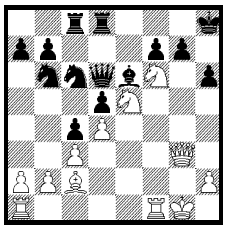

Krabbé #220

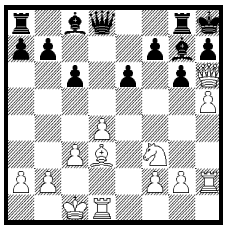

Krabbé #261

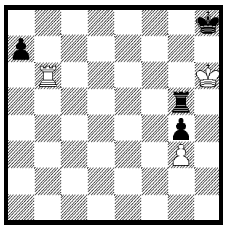

Krabbé #317

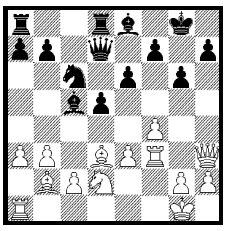

Krabbé #333

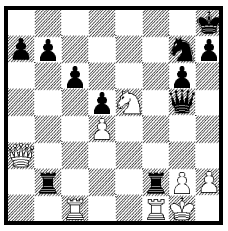

Krabbé #334

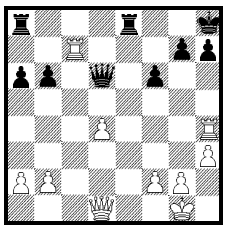

Reinfeld #1

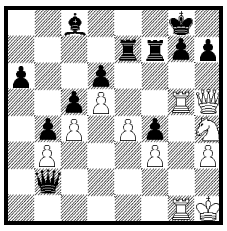

Reinfeld #4

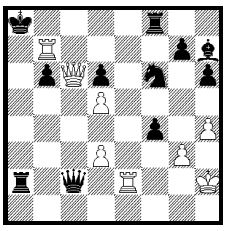

Reinfeld #6

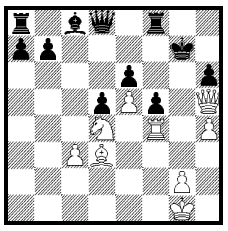

Reinfeld #14

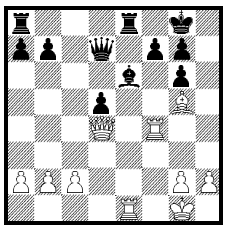

Reinfeld #27

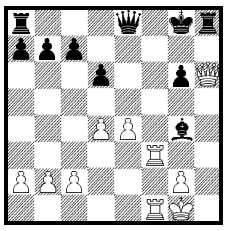

Reinfeld #35

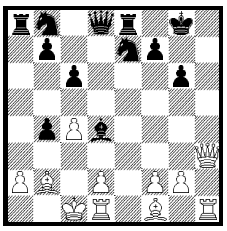

Reinfeld #49

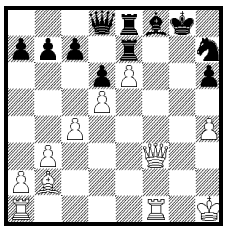

Reinfeld #50

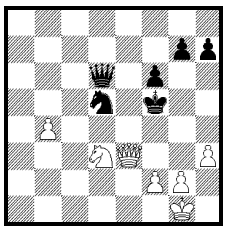

Reinfeld #51

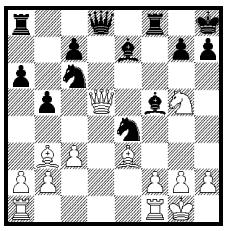

Reinfeld #55

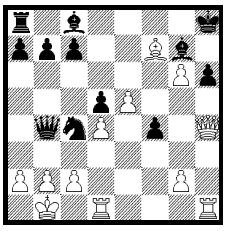

Reinfeld #57

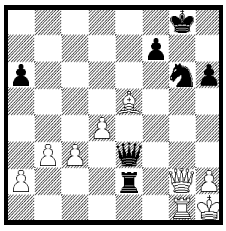

Reinfeld #60

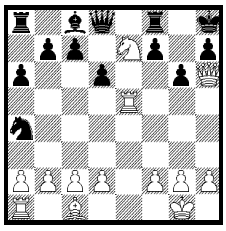

Reinfeld #61

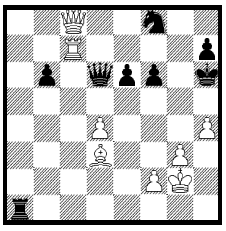

Reinfeld #64

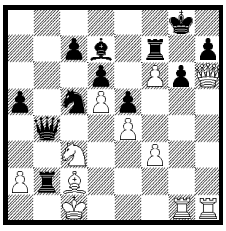

Reinfeld #84

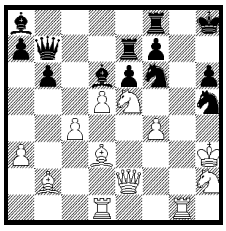

Reinfeld #96

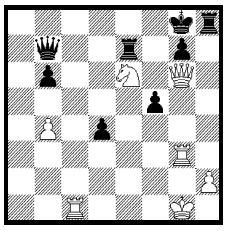

Reinfeld #97

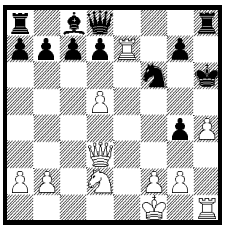

Reinfeld #99

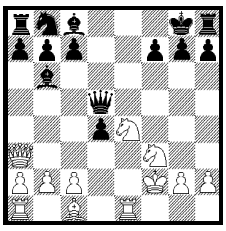

Reinfeld #102

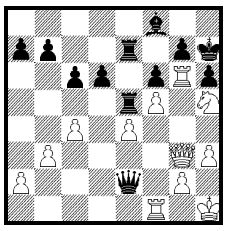

Reinfeld #103

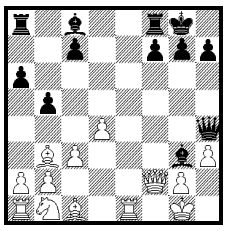

Reinfeld #104

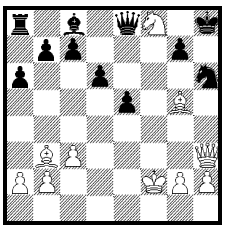

Reinfeld #136

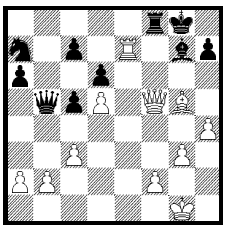

Reinfeld #138

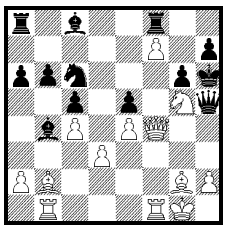

Reinfeld #139

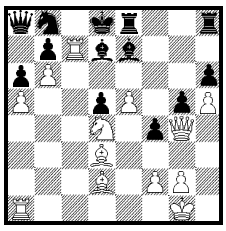

Reinfeld #143

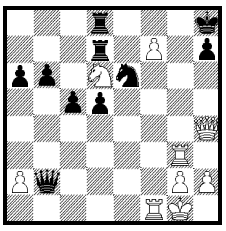

Reinfeld #154

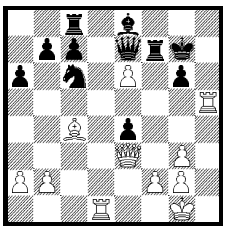

Reinfeld #156

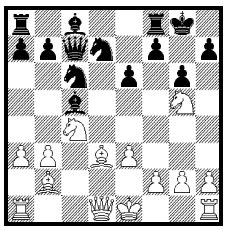

Reinfeld #158

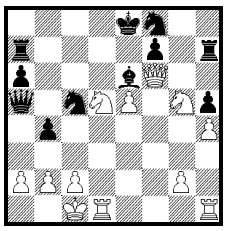

Reinfeld #159

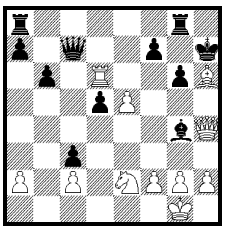

Reinfeld #160

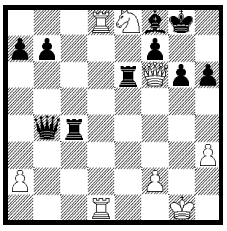

Reinfeld #161

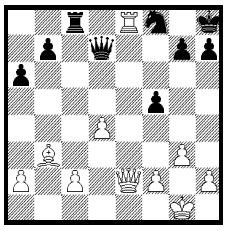

Reinfeld #173

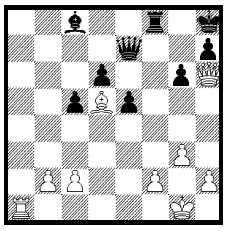

Reinfeld #182

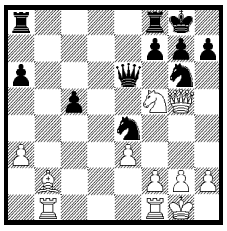

Reinfeld #184

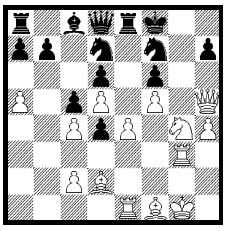

Reinfeld #186

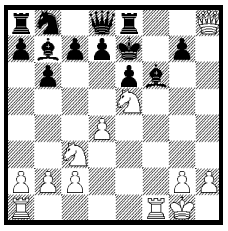

Reinfeld #188

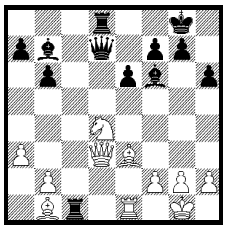

Reinfeld #191

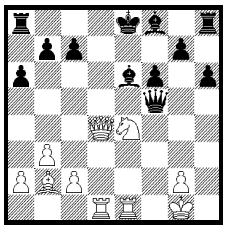

Reinfeld #201

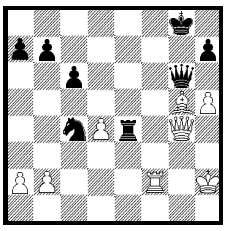

Reinfeld #203

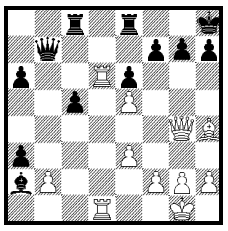

Reinfeld #211

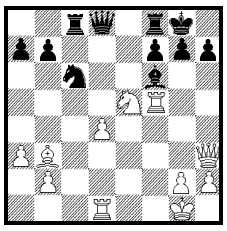

Reinfeld #212

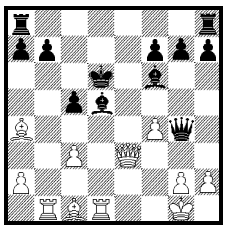

Reinfeld #215

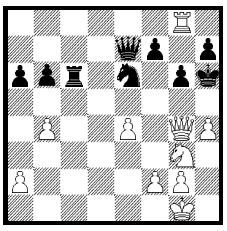

Reinfeld #217

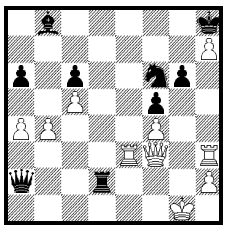

