Searching for Solutions in Games and Artificial Intelligence

Louis Victor Allis

Version 8.0 of July 1, 1994

Searching for Solutions in Games and Artificial Intelligence

Voor Petra en Cindy

Searching for Solutions in Games and Artificial Intelligence

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Rijksuniversiteit Limburg te Maastricht,
op gezag van de Rector Magnificus, Prof. dr. H. Philipsen,
volgens het besluit van het College van Dekanen,
in het openbaar te verdedigen
op vrijdag 23 september 1994 om 14.00 uur

door

Louis Victor Allis

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

Leden van de beoordelingscommissie:
Prof. dr. P.T.W. Hudson (voorzitter)
Prof. dr. ir. J.L.G. Dietz
Prof. dr. ir. W.L. van der Poel (Technische Universiteit Delft)
Prof. dr. S.H. Tijs
Dr. E. Wattel (Vrije Universiteit Amsterdam)

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Allis, L. Victor

Searching for Solutions in Games and Artificial Intelligence /
L. Victor Allis ; [ill. by the author]. - [S.l. : s.n.]
(Wageningen : Ponsen & Looijen). - Ill.
Thesis Maastricht. - With references. - With summary in Dutch
ISBN 90-9007488-0
NUGI 855
Subject headings: artificial intelligence / games / search.

Cover design: Rob Ferwerda

Contents

List of Tables

List of Figures

Preface

The research presented in this thesis would have been impossible without the help of many persons, whom I want to recognize here.

First of all, I would like to thank Jaap van den Herik for being my teacher. Jaap created an environment generously providing an abundance of learning opportunities. His efforts have been manifold, notably those aimed at teaching me how to write up scientific research as reflected in this thesis. Still, any mistakes remaining are my own.

Administrative complications unfortunately prevented my two auxiliary thesis advisors, Jonathan Schaeffer and Bob Herschberg, from due mention for their essential efforts. I would like to redress the balance. Jonathan Schaeffer's influence on this thesis has several facets. His work on cn-search forms the foundation on which pn-search has been built. During that process, his constant interest in pn-search has led to an increased understanding of the strengths and weaknesses of the algorithm. Furthermore, his comments on earlier versions of this thesis have led to major improvements, most notably in chapter 3. Bob Herschberg has scrutinized many draft versions of this thesis. The ensuing comments and explanations, regarding all different levels of the art of writing, can best be compared with a chess Grand Master introducing a young player to the many intricacies of the game. Bob's efforts have not only greatly improved the thesis at countless points, his guidance has shown me that much remains to be learned. For guiding me along this path, arduous as it may have been, I owe Bob Herschberg sincere thanks.

Besides my three thesis advisors, I want to thank Maarten van der Meulen for the research we did together. His contribution to the development of pn-search has been indispensable. I also want to thank my room mate Dennis Breuker for always being available to discuss new ideas and for all the games he beat me at over lunch. I would like to thank Matty Huntjens for creating order in the chaos of my experiments and Loek Schoenmaker for creating the X-interface for our go-moku program. I want to thank Patrick Schoo for our collaboration on the qubic program. Many thanks also go to Barney Pell. Our email discussions, as well as the times we met in person have been a source of inspiration. I would like to thank my colleagues at the Department of Computer Science of the University of Limburg, for making me feel at home. Furthermore, I would like to thank my colleagues from the AI-group at the Vrije Universiteit, who enabled me to finish this thesis. Moreover, I would like to thank the Foundation for Computer Science Research in the Netherlands (SION) and the Netherlands Organization for Scientific Research (NWO) for their financial support.

Besides the efforts of my thesis advisors, the final version of this dissertation has benefited from valuable suggestions by several people: Ingo Althöfer, Barney Pell, Loek Schoenmaker, Mark Willems and the members of the *beoordelingscommissie*.

Finally, I want to thank my family and friends for their stimulating interest in my research. Most important of all, however, has been the continuous support and stimulation of my wife Petra and my daughter Cindy.

Victor Allis
Boukoul, July 1994

Chapter 1

Introduction

In this thesis "intelligent" games are investigated from the perspective of Artificial Intelligence (AI) research. In this chapter the relevance of such investigations is discussed, leading to the formulation of a problem statement.

1.1 Speculations and AI

All through history, mankind has been fascinated by the thought of creating machines to perform the most difficult of tasks. Men of every era have dreamt of and speculated about achievements beyond the scope of the technology of their time. Yet, when confronted with a machine performing tasks at an unexplained high level, many willingly believed that science and technology had made it possible, instead of doubting the genuineness of the machine's results. For example:

In 1769, Wolfgang von Kempelen demonstrated his chess-playing automaton, the *Turk*, to the world (Carroll, 1975). It was the first machine to create the illusion of having mental abilities: playing chess at a high level. Among its successes was a victory over the Prussian king Frederick the Great. For many years, large numbers of people believed that the *Turk* was a true thinking machine, even though the technology of the 18th century did not hint at how such a machine could have been created. For exactly that reason, many others believed that the *Turk* had to be a fraud. Nevertheless, the secret of the small human chess-player hidden inside the *Turk* was well-kept until 1834.

With the creation of modern computers, the field of Artificial Intelligence emerged as a new focal point for speculations. Some of these speculations have been made by scientists working within the field, while others have been made by laymen, such as those working in the motion-picture industry. For instance, movies such as *2001: A Space Odyssey*, *Star Wars* and *War Games* feature computers (resp. HAL, R2-D2 and C-3PO, and WOPR) which seem to have minds of their own. The impact of these truly artificially intelligent entities, fictitious as they may be, on the perception of AI research by the public at large is considerable. Predictions presented by leading scientists in the field reinforce the image created by movies and science-fiction authors. As an example we refer to the Inaugural Lecture delivered by Van den Herik in which he raised the question whether computers will be able to decide issues of law (Van den Herik, 1991). Irrespective of Van den Herik's estimation of several hundreds of years necessary to create an artificial judge, the spin-off of such speeches in terms of nation-wide coverage by newspapers, radio and television strengthens the general public's idea that the creation of artificially intelligent entities is within close range.

It is important to distinguish clearly between the *state-of-the-art* in AI and *speculations* concerning future achievements. We present three well-known examples of progress in AI, each of which has led to unjustified speculations:

1. Newell *et al.* (1957) created the General Problem Solver, a new control metaphor for representing and solving problems. The name of their system led to speculations concerning the creation of a truly general problem solver. More than three decades later AI has not produced anything near such a goal.

2. In 1959, Samuel created his learning checkers[1] program which won a game against a human master player (Samuel, 1959; Samuel, 1967). From this single game, it has been wrongly concluded by many that an artificial master checkers player had been created, while some even believed that the game of checkers had been "solved" (Schaeffer *et al.*, 1991). Samuel's work on learning is classical within AI but only recently have programs begun to compete with the best human checkers players (Schaeffer *et al.*, 1992).

3. The medical diagnostic expert system MYCIN determines the infectious agent in a patient's blood, and specifies a treatment for the infection (Shortliffe, 1976; Buchanan and Shortliffe, 1984). Despite the promise

[1] In this thesis we shall use the name checkers for the game played on an 8×8 board, which is called checkers in the United States of America, and draughts in Great Britain. We reserve the term draughts for the game played on a 10×10 board.

created by successes such as MYCIN, the development of expert systems has been hindered by many problems, such as the knowledge-acquisition bottleneck (Feigenbaum, 1979). Speculations regarding machines replacing doctors of medicine so far lack a scientific basis.

The three examples illustrate that AI research in the last decades of the twentieth century is not directly involved in creating true intelligence. Instead, many of the stumbling blocks on the road to such a goal are now themselves the main subject of investigation. Only when these obstacles are removed may we start looking for the goal implicit in the name of the field.

1.2 Identifying the obstacles

It is believed by many scientists that the main hurdle to be cleared when creating artificial experts in practical domains is *common-sense knowledge* (Marr, 1977). Where humans are extraordinarily well equipped to acquire common-sense knowledge with their five senses, computers are deficient in this area. Despite efforts in areas such as computer vision, robotics, speech processing etc., no computer program exists which exhibits even a basic understanding of the real world (Marr, 1977). This lack of knowledge severely handicaps computers in becoming experts in any real-world domain, such as medicine, law, manufacturing etc. A direct consequence is the failure in dealing with natural languages. In conversations between human beings many things are left unsaid without hindering the participants. The gaps are filled by common-sense knowledge and sentences are interpreted within the context of our world view. Many AI researchers thus believe that common-sense knowledge is a vital ingredient for natural language processing (Charniak, 1978).

Another area where nature has been generous to humans is learning. Humans continuously learn from their experiences, much unlike computer programs. Whereas learning is an automatic built-in feature of infants, it is difficult to realize in computer programs, despite the efforts spent on machine learning (Michalski *et al.*, 1983; Michalski *et al.*, 1986).

The lack of common-sense knowledge and of learning have a large impact on what computers can and cannot do. Besides these known obstacles, we may wonder whether other, hidden obstacles hinder progress in AI. For instance, some argue that intuition is a human quality which cannot be implemented (De Groot, 1965), while others believe that intuition is simply a name for rule-based behavior where the rules are not accessible to consciousness (Michie, 1982). Thus, while some consider intuition to be unattainable for computers, others stress that to implement intuition, all we need to do is to uncover the rules at its basis. In general, it is of interest to know as many of the main obstacles hindering progress in AI as possible. It remains in dispute whether intuition should be regarded as such.

1.3 Uncovering hidden obstacles

Some new obstacles for AI research may become visible only after we have successfully dealt with the obstacles apparent today. Others may be discovered by concentrating on a set of domains where known obstacles play no role of importance, such as the domain of games. Many games, such as chess, checkers, go and bridge possess the property that they create a micro world (Van den Herik, 1983), in which common-sense knowledge and natural languages are not relevant. Instead, a small set of rules determines all possible states within the micro world. And yet, in most of these games, humans are (still) superior to their artificial counterparts. The game of go is a striking example: today's strongest go programs have reached a mere novice level.

By investigating a game, we envision two possible outcomes.

- If we achieve a playing strength sufficient to defeat the best human players, analysis of the means which led to this improvement may uncover new AI techniques.
- If the playing strength keeps falling short, even after prolonged attempts, of that of the best human players, a better understanding of the problems inherent in playing the game at a high level may be acquired.

We remark that the possibility remains that the results do not lead to progress (i.e., to new AI techniques or a better understanding of the inherent problems). In the first case, the improvement may be due to entirely domain-specific techniques which cannot be generalized to AI techniques (Dreyfus, 1980). In the second case, we may find that we have difficulty in isolating the problems from our failed attempts. Although a lack of progress may occur in some cases, by investigating a representative set of games in this way the probability increases that new AI techniques are developed or insight into problems hindering progress is obtained.

If similar problems are found in several different games, it may help us to uncover obstacles which are likely to exist in real-world domains as well.

It could also lead to an understanding of the restrictions of the techniques applied. We list two examples of this last phenomenon.

- After the rapid increase in playing strength of computer chess programs in the seventies and eighties, it was suggested that an increase of the search depth by an extra ply (i.e., a move by one player), was equivalent to an increase in playing strength of approximately 200 ELO points (Thompson, 1982). Now that progress in playing strength has slowed down, investigations in the relation between search depth and playing strength for checkers indicate that the added strength per ply diminishes for deeper searches (Schaeffer, 1993b). Furthermore, positions have occurred in tournament games where a search of 60 ply would be necessary to stand up against human knowledge (Schaeffer, 1993b). Because such searches are by far out of reach of current technology, it has become clear that added knowledge is a vital ingredient to world-champion level checkers and chess programs.
- In the early days of AI research, many new weak methods (i.e., using little domain-specific knowledge) were demonstrated to succeed on toy problems (Winston, 1992). It was believed that through deeper search the results on toy problems could be extrapolated to real-world problems. This has proved to be more difficult than anticipated. Using sufficient domain knowledge, state spaces can be reduced such that problems become solvable. However, when vital knowledge is excluded the explosion of possibilities makes many such problems intractable.

We postulate that when investigating sufficiently complex games with the goal of outperforming human beings, success is likely to yield new AI techniques as their products, while failure presents a better understanding of problems and obstacles encountered. This observation is the basis of the problem statement presented in the next section.

1.4 The problem statement

In this thesis, we consider games which have the following five properties. Examples of games which have these properties include chess, checkers, go and bridge.

1. *Two-player.* Most games are two-player games, as opposed to zero-player games (e.g., Conway's life (Berlekamp *et al.*, 1982)), one-player

games (e.g., the 15-puzzle (Korf, 1985), Rubik's cube and peg solitaire (Beasley, 1985)) and multi-player games (e.g., poker and diplomacy (Hall and Loeb, 1992)).

2. *Zero-sum.* These are games where one player's loss is the other player's gain. The prisoners' dilemma (Hofstadter, 1985) when considered as a game is not zero-sum.

3. *Non-trivial.* A best playing strategy should not be trivially establishable through enumeration or mathematical analysis. Examples of trivial games are tic-tac-toe and nim.

4. *Well-known.* These are games which have been played by large numbers of people, resulting in the game being known in several countries. This excludes many mathematical games, and obscure variations on well-known games (such as give-away chess).

5. *Requiring skill.* Some games serve mainly as a pastime, not requiring much skill. The more experienced player has no real advantage in those games, except maybe against novices (examples are many simple card games played by children). The games included here should exhibit a strong relation between skill and winning chances. Such a relation also exists in some games which are influenced by a chance element, such as backgammon and bridge, which are thus included.

The first two properties (two-player and zero-sum) are selected to ensure that cooperation between players can be excluded from the investigations. The third property (non-trivial) is necessary for us to have something to investigate. The last two properties (well-known and requiring skill) ensure that the results of our investigations can be checked (for instance against strong human players) and evaluated.

To be more specific, we list the set of games played at the Computer Olympiads which fulfill all these criteria (Levy and Beal, 1989; Levy and Beal, 1991; Van den Herik and Allis, 1992). This list of games will henceforth be called the *Olympic List*.

> awari, backgammon, bridge, chess, Chinese chess, checkers, connect-four, draughts, go, go-moku, nine men's morris, othello, qubic, renju, scrabble.

We do not claim that the fifteen games of the Olympic List are the only games satisfying the five properties listed above. However, as long as sufficient challenges exist for the listed games, there is no need to try to be complete.

We are now ready to present our *problem statement*, consisting of two questions.

Through an investigation of games of the Olympic List,

1. which new AI techniques can be developed and
2. which obstacles for AI research will emerge?

The goal of this thesis is to find an answer to these questions. To this end, we list below three detailed research questions, distinguishing between performance levels of systems which may be the result of investigating games of the Olympic List.

1. Which games can be *solved* (see section 1.5) and what techniques may contribute to the solution?

2. For which games can we create programs outperforming the best human players in the near future, and what techniques contribute to their performance?

3. In which games will humans continue to reign in the near future (say, at least the next decade) and what are the main obstacles to progress for computer programs?

Our attempts to answer these three questions have guided the research efforts described in this thesis.

Before we give an outline of the thesis in section 1.6, we must clarify the term *solved* in relation to games. As there is no consensus about this term, we will give a definition in section 1.5.

1.5 Solving games

Stating that a game is *solved* usually indicates in common parlance that a property with regard to the outcome of the game has been determined. Even for two-player zero-sum games with perfect information (see section 6.2), at least three different definitions could be meant, which we name *ultra-weakly solved*, *weakly solved* and *strongly solved*. The first two terms have been suggested by Paul Colley, while the third term has been suggested by Donald Michie.

ultra-weakly solved For the initial position(s), the game-theoretic value has been determined.

weakly solved For the initial position(s), a strategy has been determined to obtain at least the game-theoretic value of the game, for both players, under reasonable resources.

strongly solved For all legal positions, a strategy has been determined to obtain the game-theoretic value of the position, for both players, under reasonable resources.

We remark that the reasonable resources mentioned may be a subject of discussion. The size of the resources is meant only to give an approximate indication of the time and computing equipment allowed for reproducing a solving strategy. Without these restrictions, it could be argued that, for instance, chess could be weakly solved. As a strategy to solve chess, an α-β search through the full game tree suffices. The reasonable resources mentioned should typically allow the use of a state-of-the-art computer and several minutes of computation time per move.

The definition of ultra-weakly solved indicates that, at the start of the game, it is known what the outcome of the game would be with optimal play by both sides. It is not necessarily known how either player can achieve the optimal outcome. The game of hex, for instance, is known to be a first-player win on all diamond-shaped boards, although no constructive strategy has been determined. The game-theoretic value has been established by noting that the game does not permit draws and that having an extra move cannot be a disadvantage. Thus, since the first player does not need to lose, hex is a first-player win. This reasoning has not (yet) led to a winning strategy for the first player, which makes it of little use to practical play.

It is well-known that tic-tac-toe is a game-theoretic draw. A player who has weakly solved tic-tac-toe only needs to be able to achieve a draw, in every game she[2] plays. It is not necessary for her to win against a non-optimally playing opponent, when she is given a winning opportunity.

The definition of strongly solved demands a strategy not just from the initial position(s), but from all legal positions. Thus, against a non-optimally playing opponent, each mistake must be capitalized upon. Examples of strongly-solved games are tic-tac-toe, nim (Knuth, 1969) and many chess

[2]In contexts where the gender of a non-neutral third person is irrelevant, we will always use "she" and "her" to avoid the more cumbersome "s(he)" and "her/his".

endgames (Van den Herik and Herschberg, 1985; Thompson, 1986; Stiller, 1989).

An ordering exists between the three definitions. Any strongly-solved game, is also weakly solved, while a weakly-solved game is also ultra-weakly solved. To see the latter, it suffices to play a single game from each initial position of the game, with both sides played by the system which solved the game. The outcome of such a game is guaranteed to be the best attainable by both players, equaling the game-theoretic value of the game.

In any domain for AI research, evaluation of the practical performance of the systems produced is essential. The natural performance test of a game-playing system is a match consisting of a large number of games against a rated opponent. When claiming that a program has solved a game, it seems reasonable to require the program to exhibit skill in such a match. A program which has ultra-weakly solved a game does not guarantee being capable of playing the game at all. A program which has weakly solved a game will at least draw every match it plays (while it plays both sides equally often). Note, however, that for games where the program has shown the game to be a win for the stronger side, it is not guaranteed to exhibit any skill when playing the weaker side. The guaranteed performance level, i.e., ensuring that no single match is lost, is in our opinion sufficient to declare a game solved.

In this thesis, we consider a game *solved* when it is at least weakly solved.

1.6 Thesis outline

In 1988, research performed for a Master's thesis (Allis, 1988) led to solving connect-four, published as (Uiterwijk *et al.*, 1989a). Inspired by this result, we decided to start with the first research question, i.e., determining which other games of the Olympic List can be solved, and identifying techniques which contribute to their solution. In particular, of the fourteen remaining games of the Olympic List (i.e., excluding connect-four), we have selected four which seemed eligible for solution. These games are awari, qubic, nine men's morris and go-moku. Awari and nine men's morris are selected for their relatively small state-space complexity (see chapter 6), while qubic and go-moku are selected since human experience indicates that the first player has an overwhelming advantage. As Ralph Gasser has been investigating nine men's morris concurrently with our research (Gasser, 1991), we have concentrated on awari, qubic and go-moku.

During investigation of these games, two new search techniques have been

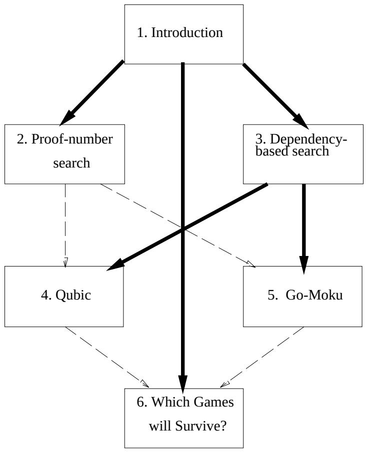


Figure 1.1: The interdependencies of chapters

developed, viz. *proof-number search* (pn-search) and *dependency-based search* (db-search). While db-search forms the basis for solving qubic and go-moku, pn-search is an important contributing factor. Although our investigations showed that applying pn-search to awari leads to promising results, awari has not (yet) been solved. The results of our investigation of the first research question are described in chapters 2 through 5.

In chapter 6 the second and third research questions are investigated leading to an evaluation of the problem statement.

The thesis is organized as follows. It consists of four parts, the first of which is this introduction. The second part consists of chapters 2 (Proof-Number Search) and 3 (Dependency-Based Search), containing descriptions of the two search techniques developed in the course of this research. Both techniques are presented independently of their application to games. Chapters 2 and 3 can each be read independently of other parts of the thesis and are of special interest to those researchers who would like to apply the techniques to their own research domains.

The third part of the thesis consists of chapters 4 (Qubic) and 5 (Go-Moku), each describing the solution to the game under investigation. Although it is recommended to read chapter 2 before any of the game-specific chapters, proof-number search is not essential foreknowledge. Dependency-based search forms the basis for solving qubic and go-moku. It is, therefore, necessary to read chapter 3 before starting on chapters 4 and 5.

The fourth part of the thesis consists of chapter 6 (Which Games Will Survive?), in which all games of the Olympic List are investigated. For each game, we determine the value of four game properties, describe the state of the art in game-playing programs, list the techniques applied and the obstacles to progress. Next we evaluate our research with respect to the problem statement. Predictions regarding the future of games conclude the chapter. The fourth part of the thesis can be read independently of the second and third parts, although it is recommended that the reader first obtains some knowledge of the contents of these parts.

The interdependencies between the chapters are pictured in figure 1.1. An arrow from chapter A to chapter B indicates that A is essential foreknowledge for B. A dashed arrow between chapters A and B indicates that it is recommended, but not essential, to read A before B.

Chapter 2

Proof-Number Search

2.1 Knowledge representation and search

Problem solving is one of the corner-stones of AI research. Within problem solving, we distinguish two subprocesses: choosing a knowledge representation and performing a search. We remark that the term knowledge representation is meant to include analysis, conceptualization and formalisation. A well-chosen representation may considerably reduce the amount of search needed to solve a problem, while a badly chosen representation may render solving a problem (virtually) impossible. As an example we present the mu-puzzle (Hofstadter, 1979).

> A production system consisting of four rewriting rules generates theorems consisting of the letters M, I and U. In each production, x and y denote any string of letters.
>
> 1. $x\mathrm{I} \rightarrow x\mathrm{IU}$
> 2. $\mathrm{M}x \rightarrow \mathrm{M}xx$
> 3. $x\mathrm{III}y \rightarrow x\mathrm{U}y$
> 4. $x\mathrm{UU}y \rightarrow xy$
>
> The goal of the mu-puzzle is to determine whether MU is a theorem in the above system, given that MI is the only axiom.
>
> In a first attempt to solve the puzzle, we represent a theorem simply by its string of letters. The rewriting rules are used to expand nodes of the search tree, where each node represents a theorem. We are now faced with a tree-search problem: to

> find a path of rewriting rules leading from the initial state MI to the goal state MU. A suitable tree-search algorithm is selected to perform the search, such as breadth-first search or depth-first search. To select a search algorithm, various criteria may be applied. For instance, breadth-first search guarantees that the first solution found is also the shortest solution (Nilsson, 1980). A disadvantage of breadth-first search is that it requires more working memory than an algorithm such as depth-first search (Nilsson, 1980). Generally, each of the applicable search algorithms has its own advantages and disadvantages. In case no solution exists, these algorithms have the disadvantage that the search will not terminate, as the set of theorems is infinite.
>
> Instead of concentrating on the selection of the best possible search algorithm, we may first try to optimize the chosen representation. For the mu-puzzle, a better representation involves an extra item of knowledge per theorem. This Boolean item, which we name ISTRIPLEI, indicates whether the theorem's total number of Is is a multiple of three. We can now verify that each of the four rewriting rules creates new theorems with ISTRIPLEI's value equal to that of the theorem it is created from. The observation that MI (false) and MU (true) have unequal ISTRIPLEI values is sufficient to prove that MU is not a theorem.

In the mu-puzzle example, it was possible to eliminate all search by enhancing the representation of the puzzle. It illustrates that choosing a representation should have the highest priority when solving problems. Choosing a knowledge representation in problem solving is mostly domain-specific. Even though general techniques (such as abstraction, here applied to the mu-puzzle) exist, their successful application remains the fruit of a thorough understanding of the domain under investigation.

For problems more complex than the mu-puzzle, a good representation generally does not eliminate all search; it merely reduces the size of the state space to, hopefully, manageable proportions. It is then important to select a search algorithm which will find a solution, if it exists, in an efficient manner. The efficient manner is to be understood here in a broad sense, including programming time, calculation time and the required amount of working memory. The weighting of these resources depends on the circumstances in which the problem has to be solved.

Thus, the domain-specific task of finding a suitable knowledge representation is performed in combination with the selection of a search algorithm well-suited for the state space. In the course of a considerable number of years of research in AI, many different search algorithms have been developed. We distinguish between several *categories* of search problems, such as those represented by single-agent trees, AND/OR trees and game trees (Nilsson, 1971). While the category that a search problem belongs to restricts our choice of search algorithms, *within* each category several search algorithms exist, each with its own characteristics. These characteristics determine the scope of problems for which the algorithm may be preferred over the other algorithms within the same category. We remark that the division into search categories is not strict. An example relevant to this thesis is that two-valued game-tree searches can also be performed by search algorithms for AND/OR trees.