Reinfeld #218

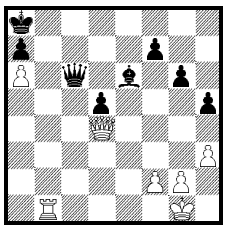

Reinfeld #222

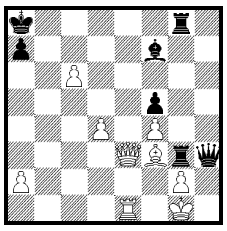

Reinfeld #241

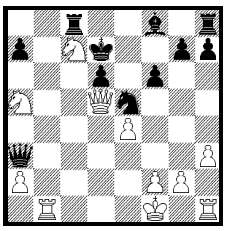

Reinfeld #244

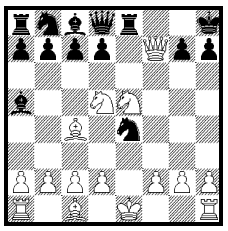

Reinfeld #246

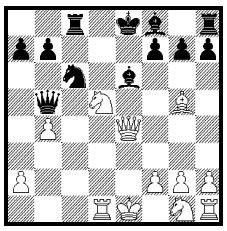

Reinfeld #250

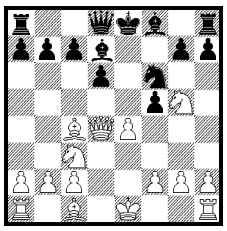

Reinfeld #251

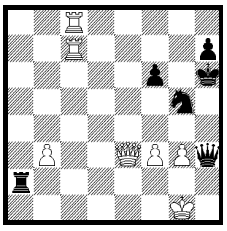

Reinfeld #253

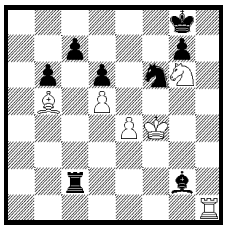

Reinfeld #260

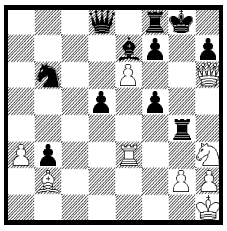

Reinfeld #263

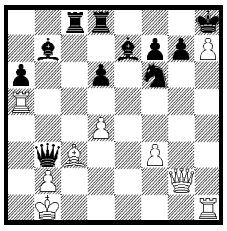

Reinfeld #267

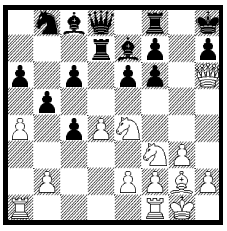

Reinfeld #278

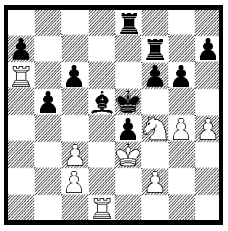

Reinfeld #281

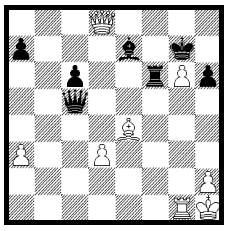

Reinfeld #282


Reinfeld #283

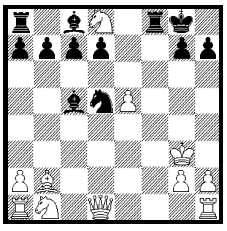

Reinfeld #285

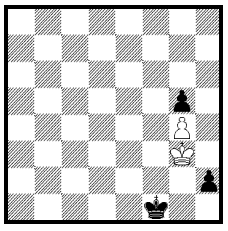

Reinfeld #293

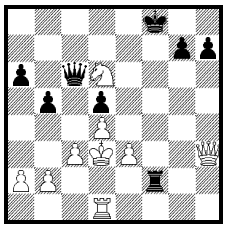

Reinfeld #295

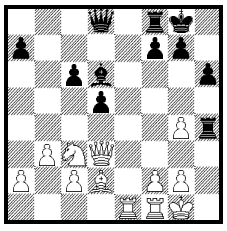

Reinfeld #298

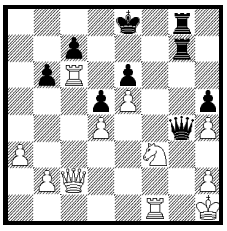
The following BTM positions from Reinfeld (1958) and Krabbé (1985) have been used:

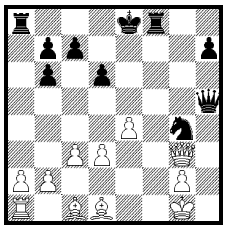

Krabbé #8

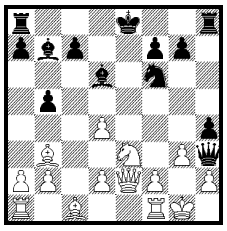

Krabbé #284

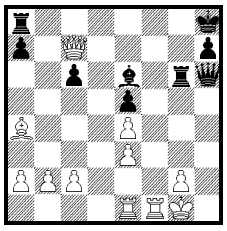

Reinfeld #5

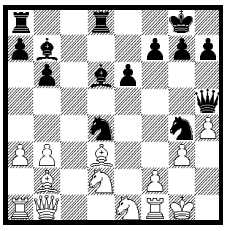

Reinfeld #9


Reinfeld #12

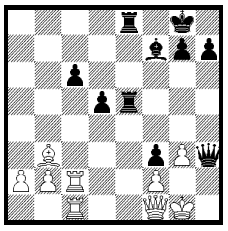

Reinfeld #54


Reinfeld #79

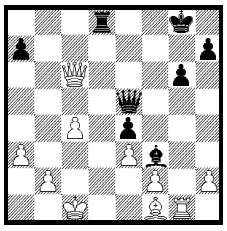

Reinfeld #88



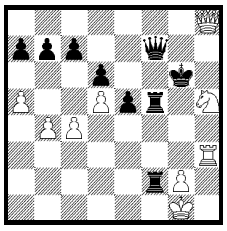
Reinfeld #105



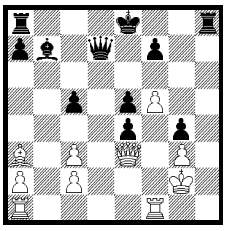
Reinfeld #132



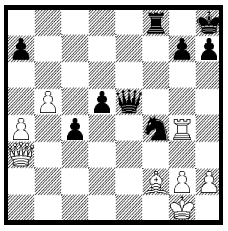
Reinfeld #134



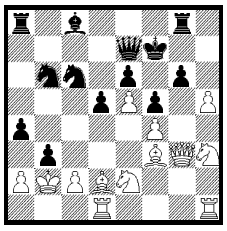
Reinfeld #167



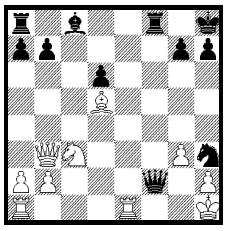
Reinfeld #168



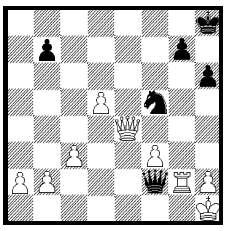
Reinfeld #172



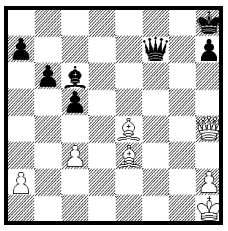
Reinfeld #177



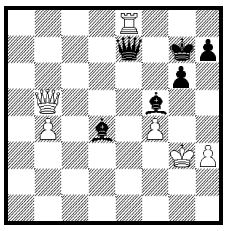
Reinfeld #179



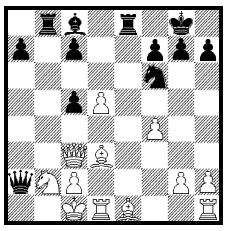
Reinfeld #197



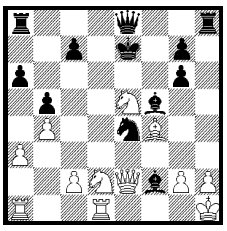
Reinfeld #219



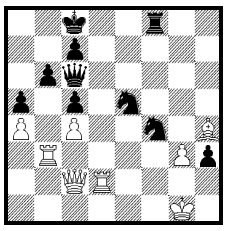
Reinfeld #225



Reinfeld #252



Reinfeld #266



Reinfeld #290

Appendix E

The pn-search versus $\alpha\beta$-search results

This appendix presents the results of the experiments with the pn-search algorithm and the $\alpha\beta$-seach algorithm described in Chapter 3. In Table E.1 all results are listed for the test set of 117 positions. The numbers refer to the number of nodes searched. A dash signifies that no solution is found due to the memory constraints (1,000,000 nodes). The first column lists the test positions. Columns two and three show the results for pn search and $\alpha\beta$ search, respectively.

	Pn search	$\alpha\beta$ search
K8	–	–
K35	296	244,122
K37	43,221	–
K38	273	57,509
K40	–	–
K44	–	–
K60	–	930,899
K61	42,228	–
K78	–	–
K192	23,290	–
K194	229,423	–
K195	–	–
K196	298,428	–
K197	323	15,831
K198	247,435	–

	Pn search	$\alpha\beta$ search
K199	370,016	–
K206	15,978	–
K207	95,418	–
K208	62,791	–
K209	–	–
K210	–	–
K211	957	155,641
K212	81,842	104,368
K214	685	–
K215	114,060	–
K216	592,890	–
K217	–	–
K218	118,361	–
K219	310,447	–
K220	–	–

Table E.1: Comparing pn search and $\alpha\beta$ search (continued on next page).

	Pn search	αβ search
K261	482	906
K284	–	224,092
K317	173,480	158,523
K333	145,922	–
K334	217,516	–
R1	7,640	18,225
R4	82	52
R5	57	22
R6	71,966	–
R9	207	124,234
R12	175	1,174
R14	324,542	127,519
R27	77	270
R35	527	3,421
R49	16,546	545,344
R50	183	830
R51	227,361	–
R54	85	1,631
R55	31,456	393,646
R57	113	106
R60	69	107
R61	78	537
R64	137	1,201
R79	152	49,502
R84	93	609
R88	759	21,862
R96	–	–
R97	107	7,186
R99	75,411	12,227
R102	279	208
R103	2,150	20,662
R104	5,047	14,017
R105	–	777,182
R132	2,301	6,340
R134	854	19,008
R136	185	114
R138	211,466	–
R139	274	47,199
R143	900	13,023
R154	117	1,784
R156	82	1,081
R158	526	5,416
R159	385,487	–
R160	110	2,783

	Pn search	αβ search
R161	2,045	4,212
R167	896	178,495
R168	596,956	–
R172	99	6,062
R173	419	5,729
R177	527	5,538
R179	184	10,342
R182	807,709	–
R184	82	1,026
R186	108	30,588
R188	117	94
R191	22,466	17,952
R197	95	762
R201	–	–
R203	19,917	6,166
R211	278	282,863
R212	458	204,009
R215	164	3,244
R217	271	331,404
R218	277,639	–
R219	157	414
R222	59,591	–
R225	342	3,034
R241	360,983	–
R244	458	70,466
R246	120	753
R250	1,147	–
R251	136,479	384,761
R252	537,628	–
R253	2,355	311
R260	807	511,553
R263	887	45,180
R266	716	289,510
R267	1,206	22,890
R278	636	389,195
R281	317,214	–
R282	749	58,274
R283	30,778	102,501
R285	218	2,372
R290	523	42,650
R293	121,720	–
R295	81	3,584
R298	150	3,900

Table E.1: Comparing pn search and αβ search (continued).

Appendix F

The pn^2-search results

This appendix lists the results of the experiments with the pn^2-search algorithm and its modifications.

In Chapter 4 it is stated that when parameter a becomes large and parameter b becomes small the fraction function approaches $f(x) = 0$, which means that standard pn search is used. When both parameter a and parameter b have a small positive value the fraction function approaches $f(x) = 1$, which means that the pn^2-search algorithm suggested by Allis (1994) is used. Table F.1 confirms these observations. In the first column of the table the algorithm is given. The second column states the number of positions solved, and the third column states the total number of nodes searched. We note that the pn-search result differs from the result with (a b) equal to (999K 1), because the first result stems from the immediate-evaluation variant of pn search, and the second from the delayed-evaluation variant of pn search.

Algorithm	Solved	# nodes
Pn search	92	2,974,602
a=999K b=1	87	2,392,664
a=1 b=1	108	67,085,784
Pn^2 search (Allis)	108	67,085,784

Table F.1: Two extremes of the fraction function.

Next, all results of the experiments described in Chapter 4 are presented in Table F.2. For these experiments, the test set consists of 108 positions. In the first two columns the values of parameters a and b are given. The third column states the number of positions solved. The sizes of the first-level and second-level tree are listed in columns four and five, respectively. Column six shows the total number of nodes searched over the solved positions. Finally, the maximum number of nodes in memory for the most difficult test position is given in column seven.

a	b	#	First level	Second level	Total	Maximum
75K	3,750	108	2,668,522	414,715,977	417,384,499	122,822
75K	7.5K	108	2,107,363	183,059,544	185,166,907	97,880
75K	15K	108	1,321,283	58,630,011	59,951,294	70,808
75K	30K	108	825,798	56,792,248	57,618,046	48,488
75K	37.5K	108	737,686	55,092,677	55,830,363	42,945
75K	45K	108	658,312	52,527,548	53,185,860	40,428
75K	60K	108	599,335	54,678,255	55,277,590	35,580
75K	75K	108	553,540	55,622,054	56,175,594	39,947
75K	90K	108	538,282	57,474,549	58,012,831	39,984
75K	120K	108	510,672	58,163,405	58,674,077	37,322
75K	150K	108	503,728	59,589,317	60,093,045	36,568
75K	180K	108	494,585	59,457,269	59,951,854	36,563
75K	210K	108	488,951	59,792,279	60,281,230	36,003
75K	240K	108	492,391	61,418,574	61,910,965	35,605
100K	10K	108	2,482,057	141,392,048	143,874,105	110,920
100K	60K	108	663,291	50,661,253	51,324,544	46,305
100K	90K	108	561,438	49,790,831	50,352,269	34,653
125K	12.5K	108	2,818,440	104,339,512	107,157,952	120,979
125K	75K	108	659,378	46,992,565	47,651,943	43,770
125K	90K	108	621,220	49,335,318	49,956,538	38,737
150K	3,750	108	4,998,281	612,351,208	617,349,489	189,770
150K	7.5K	108	4,390,473	269,286,304	273,676,777	166,447
150K	15K	108	3,113,015	80,038,932	83,151,947	135,643
150K	30K	108	1,728,353	60,406,646	62,134,999	91,605
150K	60K	108	901,278	54,777,822	55,679,100	55,430
150K	75K	108	793,386	56,532,843	57,326,229	53,954
150K	90K	108	674,730	49,173,880	49,848,610	48,665
150K	120K	108	601,153	49,506,906	50,108,059	37,630
150K	150K	108	555,561	50,273,586	50,829,147	34,647
150K	180K	108	526,613	51,573,591	52,100,204	33,413
150K	210K	108	518,422	54,791,716	55,310,138	38,152
150K	240K	108	510,498	56,215,959	56,726,457	38,249
150K	480K	108	498,061	63,168,302	63,666,363	31,185
200K	20K	108	3,642,974	58,523,263	62,166,237	164,472
200K	90K	108	853,735	54,127,970	54,981,705	55,158
200K	120K	108	696,652	50,890,364	51,587,016	48,860
250K	25K	108	4,046,147	46,244,259	50,290,406	175,617
250K	90K	108	1,057,131	55,266,557	56,323,688	68,215
250K	150K	108	697,125	49,876,992	50,574,117	49,780
300K	15K	108	6,904,133	85,195,061	92,099,194	275,206
300K	22.5K	108	5,558,725	43,618,137	49,176,862	237,670
300K	30K	108	4,391,781	43,512,164	47,903,945	200,110

Table F.2: The pn² results for varying parameters a and b
(continued on next page).