For the category of game trees, many different search algorithms have been developed. We name the best known algorithms and mention the type of search problems for which we believe they are best suited:

- By far the best-known game-tree search algorithm is α-β search (Knuth and Moore, 1975). It is a *directional* (also known as *depth-first*) algorithm, having working-memory requirements linear in the depth of the tree investigated. Knuth and Moore (1975) have shown that α-β search achieves optimal efficiency on perfectly-ordered uniform trees. Application of iterative deepening to α-β search ensures for many application domains that strongly-ordered trees are traversed, resulting in close-to-optimal efficiency on uniform trees (Campbell and Marsland, 1983).

- SSS* is a *best-first* search algorithm (Stockman, 1979). It will never investigate a node pruned by α-β search (Campbell and Marsland, 1983).

  The algorithm has two drawbacks. First, as with all best-first search algorithms, the working-memory requirements are linear in the number of nodes created, thus exponential in the depth of the tree. However, recently variants requiring less working memory have been developed (Reinefeld, 1994). Second, the reduction in the number of nodes searched compared with iterative-deepening α-β search does not outweigh the cost of maintaining the search tree (or open list) in working memory for most practical applications. However, if the cost of heuristic evaluation is large compared to the cost of traversing the tree, or if obtaining a good ordering through iterative deepening for α-β search is difficult for the domain under investigation, SSS* may be an alternative to be preferred.

- Another best-first search algorithm is B* (Berliner, 1979). It depends on the availability of reliable heuristic estimates for the upper and lower bounds on the value of internal nodes. For chess, the algorithm has been implemented in HITECH, but it remains unclear whether for this domain sufficiently accurate upper and lower bounds can be estimated to result in better move selection than by algorithms based on α-β search.

- Conspiracy-number search (cn-search) (McAllester, 1988; Schaeffer, 1989) is a best-first search algorithm which determines the cardinality of the smallest sets of (terminal) nodes which must change their value in order to change the value of the root. Once this cardinality grows beyond a pre-specified bound, it is considered unlikely that the root value will change, and the search is terminated. Cn-search has shown its merits in tactical chess positions (Schaeffer, 1989), but has failed in a comparison with α-β search in a tournament chess program (Van der Meulen, 1990). Cn-search has as disadvantages the large amount of bookkeeping necessary at each node, and the subsequent amount of working memory required to perform the search. One of the ideas underlying cn-search is that the distribution of the values over the leaf nodes of the tree, and the shape of the tree, should influence the selection of the next node to be investigated.

The last aspect of cn-search, using the shape of the tree to guide the search, has been singled out in *proof-number search* (pn-search), which can be seen as a successor to conspiracy-number search. In this chapter we present pn-search, which has the exploitation of non-uniformity as its main theme. Pn-search will be presented as an AND/OR tree search algorithm, even though all applications discussed in this thesis concern game trees.

We introduce in section 2.2 the pn-search algorithm for AND/OR trees. In section 2.3, several enhancements to the algorithm are presented. These include techniques to reduce execution time and usage of working memory, examples of the application of domain-specific knowledge, and a discussion regarding transpositions within pn-search. Results of applying pn-search to a practical domain, the game of awari, are presented in section 2.4, where its performance is compared with those of sophisticated implementations of

α-β search. Finally, section 2.5 contains a discussion of related algorithms, analyzing the similarities and differences between pn-search and conspiracy-number search, SSS*, B* and A*. (A* (Hart *et al.*, 1968), a single-agent search algorithm, has been included in this list because of its similarities with pn-search.)

2.2 Pn-search: the algorithm

In this section we introduce pn-search for AND/OR trees. First, in section 2.2.1 we define our tree model and a precise terminology for the remainder of the chapter. Then, the main assumptions of pn-search are described in section 2.2.2 and the notions of proof numbers and disproof numbers are introduced. Next, section 2.2.3 informally discusses the order in which the nodes of a pn-search tree should be created. Finally, an algorithm in pseudo-code for pn-search is presented in section 2.2.4.

2.2.1 The AND/OR-tree model

We define our tree model as follows. In the tree, there are two *types* of nodes: AND nodes and OR nodes. We assume that each node can be *evaluated*, leading to one of three values: false, true or unknown. Please note the difference between nodes which have not yet been evaluated (thus whose evaluation value is not yet known) and nodes which *have* been evaluated and obtained the value unknown.

Nodes with evaluation value unknown can be *expanded*. When a node J is expanded, a non-empty set of *child* nodes is created, each having J as *parent* node. A node which has been expanded is an *internal* node. There are three kinds of *leaf* nodes, i.e., nodes without children. First, a node evaluated to false or true is a *terminal* node. Second, a node which has evaluated to unknown is called a *frontier node*. Third, a node which has not yet been evaluated is also called a *frontier node*.

There are two *tree-creation procedures*, which we name *immediate evaluation* and *delayed evaluation*. When applying immediate evaluation each node in the tree is immediately evaluated upon creation. The tree is initialized by creating (and evaluating) the root. Then, as long as the tree has not been solved, at each step a frontier node is selected (which, since it has already been evaluated, must have value unknown), expanded and all its children are immediately evaluated. This process of expanding a node J and evaluating J's children is called *developing* node J. In case of delayed evaluation, each node is only evaluated when it is selected, instead of at creation. Thus, the tree is initialized by creating the root (without evaluation). Then, at each step a frontier node J is selected (which is guaranteed not to have been evaluated) and evaluated. If the evaluation value of J is unknown, J is expanded (without evaluating J's children). Here the process of evaluating a node, possibly followed by its expansion, is also called *developing* node J. We remark that the terms *frontier node* and *developing* each have a double meaning. However, once the tree-creating procedure has been specified, both terms are unique. This approach has been chosen so that pn-search can be explained independently of the tree-creation procedure.

The value of an expanded internal AND node A is determined as follows: if A has at least one child with value false, A also has value false; otherwise, if A has at least one child with value unknown, A has value unknown; otherwise A has value true. The value of an expanded internal OR node O is determined as follows: if O has at least one child with value true, O also has value true; otherwise, if O has at least one child with value unknown, O has value unknown; otherwise O has value false. A tree is *solved* if the value of its root has been established as either true or false. A solved tree with value true is called *proved*, while a solved tree with value false is called *disproved*.

Throughout this chapter, we depict AND nodes by circles and OR nodes by squares in each of the figures. Furthermore, AND nodes can be recognized by the arcs linking their children, in accordance with standard conventions for depicting AND/OR trees.

2.2.2 Main assumptions of pn-search

Best-first search algorithms select a best node (according to some criterion) in the search tree, develop the node and then update such information as is necessary for the algorithm to continue. The distinguishing factor of each best-first search algorithm is the manner in which a node is characterized as 'best'.

For pn-search we assume that we have no knowledge regarding *a priori* probable values of nodes, nor knowledge regarding correlations between node values, although this knowledge could be added to the program (see section 2.3.3). Instead, only the position of a node in the tree and its possible contribution to solving the tree is considered.

First, we formulate the assumptions of pn-search, implying the above. Second, we present some definitions to aid in the description of pn-search. Third, using an example, we illustrate that some nodes are better in their contribution to solving the tree than others. Finally, we summarize our findings.

Assumptions

While searching AND/OR trees, we make the following two assumptions.

1. The probability distribution of values (true, false, unknown) for a frontier node is unknown.

2. The probability distribution of values (true, false, unknown) for a frontier node is equal throughout the tree.

Even though these assumptions mean that we cannot distinguish between two nodes by looking at them independently of their context, nevertheless their position in the tree may influence their expected contribution to solving the tree.

Definitions

When searching AND/OR trees, developing a single frontier node is often insufficient to solve the tree. In most cases, several frontier nodes must obtain the value true to prove the tree or the value false to disprove it. This observation is reflected in definitions 2.1 and 2.2.

Definition 2.1 *For any* AND/OR *tree T a set of frontier nodes S is a* proof set *if proving all nodes within S proves T.*

Definition 2.2 *For any* AND/OR *tree T a set of frontier nodes S is a* disproof set *if disproving all nodes within S disproves T.*

Since it will turn out that we shall use the cardinality of proof and disproof sets, these are given names in definition 2.3 and 2.4.

Definition 2.3 *For any* AND/OR *tree T, the* proof number *of T is defined as the cardinality of the smallest proof set of T.*

Definition 2.4 *For any* AND/OR *tree T, the* disproof number *of T is defined as the cardinality of the smallest disproof set of T.*

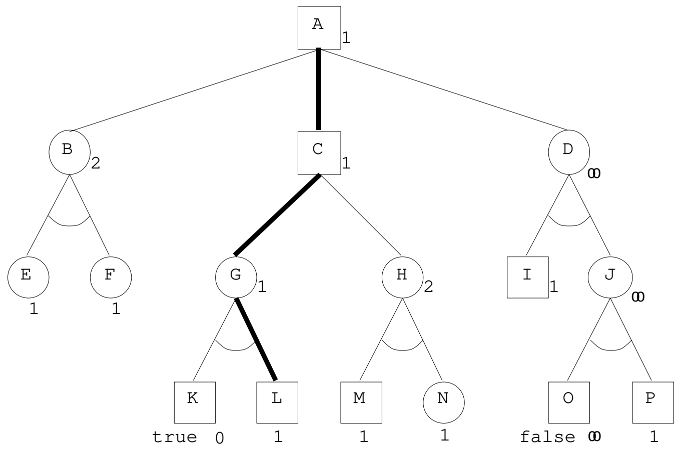


Figure 2.1: AND/OR tree with proof numbers.

Examples

To illustrate how the context can be used to distinguish between nodes, we have depicted an AND/OR tree in figure 2.1.

With each node, we have associated the *proof number* of the subtree with that node as its root, as defined in definition 2.3.

All frontier nodes (E, F, I, L, M, N and P in figure 2.1) have proof number 1. This follows from the fact that only the node itself needs to obtain the value true to prove the whole subtree (consisting of only the node itself). A terminal node with value true (node K in figure 2.1) has proof number 0, since its value has already been proved. Terminal nodes with value false (node O in figure 2.1), have proof number ∞, since there is no smallest finite set of nodes which can undo the fact that the node is disproved. Internal AND nodes obtain the value true only if all their children are proved. Thus, internal AND nodes (B, D, G, H and J in figure 2.1) have proof numbers equal to the *sum* of the proof numbers of their children. For internal OR nodes it suffices to prove one of their children, in order to have the parent obtain the value true. Thus, for internal OR nodes (A and C in figure 2.1) we establish the proof number by taking the *minimum* of the proof numbers of their children.

Root A of the tree in figure 2.1 has proof number 1. This indicates that somewhere in the tree a frontier node exists, which, by obtaining the value true, would complete the proof of the tree. The path from the root to this frontier node can be found by examining the proof numbers. To prove the

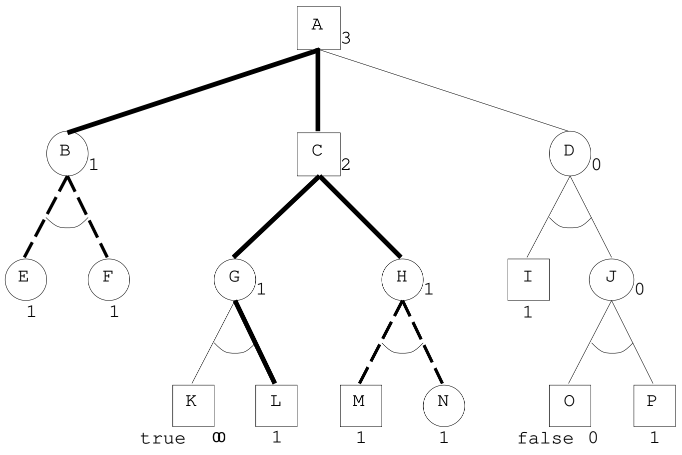


Figure 2.2: AND/OR tree with disproof numbers.

root (an OR node), it is sufficient to prove one of its children. Child C has the smallest proof number among the three children of A. The frontier node we are looking for thus lies within subtree C. In the same way, node G is preferred over node H, since G's proof number is equal to 1, while H's proof number equals 2. To prove node G (an AND node), it is necessary to prove all its children. Child K has already been proved, thus only a proof of node L is needed, which is the frontier node we have been looking for.

We could now proceed and develop node L, in an attempt to prove the tree. Instead, we will first determine which nodes may contribute to a potential disproof. In figure 2.2 we have depicted the same tree as in figure 2.1. With each node, we have associated the *disproof number* of the subtree with that node as root, as defined in definition 2.4.

The disproof numbers behave analogously to proof numbers, interchanging the roles of AND nodes and OR nodes, and the cardinalities 0 and ∞. Thus, frontier nodes (E, F, I, L, M, N and P in figure 2.2) have disproof number 1. A terminal node with value false (node O in figure 2.2) has disproof number 0, since it is already disproved. Terminal nodes with value true (node K in figure 2.2) have disproof number ∞. Internal AND nodes (B, D, G, H and J in figure 2.2) have disproof numbers equal to the *minimum* of the disproof numbers of their children. Internal OR nodes (A and C in figure 2.2) have disproof numbers equal to the *sum* of the disproof numbers of their children.

Root A of the tree in figure 2.2 has disproof number 3. This means that at least 3 nodes must obtain the value false to disprove the tree. Analysis of the tree shows that it involves one of the nodes E and F, node L and one of the nodes M and N.

Summary

The previous paragraphs illustrate that proof numbers and disproof numbers can be used to find nodes within the smallest subset of frontier nodes in the tree which, by all obtaining the same value, solve the tree.

From the assumptions underlying pn-search it follows that the probability that all nodes in a proof set obtain the value true increases with decreasing cardinality of the proof set (except in the trivial cases that the probability of evaluation to true equals either 0 or 1). As a result the total number of node developments needed to solve a tree is (on the average) reduced by first focusing on potential solutions involving a small number of nodes (i.e. subtrees with small proof and/or disproof numbers), before trying to find solutions known to require a larger number of nodes. This expectation is the basis for the pn-search algorithm as described in the following sections.

2.2.3 Informal description of pn-search

Pn-search continuously tries to solve the tree by focusing on the potentially shortest solution, i.e., consisting of the least number of nodes. At each step of the search, a node which is part of the potentially shortest solution available is selected and developed. After the development of a node, its proof number and disproof number are established anew. Then, the proof and disproof numbers of its ancestors are updated. This process of selection, development and ancestor updating is repeated until either the tree is solved or we have run out of resources (time or working memory).

The main issue yet to be resolved is to decide (1) to select a node in the smallest *proof* set, or (2) to select a node in the smallest *disproof* set. We will show in the following paragraphs that, maybe surprisingly, we can always do both at the same time. This results in the definition of a most-proving node as in definition 2.5.

Definition 2.5 *For any* AND/OR *tree T, a* most-proving node *of T is a frontier node of T, which by obtaining the value* true *reduces T's proof number by 1, while by obtaining the value* false *reduces T's disproof number by 1.*

Definition 2.5 assumes that within each unsolved tree T a frontier node exists, which is an element of the intersection of a smallest proof set and of a smallest disproof set of T. A stronger claim is that *each* pair consisting of a smallest proof set and a smallest disproof set has a non-empty intersection. We prove this stronger claim by induction.

Proof

- *Basis*
  For each frontier node J the singleton set containing J is both the only proof set, and the only disproof set. The intersection of these two sets contains node J and thus is not empty.

- *Induction step*
  Suppose that the assumption has been proved for all children J_1, .., J_n of an internal AND node J. To disprove J, only one child needs to be disproved. Let disp(J_x) be any disproof set of J_x which has minimal cardinality among all disproof sets of children of J. Then disp(J_x) is also a minimal disproof set of J. To prove J, all children must be proved. Let prove(J_i) ($1 \leq i \leq n$) be arbitrary minimal proof sets for each of the children J_i. Then $\bigcup_{i=1}^{n}$ prove(J_i) is a minimal proof set of J, which we name prove(J). Thus disp(J_x) is a minimal disproof set of J, and prove(J_x) is contained in a minimal proof set of J. As disp(J_x) and prove(J_x) are minimal disproof and proof sets of J_x, they have a non-empty intersection according to the induction assumption. Thus disp(J) and prove(J) have a non-empty intersection.

  The proof for internal OR nodes proceeds analogously.

□

We conclude that there is no conflict of strategies between trying to prove or to disprove the tree: by repeatedly selecting a most-proving node, both strategies are executed simultaneously, without one strategy delaying the other. How to select the most-proving node using proof and disproof numbers is illustrated with an example tree.

Below each node of the tree depicted in figure 2.3, we have depicted its proof number and disproof number (in that order). Thus, the least number of nodes which must be developed to prove the tree is 3. The same number of nodes is needed to disprove the tree.

First, let us analyze the effort necessary to disprove the tree. As node A is an OR node, it will only obtain value false if both children obtain value false. In other words, both children must be solved (with value false) to disprove the tree. Thus, in both subtrees frontier nodes exist which are part of the

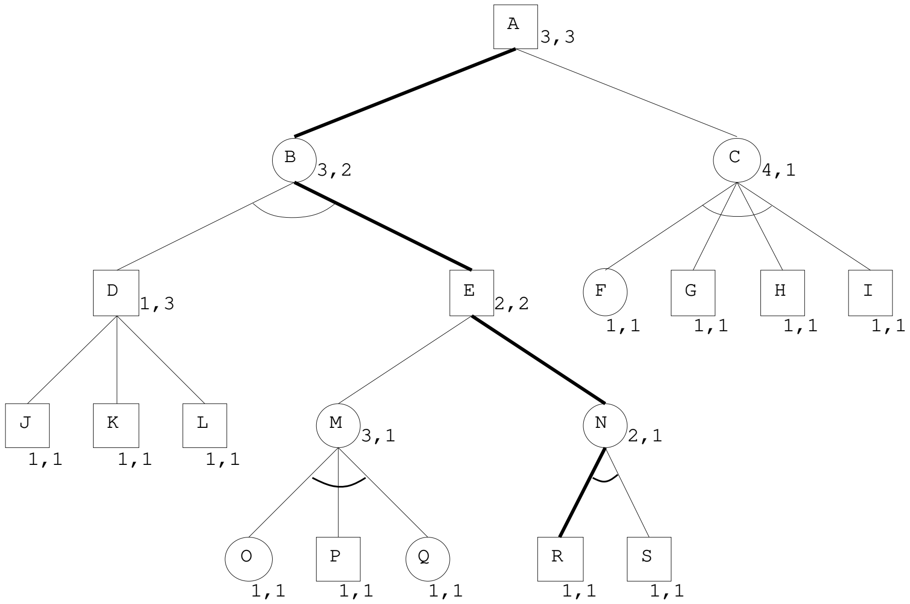


Figure 2.3: AND/OR tree with most-proving node R.

smallest disproof set of A. Second, let us look at the least number of node developments needed to prove the tree. For an OR node it is sufficient to have one child with value `true` to prove the OR node. In other words, only one child needs to be solved (with value `true`) to prove the tree. The proof number of child B (3) is one less than the proof number of child C (4). Thus, all frontier nodes of a smallest proof set lie within subtree B. We conclude that *all* most-proving nodes lie within subtree B.

With respect to subtree B an analogous analysis applies. However, since node B is an AND node, the roles of proof number and disproof number are interchanged. Thus, to prove B, both its children must be proved. Therefore, in both subtrees D and E, frontier nodes exist which are part of the smallest proof set of B. To disprove B, it is sufficient to disprove one child. Node E has disproof number 2, one less than disproof number 3 of node D. Thus, all frontier nodes of a smallest disproof set lie within subtree E. We conclude that *all* most-proving nodes lie within subtree E.

The selection within OR node E is based on the disproof numbers, as it was for node A, and thus subtree N is selected. Within AND node N no preference exists on the basis of the disproof numbers and both R and S

```
procedure ProofNumberSearch(root);
  Evaluate(root);
  SetProofAndDisproofNumbers(root);
   while root.proof ≠ 0 and root.disproof ≠ 0 and
          ResourcesAvailable() do
     mostProvingNode := SelectMostProving(root);
     DevelopNode(mostProvingNode);
     UpdateAncestors(mostProvingNode)
   od ;
   if root.proof = 0 then root.value := true
   elseif root.disproof = 0 then root.value := false
   else root.value := unknown
   fi
end
```

Table 2.1: Pn-search algorithm.

are most-proving nodes according to definition 2.5. In such cases we will, somewhat arbitrarily, always select the leftmost child. Thus, R is selected to be developed.

To summarize, the selection of a most-proving node is based on proof numbers among the children of OR nodes and on disproof numbers among the children of AND nodes.

2.2.4 The pn-search algorithm

In this section the algorithmic details of pn-search are presented in pseudocode, except for three domain-specific procedures and functions. In each of these three cases, the code for the implementation depends on the domain of investigation. The *goal* of each of these three, however, is domain-independent and has been specified below.

1. *Evaluate(node)*. Assigns to node.value one of the values true, false and unknown.

2. *GenerateAllChildren(node)*. Assigns to node.numberOfChildren the number of children of the node, and to node.children[1..node.numberOfChildren] (pointers to) the children themselves.

```
function SelectMostProving(node);
    while node.expanded do
        case node.type of
            or :
               i := 1;
                while node.children[i].proof ≠ node.proof do
                   i := i+1
                od
            and :
               i := 1;
                while node.children[i].disproof ≠ node.disproof do
                   i := i+1
                od
        esac ;
       node := node.children[i]
    od ;
    return node
end
```

Table 2.2: Most-proving node selection algorithm.

3. *ResourcesAvailable()*. Returns a Boolean value which indicates whether sufficient resources are available to continue the search. This function will typically test the availability of working memory.

The algorithm of table 2.1 encodes the main loop of pn-search. The root of the tree is created and evaluated. Then, at each iteration, a most-proving node is selected and developed, followed by updating the proof and disproof numbers of the most-proving node and its ancestors.

The algorithm terminates when the tree is solved, or the program runs out of resources. We remark that there is a choice between implementing *immediate evaluation* and *delayed evaluation*. The main difference between these two methods is the amount of information available within trees of the same size: with immediate evaluation, all nodes in the tree have been evaluated, while with delayed evaluation the frontier nodes have not been evaluated. Due to the extra information, under the same working-memory limitations, immediate evaluation is more often able to solve a tree than delayed evaluation. In rare circumstances, however, delayed evaluation may be preferable. Examples of these circumstances include trees with a large

```
procedure SetProofAndDisproofNumbers(node);
    if node.expanded then
        case node.type of
            and :
                node.proof := Σ_{N∈Children(node)} N.proof;
                node.disproof := Min_{N∈Children(node)} N.disproof
            or :
                node.proof := Min_{N∈Children(node)} N.proof;
                node.disproof := Σ_{N∈Children(node)} N.disproof
        esac
    elseif node.evaluated then
        case node.value of
            false : node.proof := ∞; node.disproof := 0
            true : node.proof := 0; node.disproof := ∞
            unknown : node.proof := 1; node.disproof := 1
        esac
    else node.proof := 1; node.disproof := 1
    fi
end
```

Table 2.3: Proof and disproof numbers calculation algorithm.

variance in the branching factor, and slow evaluation. We have chosen to implement immediate evaluation, as it is used in all our applications of pn-search to games. Thus, all frontier nodes in the tree have been evaluated.

The algorithm of table 2.2 encodes the selection of a most-proving node, in accordance with the description in section 2.2.3. Thus, at OR nodes the child with lowest proof number is selected, while at AND nodes the child with lowest disproof number is selected. In case of a tie between children, the leftmost child is selected. Selecting the child with minimal proof number (in an OR node) or disproof number (in an AND node) is equivalent to selecting a child with proof number or disproof number equal to its father's. We remark that in most applications children will not be ordered by their proof or disproof number, as the cost of updating the ordering may be prohibitive. If the children are unordered, selecting the leftmost child with equal proof or disproof number on the average reduces the selection time of the most-proving node by at least a factor two, compared with determining the minimum over all children. A detailed discussion of enhancements to the algorithm can be

```
procedure DevelopNode(node);
   GenerateAllChildren(node);
   for i := 1 to node.numberOfChildren do
      Evaluate(node.children[i]);
      SetProofAndDisproofNumbers(node.children[i])
   od
end
```

Table 2.4: Node-development algorithm.

```
procedure UpdateAncestors(node);
   while node ≠ nil do
      SetProofAndDisproofNumbers(node);
      node := node.parent
   od
end
```

Table 2.5: Ancestor-updating algorithm.

found in section 2.3.

The algorithm of table 2.3 encodes the calculation of proof and disproof numbers for a given node. It is a direct translation into pseudo-code of the case-by-case observations made in section 2.2.2. We remark that "Σ" in the algorithm indicates that the sum is calculated over all children, while "Min" indicates that the minimum over all children is calculated.

The algorithm of table 2.4 encodes the development of a node. As stated before, we have implemented immediate evaluation.

The algorithm of table 2.5 updates the proof and disproof numbers of the most-proving node and its ancestors. This is necessary to ensure that all nodes in the tree correctly reflect the new situation after the development of the most-proving node. Starting from the most-proving node, the tree is traversed in the direction of the root, updating the proof and disproof numbers of each ancestor. After the proof and disproof numbers of the root have been updated, the algorithm is terminated (indicated by the fact that the root has no parent).

This concludes our formal description of pn-search.

2.3 Enhancements

In the previous section we have presented the pn-search algorithm. Several enhancements exist. Some of these should be applied in most practical circumstances, since the added performance outweighs the additional implementation effort. The advantage associated with the other enhancements depends on the domain of application. In section 2.3.1 we focus on enhancements reducing the amount of working memory needed to execute a search. Section 2.3.2 deals with reducing the execution time necessary to select the most-proving node and to update the proof and disproof numbers of the ancestors. The role of domain-specific knowledge when enhancing the algorithm is examined in section 2.3.3. Finally, transpositions are discussed in section 2.3.4.