a	b	#	First level	Second level	Total	Maximum
300K	40K	108	3,219,230	39,278,406	42,497,636	162,991
300K	60K	108	2,003,287	46,485,285	48,488,572	117,496
300K	90K	108	1,248,374	48,739,998	49,988,372	77,854
300K	120K	108	957,952	53,597,265	54,555,217	59,768
300K	150K	108	817,824	56,359,721	57,177,545	55,957
300K	165K	108	753,545	51,515,985	52,269,530	55,570
300K	180K	108	711,403	51,190,005	51,901,408	51,969
300K	210K	108	647,192	49,930,739	50,577,931	44,686
300K	240K	108	618,720	51,738,723	52,357,443	40,312
300K	300K	108	553,488	48,820,125	49,373,613	36,234
300K	480K	108	516,863	56,576,815	57,093,678	37,855
450K	15K	88	2,685,315	77,055	2,762,370	>300,000
450K	22.5K	98	5,309,701	2,889,136	8,198,837	>300,000
450K	30K	104	6,159,818	13,468,281	19,628,099	>300,000
450K	45K	108	5,200,437	31,899,341	37,099,778	277,633
450K	60K	108	3,710,957	32,101,854	35,812,811	200,822
450K	75K	108	2,755,338	37,285,569	40,040,907	158,558
450K	90K	108	2,158,171	43,057,915	45,216,086	135,091
450K	120K	108	1,458,198	45,147,453	46,605,651	98,068
450K	150K	108	1,173,016	50,363,222	51,536,238	74,863
450K	180K	108	989,985	55,097,838	56,087,823	65,040
450K	210K	108	850,180	52,617,722	53,467,902	55,486
450K	240K	108	772,110	50,392,270	51,164,380	50,924
450K	270K	108	720,221	51,678,441	52,398,662	52,916
450K	300K	108	655,616	47,560,738	48,216,354	45,910
450K	330K	108	634,798	48,572,363	49,207,161	40,442
450K	360K	108	614,915	49,918,022	50,532,937	40,806
450K	450K	108	560,286	49,245,643	49,805,929	36,882
600K	15K	87	2,392,664	0	2,392,664	>300,000
600K	30K	91	3,539,437	433,025	3,972,462	>300,000
600K	60K	106	5,175,810	21,486,835	26,662,645	>300,000
600K	**80K**	**108**	**4,061,359**	**30,818,703**	**34,880,062**	**236,383**
600K	90K	108	3,429,832	32,596,751	36,026,583	200,682
600K	120K	108	2,198,328	37,433,618	39,631,946	136,878
600K	150K	108	1,632,654	42,467,933	44,100,587	112,288
600K	180K	108	1,310,916	46,726,924	48,037,840	87,436
600K	210K	108	1,137,297	51,167,545	52,304,842	73,973
600K	240K	108	1,006,380	55,098,476	56,104,856	68,357
600K	300K	108	826,932	55,119,547	55,946,479	59,813
600K	360K	108	719,206	49,798,501	50,517,707	47,901
600K	480K	108	614,823	49,558,639	50,173,462	39,913
600K	600K	108	558,575	48,982,121	49,540,696	36,773

Table F.2: The pn^2 results for varying parameters a and b
(continued on next page).

a	b	#	First level	Second level	Total	Maximum
750K	15K	87	2,392,664	0	2,392,664	>300,000
750K	30K	87	2,392,664	0	2,392,664	>300,000
750K	37.5K	87	2,392,664	834	2,393,498	>300,000
750K	60K	99	4,668,848	4,396,135	9,064,983	>300,000
750K	75K	104	4,879,717	12,059,792	16,939,509	>300,000
750K	90K	107	4,625,718	24,204,635	28,830,353	>300,000
750K	120K	108	3,226,125	32,090,839	35,316,964	206,505
750K	150K	108	2,296,757	39,959,428	42,256,185	149,546
750K	180K	108	1,731,845	39,282,953	41,014,798	116,815
750K	210K	108	1,447,288	46,590,740	48,038,028	98,804
750K	240K	108	1,269,027	51,306,410	52,575,437	83,348
750K	300K	108	1,015,055	55,693,427	56,708,482	68,297
750K	450K	108	720,103	49,808,809	50,528,912	48,265
750K	600K	108	614,273	49,255,159	49,869,432	39,933
750K	750K	108	564,262	49,982,423	50,546,685	36,757

Table F.2: The pn² results for varying parameters a and b (continued).

Appendix G

The BTA results for pn search

This appendix presents the results of the experiments with the pn-search algorithm and its modifications described in Chapter 5. In Table G.1 the results of the 117 test positions are listed for four pn-search variants with the same move ordering. The numbers refer to the number of nodes searched. A dash signifies that no solution was found due to the memory constraints (500,000 nodes). The first column lists the test positions. Columns two to five show the results for the tree algorithm[1], the DAG algorithm, the DCG algorithm, and the BTA algorithm, respectively.

	Tree	*DAG*	*DCG*	*BTA*
K8	–	–	–	–
K35	296	296	276	276
K37	35,724	25,737	17,981	19,886
K38	273	273	272	272
K40	–	–	–	–
K44	–	274,211	274,211	146,938
K60	–	310,251	372,634	487,969
K61	43,911	41,997	35,770	38,446
K78	–	–	–	–
K192	22,525	15,429	14,252	15,767
K194	238,085	51,427	30,699	102,336

Table G.1: The results for four pn-search variants (continued on next page).

[1] The numbers differ from the numbers given in Appendix E, because there the tree algorithm uses a different move ordering.

	Tree	*DAG*	*DCG*	*BTA*
K195	–	–	–	–
K196	318,276	97,717	88,069	93,447
K197	429	429	417	413
K198	333,165	262,255	171,720	177,929
K199	369,555	290,903	151,043	202,903
K206	11,931	11,543	9,483	9,191
K207	236,568	88,024	41,348	50,870
K208	72,468	65,279	–	31,648
K209	–	–	–	–
K210	–	–	–	–
K211	1,059	1,059	939	937
K212	83,413	59,988	52,946	55,290
K214	645	645	629	624
K215	124,984	94,108	–	74,967
K216	–	366,336	–	247,686
K217	–	–	311,027	407,633
K218	122,058	109,308	124,868	107,215
K219	277,250	129,232	63,297	83,329
K220	–	–	–	–
K261	414	388	388	424
K284	–	2,337	2,337	2,851
K317	157,424	120,358	103,033	94,043
K333	165,725	134,339	123,599	139,184
K334	145,291	88,430	74,375	82,889
R1	4,275	4,095	3,996	4,270
R4	82	82	82	82
R5	57	57	57	57
R6	96,059	32,953	11,703	13,179
R9	173	173	169	168
R12	99	99	99	99
R14	335,936	213,098	157,269	185,918
R27	77	77	77	77
R35	597	559	371	522
R49	16,935	14,767	13,797	15,625
R50	399	383	369	408
R51	270,495	191,822	173,922	204,299
R54	256	256	256	256
R55	15,245	13,293	12,749	14,552
R57	287	287	287	287
R60	69	69	69	69
R61	78	78	78	78
R64	153	153	153	153

Table G.1: The results for four pn-search variants
(continued on next page).

	Tree	*DAG*	*DCG*	*BTA*
R79	152	152	152	152
R84	93	93	93	93
R88	595	547	542	583
R96	–	–	–	–
R97	107	107	107	107
R99	31,767	31,302	27,264	27,290
R102	199	199	199	199
R103	1,837	1,742	1,731	1,780
R104	5,042	4,660	4,658	4,870
R105	–	–	–	–
R132	2,291	2,105	2,077	2,135
R134	804	798	758	760
R136	230	230	230	230
R138	192,886	164,106	118,729	137,030
R139	182	182	182	182
R143	521	520	520	519
R154	197	197	196	196
R156	82	82	82	82
R158	495	495	494	494
R159	403,797	253,108	274,275	263,275
R160	110	110	110	110
R161	1,790	1,209	1,209	1,332
R167	923	901	813	810
R168	–	317,557	209,725	301,527
R172	99	99	99	99
R173	419	418	404	402
R177	349	349	349	349
R179	156	156	156	156
R182	–	230,648	212,087	372,096
R184	82	82	82	82
R186	108	108	108	108
R188	117	117	117	117
R191	22,830	20,480	17,858	17,046
R197	95	95	95	95
R201	–	–	–	–
R203	20,980	18,429	17,265	17,397
R211	278	272	231	230
R212	545	545	543	543
R215	164	164	164	164
R217	199	199	199	199
R218	270,277	225,638	160,720	210,311
R219	140	140	140	140

Table G.1: The results for four pn-search variants
(continued on next page).

	Tree	*DAG*	*DCG*	*BTA*
R222	60,855	48,209	22,827	49,934
R225	263	263	263	263
R241	365,495	254,998	195,577	231,746
R244	323	323	323	323
R246	61	61	61	61
R250	1,102	1,101	1,076	1,071
R251	88,547	70,104	53,285	57,798
R252	–	–	352,315	386,046
R253	2,709	1,189	1,176	2,477
R260	841	794	729	804
R263	654	621	621	651
R266	716	716	711	711
R267	1,136	1,001	1,001	1,089
R278	333	333	333	333
R281	316,252	93,578	–	46,033
R282	749	729	725	742
R283	14,787	14,530	14,070	14,165
R285	218	218	218	218
R290	408	408	408	408
R293	97,666	94,138	75,283	75,509
R295	134	134	134	134
R298	150	150	150	150

Table G.1: The results for four pn-search variants (continued).