2.3.1 Reducing memory requirements

Pn-search has working-memory requirements linear in the size (number of nodes) of the search tree. Depth-first search algorithms, such as α-β search, only require working memory linear in the depth of the search. As a result, working memory is a possible bottleneck when applying pn-search. We discuss two techniques to reduce memory requirements. The first technique is concerned with the removal of solved subtrees, while the second technique performs pn-search at two levels.

Deleting solved subtrees

A node in a pn-search tree may influence the search process in two ways:

1. it is (on the path to) the most-proving node;

2. its proof and disproof numbers influence the proof and disproof numbers of its parent.

Below, we show that solved subtrees do not influence the search process in either way, except that they may solve their parent immediately after they were solved themselves.

First, we show that a solved node is never on the path to the most-proving node. As long as the search is in progress the root is not solved. We thus start the selection of the most-proving node from an unsolved node. All unsolved nodes have finite proof and disproof numbers unequal to zero. Since at each step of the selection, a child is chosen with a proof or disproof number equal to that of its parent, each subsequent node must also be unsolved. We conclude that a solved node cannot be on the path to the most-proving node.

Second, we show that the proof and disproof numbers of a solved node either solves its parent, or does not influence its parent's values. A solved node with value true has proof number 0 and disproof number infinity. A parent OR node is solved by this child, and immediately obtains the value true. A parent AND node sums its children's proof numbers, to which the 0 does not contribute, while it minimizes its children's disproof numbers, to which infinity does not contribute. Only if this child were the last unsolved child is the AND node solved and obtains the value true. To a solved child with value false an analogous reasoning applies, with false and true, proof number and disproof number, and AND node and OR node interchanged.

We conclude that a solved subtree, once its parent has been updated, no longer influences the search, and thus may be removed. An efficient way to implement this enhancement in the *SetProofAndDisproofNumbers()* algorithm of table 2.3 is by deleting solved children when calculating the sum and minimum of the childrens' proof and disproof numbers.

For a discussion of the expected gain of this technique, we refer to section 2.4.

Pn^2-search

As a second technique to reduce memory requirements, we present a short description of a recent, so far unpublished, development in pn-search, named pn^2-search. The algorithm has been developed in collaboration with Stef Keetman.

Pn^2-search consists of two levels of pn-search. The first level consists of a pn-search (pn_1), which calls as evaluation of any node J a pn-search at the second level (pn_2), with a bound N on the maximum tree size. In pn^2-search N is chosen to be the current size of the pn_1 search tree. The second level of pn-search is a standard pn-search, with a normal (either standard or domain-specific) evaluation. The result of pn_2 on node J is the value true or false in case pn_2 solved J, or the proof and disproof numbers of J, if J has not been solved. In the latter case, the proof and disproof numbers are used to initialize J in pn_1. After termination of pn_2, its tree is removed from memory. We remark that several enhancements to pn^2-search have been suggested to reduce the overhead associated with recreating deleted parts of the tree. One example of such an enhancement involves storing the M last pn_2 trees in a cache, instead of deleting them, as suggested by Schaeffer (1994). The gain achieved by such enhancements is a topic of future research. Pn^2-search has the following properties.

1. A search resulting in a pn_1 tree of size N has searched approximately $\frac{1}{2} \cdot N^2$ nodes.

2. The memory requirements during the creation of a pn_1 tree of size N are approximately $2N$ nodes.

3. Implementing pn^2-search requires only minor changes to an implementation of standard pn-search

It has been established that the memory requirements of pn^2-search are on the order of the square root of the number of nodes investigated. Comparisons on awari and draughts have shown experimentally that pn^2-search investigates on the average three times as many nodes as standard pn-search to solve the same problem. This factor of three is independent of problem size within the range investigated.

Given the approximate constancy of this factor, it follows that in cases where pn-search is bounded by trees of 10^6 nodes, pn^2-search, with the same resources of memory may usefully investigate 10^{12} nodes. This conclusion can be extrapolated to even larger problems only when the factor of three suggested by the experiments remains constant. Whether it does and whether the extrapolation therefore remains valid, is a topic for future research.

2.3.2 Reducing execution time

The main difference in execution time between a best-first search algorithm, such as pn-search, and a depth-first search algorithm, such as α-β search, is the number of node traversals necessary to select the most-proving node. The overhead specific to pn-search is the calculation of proof and disproof numbers at internal nodes, being linear in the number of node traversals. The enhancement presented in this section reduces the number of node traversals per selection of the most-proving node. We remark that the same enhancement can be and has been applied to conspiracy-number search (Klingbeil, 1989).

At each iteration of pn-search we traverse the tree starting at the root and ending at the most-proving node. After developing the most-proving node, we follow the same path backwards until we are at the root. The basis of the enhancement consists of two observations.

- If the proof and disproof numbers of an ancestor do not change, the updating process can be terminated.
- If a node J is on the path from the root to the most-proving node, and J's proof and disproof numbers are not changed by the updating process, J also lies on the path from the root to the next most-proving node.

From these two observations it follows that at each iteration a node exists where we can terminate the updating process, and start the next most-proving node selection. Such a node is called the *current node*, which is defined as follows.

Definition 2.6 *For any* AND/OR *tree T, at any time during the execution of pn-search, the* current node *of T is defined as the ancestor of the previous most-proving node J, closest to J, which had no changes to its proof and disproof numbers caused by the development of J. Initially, the current node equals the root.*

Enhancing the pn-search algorithm to use the notion of current node changes the algorithms for *ProofNumberSearch* and *UpdateAncestors*. The new algorithms are shown in the tables 2.6 and 2.7.

The current-node enhancement reduces the number of node traversals per iteration from linear in the depth of the search tree to close to constant and should therefore be included in most practical implementations of pn-search.

We remark that at the cost of storing a most-proving node per subtree, the selection process can be changed into an instant most-proving node selection. Then, the most-proving nodes of the subtrees are updated during the updating of the proof and disproof numbers within the tree. Since the working memory is the main bottleneck in most applications, we feel that small gains in terms of processing speed do not warrant the extra space requirements.

2.3.3 Applying domain-specific knowledge

Two assumptions underly the formulation of the pn-search algorithm. First, the probability distribution of expected values of frontier nodes is equal throughout the tree. Second, the distribution of probabilities over the three evaluation values (true, false, unknown) is unknown. These two assumptions describe a situation in which no domain-specific knowledge can be applied to guide the search through the tree. In many practical domains, however, at

```
procedure ProofNumberSearch(root);
   Evaluate(root);
   SetProofAndDisproofNumbers(root);
   currentNode := root;
   while root.proof ≠ 0 and root.disproof ≠ 0 and
         ResourcesAvailable() do
      mostProvingNode := SelectMostProving(currentNode);
      ExpandNode(mostProvingNode);
      currentNode := UpdateAncestors(mostProvingNode)
   od ;
   if root.proof = 0 then root.value := true
   elseif root.disproof = 0 then root.value := false
   else root.value := unknown
   fi
end
```

Table 2.6: Pn-search algorithm (with current node).

least some knowledge is available. In this section we show how such knowledge can be applied to pn-search by altering the initialization of the proof and/or disproof numbers of frontier nodes.

We can view proof and disproof numbers as lower bounds on the effort necessary to solve a tree. So far, the effort has been measured in node developments. We consider three methods to use alternative measures of effort. First, we use the number of node *evaluations* as a measure of effort. Second, a domain-specific measure of effort is applied. Third, a function of the tree depth is used to influence the shape of the tree searched. Each method is illustrated using a particular game, being give-away chess, awari and go-moku, respectively. Finally, we review the three methods applied.

Evaluations as a measure of effort

A node development, when using immediate evaluation, consists of expanding the node and evaluating each of its children. Thus, the amount of effort involved in a node development depends on the number of children. We will use as J's proof number the least number of node evaluations necessary to prove node J and as its disproof number the least number of node evaluations necessary to disprove J. Let us assume that J will have n children when

```
function UpdateAncestors(node);
   changed := true ;
   while node ≠ nil and changed do
      oldProof := node.proof;
      oldDisproof := node.disproof;
      SetProofAndDisproofNumbers(node);
      changed := (oldProof ≠ node.proof) or
                 (oldDisproof ≠ node.disproof);
      previousNode := node;
      node := node.parent
   od
   return previousNode
end
```

Table 2.7: Ancestor updating algorithm (enhanced).

expanded. J's proof and disproof numbers can be initialized using that knowledge, even before J is expanded. If J is an OR node, only one child needs to evaluate to true to prove J, thus J's proof number equals 1. To disprove J, all n children must evaluate to false. J's disproof number is therefore initialized to n. For an AND node, the proof number is initialized to n, while the disproof number becomes 1.

The advantage of using the number of evaluations as a measure of effort is that a distinction between frontier nodes can be made which is not present in standard pn-search. It allows pn-search to focus on frontier nodes with fewer children before developing frontier nodes with more children. It is expected that in this way pn-search will find solutions more quickly. Below we present results from applying this method to give-away chess.

Give-away chess is a variant of chess where a player wins as soon as she cannot make a legal move (i.e., she has no pieces left or her remaining pieces are blocked). The pieces move as in chess, with two exceptions:

1. the king has no special status and can be captured like any other piece;
2. a player is forced to make a capture move if she can (like in checkers and draughts).

Castling and *en-passant* capturing are extremely rare in give-away chess. To simplify our implementation task, we have omitted these types of moves, thus rendering them illegal. In collaboration with Barney Pell we created the give-away chess program *Prove-away*, solely based on pn-search. A node evaluates to true, if white is to move and has no legal moves, while it evaluates to false if black is to move and has no legal moves. All other nodes evaluate to unknown. Pn-search was implemented in two variants, one variant using the standard initialization, and the other one using node evaluations as measures of effort.

To enable *Prove-away* to play games against opponents, it selects its moves by performing pn-search with a predetermined bound on the number of nodes to be created. If the tree is not solved within that limit, the 1-ply nodes are inspected and the move leading to a node with the minimal ratio of proof and disproof numbers is selected. If the tree is proved within the limit, the move proving the tree is selected, ensuring a win for *Prove-away*. If the tree is disproved, the 1-ply node with the largest subtree is selected, speculating on the opponent not seeing her winning line. Although we have no clear indication of the strength of *Prove-away*, it has beaten its human opponents in all but three of its games (out of several dozen). Most games are decided by *Prove-away* finding a winning line in which the opponent is forced at each move to capture one of the program's pieces, until the program runs out of moves and wins. The maximum depth of such lines in give-away chess is 32 ply (16 moves by the program and 16 captures by the opponent).

We conducted an experiment to compare the two variants of pn-search described above. During the experiment, *Prove-away* plays random games against itself. At each move in the game, both variants of pn-search (standard initialization and using evaluations as measure of effort) create a tree, with the current game position as root. As soon as one or both variants solve the tree, the game is terminated. If in a position neither variant solves the tree within 25,000 nodes, *Prove-away* plays a random legal move to continue the game. A total of 30 games were played, which lasted on the average 5.6 ply (i.e., a little less than three moves by white and three moves by black). Three games where duplicates of other games, due to the fact that the program quickly proved that black wins after the opening moves 1. d2-d4, 1. d2-d3 or 1. e2-e4, and each of these moves was selected twice as opening move during the 30 games. In the following we disregard the three duplicate games.

The conditions of the experiment ensure that the final position of each random game has been proved a win for one of the players by at least one of the pn-search variants. In some games, both variants proved the win, while in others only the pn-search variant with the number of evaluations as the measure of effort succeeded. In none of the games did only standard

	standard initialization	*initialization by no. of moves*	*improvement factor*
developments	5928	2661	2.2
nodes visited	62323	7838	8.0
branching factor	10.5	2.9	3.6
max tree size	48935	5988	8.2
nodes per sec.	169	132	0.8

Table 2.8: Give-away chess results.

pn-search solve the tree of the final position. To compare the performances of both algorithms, we reran the standard algorithm with unlimited working memory on the positions where that variant had not found the win within the limit of 25,000 nodes. The results of the experiment are presented in table 2.8.

Measured in number of node developments, the enhanced algorithm (using evaluations as measure of effort) gains a factor of a little over 2, while in number of nodes the improvement factor is almost 8. These numbers indicate that the enhanced algorithm develops nodes with, on average, a 4 times smaller branching factor (2.9 vs. 10.5). This clearly indicates that the selection of most-proving nodes is strongly influenced by the non-standard initialization. The average amount of working memory necessary to complete the search is specified in table 2.8 as the maximal tree in memory per search. It is directly related to the total size of the tree created, resulting in an improvement by a factor 8. The extra time spent on counting the number of moves per terminal node slows the algorithm down approximately 20% per node, compared to the standard initialization. Thus, the overall gain in CPU time amounts to a factor of more than 6.

We conclude that using the number of node evaluations as a measure of effort to initialize the proof and disproof numbers may yield a significant reduction in node evaluations, node developments and CPU time.

Domain-specific measures of effort

In many domains, domain-specific properties exist which give an indication of the amount of effort involved in solving a position (i.e., in solving the AND/OR tree with the position as root).

For instance, in othello solving a position with only a few empty squares is easier than solving a position with more empty squares. In draughts, it is simpler to solve a position if both players have only four men than if both players have ten men. In these cases, we could select as domain-specific measures of effort the number of moves to the end of the game (othello) or the number of men of the opponent to be captured (draughts and checkers).

We illustrate the idea on the game awari. In the initial awari position, there are 48 stones on the board. Both players move stones around, with the goal of capturing stones. The goal of awari is to capture more stones than your opponent. It follows that a player who has captured at least 25 out of the total of 48 stones, wins the game (for a definition of the rules of awari, see section 2.4.2). We use the number of stones a player needs to capture as the measure of effort. Let us assume that we would like to determine whether north can win, or whether south can obtain at least a draw. Let us furthermore assume that south has so far captured 11 stones, while north has collected 8 stones. We build the tree from the perspective of south, thus proving the tree means showing that south can reach at least a draw. In the given position, south must capture at least another 13 stones to reach a draw, while north needs another 17 stones to obtain the 25 stones necessary for a win. These values, 13 and 17, are then used as proof and disproof numbers of the position.

In section 2.4.7 we present test results of applying pn-search to awari for both the standard initialization and the stone-based initialization as suggested here.

Depth-related measures of effort

By inspecting trees created by pn-search, we have found some occasions in which the shape of the tree indicated that much effort was spent on variations which were less likely to succeed quickly than some others. For instance, in mating problems in chess, where the weaker side was restricted to moving one piece between two squares, most variations had proof number one. As a result, variations where the attacker moved a single piece aimlessly over the board were searched very deeply. On one occasion, this resulted in a mate in 114 moves being found, while a mate in 4 moves existed. Instead of putting a hard limit on the depth of the search, examining deep variations can be somewhat discouraged by initializing the proof and disproof numbers of a node using a function of the depth of the node.

By assigning higher proof and disproof numbers to nodes deeper in the tree, it is expected that pn-search will create a somewhat shallower and broader tree. Analogously, by assigning smaller proof and disproof numbers to nodes deeper in the tree, pn-search is expected to create deeper and narrower trees. Inspection of trees created by pn-search with such alternative proof-and-disproof-numbers initializations shows that the average node depth is indeed influenced in accordance with these expectations.

Experiments on go-moku (see chapter 5), with each node's proof and disproof numbers initialized to the depth of the node measured in full moves, show that a somewhat broader, shallower tree is created, without losing pn-search's ability to find narrow, deep variations leading to a win. Comparisons on go-moku showed that this initialization was an improvement over the standard initialization. The depth-related initialization was used in the search which led to solving go-moku.

Despite this example, we do not have much ground for the assumption that such an initialization is an enhancement to pn-search for domains with behavior similar to go-moku. Furthermore, the evaluation function we developed for go-moku also influenced the success of the non-standard initialization. Although a linear function in the depth of the node worked well in go-moku, more complicated functions may be necessary for other domains. The strongest conclusion we are prepared to draw is that by using a function of the depth of the node, the shape of the tree can be somewhat influenced (either made broader and shallower, or narrower and deeper).

Reviewing the application of domain-specific knowledge

We have presented three ways in which domain-specific knowledge can be used to change the initialization of the proof and disproof numbers at frontier nodes. Although each of the three methods has been successful in improving the performance in a practical domain, some caution is in order, particularly with the second and third methods. While the use of non-standard proof-and-disproof-numbers initializations may seem useful to guide the search process, the underlying principles of pn-search are violated. Two examples of violated principles are: (1) the assumption that all frontier nodes are indistinguishable and (2) the assumption that the proof and disproof numbers are lower bounds on the effort required to solve the tree. The positive influence of different initializations may at the same time result in negative effects. We have found that for some domains, such as othello, it is necessary to perform a large number of experiments to fine-tune the initialization process, akin to the process of fine-tuning evaluation functions in game-playing programs

(Gnodde, 1993). We conclude that as yet we lack a proper understanding of the precise effects associated with knowledge-driven proof-and-disproof-numbers initializations.

2.3.4 Transpositions

The definition of pn-search depends on the graph searched being a tree. When determining the proof and disproof numbers of an internal node J, the cardinality of the smallest proof set and disproof set must be determined. In a tree, the subtrees rooted at the children of J are disjoint, ensuring that the cardinality of the smallest proof set and disproof set of J can be calculated from the cardinality of the smallest proof sets and disproof sets of the children.

In many domains, however, the same subtree may be encountered several times during the search, at different places in the tree. The standard pn-search algorithm will in such cases obtain an upper bound on the cardinality of the smallest proof and disproof sets, instead of the true proof and disproof numbers. Problems and solutions related to the problem of the common subtree (transpositions) in combination with pn-search have been investigated by Schijf (1993) and Schijf *et al.* (1994).

In the following, we shortly describe problems and practical solutions for transpositions in pn-search. We distinguish between *directed acyclic graphs*, abbreviated as DAGs and *directed cyclic graphs*, abbreviated as DCGs. We remark that practical techniques for handling transpositions in game-playing programs using α-β search have been extensively described in the literature (Greenblatt *et al.*, 1967).

Transpositions in DAGs

Transpositions resulting in DAGs necessarily occur in games where each move is a *conversion*, i.e. an irreversible alteration of the state of the game. In chess, captures and pawn moves are examples of conversions, while non-capture moves by a piece (except for castling, and castling-forbidding moves) are non-conversions. In connect-four, qubic and go-moku, each move is a conversion, as in all three games the number of stones on the board strictly increases.

As stated above, in a DAG, addition of proof numbers or disproof numbers possibly overestimates the cardinality of the minimal set of nodes needed to solve the tree. Theoretically correct algorithms exist to establish the correct proof and disproof numbers at each node, but these are slow or use

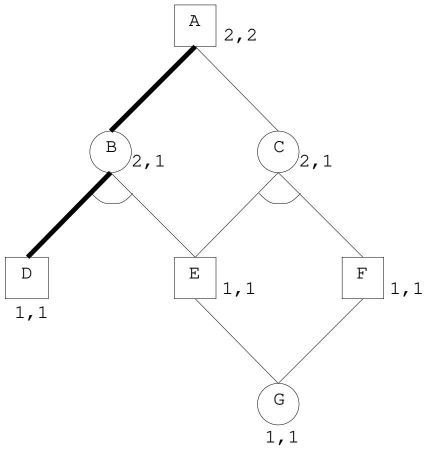


Figure 2.4: AND/OR DAG with practical solution.

an inordinate amount of working memory, or both, thus barring practical application (Schijf, 1993).

A practical solution to this problem is to treat the DAG as if it were a tree, thus calculating (incorrect) proof and disproof numbers of a node directly from its children. The main difference in the algorithm is that while updating ancestors, all parents of a node must be updated recursively. In figure 2.4, a DAG is depicted where proof and disproof numbers are calculated directly from their children. It can easily be shown that if node G is solved, root A obtains the same value as G. Thus, the proof and disproof numbers of A should equal 1. Furthermore, G should be the most-proving node. Thus, both numbers in the root are too high, and the selection mechanism incorrectly selects node D as the most-proving node. This example clearly indicates that the practical solution is no longer in accordance with the definitions of section 2.2.2. Still, our experience with connect-four, qubic and go-moku shows that this practical pn-search algorithm for DAGs has advantages similar to those of standard pn-search.

Transpositions in DCGs

Transpositions resulting in DCGs appear in games where a series of non-conversion moves leads to a position which has occurred before. Special rules govern the continuation of games after repetitions, leading by complex regulations to game-specific outcomes. There is fascination in the diversity

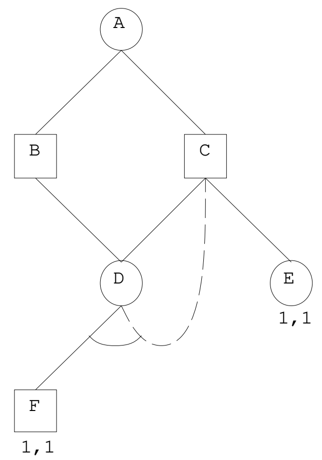


Figure 2.5: Cyclic AND/OR graph.

of these rules: in Chinese chess, some repetitions are illegal by the operation of complex rules; in go, any repetition is outlawed by the *ko* rule; in chess, finally, a repeated position can give rise to a claim of a draw from its third occurrence onwards.

Figure 2.5 depicts a DCG; in figure 2.6 we have converted that graph into a tree. Each path in the tree terminates at a frontier node of the graph, or at a repetition of positions in the path. In this example we assume that a repetition evaluates to false. The tree contains three duplicates of node D. Among these three, two have the value false, while one has proof number 2 and disproof number 1. The fact that the same node may have different proof and disproof numbers depending on the path it lies on forms the basis of the complexity of performing pn-search on DCGs. Node C has properties similar to node D. Moreover, we note that A's proof number (2) is less than the sum of the proof numbers of its children, as subtrees B and C have node E in common. The proof number at the root indicates that to prove the tree, both E and F must be proved. The disproof number 1 of A indicates that disproving either E or F disproves the tree.

The dependence of the proof and disproof numbers of a node on the path to that node forms the basis of the difficulties of cyclic transpositions. In Schijf (1993), a theoretically correct algorithm for pn-search on DCGs is described. Unfortunately, its time and working-memory requirements are too costly to warrant practical application.

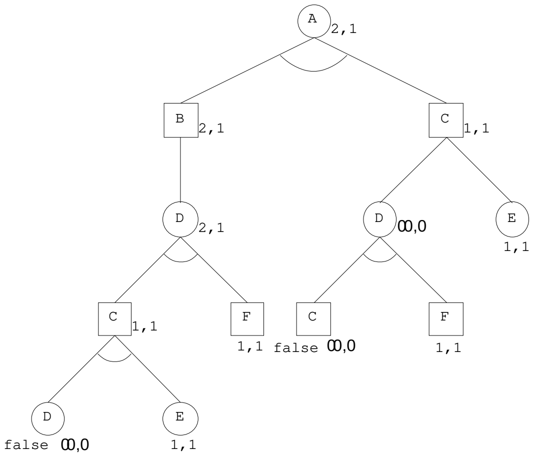


Figure 2.6: Tree version of the graph of figure 2.5.

Practical methods to apply pn-search to DCGs also exist. First, the practical algorithm for DAGs may be applied with one modification: only positions created after a conversion move are eligible to have more than one parent. As a result, some transpositions are investigated only once, while for others duplicates are created in the graph. Second, for each set of equivalent positions, at most two nodes are created: one for all paths in which the node occurs for the first time, and the second node when the node is its own ancestor. The second node is initialized to the value associated with a repetition of positions in the game under investigation. In this case, if a node is its own ancestor through at least one path, the repetition of positions is used to update the ancestors on all paths leading to the node, including those in which the node is not a repetition. Therefore, the search may incorrectly deduce that a node must have the value of a repetition of positions. Thus, if the value of the root is proved to equal the value assigned to repetitions of positions, the proof is not fully reliable. If the opposite value is proved, however, the proof is bound to be correct. For a detailed description of these two practical algorithms for pn-search in DCGs, we refer to Schijf (1993).

We believe that pn-search on directed cyclic graphs requires further investigation.

2.4 Results

2.4.1 Introduction

In this section we compare pn-search's performance with that of a sophisticated implementation of α-β search, by far the most commonly applied game-tree search algorithm in tournament programs for strategic games. As a test domain, we have selected the game of awari, one of the games on the Olympic List. We have chosen awari for two main reasons. First, awari search trees contain non-uniformity, which make them suitable for the application of pn-search. Second, all strong tournament programs competing in the Computer Olympiads selected their moves using sophisticated implementations of α-β search, establishing that awari search trees are suitable for application of α-β search.

It will be shown that, for the purpose of *proving* the game-theoretic value of a position in awari, pn-search outperforms α-β search by a wide margin. It proves that a category of search trees exists for which pn-search outperforms α-β. Further indications of pn-search's strengths can be found in chapters 4 and 5, where pn-search's contribution to solving qubic and go-moku is described.

This section is organized as follows. First, we present the rules of awari. Second, we give a description of the strongest existing awari programs, which presents evidence that our implementation of α-β search is competitive with α-β search implementations of other authors. Third, we describe in detail the implementations of pn-search and α-β search and their performances are compared. Fourth, it is explained how the nodes visited by both algorithms are counted, which is important due to the different nature of the algorithms. Fifth, we describe the set of awari positions to which the algorithms were applied. Finally, we present and analyze the empirical data.

2.4.2 The rules of awari

Awari is a two-player (south and north) zero-sum game with perfect information. It is one instance of a large family of games named mancala, of which some 1200 variants are known. The mancala games originate from Africa. Awari is mainly played in its western regions, such as Nigeria. For the game described here, the names wari or awele are also used (Deledicq and Popova, 1977).

Awari is played on a wooden board containing two rows of six pits. Each player controls the row on her side of the board. South's pits (from left to

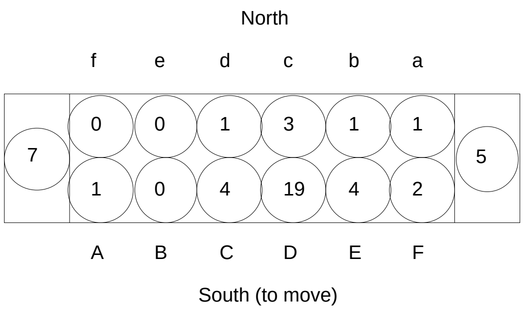


Figure 2.7: A position with legal moves $A1$, $C4 \times 2$, $D19 \times 7$, $E4$ and $F2 \times 4$.

right, as seen by south) are named A through F, while north's pits (from left to right, as seen by north) are named a through f. At the right-hand side of each row, an auxiliary pit is used to contain a player's captured stones. At the start of the game each pit (except the auxiliary pits) contains four stones, for a total of 48 stones on the board.

At each move, a player selects a non-empty pit X from her row. Starting with X's neighbor, she then sows all stones from X, one at the time, counter-clockwise over the board (omitting the two auxiliary pits). If X contains sufficient stones to go around the board (12 stones or more), pit X is skipped and sowing continues. Thus, after the move, X will always be empty. Finally, captured stones, if any, are removed and stored in the auxiliary pit. Stones are captured if the last stone sown lands in an enemy pit which after landing contains 2 or 3 stones. If such a capture is made, and the preceding pit contains 2 or 3 stones and the pit is an enemy pit, those stones are also captured. This procedure is successively repeated for the pits preceding and ends as soon as a pit is encountered containing a number of stones other than 2 or 3, or the end of the opposing row is reached.

A move is described by the name of the pit, followed by the number of stones sown (the name of the pit by itself defines the move, but such a notation is prone to error). The number of stones captured, if any, is indicated by the amount preceded by a "$\times$". In figure 2.7 an example position is shown with south to move. Legal moves for south are: $A1$, $C4 \times 2$, $D19 \times 7$, $E4$ and $F2 \times 4$.

The goal of awari is to capture more stones than the opponent. The game ends as soon as one of the players has collected 25 or more stones. Two other conditions exist which terminate the game. First, if a player is unable

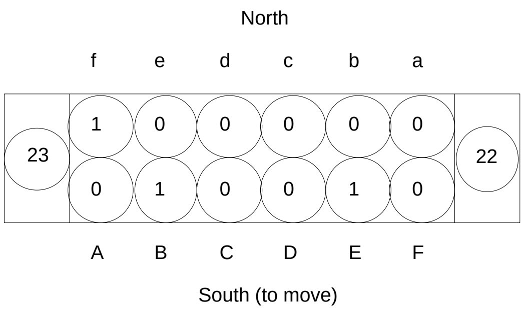


Figure 2.8: 1. $B1$ $f1$ wins. After 1. $E1$? $f1$ south must play 2. $F1$.

to move (i.e., all her pits are empty), the remaining stones are captured by her opponent. Second, if the same position is encountered for the third time, with the same player to move, the remaining stones on the board are evenly divided among the players. In all cases, after the end of the game, the winner is the player who captured the most stones. If both players capture 24 stones, the game is drawn.

A last rule exists to prevent players from running out of moves early in the game. Whenever possible, a player is forced to choose a move such that her opponent is able to make a reply move. It is, however, not compulsory to look several moves ahead to ensure that the opponent will continue to be able to reply. For instance, figure 2.8 shows a position in which south by playing *1. B1* can deliberately create a position in which she is unable to offer north any stones on her next move. By doing so, south captures all three stones remaining on the board and wins the game. However, would she have played 1. $E1$, then after 1. ... $f1$ she is forced to play 2. $F1$, leaving the game for the moment undecided (although after 2. ... $a1$ 3. $B1$ $b1$ 4. $C1$ $c1$ 5. $D1$ $d1$ 6. $A1$ $e1$ we are back at the initial position, giving south a second chance to play the winning move).

2.4.3 Tournament programs

Lithidion

In 1990 Maarten van der Meulen and the author constructed an awari-playing tournament program, named *Lithidion* (Greek for 'little stone'). *Lithidion* at the time consisted of an α-β search algorithm, and an endgame database containing the game-theoretic value of each awari position with 13 stones or fewer left on the board (Allis *et al.*, 1991c).

In 1991 *Lithidion* was enhanced with pn-search and a larger database (all positions with 17 stones or fewer). In 1992 *Lithidion* was further enhanced with an opening book. In describing *Lithidion*, we will concentrate on this last version of *Lithidion*.

The basis for *Lithidion* is its α-β search algorithm. Any position not in the opening book or the endgame database is searched with iterative-deepening α-β search. The evaluation function for leaf nodes is trivial: at each leaf node it is assumed that the players divide the remaining stones evenly. If, in the search tree, a position is encountered having 17 stones or fewer on the board, its exact value is retrieved from the endgame database. Thus, the value of the α-β search is based on a combination of crude guesses for some leaf nodes, and exact values for others (Beal, 1984). We remark that in awari the difference in the number of stones by which one wins is irrelevant. Therefore, the value retrieved from the endgame database is converted into $-\infty$ for losses, 0 for draws, and ∞ for wins. Once the game has progressed to a position contained in the endgame database, no search is needed, and at each turn the best move is played instantly.

After a move has been selected by α-β search (typically based on an 18-to-20 ply search), pn-search is called to check the move. If a proof can be found that the selected move loses, the move is rejected, α-β is asked to select a new move, and the procedure is repeated. If all moves are proved losses, the first move selected is played, hoping for an error by the opponent. While the opponent is pondering on the position, *Lithidion* performs pn-searches on her potential moves looking for wins. In case the opponent selects a losing move, *Lithidion* uses the proof by pn-search to select its winning move. The pn-search algorithm regards positions within the endgame database as terminal nodes, just as it treats positions where a player has no legal moves. All other positions are internal nodes. In summary: pn-search is only used to prevent *Lithidion* from playing losing moves and to detect winning lines after erroneous moves by the opponent. All other moves are based on α-β search.

Opponents

Lithidion has played in three tournaments: the awari tournaments of the 2nd, 3rd and 4th Computer Olympiads (London 1990, Maastricht 1991 and London 1992). *Lithidion* won the gold medal each time. The tournaments of the 2nd and 3rd Olympiads have been described in Levy and Beal (1991) and Van den Herik and Allis (1992).

In 1990, *Lithidion*'s only opponent, *Marco*, written by Remi Nierat, winner of the gold medal at the awari tournament of the 1st computer Olympiad, lost all its games. *Marco* is based on human-expert knowledge of awari, shallow α-β searches (averaging fewer than 10 ply) and no endgame databases. In most games, both *Marco* and *Lithidion* had prospects of winning, until *Lithidion*'s endgame database was reached. At that point *Marco* made one or more erroneous moves, leaving *Lithidion* with an easy win. In 1990, the main deciding factor was the endgame database (at that time, all positions of 13 stones or fewer).

In 1991, a new opponent appeared: *MyProgram* written by Eric van Riet Paap. *MyProgram* had been created using the published description of *Lithidion* (Allis *et al.*, 1991c). It contained a large endgame database (all positions of 16 stones or fewer), a fast implementation of α-β search including the singular-extension enhancement (Anantharaman *et al.*, 1989) and the same evaluation function as *Lithidion* (see above). *Lithidion* defeated *MyProgram* by the smallest possible margin, with three wins, two losses and one draw. In at least one of the games, pn-search played a decisive role, finding a deep winning line in a position unclear to α-β search. Given the small differences between the programs (a 17-stone database versus a 16-stone database, pn-search versus singular extensions, and *MyProgram* searching twice as many nodes per second), it is unclear what the exact impact of pn-search on the match has been.

In 1992, two new opponents appeared: *Marvin* and *Juju*. *Juju* turned out to be no competition for its two strong opponents and lost all its games. *Marvin* was created by Ralph Gasser with *Lithidion* as its example. The α-β search algorithms of *Marvin* and *Lithidion* performed almost equally well. *Marvin*'s endgame database (20 stones), however, was much larger than *Lithidion*'s (17 stones). A disadvantage to *Marvin* was that its database did not fit in RAM memory. Each entry retrieved from the hard disc slowed down the α-β search. Two further disadvantages to *Marvin* were its lack of a pn-search implementation and of an opening book. As a later test indicated, the opening book was the decisive factor in this match, which *Lithidion* won by a score of 4-2. The test consisted of replaying the first game from the position where *Lithidion* had exited its opening book, with *Marvin* and *Lithidion* changing places. *Marvin* easily won the game, similarly to the way *Lithidion* had won the game during the tournament. Clearly, the opening book had provided *Lithidion* with a winning advantage.

Conclusion

We have given a description of the architecture of *Lithidion*, the role of α-β search in it, and the competition it faced. From this description we conclude that *Lithidion*'s α-β-search implementation has been thoroughly tested and has performed well in competition with strong opponents. We stress this point, since *Lithidion*'s α-β search has been selected as the sparring partner for pn-search in our comparison tests on awari. Such a comparison is only valid if made against a sophisticated implementation, and we believe that practical evidence suggests that *Lithidion*'s α-β search meets those requirements.

2.4.4 The algorithms compared

For our experiments, we have compared two variants of α-β search, and two variants of pn-search. We will use the following abbreviations for the four algorithms:

α-β — α-β iterative-deepening search *without* transposition tables.

transposition — α-β iterative-deepening search *with* transposition tables.

basic pn — pn-search with standard initialization.

stones pn — pn-search with initialization based on the number of stones to be captured.

The α-β algorithm has the following characteristics. At each node, moves are pre-ordered by capture size. The largest captures are evaluated first, since the resultant positions are most likely to hit the database. Another reason for processing captures first is that they are often good moves. An iterative-deepening search is performed with a depth increase of 1 per iteration. The result of each iteration is a value and a move ordering of the full principal variation. The search terminates as soon as the value of the position has reached $-\infty$ or $+\infty$, indicating that the value of the position has been determined.

The **transposition** algorithm is the same as α-β, except that it is extended with a transposition table of a quarter of a million entries. The transposition table is implemented as a hash table, with one entry per hash code. At each node in the search tree, we first examine whether the position is present in the transposition table. Then we investigate whether the depth to which it had previously been searched is at least as large as the current depth. If both conditions are met, the range of possible values stored in the entry is used to narrow the α-β window. If after updating α exceeds or equals β, the search returns to the node's parent. Otherwise, the search is continued with the narrowed window.

After a node's value has been established, the results are stored in the transposition table. If the value of the node is equal to the initial α or β, we only know that the node's value is less or equal to α, or greater or equal to β, respectively. Only if the value lies between α and β proper, is the value reliable and can be stored as the true outcome of the search to the given depth. Values $-\infty$ and ∞ are treated separately, since these values are always indisputable. For those values, the searched depth is set to ∞ as well, since deeper searches cannot change a reliable value, making the information applicable to each following iteration. Collisions in the hash table are resolved in favor of the position which has been searched most deeply. We remark that unlike tournament chess programs, we store a full Gödel code per entry in the transposition table, ensuring that two different positions will not mistakenly be regarded as equal.

The transposition table is expected to be useful in awari in the middle and end games, when empty pits and pits containing single stones are common. A confirmation of this assumption will transpire from the results of our experiments presented in section 2.4.7.

Basic pn is the standard pn-search algorithm, enhanced with the technique which removes solved subtrees. Each frontier node is initialized to proof number 1 and disproof number 1.

Stones pn is equal to **basic pn**, except for the initialization of frontier nodes. Instead of proof and disproof numbers being initialized to 1, the number of stones still to be captured by a player to achieve her goal is used as the initialization, as explained in section 2.3.3. We remark that neither variant of pn-search uses transposition tables.

The α-β algorithm calculates approximately 10,000 nodes per second on a SUN SPARCstation 1+. The other three algorithms are roughly a factor two slower. For **transposition**, storing and retrieving information from the transposition tables is responsible for the slowed-down performance, while the pn-search variants have as extra overhead the creation and deletion of nodes, as well as the calculation of the proof and disproof numbers.

2.4.5 Comparing the performances

When selecting a search algorithm for an application, the elapsed CPU time is an important selection criterion. However, experimental results on tree searches when measured in CPU time are difficult to generalize, due to implementation details. Instead, it is customary to compare the number of nodes visited.

In this case, a careful analysis is needed to determine the fairest way to compare the number of nodes visited by α-β search and pn-search.

Let us consider the number of nodes visited by α-β iterative-deepening search. On the one hand, we could sum the number of nodes visited in each iteration. However, this would be unfair to α-β search, since a smaller number of iterations (e.g., by searching to even ply depths only) may result in almost the same ordering and thus reducing the number of nodes visited. On the other hand, we could just take the number of nodes visited in the last iteration. That would be unfair towards pn-search, as the last iteration does use the move ordering of previous iterations, and these searches should be included in the total node count somehow. Moreover, α-β search with transposition tables obtains many early cut-offs during the last iteration due to the solved subtrees stored in the transposition table.

Instead, we have chosen to count at iteration i only the nodes at depth i. Then the extra iterations are an asset to α-β search, without costing anything in terms of the number of nodes visited. Re-ordering of the moves may result in terminal nodes in a new iteration, which are not at the deepest level. These nodes are not counted at all. This slight bias in favor of α-β iterative-deepening search does not significantly influence the results.

For pn-search, we simply count the total number of nodes created during the search.

2.4.6 Test positions

As mentioned in section 2.4.3, *Lithidion* has taken part in three awari tournaments of Computer Olympiads. In total, she played 23 games (5 against *Marco* and 6 each against *MyProgram*, *Juju* and *Marvin*), of which two games were identical, which can be explained as follows. Each of the five programs described in section 2.4.3 plays deterministically. Therefore, before the next game against the same opponent, a change should be made in the opening choice of the program to avoid losing in exactly the same way. *Juju* forgot to do so once, and lost two games in identical fashion.

In the 22 different games a total of 1707 positions have occurred (from the initial position to the position after the last move had been played). Of these there were 1599 unique positions, which have been selected as the *initial* test positions.

For each of the initial test positions, a search with all four algorithms was performed. Since an awari game has three possible outcomes: win, draw and loss, and pn-search is a two-valued search algorithm, the three outcomes must be divided into two sets. We arbitrarily chose to treat a draw as equivalent to a loss for the player to move. Each of the searches has one of three possible outcomes:

- The player to move has a proved win.
- The opponent has at least a draw.
- The search ran out of resources.

Not all test positions can be used to compare the performance of the four algorithms. First, positions with 17 stones or fewer are solved immediately by all four algorithms through a single database lookup. Second, positions too early in the game are likely to be unsolvable by all four algorithms. Therefore, we have selected the relevant positions from the 1599 initial positions as follows. Each position has been investigated by all four algorithms with a resource limit of 500,000 nodes per position. If after 500,000 nodes the search had not succeeded, it was terminated. Using the outcome of the searches, the following selection was made. First, the 2 positions in which the game had just ended were discarded since all four algorithms solved the positions visiting only a single node. The reason why only 2 such positions were found out of 22 different games is that most games ended by resignation. Second, all positions with 17 stones or fewer (496 in total) were excluded. Third, all positions which were not solved by any of the algorithms (764 in total) were labeled unsolvable. The remaining 337 positions are named the *final* test positions.

We remark that in this way positions which are well suited for α-β search will be selected for the final test positions, as well as those positions well suited for pn-search. Thus, in our selection method of test positions there is no bias towards either of the algorithms.

Each of the algorithms which failed to solve one of the final test positions within the 500,000 nodes limit, was given virtually unlimited resources to try again. In practice this meant a limit of a quarter billion nodes per position for

α-β search, while for pn-search no final test position took more than about one and a half million nodes to solve.

2.4.7 Results

In this section we present the results of the comparison of the four algorithms described in section 2.4.4 on the 337 final test positions of section 2.4.6.

Each of the 337 final test positions was solved by **basic pn** and **stones pn**. Two positions were not solved by α**-**β within a quarter billion nodes, while there were two more positions not solved by both α**-**β and **transposition**. In this section we have set the solution size of unsolved positions at a quarter billion, which is a lower bound on the number of nodes necessary to solve them. Although this results in a bias in favor of α-β search, it does not influence our conclusions and it allows us to include the positions in the test results. Removing the positions from the final test set would be particularly unfair towards pn-search, as it would ignore its finest results.

First, we present figures indicating how often one algorithm outperformed another, without paying attention to the exact difference in node counts. Second, we tabulate the total number of nodes visited by each of the four algorithms, and calculate averages. Third, we group test positions by size of solution, and graphically depict the average difference in performance of the search algorithms per group.

Outperforming the other algorithms

In this section, we are interested in whether one algorithm performed better on a specific test position than another algorithm, but ignore the size of the difference. In our results we have divided the set of positions into two halves: the *easy* and the *hard* positions. To this end, we have sorted the positions according to the minimum number of nodes in which a position was solved. As a result, the 169 positions which were solved by at least one algorithm in fewer than 3200 nodes, were classified as easy positions, while the 168 positions with smallest solution larger than 3200 nodes were named the hard positions.

In table 2.9 we have listed for each algorithm how often it outperformed all other algorithms, separated for easy and hard positions. If two algorithms shared first place on a position, they were each awarded half a point.

As can be seen from table 2.9, at the easy positions there is hardly any difference between the α-β search algorithms (84 times best algorithm) and the pn-search algorithms (85 times best algorithm). For the hard positions

	α-β	transposition	basic pn	stones pn
easy	23	61	41	44
hard	0	22	$31\frac{1}{2}$	$114\frac{1}{2}$

Table 2.9: Number of times an algorithm performed best of all.

	α-β	transposition	basic pn	stones pn
α-β	-	35	79	$91\frac{1}{2}$
transposition	134	-	89	100
basic pn	90	80	-	82
stones pn	$77\frac{1}{2}$	69	87	-

Table 2.10: Comparing pairs of algorithms on easy positions.

the picture is entirely different: the pn-search variants are 146 times best, against just 22 times for the α-β search variants.

Table 2.10 shows per pair of algorithms, how often one algorithm outperformed the other, on the easy positions. Each entry at row R and column C in the table indicates how often the algorithm heading row R found a solution more quickly than the algorithm heading column C. The same information for the hard positions is displayed in table 2.11.

Table 2.10 indicates that **transposition** wins against the other three variants, albeit with a small margin compared with the two pn-search variants (89 against 80 and 100 against 69).

Table 2.11 clearly indicates that **α-β** has the worst performance of all four algorithms. It loses in all cases against **transposition**, and only 6 times outperforms the pn-search variants. **Transposition** occasionally does better than the pn-search variants, but is outperformed in more than 85% of all hard positions. Between the pn-search variants, the initialization based on the stones to be captured seems to pay off, given the 126 against 42 win compared with the standard initialization.

	α-β	transposition	basic pn	stones pn
α-β	-	0	6	6
transposition	168	-	25	24
basic pn	162	143	-	42
stones pn	162	144	126	-

Table 2.11: Comparing pairs of algorithms on hard positions.

	total nodes	average nodes	factor	tree size
α-β	2,437,035,522	7,231,559	128.8	-
transposition	1,285,839,816	3,815,548	68.0	-
basic pn	28,214,875	83,723	1.5	42,767
stones pn	18,918,032	56,136	1.0	25,505

Table 2.12: Test figures per algorithm.

Nodes visited

In this section we concentrate on the number of nodes visited by each algorithm.

In table 2.12 the first column of results lists the total number of nodes visited on the 337 test positions, per algorithm, while the second column contains the average per position. In the third column, the factor difference between each algorithm's average and the best average is presented. For both pn-search variants we have also determined the maximum number of nodes present in memory during each search. The average of these maxima have been listed in the last column of the table.

From table 2.12 a pattern similar to that seen in tables 2.10 and 2.11 becomes apparent: the pn-search variants perform best, with **stones pn** doing somewhat better than **basic pn**. With factors 68.0 and 128.8, both **α-β** and **transposition** are clearly outperformed.

The average maximum tree size in memory during the pn-searches, compared to the average solution size, indicates that removing solved subtrees during the search results in somewhat smaller memory requirements. Here approximately a factor 2 is gained. We remark that these figures only relate

	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
α-β	18	47	40	45	57	61	36	26	7
transposition	18	47	44	59	67	53	23	23	3
basic pn	14	43	52	77	92	57	2		
stones pn	14	37	57	77	101	51			

Table 2.13: Positions per group, per grouping algorithm.

to *solved* positions. In searches which are not successful, the number of solved subtrees is smaller, rendering the technique less effective.

Performance by size

Table 2.12 shows that pn-search is capable of outperforming α-β search by a large factor. The table does not indicate, however, to what extend the gain factor is related to the size of the search problems. Furthermore, we must realize that in the table the hard problems dominate the results.

Measuring the size of the search problems is not a straightforward task, since a position which is difficult to solve with α-β search may be rather simple for pn-search or vice versa. Therefore, we have grouped the test positions in four different ways, each time according to one of the algorithms applied in our experiments. We describe the grouping process based on α-β.

We have created groups for each power of 10. Thus, group i consists of all positions which were solved by α-β in more than 10^{i-1} nodes, and less than or equal to 10^i nodes. Within each group, the average number of nodes necessary to solve all positions in the group is calculated, for each of the four algorithms. These averages are then compared to see which algorithm performs best on positions of the size represented by the group.

In table 2.13 we have listed for each algorithm the number of positions per group, depending on the algorithm used as grouping criterion. These numbers indicate the size of each of the groups on which figures 2.9, 2.10, 2.11 and 2.12 are based.

Figures 2.9, 2.10, 2.11 and 2.12 contain the results per group, where the groups are created according to the solutions of α-β, **transposition**, **basic pn** and **stones pn**, respectively. For each figure, the numbers on the horizontal axis indicate the log_{10} of the size of the groups. The numbers on the vertical axis indicate the log_2 of the factor difference between the

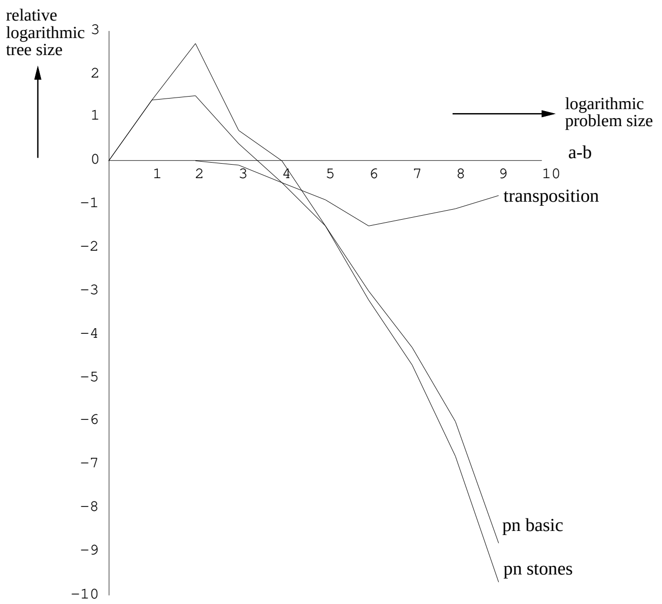


Figure 2.9: Comparison based on grouping by α-β

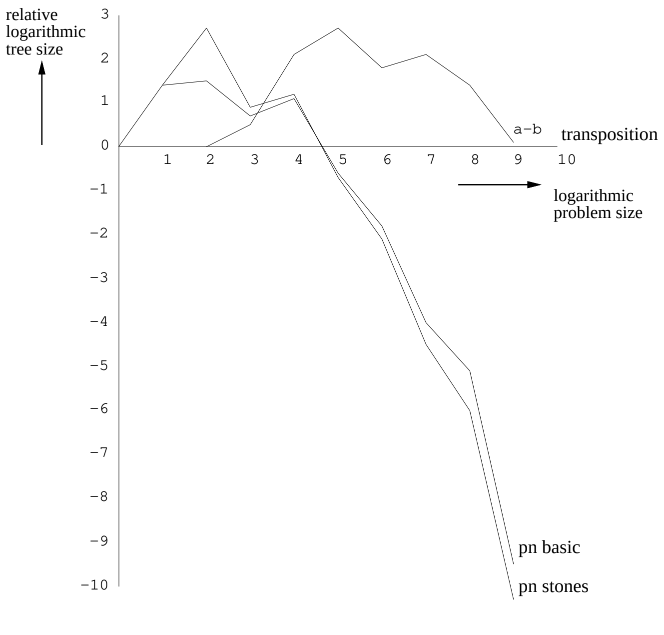


Figure 2.10: Comparison based on grouping by **transpositions**

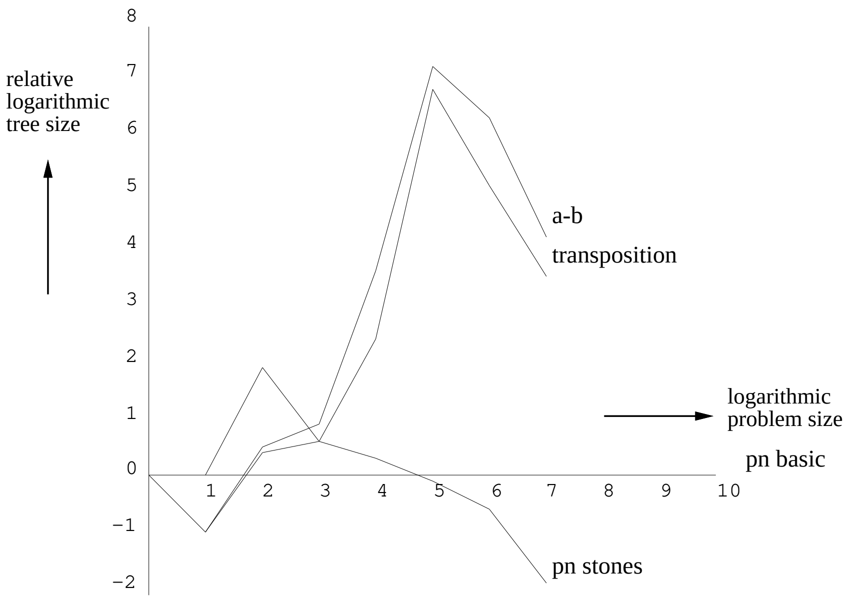


Figure 2.11: Comparison based on grouping by **basic pn**

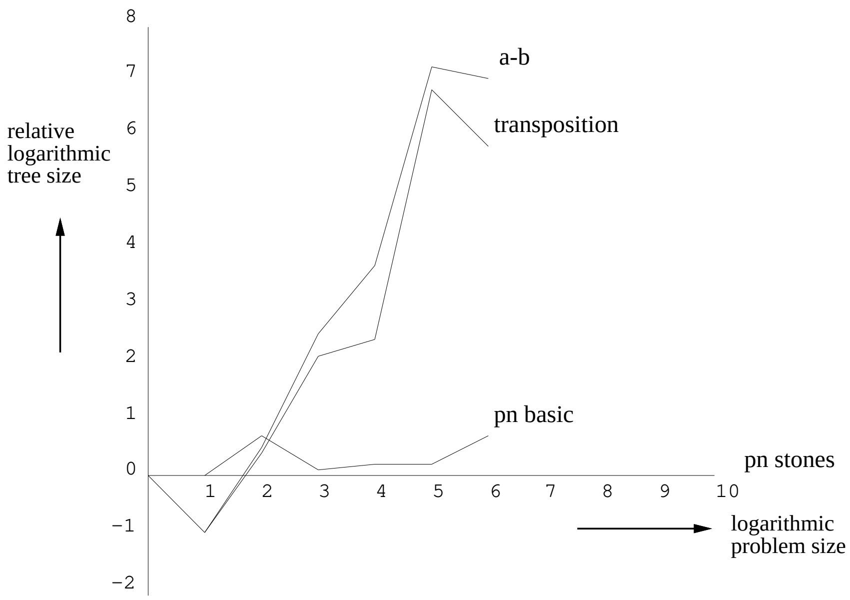


Figure 2.12: Comparison based on grouping by **stones pn**

10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
15	43	46	65	75	57	35	1

Table 2.14: Positions per group, grouped by all four algorithms.

algorithms.