References

Akl S.G. and Newborn M.M. (1977). The Principal Continuation and the Killer Heuristic. *1977 Association for Computing Machinery Annual Conference*, pp. 466–473. Association for Computing Machinery, Seattle WA, USA. (23)

Allen J.D. (1989). A Note on the Computer Solution of Connect-Four. *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 134–135. Ellis Horwood Ltd., Chichester, United Kingdom. (4)

Allis L.V. (1988). A Knowledge-based Approach of Connect-Four. Technical Report IR-163, Vrije Universiteit Amsterdam, Amsterdam, the Netherlands. Reprinted (1992) by University of Limburg, Maastricht, The Netherlands. (4)

Allis L.V., Herik H.J. van den, and Herschberg I.S. (1991). Which Games Will Survive? *Heuristic Programming in Artificial Intelligence 2: The Second Computer Olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 232–243. Ellis Horwood Ltd., Chichester, United Kingdom. (3)

Allis L.V. and Schoo P.N.A. (1992). Qubic Solved Again. *Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad* (eds. H.J. van den Herik and L.V. Allis), pp. 192–204. Ellis Horwood Ltd., Chichester, United Kingdom. (3)

Allis L.V., Herik H.J. van den, and Huntjens M.P.H. (1993). Go-Moku Solved by New Search Techniques. *Proceedings of the 1993 AAAI Fall Symposium on Games: Planning and Learning*. AAAI Press Tecnical Report FS93-02, Menlo Park CA, USA. (4)

Allis L.V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. thesis, University of Limburg, Maastricht, The Netherlands. (4, 51, 52, 54, 58, 70, 72, 110, 139)

Allis L.V., Meulen M. van der, and Herik H.J. van den (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91–124. (13, 51, 54, 69, 87)

Allis L.V., Herik H.J. van den, and Huntjens M.P.H. (1996). Go-Moku Solved by New Search Techniques. *Computational Intelligence*, Vol. 12, No. 1, pp. 7–23. (4)

Anantharaman T.S., Campbell M., and Hsu F.-h. (1988). Singular Extensions: Adding Selectivity to Brute-Force Searching. *AAAI Spring Symposium, Computer Game Playing*, pp. 8–13. Also published (1990) in *Artificial Intelligence*, Vol. 43, No. 1, pp. 99–109. (27)

Anantharaman T.S. (1991). Confidently Selecting a Search Heuristic. *ICCA Journal*, Vol. 14, No. 1, pp. 3–16. (28, 29)

Averbakh Y. (1987). *Erfolg im Endspiel*. Sportverlag Berlin, Berlin, Germany. In German. (30)

Baum E.B. and Smith W.D. (1995). *Best Play for Imperfect Players and Game Tree Search*. Accepted for publication in *Artificial Intelligence*. A preliminary version is available from
`http://www.neci.nj.nec.com/homepages/smith/works.html`. (87)

Baum E.B. and Smith W.D. (1997). A Bayesian Approach to Relevance in Game Playing. *Artificial Intelligence*, Vol. 97, Nos. 1–2, pp. 195–242. (69)

Beal D.F. and Smith M.C. (1996). Multiple Probes of Transposition Tables. *ICCA Journal*, Vol. 19, No. 4, pp. 205–211. (19)

Berkey D.D. (1988). *Calculus*. Saunders College Publishing, New York NY, USA. (71)

Berlekamp E.R., Conway J.H., and Guy R.K. (1982a). *Winning Ways for your Mathematical Plays. Volume 2: Games in Particular*, pp. 670–671. Academic Press Inc., London, United Kingdom. (3)

Berlekamp E.R., Conway J.H., and Guy R.K. (1982b). *Winning Ways for your Mathematical Plays. Volume 1: Games in General*. Academic Press Inc., London, United Kingdom. (41)

Berlekamp E.R. (1988). Blockbusting and Domineering. *Journal of Combinatorial Theory, Series A*, Vol. 49, pp. 67–116. (41, 110)

Berliner H.J. (1974). *Chess as Problem Solving: The Development of a Tactics Analyzer*. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh PA, USA. (17, 22)

Berliner H.J. (1979). The B* Tree Search Algorithm: A Best-First Proof Procedure. *Artificial Intelligence*, Vol. 12, pp. 23–40. (69)

Berliner H.J. (1984). Search vs. Knowledge: An Analysis from the Domain of Games. *Artificial and Human Intelligence* (eds. A. Elithorn and R. Banerji), pp. 105–117. Elsevier Science Publishers B.V., Amsterdam, The Netherlands. (4)

Berliner H.J. and Ebeling C. (1990). Hitech. *Computers, Chess, and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 79–109. Springer-Verlag, New York NY, USA. (17)

Berliner H.J., Kopec D., and Northam E. (1991). A Taxonomy of Concepts for Evaluating Chess Strength: Examples from Two Difficult Categories. *Advances in Computer Chess 6* (ed. D.F. Beal), pp. 179–191. Ellis Horwood Ltd., Chichester, United Kingdom. (28)

Berliner H.J. and McConnell C. (1996). B* Probability Based Search. *Artificial Intelligence*, Vol. 86, No. 1, pp. 97–156. (85)

Bonsdorff E., Fabel K., and Riihimaa O. (1978). *Schach und Zahl*. Walter Rau Verlag, Düsseldorf, Germany, third edition. (10)

Bouton C.L. (1901). Nim, a Game with a Complete Mathematical Theory. *Annals of Mathematics*, Vol. 2, No. 3, pp. 35–39. (3)

Bouwmeester H. (1966). *Het Eindspel*. Prisma-schaakboek 7. Het Spectrum N.V., Utrecht, The Netherlands. In Dutch. (30)

Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1994a). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 17, No. 4, pp. 183–193. (9, 31)

Breuker D.M., Allis L.V., and Herik H.J. van den (1994b). How to Mate: Applying Proof-Number Search. *Advances in Computer Chess 7* (eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 251–272. University of Limburg, Maastricht, The Netherlands. (51, 59)

Breuker D.M. and Uiterwijk J.W.H.M. (1995). Transposition Tables in Computer Chess. *New Approaches to Board Games Research: Asian Origins and Future Perspectives* (ed. A.J. de Voogt), pp. 135–143. International Institute for Asian Studies, Leiden, The Netherlands. (9, 31)

Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1996). Replacement Schemes and Two-Level Tables. *ICCA Journal*, Vol. 19, No. 3, pp. 175–180. (9, 31)

Breuker D.M., Herik H.J. van den, Allis L.V., and Uiterwijk J.W.H.M. (1997a). A Solution to the GHI Problem for Best-First Search. *Proceedings of the Ninth Dutch Conference on Artificial Intelligence* (eds. K. van Marcke and W. Daelemans), pp. 457–468. University of Antwerp, Antwerp, Belgium. (81)

Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1997b). Information in Transposition Tables. *Advances in Computer Chess 8* (eds. H.J. van den Herik and J.W.H.M. Uiterwijk), pp. 199–211. Universiteit Maastricht, Maastricht, The Netherlands. (9, 31)

Breuker D.M., Herik H.J. van den, Allis L.V., and Uiterwijk J.W.H.M. (1998a). A Solution to the GHI Problem for Best-First Search. Submitted as journal publication. Also published (1997) as Technical Report CS 97-02, Universiteit Maastricht, Maastricht, The Netherlands. (81)

Breuker D.M., Uiterwijk J.W.H.M., and Herik H.J. van den (1998b). Solving Domineering. Submitted as journal publication. Also published (1998) as Technical Report CS 98-05, Universiteit Maastricht, Maastricht, The Netherlands. (9, 31)

Brudno A.L. (1963). Bounds and Valuations for Abridging the Search of Estimates. *Problems of Cybernetics*, Vol. 10, pp. 225–241. Translation of Russian original in *Problemy Kibernetiki*, Vol. 10, May 1963, pp. 141–150. (17, 22)

Buro M. (1994). *Techniken für die Bewertung von Spielsituationen anhand von Beispielen*. Ph.D. thesis, Universität–GH–Paderborn, Paderborn, Germany. In German. (2)

Buro M. (1995). ProbCut: An Effective Selective Extension of the α-β Algorithm. *ICCA Journal*, Vol. 18, No. 2, pp. 71–76. (17)

Buro M. (1997). The Othello Match of the Year: Takeshi Murakami vs. Logistello. *ICCA Journal*, Vol. 20, No. 3, pp. 189–193. (2)

Campbell M. (1985). The Graph-History Interaction: On Ignoring Position History. *1985 Association for Computing Machinery Annual Conference*, pp. 278–280. (82, 83, 86, 105)

Chinchalkar S. (1996). An Upper Bound for the Number of Reachable Positions. *ICCA Journal*, Vol. 19, No. 3, pp. 181–183. (10)

Clarke M.R.B. (1977). A Quantitative Study of King and Pawn against King. *Advances in Computer Chess 1* (ed. M.R.B. Clarke), pp. 108–115. Edinburgh University Press, Edinburgh, United Kingdom. (3)

Conway J.H. (1976). *On Numbers and Games*. Academic Press Inc. Ltd., London, United Kingdom. (26, 110)

Diepen P. van and Herik H.J. van den (1987). *Schaken voor Computers*. Academic Service, Schoonhoven, The Netherlands. In Dutch. (19)

Ebeling C. (1986). *All the Right Moves: A VLSI Architecture for Chess*. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh PA, USA. (17, 30, 32, 33, 36, 38)

Elo A. (1978). *The Rating of Chess Players, Past and Present*. Arco Publishing Inc., New York NY, USA. (27)

Feldmann R. (1993). *Game Tree Search on Massively Parallel Systems*. Ph.D. thesis, University of Paderborn, Paderborn, Germany. (15, 21)

Feldmann R. (1994). *Personal communication: response on a questionnaire*. (19, 20)

Feldmann R. (1996). *Personal communication*. (45)

Feldmann R. (1997). Fail-High Reductions. *Advances in Computer Chess 8* (eds. H.J. van den Herik and J.W.H.M. Uiterwijk), pp. 111–127. Universiteit Maastricht, Maastricht, The Netherlands. (17)

Feller W. (1950). *An Introduction to Probability Theory*. Wiley, New York NY, USA. (21)

Fine R. (1941). *Basic Chess Endings*. David McKay Company, New York NY, USA. (18, 30)

Fotland D. (1997). *Personal communication*. (40, 41)

Fraenkel A.S. (1996). Combinatorial Games: Selected Bibliography with a Succint Gourmet Introduction. *Games of No Chance. Combinatorial Games at MSRI, 1994* (ed. R.J. Nowakowski), pp. 493–537. Cambridge University Press, Cambridge, United Kingdom. (3)

Gardner M. (1974). Mathematical Games. *Scientific American*, Vol. 230, No. 2, pp. 106–108. (26)