In figures 2.9 and 2.10 we see that on small problems α-β search does somewhat better, while with increasing problem size, pn-search does better and better. For the largest problems, the gain factor is around 500.

In figures 2.11 and 2.12, again pn-search does worse on the smallest problems and quickly starts doing better on increasing problem size. It is remarkable that the gain factor reduces when the problem size further increases. The cause of this phenomenon is described below.

In each figure the algorithm used as grouping criterion plays an important role. In the first few groups we find positions which were suitable for that type of algorithm, while in the last few groups the positions found were difficult to solve for the algorithm. It is thus to be expected that in the graphs the other algorithms will do somewhat worse in the first groups, while they do somewhat better on the last groups.

This is exactly what can be seen in all four graphs. In figures 2.9 and 2.10 pn-search outperform α-β search starting from group 4, while in figures 2.11 and 2.12 pn-search is the better algorithm from group 2 onwards. Furthermore, in the first two graphs pn-search's gain factor towards the last few groups grows remarkably fast, while in the second two graphs, with pn-search as the grouping criterion, pn-search's advantage reduces in the last two groups.

Thus, when looking at the groups for the hard problems, figures 2.9 and 2.10 are too flattering towards pn-search while figures 2.11 and 2.12 do not give pn-search full credit.

As a solution to this problem, we present one final graph. This time we have determined the size of a problem in a more elaborate way. For each solution by an algorithm, we determine the log_{10} of the number of nodes visited. For the four algorithms we then determine the average of these exponents and use it as group number. The number of positions per group has been tabulated in table 2.14. We average the logs since node counts tend to grow exponentially instead of linearly.

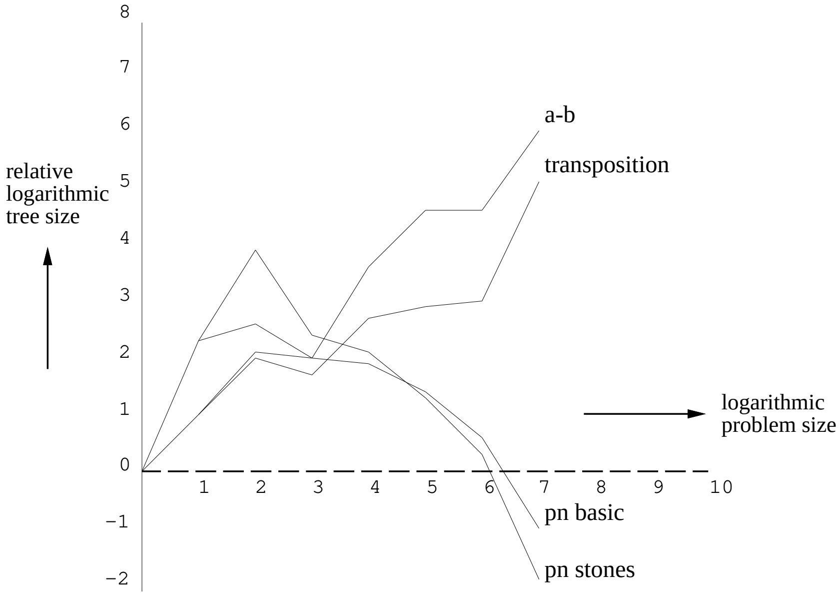


Figure 2.13: Comparison based on grouping using all four algorithms.

The singleton last group has been deleted, and its position has been added to the second last group, making a total of 36 entries in that group. The graph produced by this grouping criterion is pictured in figure 2.13. The numbers on the axes have the same meaning as in figures 2.9, 2.10, 2.11 and 2.12. In it, the bias towards a single algorithm no longer exists. The figure confirms the suggestion from the previous four figures, that pn-search's gain factor, compared with α-β search, grows with increasing problem size.

2.4.8 Conclusions

In this section we have compared the behavior of two pn-search variants with two variants of α-β search. The comparisons lead to clear conclusions: pn-search significantly outperforms both variants of α-β search (cf. table 2.12) in proving game-theoretic values in awari. The gain factor difference between pn-search and the α-β variants tends to increase with increasing problem size (cf. figure 2.13).

We further conclude from table 2.12 that a domain-dependent initialization can be beneficial on awari, with the enhancement yielding a profit of about a factor 2. Moreover, the removal of solved subtrees in pn-search decreases the working memory requirements by a factor of about 2, in problems which are ultimately solved.

We believe that the success of pn-search on awari is due to the non-uniformity of the tree. Allis *et al.* (1991b) have attempted to measure the degree of non-uniformity necessary for pn-search to outperform alternative algorithms. The results of this section show that awari's non-uniformity warrants the selection of pn-search for proving game-theoretic values instead of α-β search variants.

We tentatively conclude from these results that pn-search has contributed significantly to proving the game-theoretic values of other non-uniform trees, such as those of connect-four, qubic (see chapter 4) and go-moku (see chapter 5).

2.5 Related algorithms

In this chapter we have presented pn-search as an AND/OR tree search algorithm. Its roots, however, lie within the game-tree search algorithms. So far we have applied pn-search only to game trees, including awari, chess (Breuker *et al.*, 1994), connect-four (Allis, 1988), give-away chess, go-moku (see chapter 5), othello (Gnodde, 1993) and qubic (see chapter 4). In our discussion of related algorithms we will therefore focus mainly on game-tree search algorithms. In this section, we discuss the relationships with conspiracy-number search, SSS*, B* and A*, the latter being the only single-agent search algorithm in the list.

2.5.1 Conspiracy-number search

Conspiracy-number search (cn-search) is pn-search's direct ancestor. Cn-search was developed in the middle of the 1980s by McAllester, and has received attention of many researchers since then (McAllester, 1988; Klingbeil and Schaeffer, 1988; Klingbeil, 1989; Schaeffer, 1989; Van der Meulen, 1990; Allis *et al.*, 1991b; Lister and Schaeffer, 1994).

While pn-search focuses on the minimum number of nodes which must conspire to *prove* the value of a position, cn-search determines the minimum number of nodes which must conspire to *change* the value of a position. This main difference is more apparent when looking at the search tree: pn-search does not use a heuristic evaluation function to evaluate non-terminal nodes, while cn-search does.

Subtle differences between cn-search and pn-search can be identified by creating an instantiation of cn-search which resembles pn-search as much as possible. To do so, we define a three-valued evaluation function for cn-search, which returns -1 for a disproved node, 0 for a node with value unknown, and 1 for a proved node. In such a tree, the conspiracy numbers for -1 and 1 of a node correspond to the proof and disproof numbers of that node. These algorithms only differ in the manner in which the next node to be developed is selected, for which unpublished experiments on connect-four have shown that the selection mechanism of pn-search performs better than the selection mechanism of cn-search.

In cn-search, for any potential value v of the evaluation function it is determined for each subtree how many nodes, say N_v, within the subtree must change their evaluation value to v, to change the value of the subtree to v. If N_v for the root exceeds a certain limit, for all v unequal to the current root value, cn-search assumes that the current root value is reliable and terminates the search. Schaeffer's implementation showed that cn-search could achieve good results in tactical chess positions (Schaeffer, 1989). Unfortunately, experiments with tournament chess programs (Van der Meulen, 1990) have not been successful.

We remark that, despite pn-search's success in analyzing awari positions, we do not claim that pn-search is better suited than cn-search to perform well in a tournament chess program. Instead, we claim that the ideas behind cn-search, such as applied in pn-search, are better suited for *proving* values, than for determining the reliability of heuristic root values. Pn-search capitalizes on this suitability, concentrating on proving only. We do envision applications in tournament programs, as we have in our awari program. For instance, Breuker *et al.* (1994) have shown that pn-search may be an asset to chess programs, to prove quickly whether a mating sequence exists in a given chess position.

We conclude that cn-search and pn-search are closely related, with pn-search focusing on a different goal and being successful at it.

2.5.2 SSS*

With the availability of large internal memories, algorithms which store the search in working memory have become of practical interest. One of the earliest game-tree search algorithms which uses a stored tree is SSS* (Stockman, 1979; Campbell and Marsland, 1983).

SSS* and pn-search are both best-first search algorithms. At each step in the algorithm a node is selected according to a certain criterion and then developed. This process is repeated until the tree has been solved, or resources have run out. An important similarity between SSS* and pn-search is that neither algorithm uses a heuristic evaluation function for internal nodes. Only leaf nodes are assigned a value, either by a heuristic evaluation function or by reliable game knowledge.

The main difference between the algorithms is the criterion which determines the selection of the next node. SSS* selects a node purely based on the upper bound still achievable. At any point during the search the node which has the highest possible upper bound is selected, while among equals the leftmost node in the tree is preferred. Pn-search does not use a range of terminal-node values. Instead, the set of possible terminal-node values is split into two. Solving the tree means determining in which of the two sets the true value lies. If the exact value from a larger range must be determined, pn-search should be called repeatedly, for instance by having pn-search be the discriminating function in a binary search. While pn-search does not use a range of values, it bases its selection on the proof and disproof numbers implying that a node is tried which may be part of a solution with minimal effort.

A predecessor of pn-search, viz. $\alpha\beta$-cn search (Allis *et al.*, 1991b), can be seen as a hybrid form of pn-search and SSS*. It uses both a range of values *and* proof and disproof numbers (although these were named differently) to determine the next node to be developed. The main criterion is the range of possible values, like in SSS*, while in case of a tie the proof and disproof numbers are used. It can be shown, however, that trees exist with solutions of only a few nodes, in which $\alpha\beta$-cn search could spend a long time in irrelevant subtrees (Allis *et al.*, 1994). The solution to this problem consisted of reducing the impact of the range of values, while enlarging the role of the proof and disproof numbers. The result of this change has been the development of pn-search.

For a comparison of SSS* and $\alpha\beta$-cn search on random trees, see Allis *et al.* (1991b).

2.5.3 B*

B* is a best-first game-tree search algorithm introduced by Berliner (1979). It assumes that at each frontier node a special evaluation function returns a reliable lower and upper bound on the true value of the node. After a node is expanded, the lower and upper bounds of a node are calculated by maximizing (or minimizing, depending on the node type) the lower and upper bounds of its children. Let us assume that the root of the tree is a max node. Let us further assume that the root has two children A and B, with values in the intervals $[0, 2]$ and $[1, 3]$. B*'s main goal is to determine the best move, without necessarily knowing the exact *value* of such a move. In our example, B is the most-promising child of the root. Before we can terminate the search, however, we should either prove that the upper bound (2) on A's interval can be reduced to a value below the lower bound of B, which currently equals 1, or we should prove that the lower bound of B can be raised to at least the value of A's upper bound. These two different strategies are called *prove* and *disprove*.

Focusing both on proving and disproving is a similarity with pn-search. However, a difference with pn-search is that there is no way to simultaneously work on both strategies. Thus, in B*, at each step first a choice must be made for one of the strategies, followed by the selection of a node. Of course, after each node expansion, a change of strategies may take place. Since B* does not assume that some nodes may change their bounds more easily than others, we suggest that the concept of proof and disproof numbers could be a useful addition to B*.

An important prerequisite of B* is the *reliable* evaluation function which determines the lower and upper bound per node. Such an evaluation function heavily depends on domain-specific knowledge, and may be a serious obstacle in many domains. If, however, the knowledge to create such a function is readily available, B* provides a sound mechanism to incorporate it to guide the search process. An alternative way to obtain these bounds through a small search has been described by (Palay, 1982). For pn-search such a clear mechanism has not yet been formulated. In this respect B* has advantages above pn-search.

2.5.4 A*

A*, a single-agent search algorithm, has links with pn-search. A* is a best-first search algorithm, which uses an admissible evaluation function at each frontier node. Such a function calculates a lower bound on the total costs of the path from the root to a solution through that node. At each step a node with minimal lower bound on the solution costs is developed. A* thus guarantees finding an optimal solution (Hart *et al.*, 1968; Hart *et al.*, 1972; Nilsson, 1980).

Where A* concentrates on the cheapest *overall* solution, including the effort already spent (i.e., the cost of the path from root to frontier node), pn-search selects a node on the basis of the cheapest *remaining* solution, thus ignoring the contribution of already solved nodes and the path length from the root to the most-proving node. As a result, pn-search is not guaranteed to find the solution tree of minimal size.

Surprisingly, a small change to pn-search is sufficient to let it find the minimal solution tree. If, at each internal node, we add one to the proof number and disproof number as calculated from its children's proof and disproof numbers, then the proof number and disproof number at each node are a lower bound on the size of a solution tree for the node. We remark that proof and disproof numbers now can only increase, making some changes to the algorithm necessary. This algorithm, originating from discussions with Ingo Althöfer, has been named MST*, short for Minimal Solution-Tree search.

MST*, as variant of pn-search, will be subject of future research.

Chapter 3

Dependency-Based Search

3.1 Introduction

In section 2.1, we argued that choosing a representation and performing a search are two interacting subprocesses of problem solving. Better representations of a problem may result in smaller state spaces, and better search algorithms may traverse a given state space more efficiently. While the game-tree search algorithm pn-search (chapter 2) focuses on the latter, the single-agent search algorithm dependency-based search (db-search) introduced in this chapter, focuses on the former.

Atomic vs. structured states

Search problems are often modeled by treating states as *atomic* entities. This means that two states are considered as either equal or different, without the option of a measure of similarity between states.

As an alternative to atomic state representations, states can be *structured*, such as in STRIPS (Fikes and Nilsson, 1971). In STRIPS, each state is defined as a set of attributes. Each operator f is specified by a *precondition set*, a *delete set* and an *add set*. In any state A containing the attributes of the precondition set of f, f can be applied, yielding a state B. B consists of the attributes of A with the attributes of the delete set of f removed and with the attributes of the add set of f added.

To see how a structured state representation may help in reducing the size of a state space consider a production system P consisting of 10 rewriting rules $r_0, r_1, \ldots, r_9$.

$$\begin{array}{ccc}
0 & \xrightarrow{r_0} & a \\
1 & \xrightarrow{r_1} & l \\
2 & \xrightarrow{r_2} & t \\
3 & \xrightarrow{r_3} & o \\
4 & \xrightarrow{r_4} & g \\
5 & \xrightarrow{r_5} & e \\
6 & \xrightarrow{r_6} & t \\
7 & \xrightarrow{r_7} & h \\
8 & \xrightarrow{r_8} & e \\
9 & \xrightarrow{r_9} & r
\end{array}$$

Furthermore, we consider production system P', which contains the 10 rules of P as well as the rule r_{10}.

$$\text{altogether} \xrightarrow{r_{10}} \text{goal}$$

Rule r_{10} states that the string *altogether* may be replaced by the string *goal*. For both P and P', we start with the initial string 0123456789. The goal of both P and P' is to create the string *goal*. Clearly, in P there is no solution, while any order of applying rules r_0 to r_9, followed by the application of r_{10} leads to the goal in P'.

First, let us represent P using atomic states. The state space will consist of $2^{10} = 1024$ states, each representing a mixture of digits and lower-case letters. The state space of P' consists of the same 1024 states as P, with one additional state consisting of the string *goal*. Without the application of domain-specific knowledge, searching P consists of traversing the full state space of 1024 states. The number of states visited in P' depends on the search algorithm applied. Depth-first search visits the goal as 11th state, while breadth-first search visits the goal state as number 1025.

Second, let us represent P and P' using structured states. A possible structure consists of attributes of the form $a(i, z)$, where $i \in \{0, \ldots, 9\}$, and $z \in \{0, \ldots, 9, a, e, g, h, l, o, r, t\}$. An attribute $a(i, z)$ indicates that letter or digit z occupies position i in the string represented by the set of attributes. In P' we have an additional attribute g representing the string *goal*. The rule r_0 can now be represented by its precondition set $\{a(0, 0)\}$, its delete set $\{a(0, 0)\}$ and its add set $\{a(0, a)\}$. Similarly, rule r_5 is represented by its precondition set $\{a(5,5)\}$, its delete set $\{a(5,5)\}$ and its add set $\{a(5,e)\}$. The rule r_{10} is represented by its precondition set

$$\{a(0,a), a(1,l), a(2,t), a(3,o), a(4,g), a(5,e), a(6,t), a(7,h), a(8,e), a(9,r)\},$$

its delete set

$$\{a(0,a), a(1,l), a(2,t), a(3,o), a(4,g), a(5,e), a(6,t), a(7,h), a(8,e), a(9,r)\},$$

and its add set

$$\{a(0,g), a(1,o), a(2,a), a(3,l)\}.$$

The number of states in the state space, as well as the number of states visited by depth-first search and breadth-first search algorithms are equivalent to the numbers found for atomic states.

The difference between the atomic and the structured state representations is that the structure of states provides us with a framework for reasoning about relations between states and operators (e.g., rewriting rules), *without* having to rely on domain-specific knowledge. As an example of such a relation between operators we state that any two rules r_i and r_j, for $0 \leq i < j \leq 9$ are *independent*, meaning that in any state where both rules can be applied, changing the order of application does not influence the outcome.

Clearly, all relations which can be found by using structured state representations can also be found through a domain-specific analysis of the problem at hand. The advantage of a general framework using structured states as introduced in this chapter is that the analysis is performed once and for all for a category of problems.

In this chapter we define a framework, based on structured states and STRIPS-like operators. Within the framework, a set of conditions has been identified which are sufficient to prove that a reduction of the state space can be performed without the loss of solutions in the state space.

Conventional search algorithms cannot traverse the reduced state space; but the db-search algorithm can. It is proved that db-search, introduced for the purpose, traverses exactly the reduced state space.

To give an indication of the amount of state-space reduction achieved by our framework, we once again look at the state space defined for production systems P and P'. For P the reduced state space consists of 11 elements (an initial state and 10 states representing the changes by rules $r_0, \ldots, r_9$). For P' the reduced state space consists of 12 elements (one additional state representing *goal*). These numbers should be compared with the 1024 and 1025 found for the atomic-state representation.

Overview of the chapter

In section 3.2 we describe the double-letter puzzle (DLP), which is used as an example throughout the chapter. In section 3.3 we formally define a framework for a category of single-agent searches based on structured-state representations. Each definition in this section is illustrated by its application to DLP. In section 3.4 db-search is described informally using the framework introduced in the previous section, by applying it to an instance of DLP. In section 3.5 we present algorithms in pseudo-code for db-search. In section 3.6 we compare the performances on DLP of db-search and depth-first search. Finally, in section 3.7 the scope of applicability of db-search is discussed. For practical results of db-search we refer to chapters 4 and 5.

3.2 The double-letter puzzle

The double-letter puzzle (DLP) is a production system consisting of an *axiom* and a set of 10 *rewriting rules*. The axiom is an element of $\{a, b, c, d, e\}^+$. The rewriting rules are listed below.

$$\begin{array}{rcl} aa & \rightarrow & e \mid b \\ bb & \rightarrow & a \mid c \\ cc & \rightarrow & b \mid d \\ dd & \rightarrow & c \mid e \\ ee & \rightarrow & d \mid a \end{array}$$

The rewriting rules can be informally described as allowing any double occurrence of a letter to be replaced by a single instance of its alphabetical predecessor or successor in a circular alphabet.

We define the set of theorems of DLP as follows:

1. The axiom is a theorem

2. If x is a theorem and there exists a rewriting rule r such that $x \xrightarrow{r} y$, then y is a theorem.

3. There are no theorems except as defined by 1. and 2.

Each theorem of length 1 (i.e., a theorem consisting of a single letter) is called a solution to DLP.

Two solutions to instance $aabdcbbdcaa$ of DLP are presented below.

$$aabdcbbdcaa \stackrel{aa\to b}{\longrightarrow} bbdcbbdcaa \stackrel{bb\to a}{\longrightarrow} adcbbdcaa \stackrel{bb\to c}{\longrightarrow} adccdcaa \stackrel{cc\to d}{\longrightarrow}$$
$$\stackrel{cc\to d}{\longrightarrow} adddcaa \stackrel{dd\to c}{\longrightarrow} adccaa \stackrel{cc\to d}{\longrightarrow} addaa \stackrel{dd\to e}{\longrightarrow} aeaa \stackrel{aa\to e}{\longrightarrow} aee \stackrel{ee\to a}{\longrightarrow}$$
$$\stackrel{ee\to a}{\longrightarrow} aa \stackrel{aa\to b|e}{\longrightarrow} b \mid e$$

$$aabdcbbdcaa \stackrel{aa\to b}{\longrightarrow} bbdcbbdcaa \stackrel{bb\to c}{\longrightarrow} cdcbbdcaa \stackrel{bb\to c}{\longrightarrow} cdccdcaa \stackrel{cc\to d}{\longrightarrow}$$
$$\stackrel{cc\to d}{\longrightarrow} cdddcaa \stackrel{dd\to c}{\longrightarrow} ccdcaa \stackrel{cc\to d}{\longrightarrow} ddcaa \stackrel{dd\to c}{\longrightarrow} ccaa \stackrel{cc\to b}{\longrightarrow} baa \stackrel{aa\to b}{\longrightarrow}$$
$$\stackrel{aa\to b}{\longrightarrow} bb \stackrel{bb\to a|c}{\longrightarrow} a \mid c$$

From the examples we see that a, b, c and e can be deduced. For a proof that d cannot be deduced from $aabdcbbdcaa$, we refer to appendix A.

3.3 A formal framework for db-search

In this section we define a formal framework for db-search. The framework is described in four steps. In section 3.3.1 we define states and operators. In section 3.3.2 we define paths through the state space and classes of equivalent paths. It is shown that conventional search algorithms traverse exactly the set of all paths. In section 3.3.3 key classes are defined. These form a subset of the classes of paths defined previously. It is shown that, under accurately defined circumstances, the set of all key classes is complete, meaning that each solution path is represented by a key class. In section 3.3.4 we define a meta-operator for traversing the state space defined by the set of all key classes. It is shown that the meta-operator is sound and complete, meaning that through application of the meta-operator exactly all key classes are visited. Finally, in section 3.3.5, we summarize the properties of our framework.

The description of the framework for db-search requires a large number of definitions. For reference purposes, we have listed the symbols used in this section and a short description of their meaning in table 3.1. Each definition in this section is illustrated by its application to the instance of DLP with axiom $aacc$.

3.3.1 States and operators

In this section we first define the set of attributes U and the state space U_s. Then we define operators (consisting of a precondition set, a delete set and an add set) which map states onto other states, followed by the set of all

symbol	description
U	the set of all attributes
U_s	the state space
U_i	the set of all initial states
U_g	the set of all goal states
U_f	the set of all operators
U_p	the set of all paths applicable to initial states
U_k	the set of all key classes
f	an operator
f^{pre}	the precondition set of operator f
f^{del}	the delete set of operator f
f^{add}	the add set of operator f
$f(S)$	the state reached when applying operator f to S
$f_1 \prec f_2$	f_1 supports f_2, f_2 depends on f_1
$f_1 \ll f_2$	f_1 precedes f_2
$f_{(p,q,r,z_1,z_2)}$	an operator in DLP
P	a path
$P \sim Q$	the concatenation of paths P and Q
$P \equiv Q$	paths P and Q are equivalent
$P \cong Q$	P and Q are transpositions
$P(S)$	the state resulting from applying path P to state S
$[P]_\equiv$	the equivalence class of path P
$U_p/_\equiv$	the set of equivalence classes of U_p
$key(P)$	the key operator (last operator) of path P
$P \parallel Q$	the merge of paths P and Q
$Par_f(P)$	the set of parents of operator f in path P
$Anc_f(P)$	the set of ancestors of operator f in path P
Ax	the axiom state of DLP

Table 3.1: Symbols used in db-search framework

operators U_f. Finally, we define the set U_i of all initial states, and the set U_g of all goal states.

Definition 3.1 *Let U be a set of attributes. Then the* state space *U_s is defined as 2^U, the power set of U.*

We index the letters of the axiom in DLP from 0 to $n-1$, where n is the length of the axiom (i.e., 4 for DLP with axiom *aacc*). In the axiom, the first a has index 0, the second a has index 1, the first c has index 2 and the second c has index 3. Each letter in a theorem originates from a substring of the axiom. We represent a letter z in a theorem by three values: the first and last index of the substring of the axiom z originates from, and z itself. If *aab* is produced from *aacc*, the letter b originates from the substring *cc* in the axiom, which has first index 2 and last index 3. Therefore, the b in *aab* is represented by $A(2,3,b)$.

The set of all attributes U is specified as follows.

$$U = \{A(i,j,z) \mid 0 \leq i \leq j \leq 3 \wedge z \in \{a,b,c,d,e\}\}$$

As the axiom will play a special role in many of the definitions of this section, we denote the state representing the axiom *aacc* by Ax. In accordance with definition 3.1, $Ax \in U_s$ is represented as follows.

$$Ax = \{A(0,0,a), A(1,1,a), A(2,2,c), A(3,3,c)\}$$

Definition 3.2 *We define an* operator *f as a 3-tuple $\langle f^{pre}, f^{del}, f^{add}\rangle$, with $f^{pre}, f^{del}, f^{add} \subseteq U$ and $f^{del} \subseteq f^{pre}$. The elements in the 3-tuple are named the* precondition set*, the* delete set *and the* add set *of f, respectively. Operator f is a partial function $f : U_s \longrightarrow U_s$, defined as $f(S) = (S \setminus f^{del}) \cup f^{add}$, for all $S \supseteq f^{pre}$.*

Definition 3.2 states that an operator f is applicable to each state containing all attributes in the precondition set of f. Applying operator f to state S yields a state T, by deleting the attributes of the delete set of f from S and adding to the result the attributes of the add set of f. In DLP, two equal adjacent letters z_1 are replaced by z_2, which is either the successor or predecessor of z_1. The two z_1s originate from two adjacent substrings in the axiom. Let the first z_1 originate from the substring with start index p and end index q, and let the second z_1 originate from the substring with start index $q+1$ and end index r. Then, the indices p, q, and r, and the letters z_1

and z_2 are sufficient information to define an operator. In the following, we denote with $f_{(p,q,r,z_1,z_2)}$ the operator

$$\langle\{A(p,q,z_1), A(q+1,r,z_1)\}, \{A(p,q,z_1), A(q+1,r,z_1)\}, \{A(p,r,z_2)\}\rangle.$$

An example operator in DLP is

$$f_{(0,0,1,a,b)} = \langle\{A(0,0,a), A(1,1,a)\}, \{A(0,0,a), A(1,1,a)\}, \{A(0,1,b)\}\rangle$$

Applying $f_{(0,0,1,a,b)}$ to axiom state Ax yields $\{A(0,1,b), A(2,2,c), A(3,3,c)\}$.

Definition 3.3 *The set of operators defined within a domain is denoted by U_f.*

Using definition 3.3 we define for our instance of DLP the set of operators U_f as

$$\{f_{(p,q,r,z_1,z_2)} \mid 0 \leq p \leq q < r \leq 3 \wedge z_1 \in \{a,b,c,d,e\} \wedge z_2 \in \mathit{succpred}(z_1)\}$$

Here *succpred*(z) denotes a set containing the circular alphabetical successor and predecessor of z.

Definition 3.4 *We denote the set of* initial states *by U_i, with $U_i \subseteq U_s$. We denote the set of* goal states *by U_g, with $U_g \subseteq U_s$.*

For our instance of DLP,

$$\begin{aligned} U_i &= \{\{A(0,0,a), A(1,1,a), A(2,2,c), A(3,3,c)\}\} \\ U_g &= \{\{A(0,3,a)\}, \{A(0,3,b)\}, \{A(0,3,c)\}, \{A(0,3,d)\}, \{A(0,3,e)\}\}. \end{aligned}$$

3.3.2 Paths

In this section we first define paths, which are just sequences of operators. We define the application of a path to a state S, as one by one applying the operators, starting from state S. Then solutions for a state S are defined as the paths which, if applied to S yield a superset of a goal state. We then define the extension of a path P, which is a path consisting of all operators of P, in the same order, plus one additional operator. An equivalence relation for paths is defined, which states that two paths are equivalent if one is a permutation of the other. Then, a notation for equivalence classes of paths is introduced. Finally, we describe the behavior of conventional search algorithms in terms of paths.

Definition 3.5 *Any element P of ${U_f}^*$ is a* path. *Let $P = (f_1, \ldots, f_n)$ be a path. Let concatenation of two paths P and Q be denoted by $P \sim Q$. Then, P* is applicable *to S if (1) $P = \epsilon$, or (2) $P = (f) \sim Q$ and $f(S)$ is defined and Q is applicable to $f(S)$. If P is applicable to S, then*

$$P(S) = f_n(f_{n-1}(\ldots(f_2(f_1(S)))\ldots)).$$

For path $P = (f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}, f_{(0,1,3,b,c)})$, applicable to the axiom state Ax, it follows from definition 3.5 that

$$\begin{aligned}
P(Ax) &= f_{(0,1,3,b,c)}(f_{(2,2,3,c,b)}(f_{(0,0,1,a,b)}(Ax))) = \\
&= f_{(0,1,3,b,c)}(f_{(2,2,3,c,b)}(\{A(0,1,b), A(2,2,c), A(3,3,c)\})) = \\
&= f_{(0,1,3,b,c)}(\{A(0,1,b), A(2,3,b)\}) = \\
&= \{A(0,3,c)\}
\end{aligned}$$

Definition 3.6 *The set of paths U_p, is defined as follows.*

$$U_p = \{P \mid S \in U_i \wedge P \text{ is applicable to } S\}$$

It can be checked that for our instance of DLP with initial state Ax, U_p (definition 3.6) consists of 17 paths.

$$\begin{aligned}
U_p \;=\; &\{\epsilon, (f_{(0,0,1,a,b)}), (f_{(0,0,1,a,e)}), (f_{(2,2,3,c,b)}), (f_{(2,2,3,c,d)}), \\
&(f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}), (f_{(2,2,3,c,b)}, f_{(0,0,1,a,b)}), (f_{(0,0,1,a,b)}, f_{(2,2,3,c,d)}), \\
&(f_{(2,2,3,c,d)}, f_{(0,0,1,a,b)}), (f_{(0,0,1,a,e)}, f_{(2,2,3,c,b)}), (f_{(2,2,3,c,b)}, f_{(0,0,1,a,e)}), \\
&(f_{(0,0,1,a,e)}, f_{(2,2,3,c,d)}), (f_{(2,2,3,c,d)}, f_{(0,0,1,a,e)}), \\
&(f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}, f_{(0,1,3,b,a)}), (f_{(2,2,3,c,b)}, f_{(0,0,1,a,b)}, f_{(0,1,3,b,a)}), \\
&(f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}, f_{(0,1,3,b,c)}), (f_{(2,2,3,c,b)}, f_{(0,0,1,a,b)}, f_{(0,1,3,b,c)})\}
\end{aligned}$$

Definition 3.7 *Let $P = (f_1, \ldots, f_n)$ be a path applicable to S. We define the following terminology with respect to P.*

1. *P is a* solution *for S, if $\exists x \in U_g\ x \subseteq P(S)$.*
2. *A path Q is an* extension *of P, if $Q = P \sim (f)$, for some operator f.*

We give examples for definition 3.7 using path

$$P = (f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}, f_{(0,1,3,b,c)}).$$

1. P is a solution for axiom state Ax, because $P(Ax) = \{A(0,3,c)\}$ and $\{A(0,3,c)\} \in U_g$.

2. P is an extension of path $(f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)})$.

.

Definition 3.8 *Let P and Q be paths. P and Q are* equivalent*, denoted by $P \equiv Q$, if P is a permutation of Q.*

An example of definition 3.8 from the set of paths in DLP applicable to Ax is

$$(f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}, f_{(0,1,3,b,c)}) \equiv (f_{(2,2,3,c,b)}, f_{(0,0,1,a,b)}, f_{(0,1,3,b,c)})$$

Definition 3.9 *Let $P \in U_p$ be a path. We denote the set of all paths $Q \in U_p$ such that $P \equiv Q$ by $[P]_\equiv$ (the equivalence class of P modulo $\equiv$). The set of all equivalence classes of U_p modulo $\equiv$ is denoted by $U_p/_\equiv$.*

From definition 3.9 and the example after definition 3.6 it follows that for

$$P = (f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}, f_{(0,1,3,b,c)}),$$

$$[P]_\equiv = \{(f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}, f_{(0,1,3,b,c)}), (f_{(2,2,3,c,b)}, f_{(0,0,1,a,b)}, f_{(0,1,3,b,c)})\}.$$

We mention that in our instance of DLP $U_p/_\equiv$ consists of 11 equivalence classes.

Traversing U_p

In this section we describe how a conventional tree search algorithm traverses U_p, as defined within our framework.

As an example tree search algorithm, we discuss *depth-first search* (DFS). Starting from initial state Ax, DFS traverses a tree such that each node N represents a path P applicable to initial state Ax. At node N, an operator f of U_f can be applied, if f is applicable to $P(Ax)$. In other words, f can be applied at node N, if $P \sim (f)$ is applicable to Ax, i.e., $P \sim (f) \in U_p$. Clearly, DFS will traverse a finite U_p fully, unless terminated early.

A reduction of state space U_p is applied in many practical domains. We say that $P \cong Q$ if $P(Ax) = Q(Ax)$. Thus, if $P \cong Q$, then $P(Ax)$ and $Q(Ax)$ are transpositions. From the definition of a path, it is clear that in such a case P and Q can be extended in exactly the same way. Thus, even though several paths may lead to the same state, the continuations from that state need to be investigated only once. Instead of traversing U_p, we may therefore restrict ourselves to traversing $U_p/_{\cong}$. To do so, *transposition tables* are used to store the results of investigating the continuations starting at each node. Before investigating a node, it is checked whether the node has already been investigated (indicating that the node is a transposition) (Greenblatt *et al.*, 1967).

We conclude that conventional tree search algorithms traverse the state space U_p, which may be reduced by investigating each transposition only once.

3.3.3 Key classes

In this section we define the key operator of a path (which is just the last operator of the path), key classes (which are equivalence classes of paths where all paths have the same key operator), and the set of all key classes. We define monotonicity, which indicates that in the course of executing operators, an attribute can never be recreated after it has been deleted. We define singularity, which means that each goal state consists of a single attribute. Furthermore, we define redundant paths, which are extensions of solutions. Finally, we show that the set of all key classes is complete under the condition of monotonicity, singularity and the absence of redundancy. Completeness means that each solution in U_p is an element of a key class.

Definition 3.10 *Let $P = (f_1, \ldots, f_n)$ be a path applicable to S. The last operator of a non-empty path P (i.e., f_n), is called the* key operator *of the path. Notation: $key(P) = f_n$.*

For path $P = (f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}, f_{(0,1,3,b,c)})$ we obtain from definition 3.10 that $key(P) = f_{(0,1,3,b,c)}$.

Definition 3.11 *Let $C \in U_p/_{\equiv}$ be a class. C is a* key class*, if for all $P_i, P_j \in C$, $key(P_i) = key(P_j)$. The set of all key classes of $U_p/_{\equiv}$ is denoted by U_k. The* key *of a key class C is defined to equal the key of the paths in C and is denoted by $key(C)$.*

From definition 3.11 and the example after definition 3.9, it follows that for $P = (f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}, f_{(0,1,3,b,c)})$, $[P]_{\equiv}$ is a key class. For $Q = (f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)})$, $[Q]_{\equiv}$ is not a key class, since Q has key $f_{(2,2,3,c,b)}$, while $(f_{(2,2,3,c,b)}, f_{(0,0,1,a,b)})$ has key $f_{(0,0,1,a,b)}$. We note that U_k for our instance of DLP consists of 7 key classes.

Definition 3.12 *A path $P = (f_1, \ldots, f_n)$ applicable to S is* monotonous for *S if*

$$\forall_{i \neq j}\ f_i^{add} \cap f_j^{add} = \emptyset \ \wedge\ \forall_i\ S \cap f_i^{add} = \emptyset.$$

U_p is monotonous if all paths in U_p are monotonous for all $S \in U_i$.

In our instance of DLP, there are 17 paths. Investigation shows that each path P is monotonous for Ax. From definition 3.12 it follows that U_p is monotonous.

Definition 3.13 *We say that U_g is* singular *if each $S \in U_g$ consists of a single attribute, i.e., $|S| = 1$.*

The U_g defined for DLP,

$$U_g = \{\{A(0,3,a)\}, \{A(0,3,b)\}, \{A(0,3,c)\}, \{A(0,3,d)\}, \{A(0,3,e)\}\}$$

is singular, according to definition 3.13.

Definition 3.14 *A path Q is* redundant, *if Q is an extension of P, and P is a solution for an initial state, or P is redundant. U_p is non-redundant, if no path in U_p is redundant.*

In DLP, there are no operators applicable to goal states. Therefore, there are no redundant paths in DLP, as defined in definition 3.14.

Completeness of U_k

In section 3.3.2 we have shown that conventional search algorithms traverse U_p. Through the equivalence relation $\equiv$, we have defined classes of paths, $U_p/_\equiv$. Of these classes, the subset U_k of key classes has been singled out. In this section we will show that to find all solutions in U_p, it is sufficient to consider only paths which are elements of key classes, thereby restricting the size of the state space. Our proof is based on the assumption that U_p is monotonous and non-redundant, and that U_g is singular.

Our proof consists of three steps. First, in lemma 3.1 we show that either all paths in a class are a solution, or none are. It follows that instead of focusing on paths, we need only to focus on classes of paths, thereby restricting our state space to $U_p/_\equiv$. Second, in lemma 3.2 we show that each equivalence class containing a solution must be a key class. Third, in theorem 3.1 we combine these two results to show that it is sufficient to examine the set of all key classes U_k.

Lemma 3.1 *Let P and Q, paths applicable to S, be elements of $[P]_\equiv$, for P and Q monotonous for S. Then $P(S) = Q(S)$.*

Proof

We assume without lack of generality that $P = (f_1, \ldots, f_n)$ for some natural n. Let $a \in P(S)$ be an attribute. Then, because of monotonicity, a is an element of exactly one of the following sets: $S, f_1^{add}, f_2^{add}, \ldots, f_n^{add}$. Now let us suppose that $a \in f_i^{del}$, for some i. Then from definition 3.2 it follows that $a \in f_i^{pre}$, restricting a to membership of exactly one of the following sets: $S, f_1^{add}, f_2^{add}, \ldots, f_{i-1}^{add}$. But then, since $a \notin f_i(\ldots(f_1(S)))$, also $a \notin P(S)$. This contradicts our assumption that $a \in P(S)$. Thus, there is no $i \in \{1, \ldots, n\}$ such that $a \in f_i^{del}$. Since Q is a permutation of P, $a \in Q(S)$ and $P(S) \subseteq Q(S)$. Analogously, $Q(S) \subseteq P(S)$. □

Lemma 3.2 *Let U_g be singular and let U_p be monotonous and non-redundant. If P is a solution applicable to S then $[P]_\equiv$ is a key class.*

Proof

Let $Q \in [P]_\equiv$. We assume without lack of generality that $Q = (f_1, \ldots, f_n)$ for some natural n. Let x be an attribute in an element of U_g. If $x \in f_p^{add}$, for some $p \in \{1, \ldots, n\}$, then $(f_1, \ldots, f_p)$ is a solution, since U_g is singular. Since U_p is non-redundant, Q is non-redundant. Thus, f_p must be the last operator (i.e., the key operator) of Q. As f_p occurs in all paths in $[P]_\equiv$, it must be the key operator in each of these paths. Thus, $[P]_\equiv$ is a key class □

Theorem 3.1 *Let U_p be monotonous and non-redundant and let U_g be singular. Then U_k is complete (i.e., each solution path in U_p is element of a class in U_k, and each class in U_k either consists of only solutions, or no solutions).*

Proof

From lemma 3.1 it follows that either all paths in the equivalence classes of $U_p/_\equiv$ are solutions, or none are. From lemma 3.2 it follows that the equivalence class modulo $\equiv$ for any solution path is a key class. Thus, for any solution path, its equivalence class is a key class, of which each representative is a solution. Thus U_k is complete. □

3.3.4 Traversing U_k

In this section we define two relations, to *support* and to *precede*, between operators. These relations create a partial order between operators in monotonous paths. Using the partial order we can define the parents (operators which directly support or precede an operator) and ancestors (operators which directly or indirectly support or precede an operator). Last, we define the merge of a set of classes, which itself is a class. The merge of a set of classes consists of paths containing exactly the operators in the paths of the classes merged. Stated more simply, if we merge a class containing path P with a class containing path Q, the merge contains all paths consisting of exactly the operators in P and Q. Operators in both P and Q occur only once in the paths of the merge.

The purpose of these definitions is to create a meta-operator which is capable of traversing exactly U_k. We have shown in section 3.3.3 that U_k is complete. Together with a proof that we have a meta-operator which traverses exactly U_k, we have shown that a restricted state space can be traversed, without reduced efficacy. The definition of the meta-operator and the proof of its soundness (each application leads to a key class) and completeness (all key classes will be created by application of the meta-operator) follow the definitions in this section.

Definition 3.15 *Let $f_1, f_2 \in U_f$. We define the two relations $\prec$* (supports) *and $\ll$* (precedes) *on $U_f \times U_f$ as follows.*

1. $f_1 \prec f_2 \iff f_1^{add} \cap f_2^{pre} \neq \emptyset$.
2. $f_1 \ll f_2 \iff f_1^{pre} \cap f_2^{del} \neq \emptyset$.

We remark that we will use both the phrases f_1 *supports* f_2 and f_2 *depends on* f_1 to describe $f_1 \prec f_2$. We provide examples in our instance of DLP, for the two relations of definition 3.15.

1. $f_{(0,0,1,a,b)} \prec f_{(0,1,3,b,a)}$, as $f^{add}_{(0,0,1,a,b)} \cap f^{pre}_{(0,1,3,b,a)} = \{A(0,1,b)\}$.
2. $f_{(0,0,1,a,b)} \ll f_{(0,0,1,a,e)}$, as $f^{pre}_{(0,0,1,a,b)} \cap f^{del}_{(0,0,1,a,e)} = \{A(0,0,a), A(1,1,a)\}$. We remark that also $f_{(0,0,1,a,e)} \ll f_{(0,0,1,a,b)}$. Which shows that $f_{(0,0,1,a,e)}$ and $f_{(0,0,1,a,b)}$ cannot occur in the same monotonous path.

Definition 3.16 *Let P be a non-empty path applicable to S, and let f be an operator in path P. The set of* parents *of f in P is defined as follows.*

$$Par_f(P) = \{f_i \mid f_i \in P \wedge (f_i \prec f \vee f_i \ll f)\}$$

Between $f_{(0,0,1,a,b)}$, $f_{(2,2,3,c,b)}$ and $f_{(0,1,3,b,c)}$ the following two relations hold: $f_{(0,0,1,a,b)} \prec f_{(0,1,3,b,c)}$ and $f_{(2,2,3,c,b)} \prec f_{(0,1,3,b,c)}$. Thus, for path $P = (f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}, f_{(0,1,3,b,c)})$, definition 3.16 states that $Par_{f_{(0,1,3,b,c)}}(P) = \{f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}\}$.

Definition 3.17 *Let P be a non-empty path applicable to S, and let f be an operator in P. The set of* ancestors *of f in P is defined as follows.*

$$Anc_f(P) = \{f\} \;\cup \bigcup_{f_i \in Par_f(P)} Anc_{f_i}(P)$$

Furthermore, a parent f_i of f is named a relevant parent *if for all parents f_j of f, with $f_j \neq f_i$, $f_i \notin Anc_{f_j}(P)$.*

In our example instance of DLP, $Anc_f(P) = \{f\} \cup Par_f(P)$, for all paths P and all operators f. In more complex instances of DLP, however, not all ancestors of f as defined in definition 3.17 will be parents of f (or f itself). In each instance of DLP, each parent is a relevant parent.

Definition 3.18 *Let $P_1, \ldots, P_n$ be paths applicable to S. Then the* merge *of $P_1, \ldots, P_n$, denoted by $P_1 \parallel \ldots \parallel P_n$, is defined as the set of all paths Q applicable to S, such that Q is a permutation of the set of all operators in the P_i. The merge of a set of classes $[P_i]_\equiv$ is defined as the merge of a set of representatives of the classes. Thus,*

$$[P_1]_\equiv \parallel \ldots \parallel [P_n]_\equiv = P_1 \parallel \ldots \parallel P_n.$$

We present three examples of merges of paths, as defined in definition 3.18.

$$\begin{aligned}
(f_{(0,0,1,a,b)}) \parallel (f_{(2,2,3,c,b)}) &= \{(f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}), (f_{(2,2,3,c,b)}, f_{(0,0,1,a,b)})\} \\
(f_{(0,0,1,a,b)}) \parallel (f_{(0,0,1,a,e)}) &= \emptyset \\
(f_{(0,0,1,a,b)}) \parallel (f_{(2,2,3,c,b)}) &= \{(f_{(0,0,1,a,b)})\} \parallel \{(f_{(2,2,3,c,b)})\}
\end{aligned}$$

A meta-operator

So far, we have defined U_k and proved its completeness, under the assumptions of singularity, non-redundancy and monotonicity. For the remainder of this section we assume that these three conditions hold, unless stated otherwise.

Traversing U_k is not as straightforward as traversing U_p. For instance, $\{(f_{(0,0,1,a,b)})\}$ is a key class in the unsolvable instance *aacaa* of DLP, whose axiom is represented by

$$\{A(0,0,a), A(1,1,a), A(2,2,c), A(3,3,a), A(4,4,a)\}.$$

We can extend the only path in the key class to the paths $(f_{(0,0,1,a,b)}, f_{(3,3,4,a,b)})$ and $(f_{(0,0,1,a,b)}, f_{(3,3,4,a,e)})$. However, in both cases the equivalence classes of these paths are not key classes. Thus, extending elements of key classes may lead to paths which are not element of a key class. We conclude that traversing U_k involves more than just extending paths.

In this section we introduce the *meta-operator* $F(N, f)$ which is capable of traversing U_k. First, we define $F(N, f)$. Then we prove that each application of $F(N, f)$ in a graph where each node represents a key class, creates only nodes representing key classes. Finally, we prove through induction that each key class is created through application of $F(N, f)$.

Definition of the meta-operator

We define meta-operator $F(N, f)$ in definition 3.19.

Definition 3.19 *Let $N \subseteq U_k$, with $N = \{C_1, \ldots, C_n\}$, all $C_i \neq \emptyset$, and $n \geq 1$. Let $C_1 \parallel \ldots \parallel C_n = C$, with $C \neq \emptyset$. Let operator $f \in U_f$, such that $\forall_{1 \leq i \leq n}$ $(key(C_i) \prec f \vee key(C_i) \ll f)$, and let f be an extension to a path $P \in C$. We then say that f is* valid *in N. $F(N, f)$ is* applicable *if and only if f is valid in N and there is no proper subset M of N, such that f is valid in M. If $F(N, f)$ is applicable, then $F(N, f) = [P \sim (f)]_{\equiv}$. Furthermore, $F(\emptyset, f)$ is applicable if and only if $f(Ax)$ is defined. In those cases, $F(\emptyset, f) = \{(f)\}$.*

An informal interpretation of $F(N, f)$ is as follows. Operator f can only be applied to states containing all elements of f^{pre}. Each element C_i of the set of key classes N contributes one or more attributes of f^{pre}, implying that f depends on or is preceded by the key operators of each C_i. If all operators in the C_i can be combined without conflicts (i.e., the merge of all C_i is not empty) and paths in the merge extended with f are applicable, then $F(N, f)$ is applicable.

We give two examples. First, we look at instance *aacc* of DLP. Both $C_1 = \{(f_{(0,0,1,a,b)})\}$ and $C_2 = \{(f_{(2,2,3,c,b)})\}$ are key classes. Operator $f = f_{(0,1,3,b,a)}$, with $f^{pre} = \{A(0,1,b), A(2,3,b)\}$ depends on the keys of the paths of C_1 and C_2. Furthermore, $C_1 \parallel C_2 = \{(f_{(0,0,1,a,b)}, f_{(2,2,3,c,b)}), (f_{(2,2,3,c,b)}, f_{(0,0,1,a,b)})\}$.

For both paths Q_1 and Q_2 in the merge, $Q_1 \sim (f)$ and $Q_2 \sim (f)$ are applicable to Ax. Finally, $F(\{C_1\}, f)$ and $F(\{C_2\}, f)$ are not valid. Thus, $F(\{C_1, C_2\}, f)$ is applicable.