Gasser R.U. (1995). *Harnessing Computational Resources for Efficient Exhaustive Search*. Ph.D. thesis, Swiss Federal Institute of Technology, Zürich, Switzerland. (4)

Gillogly J.J. (1972). The Technology Chess Program. *Artificial Intelligence*, Vol. 3, Nos. 1–3, pp. 145–163. (17)

Gillogly J.J. (1978). *Performance Analysis of the Technology Chess Program*. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh PA, USA. (17, 22, 28)

Gillogly J.J. (1989). Transposition Table Collisions. *Workshop on New Directions on Game-tree Search (pre-prints)* (ed. T.A. Marsland), p. 12. University of Alberta, Edmonton, Canada. (21)

Gillogly J.J. (1994). *Personal communication*. (21)

Ginsberg M.L. (1996). Partition Search. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 228–233. (110)

Goodman D. and Keene R. (1997). *Man versus Machine: Kasparov versus Deep Blue*. H3 publications, Cambridge MA, USA. (2)

Greenblatt R.D., Eastlake D.E., and Crocker S.D. (1967). The Greenblatt Chess Program. *Proceedings of the AFIPS Fall Joint Computer Conference 31*, pp. 801–810. Reprinted (1988) in *Computer Chess Compendium* (ed. D.N.L. Levy), pp. 56-66. B.T. Batsford Ltd., London, United Kingdom. (13, 31)

Groot A.D. de (1946). *Het Denken van den Schaker, een Experimenteel-psychologische Studie*. Ph.D. thesis, University of Amsterdam, Amsterdam, The Netherlands. In Dutch. Translated (1965) as *Thought and Choice in Chess* by Mouton Publishers, The Hague-Paris-New York. (3)

Groot A.D. de and Gobet F. (1996). *Perception and Memory in Chess*. Van Gorcum, Assen, The Netherlands. (With R.W. Jongman). (3)

Guy R.K. (1991). *Combinatorial Games*. American Mathematical Society, Providence. (41)

Herik H.J. van den (1983). *Computerschaak, Schaakwereld en Kunstmatige Intelligentie*. Ph.D. thesis, Delft University of Technology. Academic Service, Den Haag, The Netherlands. In Dutch. (2)

Herik H.J. van den (1991). *Kunnen computers rechtspreken?* Inaugural Address University of Leiden. Gouda Quint, Arnhem, The Netherlands. In Dutch. (2)

Howard K.S. (1961). *The Enjoyment of Chess Problems*. Dover Publications Inc., New York NY, USA. (65)

Hsu F.-h., Anantharaman T.S., Campbell M.S., and Nowatzyk A. (1990). Deep Thought. *Computers, Chess, and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 55–78. Springer-Verlag, New York NY, USA. (2)

Hyatt R.M., Gower A.E., and Nelson H.L. (1984). Cray Blitz. *Advances in Computer Chess 4* (ed. D.F. Beal), pp. 8–18. Pergamon Press, Oxford, United Kingdom. (18)

Hyatt R.M., Gower A.E., and Nelson H.L. (1990). Cray Blitz. *Computers, Chess, and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 111–130. Springer-Verlag, New York NY, USA. (16, 17, 31)

Hyatt R.M. (1994). *Personal communication: response on a questionnaire*. (19, 20, 31)

Junghanns A. and Schaeffer J. (1997). Sokoban: A Challenging Single-Agent Search Problem. *IJCAI-97 Workshop Proceedings: Using Games as an Experimental Testbed for AI Research* (ed. H. Iida), pp. 27–36. Nagoya, Japan. (38)

Junghanns A., Schaeffer J., Brockington M., Björnsson Y., and Marsland T.A. (1997). Diminishing Returns for Additional Search in Chess. *Advances in Computer Chess 8* (eds. H.J. van den Herik and J.W.H.M. Uiterwijk), pp. 53–67. Universiteit Maastricht, Maastricht, The Netherlands. (36)

Kažić B., Keene R., and Lim K.A. (1985). *The Official Laws of Chess and Other FIDE Regulations*. B.T. Batsford Ltd., London, United Kingdom. (82)

King D. (1997). *Kasparov versus Deeper Blue: The Ultimate Man versus Machine Challenge*. B.T. Batsford Ltd., London, United Kingdom. (2)

Klingbeil N. and Schaeffer J. (1990). Empirical Results with Conspiracy Numbers. *Computational Intelligence*, Vol. 6, pp. 1–11. (61)

Kmoch H. (1959). *Pawn Power in Chess*. David McKay Company, New York NY, USA. (27)

Knuth D.E. (1973). *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, Reading MA, USA. (14, 19, 20, 44)

Knuth D.E. and Moore R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. (11, 14, 18)

Kopec D. and Bratko I. (1982). The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp. 57–72. Pergamon Press, Oxford, United Kingdom. (27)

Krabbé T. (1985). *Chess Curiosities*. George Allen and Unwin Ltd., London, United Kingdom. (58, 62, 63, 104, 129, 134)

Lang K.J. and Smith W.D. (1993). A Test Suite for Chess Programs. *ICCA Journal*, Vol. 16, No. 3, pp. 152–161. (28)

Levenfish G. and Smyslov V. (1971). *Rook Endings*. B.T. Batsford Ltd., London, United Kingdom. (30)

MacWilliams F.J. and Sloane N.J.A. (1977). *The Theory of Error-Correcting Codes*. Elsevier Science Publishers B.V., Amsterdam, The Netherlands. (15)

Marsland T.A. and Rushton P.G. (1973). Mechanics for Comparing Chess Programs. *1973 Association for Computing Machinery Annual Conference*, pp. 202–205. (28)

Marsland T.A. and Campbell M.S. (1982). Parallel Search of Strongly Ordered Game Trees. *Computing Surveys*, Vol. 14, No. 4, pp. 533–551. (17)

Marsland T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3–19. (16, 22, 24, 30, 31)

McAllester D.A. (1988). Conspiracy Numbers for Min-Max Search. *Artificial Intelligence*, Vol. 35, No. 1, pp. 287–310. (51)

Michie D. (1980). Chess with Computers. *Interdisciplinary Science Reviews*, Vol. 5, No. 3, pp. 215–227. (2)

Minsky M. (1968). *Semantic Information Processing*. M.I.T. Press, Cambridge MA, USA. (2)

Morita K. (1997). *Personal communication*. (40)

Nelson H.L. (1985). Hash Tables in Cray Blitz. *ICCA Journal*, Vol. 8, No. 1, pp. 3–13. (17)

Newborn M. (1997). *Kasparov versus Deep Blue: Computer Chess Comes of Age*. Springer-Verlag, New York NY, USA. (2)

Newell A., Shaw J.C., and Simon H.A. (1958). Chess-Playing Programs and the Problem of Complexity. *IBM Journal of Research and Development*, Vol. 2, pp. 320–335. Reprinted (1988) in *Computer Games I* (ed. D.N.L. Levy), pp. 89-115. Springer-Verlag, New York NY, USA. (3)

Newell A. and Simon H.A. (1972). *Human Problem Solving*. Prentice-Hall Inc., Englewood Cliffs NY, USA. (3)

Nielsen J.B. (1991). A Chess-computer Test Set. *ICCA Journal*, Vol. 14, No. 1, pp. 33–37. (27)

Nilsson N.J. (1971). *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill Book Company, New York NY, USA. (3)

Palay A.J. (1985). *Searching with Probabilities*. Ph.D. thesis, Boston University, Boston MA, USA. (82, 85)

Patashnik O. (1980). Qubic: 4x4x4 Tic-Tac-Toe. *Mathematics Magazine*, Vol. 53, pp. 202–216. (3)

Pearl J. (1980). Asymptotic Properties of Minimax Game Trees and Game Searching Procedures. *Artificial Intelligence*, Vol. 14, No. 2, pp. 113–138. (17)

Pearl J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading MA, USA. (10)

Pijls W. and Bruin A. de (1994). Generalizing Alpha-Beta. *Advances in Computer Chess 7* (eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 219–236. University of Limburg, Maastricht, The Netherlands. (69)

Plaat A. (1996). *Research Re:search & Re-search*. Ph.D. thesis, Erasmus University Rotterdam, Rotterdam, The Netherlands. (13)

Plaat A., Schaeffer J., Pijls W., and Bruin A. de (1996). Best-First Fixed-Depth Minimax Algorithms. *Artificial Intelligence*, Vol. 87, No. 2, pp. 255–293. (13, 105)

Pronk T. (1987). Transpositietabellen in Schaakprogramma's. M.Sc. thesis, Gemeentelijke HTS, Den Haag, The Netherlands. In Dutch. (19)

Reinefeld A. (1983). An Improvement to the Scout Tree Search Algorithm. *ICCA Journal*, Vol. 6, No. 4, pp. 4–14. (17, 22)

Reinefeld A. (1989). *Spielbaum-Suchverfahren*. Springer-Verlag, Berlin, Germany. In German. (22)

Reinfeld F. (1958). *Win at Chess*. Dover Publications Inc., New York NY, USA. Originally published (1945) as *Chess Quiz* by David McKay Company, New York NY, USA. (27, 58, 64, 66, 129, 134)

Samuel A.L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, Vol. 3, No. 3, pp. 210–229. Reprinted (1963) in *Computers and Thought* (eds. E.A. Feigenbaum and J. Feldman), pp. 71-105. McGraw-Hill Book Company, New York NY, USA. (1, 11)

Samuel A.L. (1967). Some Studies in Machine Learning Using the Game of Checkers II – Recent Progress. *IBM Journal of Research and Development*, Vol. 11, No. 6, pp. 601–617. Reprinted (1970) in *Human and Artificial Intelligence* (ed. F.J. Crosson), pp. 81-116. Appleton-Century-Crofts, Educational Division, Meredith Corporation, New York NY, USA. (1)

Schaeffer J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19. (23)

Schaeffer J. (1986). *Experiments in Search and Knowledge*. Ph.D. thesis, University of Waterloo, Ontario, Canada. Also published (1986) as Technical Report TR 86-12, University of Alberta, Edmonton, Canada. (28)

Schaeffer J. (1989a). Conspiracy Numbers. *Advances in Computer Chess 5* (ed. D.F. Beal), pp. 199–217. Elsevier Science Publishers B.V., Amsterdam, The Netherlands. (64)

Schaeffer J. (1989b). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203–1212. (23, 30)

Schaeffer J. (1990). Conspiracy Numbers. *Artificial Intelligence*, Vol. 43, No. 1, pp. 67–84. (51, 64)

Schaeffer J. (1994). *Personal communication: response on a questionnaire*. (20, 32, 33, 45)

Schaeffer J. (1996a). Marion Tinsley: Human Perfection at Checkers? *Games of No Chance. Combinatorial Games at MSRI, 1994* (ed. R.J. Nowakowski), pp. 115–118. Cambridge University Press, Cambridge, United Kingdom. (2)

Schaeffer J. (1996b). *Personal communication*. (46)

Schaeffer J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag, New York NY, USA. (1)

Schaeffer J. and Plaat A. (1997). Kasparov versus Deep Blue: The Rematch. *ICCA Journal*, Vol. 20, No. 2, pp. 95–101. (2)

Schaeffer J. (1998). *Personal communication*. (63)

Schaeffer J., Culberson J., Treloar N., Knight B., Lu P., and Szafron D. (1992). A World Championship Caliber Checkers Program. *Artificial Intelligence*, Vol. 53, Nos. 2–3, pp. 273–290. (1)

Schijf M. (1993). Proof-Number Search and Transpositions. M.Sc. thesis, University of Leiden, Leiden, The Netherlands. (87, 88, 102)

Schijf M., Allis L.V., and Uiterwijk J.W.H.M. (1994). Proof-Number Search and Transpositions. *ICCA Journal*, Vol. 17, No. 2, pp. 63–74. (87, 88, 102, 104)

Schrüfer G. (1989). A Strategic Quiescence Search. *ICCA Journal*, Vol. 12, No. 1, pp. 3–9. (22)

Shannon C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256–275. (1, 10, 22)

Slate J.D. and Atkin L.R. (1977). CHESS 4.5: The Northwestern University Chess Program. *Chess Skill in Man and Machine* (ed. P.W. Frey), pp. 82–118. Springer-Verlag, New York NY, USA. Second Edition, 1983. (13, 14, 17, 29, 31)

Stanback J.S. (1994). *Personal communication: response on a questionnaire*. (19, 31, 45)

Stockman G. (1979). A Minimax Algorithm Better than Alpha-beta? *Artificial Intelligence*, Vol. 12, pp. 179–196. (13)

Thompson K. (1982). Computer Chess Strength. *Advances in Computer Chess 3* (ed. D.F. Beal), pp. 55–56. Pergamon Press, Oxford, United Kingdom. (36)

Thompson K. (1995). *Personal communication*. (86)

Thompson K. (1996a). *Personal communication*. (45)

Thompson K. (1996b). 6-Piece Endgames. *Advances in Computer Chess 8* (eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 9–26. Universiteit Maastricht, Maastricht, The Netherlands. An abbreviated version is published (1996) in *ICCA Journal*, Vol. 19, No. 4, pp. 215–226. (5)

Truscott T.R. (1981). Techniques Used in Minimax Game-Playing Programs. M.Sc. thesis, Duke University, Durham NC, USA. (46)

Turing A.M. (1953). Digital Computers Applied to Games. *Faster than Thought* (ed. B.V. Bowden), pp. 286–297. Pitman, London, United Kingdom. (1)

Uiterwijk J.W.H.M., Herik H.J. van den, and Allis L.V. (1989). A Knowledge-Based Approach to Connect-Four. The Game is Over: White to Move Wins! *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 113–133. Ellis Horwood Ltd., Chichester, United Kingdom. (4)

Uiterwijk J.W.H.M. (1994). *Personal communication: response on a questionnaire.* (20)

Uiterwijk J.W.H.M. (1996). The Kasparov – Deep Blue Match. *ICCA Journal*, Vol. 19, No. 1, pp. 38–41. (2)

Warnock T. and Wendroff B. (1988). Search Tables in Computer Chess. *ICCA Journal*, Vol. 11, No. 1, pp. 10–13. (15, 18)

Weill J.-C. (1994). *Personal communication: response on a questionnaire.* (19, 20)

Wendroff B. (1994). *Personal communication: response on a questionnaire.* (20)

West J. (1996). Championship-Level Play of Domineering. *Games of No Chance. Combinatorial Games at MSRI, 1994* (ed. R.J. Nowakowski), pp. 85–91. Cambridge University Press, Cambridge, United Kingdom. (26)

Zermelo E. (1912). Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. *Proceedings of the fifth International Congress of Mathematics*, Vol. 2, pp. 501–504. Cambridge, United Kingdom. (10)

Zobrist A.L. (1970). A New Hashing Method with Application for Game Playing. Technical Report #88, Computer Science Department, The University of Wisconsin, Madison WI, USA. Reprinted (1990) in *ICCA Journal*, Vol. 13, No. 2, pp. 69-73. (15, 20)

Index

Summary

Memory versus search in games

In this thesis, research is presented on the trade-off between memory and search. The domain under investigation is the domain of two-player zero-sum games, in particular the games of chess and domineering. The trade-off between memory and search is enhanced by the increase in availability of computer memory and the increase in processor speed.

Currently, the prices of computer memory are decreasing. Therefore, acquiring larger memory configurations is no longer an obstacle, making it easier to equip a computer with more memory. A depth-first search algorithm (such as $\alpha\beta$ search) uses little memory. The large amount of remaining memory can be used, e.g., to prevent the re-search of transpositions (identical positions in the tree). For this purpose, a transposition table, holding the results of previous searches, is maintained in the remaining memory. The trade-off transpires in more memory to be used, in favour of less searching. This leads to the formulation of the first problem statement.

Problem statement 1: Which methods exist to improve the efficiency of a transposition table?

In Chapter 2 three methods for improving the efficiency of a transposition table are described. The first method addresses the use of an adequate replacement scheme. When a conflict arises, a replacement scheme decides which positions to keep in the table, and which positions to discard. Experiments show that in this area improvements can still be found. A new replacement scheme, called TwoBig1, based on a two-level table and the number of nodes of the subtree investigated, outperforms all other schemes. It enabled us to solve the game of domineering for several boards, including the standard board. The second method addresses doubling the number of positions in the transposition table. Experiments show that doubling the number of positions is a good method for improving the efficiency of a transposition table. However, beyond a certain table size not much is to be gained from doubling the number of positions. Therefore, the third method concentrates on using the remaining memory not for doubling the number of positions of the table, but for enlarging the size of an entry, by storing more information in an entry. A limited

set of experiments show that – beyond a certain table size – this method gains more than doubling the number of positions in the table, although more experiments are needed to substantiate this claim.

In Chapter 3 proof-number search (pn search) is described. This is a best-first search algorithm, storing the complete search tree in memory. Experiments show that pn search is suitable for solving mate problems in chess. However, there are two drawbacks: (1) a solution cannot be found if the search tree takes up all memory, and (2) identical positions in the search tree (and their subtrees) are doubly searched. These drawbacks are taken care of in Chapters 4 and 5.

Every year there is a large increase in computer speed. Increasing computer speed causes acceleration of search algorithms. A best-first search algorithm (such as pn search) stores the complete search tree in memory. After a relatively short search time no more memory is available since the fast search has generated too many nodes. The increase in computer speed can also be used to do more search at nodes, thereby gaining more knowledge per node. The trade-off transpires in more searching, in favour of less memory to be used. This leads to the formulation of the second problem statement.

Problem statement 2: Which methods exist for best-first search to reduce the need for memory by increasing the search, thereby gaining more knowledge per node?

In Chapter 4 the pn^2-search algorithm is presented. The concept behind this algorithm is that the leaves are not evaluated by an evaluation function, but by a secondary pn-search process. Several experiments with different sizes of the secondary search tree show that much can be gained by choosing the right size of the secondary search tree. The conclusion is that the pn^2-search algorithm is a good method to use the increase in computer speed for additional searching, thereby gaining a better assessment of the values of the leaves.

As mentioned above, in pn search identical positions in the search tree (and their subtrees) are doubly searched. In depth-first search algorithms the re-search of a transposition is avoided by implementing a transposition table. A logical way to avoid the re-search of a transposition in best-first search is to store a transposition only once, thereby transforming the tree into a Directed Cyclic Graph (DCG). However, an important aspect of a position is the path leading to it (the history). Ignoring the history of a position introduces the graph-history-interaction (GHI) problem. This leads to the third problem statement.

Problem statement 3: Is it possible to give a solution for the GHI problem for best-first search?

In Chapter 5 the GHI problem is analyzed in the domain of pn search. A different implementation of a DCG is suggested, and the pn-search algorithm is modified to be able to search this DCG implementation. The new BTA (Base-Twin Algorithm) algorithm is based on the distinction of two types of nodes, termed *base nodes* and

twin nodes. The purpose of these types is to distinguish between equal positions with different history. Experiments with this pn-search algorithm for DCGs confirm our solution of the GHI problem. In the test positions submitted the BTA algorithm solves them all and hence outperforms other attempts to overcome the GHI problem as well as the standard tree algorithm.