Second, we look at the production system P' of section 3.1. In P', each of the applications of $r_0, \ldots, r_9$ results in a key class of one element, which we name C_0 through C_9. Rule r_{10} depends on each of the applications of r_0 to r_9 to have been executed. Thus, $F(\{C_0, \ldots, C_9\}, r_{10})$ is applicable and yields the solution of P'.

Soundness of the meta-operator

In theorem 3.2 we prove that each application of meta-operator $F(N, f)$ creates a key class. Before we give the proof of theorem 3.2, we prove lemmas 3.3 and 3.4.

Lemma 3.3 *Let $P = (f_1, \ldots, f_n)$ be a path applicable to S. Let $f_i \not\prec f_{i+1} \wedge f_i \not\ll f_{i+1}$. Then $(f_1, \ldots, f_{i-1}, f_{i+1}, f_i, f_{i+2}, \ldots, f_n)$ is also a path applicable to S.*

Proof

Let $f_{i-1}(\ldots(f_1(S))\ldots) = T$. Then $f_i(T)$ is defined, and $f_i^{pre} \subseteq T$. Since $f_i \not\prec f_{i+1}$ we know that $f_i^{add} \cap f_{i+1}^{pre} = \emptyset$. Thus, $f_{i+1}^{pre} \subseteq T$ and $f_{i+1}(T)$ is defined. Furthermore, $f_i \not\ll f_{i+1}$ implies that $f_{i+1}^{del} \cap f_i^{pre} = \emptyset$. Therefore $f_i(f_{i+1}(T))$ is defined. Since $f_i(f_{i+1}(T)) = f_{i+1}(f_i(T))$ according to lemma 3.1, $(f_1, \ldots, f_{i-1}, f_{i+1}, f_i, f_{i+2}, \ldots, f_n)$ is a path. □

Lemma 3.4 *Let C be a key class with key f_n. Let $P = (f_1, \ldots, f_n)$ be a path in C. Then the following two statements are true.*

$$1.\quad \forall_{1 \leq i \leq n-1} \quad \exists_{j>i}\ (f_i \prec f_j \vee f_i \ll f_j)$$
$$2.\quad \forall_{1 \leq i \leq n-1} \quad (f_n \not\prec f_i \wedge f_n \not\ll f_i)$$

Proof

1. Suppose that there exists an f_p, with $1 \leq p \leq n-1$, such that $\forall_{j>p}(f_p \not\prec f_j \wedge f_p \not\ll f_j)$. Then, by repeated application of lemma 3.3, we can move f_p to the end of P. However, this contradicts the assumption that C is a key class. Thus, $\forall_{1 \leq i \leq n-1} \exists_{j>i}(f_i \prec f_j \vee f_i \ll f_j)$.

2. Suppose that there exists an f_p, with $1 \leq p \leq n-1$, such that $f_n \prec f_p$. Then $f_n^{add} \cap f_p^{pre} \neq \emptyset$. Let $x \in f_n^{add} \cap f_p^{pre}$. Then, by monotonicity, $x \notin S$ and $x \notin f_i^{add}$, for all $i < n$. Thus, f_p is only applicable if $p \geq n$, which is a contradiction. Thus, $f_n \not\prec f_p$. Suppose that there exists an f_p, with $1 \leq p \leq n-1$, such that $f_n \ll f_p$. Then $f_n^{pre} \cap f_p^{del} \neq \emptyset$. Let $x \in f_n^{pre} \cap f_p^{del}$. Then, by definition 3.2 $x \in f_p^{pre}$, and either $x \in S$, or $x \in f_i^{add}$ for exactly one $i < p$, but not both. And thus, $x \notin f_j^{add}$ for all $j \geq p$. Thus, f_n is only applicable, if $n \leq p$, which is a contradiction. Thus, $f_n \not\ll f_p$. Therefore, $\forall_{1 \leq i \leq n-1}\ (f_n \not\prec f_i \wedge f_n \not\ll f_i)$. □

Theorem 3.2 *If $F(N, f)$ is applicable, then $F(N, f)$ is a key class.*

Proof

Consider arbitrary $P \in F(N, f)$ and suppose that $key(P) \neq f$. Then either $key(P) = key(P_i)$ for some P_i in a class in N or $key(P)$ is a non-key operator f_j in a path in some class in N. The first case leads to a contradiction, since $key(P_i) \prec f$ according to definition 3.19, which contradicts lemma 3.4. The second case also leads to a contradiction, since $[P_i]_\equiv$ is a key class, and from lemma 3.4 it follows that f_j precedes or supports at least one operator f_k in P_i and thus cannot be the key in P_i. We conclude that the assumption $key(P) \neq f$ is invalid, thus $F(n, f)$ is a key class with key f. □

Completeness of the meta-operator

In this section we prove by induction that each key class can be created through applications of $F(N, f)$, as formulated in theorem 3.3. Before we present the proof of theorem 3.3, we prove lemmas 3.5 and 3.6.

Lemma 3.5 *Let P be a path applicable to S, and $f \in P$. Then there is a path Q applicable to S, such that*

1. *Q consists of exactly the operators in $Anc_f(P)$.*

2. *$[Q]_\equiv$ is a key class, with key f.*

We name $[Q]_\equiv$ the key class induced by *f in P.*

Proof

1. Let $P = (f_1, \ldots f_n)$ and Q be the path consisting of the operators in $Anc_f(P)$ in the same order as they appear in P. Now let us suppose that Q is not applicable to S, i.e., there is an operator f_i in Q, such that f_i is not applicable. Then, there is an attribute $x \in f_i^{pre}$, such that $x \in f_j^{add}$, while $f_j \notin Anc_f(P)$. However, then $f_j^{add} \cap f_i^{pre} \neq \emptyset$, and thus $f_j \prec f_i$, and thus $f_j \in Anc_f(P)$ if $f_i \in Anc_f(P)$. Thus, Q is applicable to S.

2. By definition of $Anc_f(P)$, for each operator $f_i \in Anc_f(P)$ with $f_i \neq f$, there is an operator f_j, such that $f_i \prec f_j \vee f_i \ll f_j$. And thus, f_i must occur before f_j in any path containing both. Thus, only f may be the last operator in a path containing all operators in $Anc_f(P)$. Therefore, $[Q]_\equiv$ is a key class, with key f.

$\square$

Lemma 3.6 *Let C be a key class with key f_n and let $P \in C$ be a path, with $P = (f_1, \ldots, f_n)$. Let N be the set of relevant parents of f_n in P. Then the merge M of the key classes induced by the elements of N is non-empty, and for each path $Q \in M$, $Q \sim (f_n) \in [P]_\equiv$.*

Proof

Let $f_i \in P$ $(1 \leq i \leq n-1)$ be the operator with highest index such that f_i is not in any path of the key classes induced by the relevant parents of f_n. Since P is a path in a key class, it follows from lemma 3.4 that there exists an f_j such that $f_i \prec f_j \vee f_i \ll f_j$. If $f_j = f_n$, then f_i is a parent of f_n and by definition a relevant parent. If $f_j \neq f_n$, then f_i is in a path in the same key class as f_j induced by a relevant parent of f_n. Thus, in both cases, f_i is in a path in a key class induced by a relevant parent of f_n. From this contradiction, it follows that all $f_i \in P$ are in a path in a key class induced by a relevant parent of f_n. Thus, the merge of all these key classes contains at least the path Q such that $Q \sim (f_n) = P$. From this it follows immediately that for each path $Q \in M$, $Q \sim (f_n) \in [P]_\equiv$. $\square$

Theorem 3.3 *For each non-empty key class $C \in U_k$, there is a set N of key classes and an operator f, such that $F(N, f) = C$.*

Proof

- *Basis.*
  Let $C = \{(f_1)\}$. Then by definition $F(\emptyset, f_1) = C$.

- *Induction step.*
  We assume that each class C consisting of paths with length less than n is the result of an application of $F(N, f)$. Let $P = (f_1, \ldots, f_n)$ and let $C = [P]_\equiv$ be a key class, with key f_n. Let R be the set of relevant parents of f_n in P. Furthermore, let N be the set of key classes induced by the elements of R (cf. lemma 3.5). Then, from lemma 3.6 it follows that the merge M of all paths in N is non-empty, and that for each path $Q \in M$, $[Q \sim (f_n)]_\equiv = C$. Thus, $F(N, f_n) = C$.

□

3.3.5 Summary

In this section we have created a framework for db-search. We have shown that conventional search algorithms traverse the set of all paths U_p. The object of determining the state space traversed by conventional search algorithms was to create a standard for comparison with db-search.

Next we have defined the set of key classes U_k, which is a subset of the equivalence classes of U_p modulo $\equiv$. We have proved that U_k is complete, which means that all solutions in U_p are elements of classes in U_k under the conditions of monotonicity, non-redundancy and singularity. Thus, even though the cardinality of U_k is not larger than that of U_p, and often (much) smaller, all solutions are present in the smaller state space.

Finally, we determined a meta-operator which can be used to traverse the smaller state space U_k. The meta-operator $F(N, f)$ was defined, and we have shown that it is both sound and complete. The former indicates that each operation of the meta-operator yields an element of the reduced state space, while the latter indicates that each element of the reduced state space can be reached by application of the meta-operator.

Summarizing, we have succeeded in creating a framework which allows us to search a smaller state space, while being assured that the smaller state space contains all solutions of the original state space, and that the smaller state space is fully traversed. What remains to be done, is to describe practical algorithms for applying the meta-operator in an efficient manner. This is the topic of the next section.

3.4 Informal description of db-search

In section 3.3.4 we have defined meta-operator $F(N, f)$. $F(N, f)$ can be applied to a set of nodes N (each node representing a key class) and an operator f, under the three conditions that (1) the merge M of all key classes in N is non-empty; (2) the concatenation of a path P in M with f is applicable to the initial state; (3) for each of the key operators f_i of the classes in N, $f_i \prec f$ or $f_i \ll f$.

Clearly, trying all subsets N of nodes of a tree T as parameter for $F(N, f)$, has a search complexity exponential in the number of nodes of T. In such a case, searching U_k may be more expensive than searching the larger set U_p using a conventional search algorithm. The way in which db-search traverses the search graph is designed to limit the cost of applying $F(N, f)$ as much as possible. We present a short informal description of db-search, followed by an explanation of the application of db-search to an instance of DLP

Db-search repeatedly executes *levels*, where each level consists of two *stages*. In the first stage, named the *dependency stage*, only sets of nodes with cardinality 1 are selected for application of $F(N, f)$. If new eligible sets of nodes with cardinality 1 are created during a stage, $F(N, f)$ is applied to these sets as well. The *dependency stage* ends when $F(N, f)$ has been applied to all such sets. In the second stage, called the *combination stage*, sets of nodes with larger cardinality are considered. A node A created during the combination stage may not be element of a set N to which $F(N, f)$ is applied during the same stage. This ensures that the computationally expensive combination stage does not continue any longer than is strictly necessary.

We remark that during each combination stage of db-search, we only perform preparatory work for application of $F(N, f)$. We create a combination node A for each set of nodes N, such that at least one f exists allowing the execution of $F(N, f)$. During the dependency stage of the next level, f will be executed from A. Thus, nodes created during the combination stage do not themselves represent elements of U_k, but are aids to a clear implementation. They correspond to the merge of the classes represented by the nodes in N.

In the following, we describe the application of db-search to instance *aaccadd* of DLP.

Figure 3.1 shows the search graph after executing the first dependency stage for axiom *aaccadd*. In each child node we have capitalized the letter which has been created through the last applied operator. In each of the 1-ply nodes of the tree four operators are applicable. However, none of these

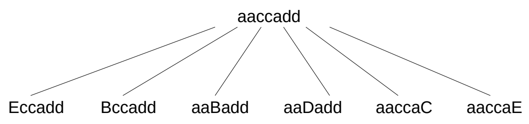


Figure 3.1: Search graph after 1st dependency stage for theorem *aaccadd*.

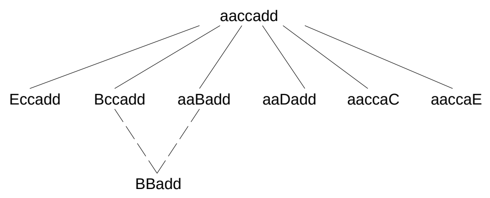


Figure 3.2: Search graph after 1st combination stage for theorem *aaccadd*.

correspond to an application of meta-operator $F(N, f)$, since the operator f does not depend on the operator leading to the 1-ply node. To clarify this, we look at the node representing theorem *Eccadd*. The rules $cc \rightarrow b|d$ and $dd \rightarrow c|e$ are applicable and correspond to the operators $f_{(2,2,3,c,b)}$, $f_{(2,2,3,c,d)}$, $f_{(5,5,6,d,c)}$ and $f_{(5,5,6,d,e)}$. Neither of these operators depends on the operator $f_{(0,0,1,a,e)}$ which has led to the creation of this node. Therefore, the meta-operator is not applicable in node *Eccadd*.

Having finished the first dependency stage, we proceed with the first combination stage. In DLP, each precondition set of an operator consists of two attributes. As a result, during the combination stage only combinations of exactly two nodes need to be considered. Figure 3.2 shows the search graph for our instance of DLP after finishing the first level of db-search. It was created by examining all 15 combinations of two 1-ply nodes, to see if the combination of two nodes would lead to a valid application of the meta-operator. In one case it did, resulting in the creation of node *BBadd*. The operators which led to the creation of the parents *BBadd* are $f_{(0,0,1,a,b)}$ and $f_{(2,2,3,c,b)}$. Depending on both these operators are $f_{(0,1,3,b,a)}$ and $f_{(0,1,3,b,c)}$. Thus, two operators are applicable in *BBadd*, for which reason the combination node representing theorem *BBadd* was created.

Next, we execute the dependency stage of the second level of db-search. For this stage, we apply $F(N, f)$ to the combination node created in the first level. The application of $f_{(0,1,3,b,a)}$ and $f_{(0,1,3,b,c)}$ from the combination lead to the creation of *Aadd* and *Cadd*. From *Aadd* we can apply two more

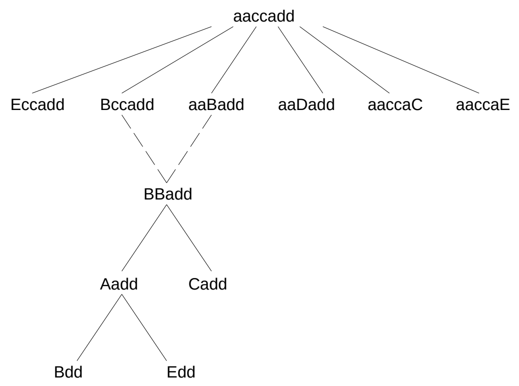


Figure 3.3: Search graph after 2nd dependency stage for theorem *aaccadd*.

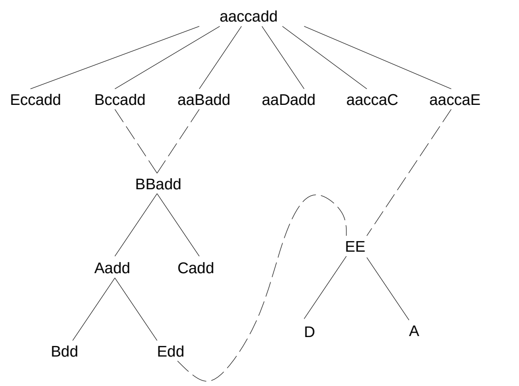


Figure 3.4: Complete dependency-based search graph for theorem *aaccadd*.

operators which depend on the operator leading to *Aadd*. Thus, a total of four nodes is added in the second dependency stage. Figure 3.3 shows the search graph after the second dependency stage.

For the second level of combination nodes, not all combinations of nodes in the tree need to be checked. Only combinations involving at least one node created during the second dependency stage need to be investigated. In our example this leads to a combination between second-level node *Edd* and first-level node *aaccaE*. Using the new combination node, the third level of nodes is created, again consisting of a dependency stage and a combination stage.

The complete dependency-based search graph for theorem *aaccadd* is

```
procedure DbSearch()
   CreateRoot(root);
   level := 1;
    while ResourcesAvailable() and TreeSizeIncreased() do
       AddDependencyStage(root);
       AddCombinationStage(root);
       level := level + 1
    od
end
```

Table 3.2: Main db-search algorithm.

```
procedure AddDependencyStage(node)
    if node ≠ nil then
        if level = node.level+1 and
                node.type in [Root, Combination] then
            AddDependentChildren(node)
        fi ;
       AddDependencyStage(node.child);
       AddDependencyStage(node.sibling)
    fi
end
```

Table 3.3: Dependency-stage algorithm.

shown in figure 3.4.

The graph consists of three dependency levels, and two combination levels. The third combination level is empty, which terminates the search. From figure 3.4 we see that the instance of DLP with axiom *aaccadd* has two solutions: single-letter theorems a and d can be created.

3.5 Algorithms

In this section we present the db-search algorithms in pseudo-code. We remark that many implementation details have been omitted in the algorithms.

Table 3.2 shows the main loop of db-search. Repeatedly, a level is created,

```
procedure AddDependentChildren(node)
    for operator in LegalOperators(node) do
        if Applicable(operator, node) then
            LinkNewChildToGraph(node, operator);
            AddDependentChildren(node.newChild)
        fi
    od
end
```

Table 3.4: Dependent-children algorithm.

```
procedure AddCombinationStage(node);
    if node ≠ nil then
        if node.type = Dependency and node.level = level then
            FindAllCombinationNodes(node, root);
        fi ;
        AddCombinationStage(node.child);
        AddCombinationStage(node.sibling)
    fi
end
```

Table 3.5: Combination-level algorithm.

consisting of a dependency stage and a combination stage, as described in section 3.4.

Table 3.3 shows the algorithm for creating the dependency stage. It is assumed that each node has a child pointer and a sibling pointer. The child pointer points to the first child of the node, while the child's sibling pointer points to the next child, etc. This assumption explains the recursive calls in *AddDependencyStage()*. In the graph, we distinguish between three types of nodes: *Root*, *Combination* and *Dependency*. A dependency stage is started only from combination nodes, and, for the first level, from the root.

The algorithm of table 3.4 determines all operators dependent on a node and creates children for each eligible operator. The function *Applicable()* tests to see if the selected operator and node form a pair of parameters which is eligible for application of the meta-operator $F(N, f)$.

The second stage of each level of db-search consists of creating the combinations of independent paths. In our example algorithm (see table

```
procedure FindAllCombinationNodes(partner, node);
  if node ≠ nil then
    if NotInConflict(partner, node) then
      if node.type = Dependency then
        combination := Combine(partner, node);
        operators := DependingOn(combination);
        if operators ≠ nil then
          AddCombinationNode(node, combination)
        fi
      fi ;
      FindAllCombinationNodes(partner, node.child)
    fi ;
    FindAllCombinationNodes(partner, node.sibling)
  fi
end
```

Table 3.6: Algorithm to find combinations of nodes.

3.5) we have assumed that each combination consists of exactly two nodes. In the double-letter puzzle and qubic, this is indeed the case. In go-moku, combinations of up to four nodes exist. Extending the algorithm to include combinations of three or more nodes is not difficult. A disadvantage is, however, that searching for combinations of c nodes in a graph of size N has a time complexity in the order of N^c. Domain-specific reductions of the complexity may often be possible. We have therefore refrained from presenting a general algorithm for combinations of other than two nodes.

The algorithm of table 3.6 finds a node in the graph for a selected partner. It is checked that the nodes are not in conflict, that its type is a dependency node, and that the combination of the two nodes allows at least one application of the meta-operator. This last condition is important to prevent the creation of a large number of useless combination nodes.

3.6 Test results

Earlier, we stated that conventional search algorithms traverse U_p, while db-search traverses U_k. In this section we investigate through experiments on DLP the difference in cardinality between U_p and U_k.

First, we describe the four algorithms used in the experiments. Second, we describe the set of test problems used for the experiments, as well as the conditions in which the experiments took place. Third, we present the results of the experiments.

Selected algorithms

As a conventional tree-search algorithm for our experiments, we have selected DFS of which we have implemented two variants: (1) without transposition tables (DFS-), and (2) with transposition tables (DFS+). Since we intend to run the algorithms in our experiments until the complete state space has been traversed, the performance of alternatives like breadth-first search are equivalent to the performance of DFS.

The other two implemented algorithms are the domain-specific algorithm TRIANGLE, presented in appendix A, and, of course, db-search. An advantage of db-search over TRIANGLE is that in cases where only few theorems can be deduced, db-search may search less nodes than the fixed number of entries needed for TRIANGLE. A disadvantage of our implementation of db-search is that we did not implement a transposition table. However, transpositions resulting from the order in which operators are executed are non-existent in U_k, as they are all part of the same key class. As a result, transpositions have only a minor influence on the performance of db-search on DLP.

Test problems

We have generated random instances of DLP. For each string length of 1 to 20, 100 strings were generated, for a total of 2000 axioms. For each of these 2000 axioms, all four algorithms were to run to completion. However, in order not to have extremely large state spaces dominate the results and to keep the required resources within practical limits, we have set limits for the state spaces examined by DFS+ and DFS-. We terminated DFS+ as soon as the tree size exceeded 100,000 nodes, while DFS- was terminated as soon as the tree size exceeded 1,000,000 nodes. Both TRIANGLE and db-search were run to completion on all selected test problems.

Results

The tree-size limit set for DFS+ terminated the search 26 times out of the 2000 runs. Only once did the early termination result in missing a solution. For DFS- a million nodes was insufficient to complete the search in 129 of the 2000 runs. In 24 of these, at least one of the solutions was missed.

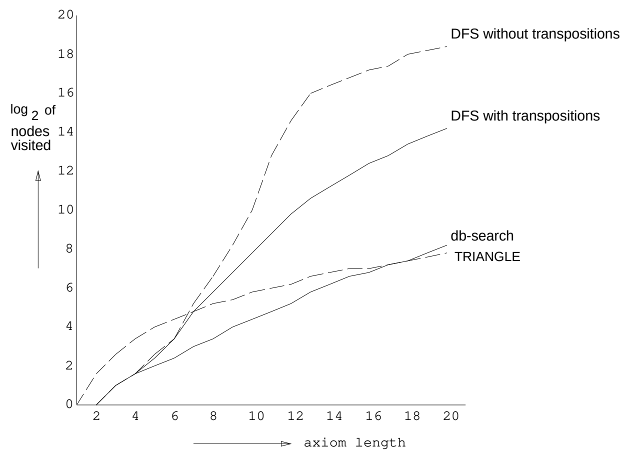


Figure 3.5: Tree size per algorithm applied to the double-letter puzzle.

Db-search's most difficult problem was *dbdeabbaacccddaeecda*, for which it needed 3934 nodes to determine that it has no solutions. Both variations of depth-first search did not complete the search on this axiom within their respective tree-size limits.

The average number of nodes visited by each algorithm is illustrated in figure 3.5. The horizontal axis is the axiom length, while the vertical axis is the log_2 of the number of nodes created.

Up to strings of length 18, db-search outperforms TRIANGLE. For those strings, transpositions do not outweigh the gain db-search makes in terminating the search early if possible. Still, the time complexity of db-search, in particular in the combination stage of each level, is higher than for the domain-specific algorithm. Therefore, we do not claim that db-search outperforms TRIANGLE.

The trees traversed by both variants of DFS suffer from a combinatorial explosion. At theorem length 20 the average cardinality of U_p (the size of the trees searched by DFS-) is more than 1200 times the average cardinality of U_k (the size of the graphs searched by db-search). As can be seen from the size of the graph traversed by DFS+, transpositions are responsible for a factor 20. The more than 60 times smaller graph traversed by db-search compared to DFS+ indicates that db-search is far more efficient on DLP than conventional search algorithms.

In chapters 4 and 5 db-search has been applied to qubic and go-moku, resulting in significantly reduced state spaces, while no domain-specific algorithm has yet been developed which does the same.

3.7 Applicability

Db-search is a single-agent search algorithm. The main source of applications therefore lies within that area. In some games, such as qubic and go-moku, a restricted search concentrates on sequences of threatening moves only. If the opponent is constantly restricted to only a single reply, the state space is conceptually transformed into a single-agent state space. In those circumstances db-search may be applied to games. For details of such transformations on qubic and go-moku see chapters 4 and 5.

In section 3.3.3 we have proved that U_k is complete if three conditions are met. While these conditions all hold for DLP, they do not hold fully in domains such as qubic and go-moku (i.e., after the transformation to a single-agent state space). As a result, U_k may neither be sound nor complete. Searching a non-complete U_k may still be favorable to searching U_p, if the size of U_p prohibits full investigation. However, further research is necessary to understand the implications of applying db-search to such domains in general.

Chapter 4

Qubic

In chapters 2 and 3 two new search techniques, pn-search and db-search, were introduced. Pn-search attempts to use non-uniformity in AND/OR trees to traverse the state space more efficiently than the various conventional search algorithms. Db-search traverses a smaller graph than conventional search algorithms. Still, for a special class of problems it has been shown that the smaller graph is sound and complete. This means that each solution found by a conventional search algorithm will also be found by db-search.

Pn-search and db-search were developed during the investigation of several games: connect-four (Allis, 1988), awari (Allis *et al.*, 1994), qubic (Allis and Schoo, 1992) and go-moku (Allis *et al.*, 1993). The application of pn-search and db-search to qubic and go-moku are discussed in this and the next chapter. The purpose of these chapters is twofold:

1. to explain in detail how pn-search and db-search were applied to two combinatorially complex problems, and
2. to show that qubic and go-moku can be solved, thereby positively answering our first research question (cf. section 1.4) for two specific games.

At this point it is important to mention that qubic was solved more than a decade before we started our research. Oren Patashnik solved qubic in 1977 and his solution was confirmed by Ken Thompson (Patashnik, 1980).