Summarizing, the main contributions of this thesis are as follows.

1. The discovery of a new replacement scheme (TwoBig), based on a two-level transposition table and number of nodes of the subtree investigated.
2. Solving the game of domineering.
3. The pn^2-search algorithm.
4. The BTA algorithm (implemented for pn search), solving the GHI problem.

Samenvatting

Geheugen versus zoeken in spelen

In dit proefschrift wordt onderzoek gepresenteerd betreffende de uitwisseling tussen geheugen en zoeken. Het onderzoeksdomein is het domein van de tweepersoons nulsom spelen, in het bijzonder de spelen schaken en domineering. De uitwisseling tussen geheugen en zoeken wint aan belangrijkheid door het beschikbaar komen van meer computergeheugen en meer processorsnelheid.

Computergeheugen wordt steeds goedkoper, en komt daardoor in steeds grotere mate beschikbaar. Een *depth-first* zoekalgoritme (zoals $\alpha\beta$ *search*) gebruikt weinig geheugen. Het resterende geheugen kan bijvoorbeeld gebruikt worden om het heronderzoeken van identieke stellingen, de zogenoemde transposities, te vermijden. Daartoe kan een transpositietabel, die resultaten van voorgaande zoekprocessen bewaart, in het resterende geheugen worden opgeslagen. De uitwisseling zien we terug in het gebruik van meer geheugen, zodat minder hoeft te worden gezocht. Dit resulteert in de formulering van de eerste probleemstelling.

Probleemstelling 1: Welke methoden bestaan er om de efficiëntie van een transpositietabel te verbeteren?

In Hoofdstuk 2 worden drie methoden beschreven om de efficiëntie van een transpositietabel te verbeteren. De eerste methode betreft het gebruik van een adequaat vervangingsschema. Deze methodiek bepaalt bij een conflict welke stellingen wel, en welke niet opgeslagen worden. Experimenten tonen aan dat op dit terrein nog steeds verbeteringen gevonden kunnen worden. Een nieuw vervangingsschema, genaamd TwoBig1, gebaseerd op een *two-level* tabel en het aantal knopen van de onderzochte subboom, presteert beter dan alle andere schema's. Dit schema maakte het mogelijk om het spel domineering op te lossen voor verscheidene borden, waaronder het standaard bord. De tweede methode betreft het verdubbelen van het aantal stellingen in een transpositietabel. Experimenten tonen aan dat het verdubbelen van het aantal stellingen een goede methode is om de efficiëntie van een transpositietabel te verbeteren. Vanaf een bepaalde tabelgrootte valt echter weinig winst meer te behalen met het verdubbelen van het aantal stellingen. De derde methode gebruikt het resterende geheugen daarom niet om het aantal stellingen in de tabel te verdubbelen, maar om het formaat van een tabelingang te vergroten, door meer informatie

in een ingang op te slaan. Een beperkt aantal experimenten toont aan dat – vanaf een bepaalde tabelgrootte – deze methode meer oplevert dan het verdubbelen van het aantal stellingen in de tabel.

In Hoofdstuk 3 wordt *proof-number search (pn search)* beschreven. Dit is een *best-first* zoekalgoritme, dat de gehele zoekboom in het geheugen opslaat. Experimenten tonen aan dat *pn search* geschikt is voor het oplossen van matproblemen in schaken. Er zijn echter twee nadelen: (1) er wordt geen oplossing gevonden als het geheugen vol raakt, en (2) identieke stellingen in de zoekboom (en hun subbomen) worden dubbel onderzocht. In Hoofdstukken 4 en 5 wordt op deze nadelen ingegaan.

De snelheid van computers wordt ieder jaar groter. Vergroting van de computersnelheid betekent automatisch ook versnelling van het zoeken. Een *best-first* zoekalgoritme (zoals *pn search*) slaat de gehele zoekboom op in het geheugen. Na een relatief korte zoektijd is geen geheugen meer beschikbaar omdat het snelle zoekproces teveel knopen heeft gegenereerd. De vergroting van de computersnelheid kan echter ook gebruikt worden om meer te zoeken bij de knopen, waardoor meer kennis per knoop wordt verkregen. De uitwisseling komt terug in meer zoeken, zodat minder geheugen gebruikt hoeft te worden. Dit resulteert in de formulering van de tweede probleemstelling.

Probleemstelling 2: Welke methoden bestaan er voor *best-first* zoekalgoritmen om de vraag naar geheugen te verminderen, door meer te zoeken en daardoor meer kennis per knoop te verkrijgen?

In Hoofdstuk 4 wordt het *pn*^{2}*-search* algoritme gepresenteerd. Het concept achter dit algoritme is dat de bladeren niet door een evaluatiefunctie worden geëvalueerd, maar door een tweede *pn search* proces. Verscheidene experimenten met verschillende grootten van de tweede zoekboom tonen aan dat veel gewonnen kan worden door de juiste grootte van de tweede zoekboom te kiezen. De conclusie is dat *pn*^{2} *search* een goede methode is om de vergroting van de computersnelheid te gebruiken om meer te zoeken, waarbij een betere schatting van de waarden van de bladeren wordt verkregen.

Zoals hierboven is genoemd worden in *pn search* identieke stellingen in de zoekboom (en hun subbomen) dubbel onderzocht. In *depth-first* zoekalgoritmen wordt de heronderzoeking van een transpositie vermeden door het gebruik van een transpositietabel. Een logische manier om de heronderzoeking van een transpositie in een *best-first* zoekalgoritme te vermijden is om de transpositie maar één keer op te slaan, waardoor de boom wordt veranderd in een Gerichte Cyclische Graaf (DCG). Een belangrijk aspect van een stelling is echter het pad dat tot deze stelling leidt (de geschiedenis). Het negeren van van de geschiedenis van een stelling introduceert het *graph-history-interaction* (GHI) probleem. Dit resulteert in de derde probleemstelling.

Problemstelling 3: Is het mogelijk om een oplossing te geven voor het GHI probleem voor *best-first* zoekalgoritmen?

In Hoofdstuk 5 wordt het GHI probleem geanalyseerd in het domein van *pn search*. Er wordt een andere implementatie van een DCG geopperd, en het *pn search* algoritme wordt gewijzigd om het mogelijk te maken om deze DCG implementatie te onderzoeken. Het nieuwe BTA (*Base-Twin Algorithm*) algoritme is gebaseerd op het onderscheid tussen twee typen knopen, genaamd *base nodes* en *twin nodes*. Het doel van deze typen is om een onderscheid te kunnen maken tussen knopen met verschillende geschiedenissen. Experimenten met dit *pn-search* algoritme voor DCGs bekrachtigen onze oplossing van het GHI probleem. Het BTA algoritme lost alle teststellingen op en presteert dientengevolge beter dan zowel andere pogingen om het GHI probleem te overwinnen als het standaard algoritme voor bomen.

Samenvattend kunnen de hoofdbijdragen van dit proefschrift als volgt geformuleerd worden.

1. De ontdekking van een nieuw vervangingsschema (TwoBig), gebaseerd op een *two-level* transpositietabel en het aantal knopen van de onderzochte subboom.
2. Het oplossen van het spel domineering.
3. Het pn^2-*search* algoritme.
4. Het BTA algoritme (geïmplementeerd in *pn search*), dat het GHI probleem oplost.

Curriculum Vitae

Dennis Michel Breuker was born in Amsterdam on Friday the thirteenth of January 1967. From 1979 to 1985 he attended the *Christelijk Lyceum* in Alphen aan den Rijn. After graduation (Atheneum B) he began his study of Computer Science in 1985 at the University of Leiden, specializing in Artificial Intelligence. He worked as a student assistant for the Department of Computer Science from 1987 to 1988. From 1989 to 1991 he created a bidding program for the game of bridge, obtaining his M.Sc. degree under supervision of prof. dr. A. Ållongren and prof. dr. H.J. van den Herik.

In 1991 he started working as a Ph.D. researcher at the Universiteit Maastricht, then called the University of Limburg at the Department of Computer Science. The subject of his research was synthesis of reliable information using the knowledge of experts. Somewhat later he was accepted as NWO researcher (Netherlands Organization for Scientific Research). He started in the domain of automatic hyphenation, but gradually shifted to the domain of computer games, in particular the games of chess and domineering. The research resulted in several publications and this thesis. As of March 1998 he is employed at the division ATS (Advanced Technology Services) of Cap Gemini Nederland B.V. in Utrecht.

SIKS Dissertatiereeks

In 1998 zijn de volgende SIKS-dissertaties verschenen.

98-1 Johan van den Akker (CWI)
DEGAS — An Active, Temporal Database of Autonomous Objects
promotor: prof. dr. M.L. Kersten (CWI/UvA)
co-promotor: dr. A.P.J.M. Siebes (CWI)
promotie: 30 maart 1998

98-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information
promotores: prof. dr. ir. A. Hasman (UM)
prof. dr. H.J. van den Herik (UM/RUL)
prof. dr. ir. J.L.G. Dietz (TUD)
promotie: 7 mei 1998

98-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective
promotores: prof. dr. ir. J.L.G. Dietz (TUD)
prof. dr. P.C. Hengeveld (UvA)
promotie: 22 juni 1998

98-4 Dennis Breuker (UM)
Memory versus Search in Games
promotor: prof. dr. H.J. van den Herik (UM/RUL)
promotie: 16 oktober 1998