Our interest in qubic sprang from its potential as a test bed for go-moku, due to the similarity between these two games. While *threat sequences* (see section 4.2.2) play an important role in both games, threat sequences in go-moku are more complex than threat sequences in qubic.

Being ignorant of Patashnik's work, there was the added challenge of solving the game. After we were informed of Patashnik's work by Ingo Althöfer and Ralph Gasser, we nevertheless decided to finish our work on the game. The experience gained has helped to solve go-moku, while it also provided the means for a comparison of db-search and pn-search with the search techniques applied by Patashnik.

The chapter is organized as follows. In section 4.1 we provide a background to the investigations in qubic. The rules of qubic and common strategies are presented in section 4.2. The application of db-search to qubic is described in section 4.3. The role of pn-search in the solution of qubic transpires from section 4.4. The results of our investigations, as well as comparisons with the results of Patashnik, are presented in section 4.5.

4.1 Background

Among the games of the Olympic List, qubic is one of the lesser-known games. Despite its simple rules, qubic has a severe handicap: it is played on a three-dimensional board. Therefore, visualizing sequences of moves is a difficult task for human players, while most games end in a long sequence of threatening moves requiring careful analysis.

Nevertheless, at least some strong human players exist, as is apparent from Patashnik (1980), who describes how qubic is solved using a combination of human expert knowledge and a standard search algorithm.

Patashnik assumed that qubic would be a first-player win. Therefore, to prove a win in a position with white (the first player) to move, only one winning move had to be selected. To prove a win in a position with black (the second player) to move, all moves had to lead to wins for white. Using a standard $\alpha-\beta$ search, Patashnik created a tactical module which determined in a given position whether the player to move had a forced win. For each position in the search tree, it was determined whether the player to move had to make a forced move. Otherwise, if black was to move, for each legal black move a child position was created. If white was to move, a so-called *strategic move* had to be made. These moves were selected by hand by Patashnik. Using some 1500 hours of CPU time, and 2929 strategic moves, qubic was solved. The database with the solution tree has been checked by Ken Thompson, who confirmed Patashnik's results.

Our research in 1991 consisted of creating a tactical module based on db-search. Furthermore, instead of selecting strategic moves by hand, pn-search guided the search process. After the program was created we were informed that qubic had already been solved. Nevertheless, as qubic was not yet removed from the Computer Olympiad, we finished our solution in collaboration with Patrick Schoo. Since then our understanding of db-search has improved, resulting in a new implementation of our qubic program. In this chapter we describe the 1993 implementation and its results, which differ somewhat from Allis and Schoo (1992).

In earlier publications (Allis and Schoo, 1992; Allis *et al.*, 1993) we used the term *threat-space search* for the application of db-search to qubic and go-moku. In this text we only use the term db-search. We gladly acknowledge that both names were suggested by Barney Pell.

4.2 Rules and strategies

Qubic is a three-dimensional instance of a category of games of which well-known two-dimensional analogs are tic-tac-toe, go-moku and renju. First, we present the rules in section 4.2.1. Second, in section 4.2.2 we discuss the role of threats and threat sequences in qubic. Finally, we analyze the automorphisms (i.e., mappings of the playing board onto itself, such that all relevant properties of the board are preserved) of the qubic board and its two different types of cubes in 4.2.3.

4.2.1 Rules

Qubic is played on a $4 \times 4 \times 4$ cube, thus consisting of 64 small cubes. Players move alternately by occupying any empty cube. The game ends as soon as one of the players has occupied four consecutive cubes in a straight line (either in one, two or three dimensions). Such a set of four cubes in a straight line is called a *group*. There are $3 \times 16 = 48$ one-dimensional groups, $3 \times 8 = 24$ two-dimensional groups and 4 three-dimensional groups, for a total of 76 groups.

In figure 4.1 the three different types of groups are shown. Group a is one-dimensional, group b is two-dimensional, while group c is three-dimensional.

4.2.2 Threats and threat sequences

If a player has occupied three cubes in a group, with the fourth cube empty, she threatens to win at her next move. In such a position, the opponent is forced to refute the threat (unless she can win at her next move). The game

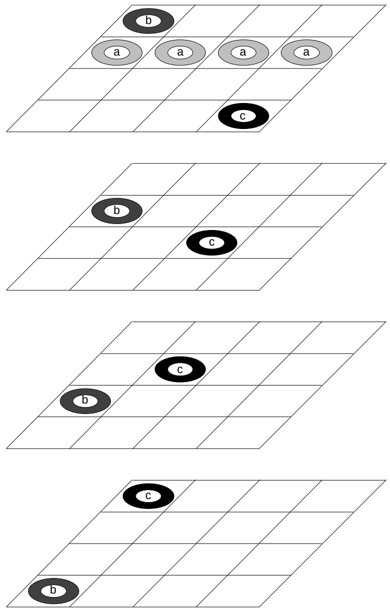


Figure 4.1: Three types of groups in qubic.

is usually decided by a player creating a threat sequence ending in a double threat, which cannot be stopped by the opponent.

In figure 4.2 an example winning threat sequence in a single plane is shown. White has occupied three cubes in the plane (in the corners), while black has played her moves elsewhere (i.e., in other planes). White now has an 11-ply winning threat sequence starting with moves 1 through 9 in figure 4.2. After move 9, white threatens to win at a and b, which cannot both be countered by black's next move.

In general, a threat sequence may end in one of three possible ways. First, a double threat may be created, resulting in a win for the attacker. Second, the attacker may run out of threats. Third, the forced moves of the defender may result in her accidentally creating a threat of her own, and changing her role from defender to attacker.

If a threat sequence ends without success for the attacking player, she has normally exhausted most of her threat potential, reducing her winning chances. Therefore, early in the game, both players try to occupy cubes which increase their potential for creating threats, without actually executing those threats.

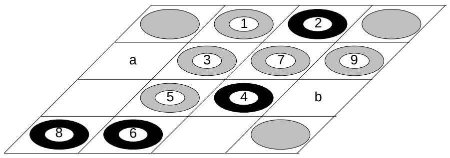


Figure 4.2: An 11-ply winning threat sequence.

4.2.3 Cube types and automorphisms

The 64 cubes fall into two categories. The 8 corner cubes and 8 center cubes are named 7-cubes, as each is part of 7 groups (3 one-dimensional groups, 3 two-dimensional groups and 1 three-dimensional group). The other 48 cubes are called 4-cubes as they are part of four groups only (3 one-dimensional groups and 1 two-dimensional group).

The number of automorphisms in qubic is surprisingly high: 192. This can be explained as follows. By rotation, each of the six sides of the cube can be brought on top in four different ways, resulting in a total of 24 automorphisms by rotation. There are three more operations, each doubling the number of automorphisms. First, *reflection* in a plane through the center of the cube. Second, *turning the cube inside out*, i.e., exchanging (in all three dimensions) the inner planes with the outer planes. Third, *internal exchange*, i.e., exchanging the inner planes in all three dimensions, while leaving the outer planes untouched.

Due to the automorphisms, there are only two distinct opening moves in qubic, one at any 7-cube, and one at any 4-cube. After White's first move at a 7-cube, black has 12 distinct answers, as presented in figure 4.3. Each of the empty 51 cubes in the figure can be mapped to at least one of the 12 black cubes, through at least one of the automorphisms of qubic.

4.3 Applying db-search

As mentioned before, threat sequences play a dominant role in qubic. Obviously, to play qubic well, it would be advantageous to have a module which determines whether a winning threat sequence exists. Our application of db-search to qubic is restricted to searching for winning threat sequences.

This section consists of three parts. First, in section 4.3.1 we describe how the adversary-agent state space, when restricted to threat sequences, can be transformed into a single-agent state space. Second, in section 4.3.2

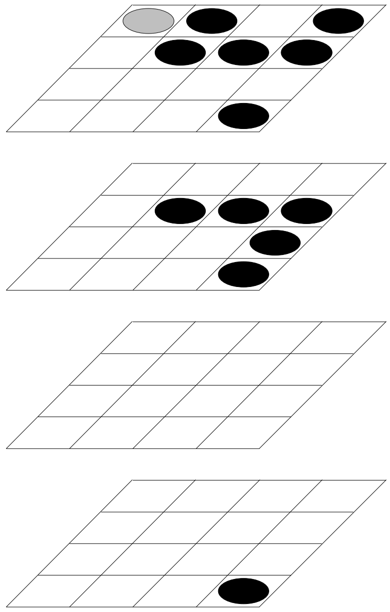


Figure 4.3: The 12 two-ply moves.

we illustrate how the single-agent state space thus created for qubic fits in the framework for db-search presented in chapter 3. Third, in section 4.3.3 we discuss three properties of the single-agent state space for qubic which have not been included in the framework of section 4.3.2. For each of these properties there is an explanation of how our implementation of db-search handles them.

4.3.1 A single-agent search in qubic

Our description of the single-agent state space of threat sequences in qubic consists of a set of definitions, an interpretation of the definitions, and the transformation of the adversary-agent state space to a single-agent state space.

Definitions

In the previous sections we informally introduced the concept of threats, threat sequences and winning threat sequences in qubic. These notions are defined in definitions 4.1, 4.2 and 4.3.

Definition 4.1 *A* threat *in* qubic *is a move by the attacker leading to a position such that*

1. *The defender cannot win at her next move, and*

2. *The defender has at most one move stopping the attacker from winning at her next move.*

If a threat leaves the defender without any moves to stop the attacker from winning at her next move, it is called a *double threat*, otherwise the threat is called a *single threat*.

Definition 4.2 *A* threat sequence $(a_1, d_1, a_2, d_2, \ldots, a_n, d_n)$*, with* $n \geq 1$*, is any sequence of moves such that each* a_i*,* $1 \leq i \leq n$ *is a single threat, and each* d_i *the single response to* a_i *which does not lose immediately,*

Definition 4.3 *A* winning threat sequence *in* qubic *is a sequence of moves* $(a_1, d_1, \ldots, a_n, d_n, a_{n+1}, d_{n+1})$*, such that* $(a_1, d_1, \ldots, a_n, d_n)$ *is a threat sequence,* a_{n+1} *is a double threat and* d_{n+1} *is any legal move.*

Interpretation

Here we elaborate on the definitions presented above, interpreting them in the context of groups.

To win in qubic, a player must occupy all four cubes in a group. Thus, a player who occupies three cubes in a group, while the last cube is empty, threatens to win. According to definition 4.1, such a move is only a threat if the opponent has not obtained three cubes in a group herself. In other words, a threat consists of a *local* property for the attacker (i.e., the state of one specific group) and the *global* lack of a similar property for the defender (i.e., no group on the board having the property).

In a threat sequence, each attacker move occupies the third attacker cube in a group, while the fourth cube is empty. Each defender move occupies the fourth cube in that group. In each case, the defender has no alternative move which wins immediately and, although the rules of qubic allow playing anywhere else, alternative moves are *blunders* as they would result in losing at the next move. In other words, a threat sequence consists of a sequence of moves where each attacker move is followed by its only non-blundering reply.

A winning threat sequence is a threat sequence followed by a double threat and any legal move. Since there are at least two places where the attacker threatens to win at the next move, and the defender cannot win herself immediately, all moves are equally bad. Therefore, any legal move may be selected.

Adversary-agent vs. single-agent

As we have seen, in threat sequences and winning threat sequences each move by the defender is implied by the previous attacker move. Therefore, we may conceptually merge these two moves into a single meta-move.

If we examine the state space created by these meta-moves, it is no longer an adversary-agent state space, but instead a single-agent state space. For each meta-move, the attacker selects any of the possible threats in a position. If the threat is a single threat, the move by the opponent is implied by the previous move. If the threat is a double threat, all moves by the opponent are equally bad, and a random move may be selected to represent all possible moves. In both cases the defender has no real choice, effectively transforming the state space into a single-agent state space. In the remainder of this section, we will only regard meta-moves, and assume that the attacker move and defender move in a meta-move are made at the same time.

4.3.2 A db-search framework for qubic

In this section, we describe a db-search framework for the single-agent state space of qubic. We mention that the framework only involves *local* properties, i.e., occupation of single groups, while ignoring *global* properties, i.e., possible counter threats of the defender. Global properties of a position will be handled in section 4.3.3. The terminology introduced in chapter 3 is used throughout this section.

Attributes

The set U of all attributes is defined as follows. $U = \{C(i,x) | 0 \leq i \leq \text{SIZE} - 1 \wedge x \in \{\circ, \bullet, \cdot\}\}$. Attribute $C(i,x)$ represents the fact that cube i is occupied by the attacker ($\circ$), occupied by the defender ($\bullet$) or empty ($\cdot$). The constant SIZE equals the number of cubes on the playing board (i.e., $4^3 = 64$). It can easily be checked that U has 192 elements.

Operators

The operator f_{c_1,c_2,c_3,c_4} is defined as follows.

$$f^{pre}_{c_1,c_2,c_3,c_4} = \{C(c_1,\circ), C(c_2,\circ), C(c_3,\cdot), C(c_4,\cdot)\}$$

$$f^{del}_{c_1,c_2,c_3,c_4} = \{C(c_3,\cdot),C(c_4,\cdot)\}$$
$$f^{add}_{c_1,c_2,c_3,c_4} = \{C(c_3,\circ),C(c_4,\bullet)\}$$

The set of all operators U_f is defined as follows.

$$U_f = \{f_{c_1,c_2,c_3,c_4}|\{c_1,c_2,c_3,c_4\} \text{ is a group}\}$$

We remark here that a group is a set of four squares which, if all occupied by one player, result in that player winning the game. In qubic there are 76 groups. For each group, the 4 elements can be ordered in $4! = 24$ possible ways. Thus, there are $24 \cdot 76 = 1824$ operators in U_f. Since c_1 and c_2 can be exchanged without changing the operator, there are effectively 912 operators in U_f.

Initial state and goal states

The initial state consists of exactly 64 attributes, one per cube indicating the contents of the cube. Each qubic position which is to be checked for the existence of a winning threat sequence can serve as an initial state. The set U_g of goal states is independent of the initial state, and is defined as follows.

$$U_g = \{\{C(c_1,\circ),C(c_2,\circ),C(c_3,\circ),C(c_4,\cdot)\} \mid \{c_1,c_2,c_3,c_4\} \text{ is a group}\}$$

In other words, any state in which a group exists of which three cubes have been occupied by the attacker and the fourth cube is empty, is a goal state. We remark that each meta-move starts with a move by the attacker. Therefore, a state as described here in the single-agent search, ensures that in the adversary-agent search the attacker can win at her next move. U_g has 304 elements and is not singular.

Properties of the qubic framework

The framework we have described above is monotonous. Furthermore, we can easily restrict ourselves to non-redundant paths. If U_g were singular, our U_k would be complete.

We can create a singular $U_{g'} = \{\ \{G\}\ \}$, by defining a special goal attribute G and operators which transform any element of U_g into G, which would result in a complete U_k. A discussion of the completeness of U_k would be premature, however, since we have ignored the global properties of qubic so far.

4.3.3 Qubic-specific enhancements to db-search

The db-search framework for qubic presented in the previous section focuses only on the local properties of threats. In this section we discuss the three global properties which need to be incorporated in db-search. Each property is described followed by the method of inclusion in db-search.

Defender four

In each winning threat sequence, both the attacker and defender occupy cubes. Even though the defender has no choices of which cubes to occupy, the attacker may, accidentally, force her to occupy all four cubes in a group. Such a group is named a *defender four*. If this happens, the threat sequence by the attacker has failed.

During the dependency stage of each level of db-search, it is easy to check after each meta-move (a, d), consisting of attacker move a and defender move d, whether d has created a defender four. It is sufficient to investigate the 4 or 7 groups in which d lies. During the combination stage of each level of db-search, a defender four could be created by the merge of two or more paths. To detect such a defender four, all 76 groups must be investigated when creating a combination.

We remark that the qubic-specific enhancements mentioned below render the dependency-stage test for defender fours superfluous and it has therefore been omitted in our implementation.

Closed defender three

Each meta-move results in a group containing three attacker cubes and one defender cube. Such a group is named a *closed attacker three*. Similarly, a *closed defender three* is a group containing three defender cubes and one attacker cube. A group where one player has occupied three cubes, while the fourth cube is empty are named *open attacker three* or *open defender three*.

Even though closed defender threes cannot be converted into a winning group, they may represent a subtle problem. If two paths in db-search are merged they may create one or more closed defender threes on the board. Let us assume that the three defender cubes are occupied during meta-moves m_1, m_2 and m_3, while the attacker cube is occupied during meta-move m. Furthermore, let us assume that a path P in the merge exists, consisting of the following sequence of moves: (m_1, m_2, m_3, m_4, m), where m_4 is any meta-move. Then, after move m_3, an open defender three exists. Clearly, the only way for the attacker to stop the open defender three is to immediately play move m. In P move m_4 is played first, which means that meta-move m_4 erroneously ignores the option for the defender to win. We remark that (some of) the cubes in a closed defender three need not be part of a meta-move, but could be part of the initial state.

Summarizing, closed defender threes present a problem when the meta-move occupying the attacker cube is played later than immediately after the third defender cube has been occupied. In other words, an ordering exists between the set of meta-operators occupying the defender cubes in the closed defender three, and the operator occupying the attacker cube.

During the dependency stage of db-search, to create a closed defender three, first an open defender three must be created, otherwise the closed defender three does not pose a problem. As these are monitored separately, we can safely ignore closed defender threes during the dependency stage. During the combination stage, a merge may create one or more closed defender threes. Only paths in which the attacker cube for each closed defender three is occupied in time (i.e., not later than immediately after the third defender cube has been occupied) should be included in the merge.

Determining whether a merge is non-empty may be time-consuming when fully incorporating the closed defender tests. Instead, we have implemented a simple and surprisingly effective heuristic. Previously, for each combination node (i.e., for each merge), a path representing the merge was selected randomly. The heuristic consists of investigating whether the selected path honors the ordering criteria imposed by the closed defender threes. If so, the merge is not empty. If not, the merge is assumed to be empty. Clearly, in this way valid merges may be rejected, but invalid merges are never wrongfully accepted.

To investigate the amount of error created through the use of this heuristic, we ran the program twice on a set of test positions. The first variant of the program contained the heuristic test, while the second variant did not test for closed defender threes at all. In less than 1% of the test positions did the second variant suggest a winning line, while the first variant failed to find any winning line, although several times the first variant suggested a different winning line. We remark that in the extra 1%, the suggested winning line may have been incorrect, due to defender threes, or may have been valid and have been accidentally rejected by the above heuristic. A non-heuristic implementation for investigating closed defender threes is expected to yield only a small gain in efficacy while causing a significant decrease in efficiency. Such an implementation has therefore been omitted.

Open defender three

When a threat sequence contains an open defender three, the attacker must respond to that defender three immediately or lose at the next move.

During the dependency stage of db-search, only meta-moves are considered which depend on, or are preceded, by the node from which the meta-move is made. Therefore, during the dependency stage it is often not possible to counter a defender three. Instead, we solve the problem of open defender threes during the combination stage.

In standard db-search, to apply meta-operator $F(N, f)$, set N must be a minimal set of key classes, such that f depends on, or is preceded by, the key operator in each of those classes. We extend the application of meta-operator $F(N, f)$ as follows.

Let $F(N, f)$ be applicable and let P be an element in the merge of classes of N. Furthermore, we assume that $(x_1, \ldots, x_n)$ is the priority queue of empty cubes in open defender threes in $P(A)$, where A is the initial state. Then, we define $F(N', f)$, with $N' = N \cup P_1$, to be applicable, if (1) the key operator f_1 of P_1 occupies with its attacker move x_1, and (2) the merge of all elements in N' is non-empty.

Using the extended meta-operator, we can create combinations of paths to counter open defender threes. Clearly, a combination should only be created if $F(N, f)$ is applicable and its priority queue of empty cubes in open defender threes is empty. In our db-search implementation for qubic we have implemented the extended meta-operator.

Summary of db-search enhancements

In this section we have introduced three qubic-specific enhancements to db-search. The main question yet to be answered is whether the state space searched by db-search with these qubic-specific enhancements is complete. Of course, the heuristic applied to counter closed defender threes renders the state space incomplete, but as has been argued, only a marginal number of solutions are incorrectly rejected. Without proof we state that, except for the aforementioned heuristic, our implementation of db-search is complete.

In other words, in each position where a winning threat sequence exists, db-search finds a winning threat sequence, unless the meta-moves within each winning threat sequence can be reordered such that a closed defender three is countered too late by the attacker.

4.4 Applying pn-search

To apply pn-search to qubic, we need to convert the qubic game tree into an AND/OR tree. This is described in section 4.4.1. The enhancements to basic pn-search adopted for our qubic implementation are described in section 4.4.2.

4.4.1 Qubic as an AND/OR tree

Proof-number search (as described in chapter 2) is an AND/OR-tree algorithm. To apply it to qubic, we represent positions where white is to move as OR nodes, and positions where black is to move as AND nodes. A win for white is represented by the value true, while a draw and a win for black are represented by the value false. Thus, proving the pn-search tree means that white can win in the root position, while disproving the pn-search tree means that black can achieve at least a draw.

At each OR node, white is to move. At such nodes, db-search with white as attacker is used as evaluation function. If db-search finds a winning threat sequence, the node is proved, otherwise it obtains the value unknown. In AND nodes, black is to move. In such nodes, db-search with black as attacker is used as evaluation function. If a winning threat sequence is found, the node is disproved, otherwise the value of the node is unknown. A node representing a position with all 64 cubes occupied, while neither player has created a winning configuration, is a draw and therefore has value false, without applying the evaluation function.

4.4.2 Enhancements

The above description explains how standard pn-search is applied to qubic. However, four enhancements have been added to speed up the search. The enhancements are discussed in this section.

Transpositions

A DAG is created instead of a tree, using the algorithm described in section 2.3.3. This ensures that if a position has already occurred in the DAG, or if a position is equivalent through automorphisms to another position in the DAG, the position is not investigated again.

Threatening moves by white

Pn-search favors subtrees in which the mobility (i.e., the number of choices available to a player) of one player is restricted, while the mobility of the other is enlarged. In qubic, this means that threatening moves are favored above all other moves, as they leave the opponent with just a single response. After a threatening move, and the forced response by the opponent, again threatening moves are favored above all other moves, and so on. Thus, pn-search automatically focuses on the space of threatening moves. This is undesirable for pn-search in qubic, since the evaluation function (db-search) will already have investigated whether a winning threat sequence exists. If such a sequence does not exist, the potential for threats should be increased, instead of decreased by executing them. Therefore, in our pn-search tree, we have restricted white to non-threatening moves, simply by omitting moves which create a threat in the move-generation module. For black, of course, all moves are investigated.

Heuristic (dis)proof number initialization

In chapter 2 we have suggested several methods to include some domain-specific knowledge in the initialization of proof and disproof numbers. Here we describe our qubic-specific initializations.

After expansion of an AND node (black to move), usually many nodes are proved immediately by db-search. Nodes in which black has just created a threat, however, are not proved immediately, because white is forced to counter the threat. A good estimate of the number of nodes which must still be proved at an AND node is the number of threatening moves black can make. Therefore, the proof number of an AND node is initialized to the number of threatening moves for black (with a minimum of 1), while the disproof number is initialized to 1.

After expansion of an OR node (white to move), usually several nodes are disproved immediately by db-search. Moves which create potential threats (named *positional moves*), however, are usually not disproved immediately. We determine the number of *positional* moves using the following heuristic. For a move M we consider the set of groups G which contain M, while not containing any black cubes. M is named positional if G consists of at least three groups, each containing zero or one white cubes (besides M), or at least two groups, each containing at least one white cube (besides M). At OR nodes, the disproof number is initialized to the number of positional moves for white (with minimum 1), while the proof number is set to 1.

Removing solved terminal nodes

In section 2.3.1 it was described how solved subtrees in a pn-search tree may be removed. Such a technique has disadvantages when applied to a DAG instead of a tree.

Assume that a node J has been solved and is subsequently removed from the DAG. If in another subtree a new instance of node J is created, the work to solve J will be duplicated. The decision of which solved nodes to remove may depend on the size of the working memory and the probability that this scenario will occur. Generally, nodes which have been solved with little effort may be removed with less cost than nodes which have been solved only after a large search.

We have decided to remove nodes from the DAG only if they were solved through evaluation. As evaluations of nodes require only a small amount of time, the reduced memory requirements were judged to outweigh the cost of re-evaluation for the terminal nodes which occur more than once in the search. In our experiments the memory requirements were thus reduced by approximately 70%.

4.5 Solving qubic

In this section we describe how we solved qubic. First, in section 4.5.1 we describe how we subdivided the game tree into 195 subtrees. Second, in section 4.5.2 we present statistics on solving each of the 195 subtrees. Third, we compare our results with those of Patashnik (1980) in section 4.5.3. Finally, in section 4.5.4 we discuss the reliability of our results.

4.5.1 Subdividing the game tree

In this section we explain why and how we subdivided the qubic game tree in 195 subtrees. First, we explain why this was necessary. Second, we show how we subdivided the game tree into four-ply subtrees. Third, we explain how each of the four-ply subtrees was investigated.

Necessity of subdividing the game tree

Before white can create a threat, she must have occupied two cubes in the same group. After the threat is executed by white and countered by black, white has three cubes in one group together with a black cube. To create a new threat she must have occupied at least one other cube. Thus, winning threat sequences can only be found in positions with at least six cubes (three white and three black) on the board. As we have seen in figure 4.2, in some positions with exactly three white cubes, winning threat sequences exists.

From the above, it follows that any evaluation by db-search of positions with 0 to 5 cubes occupied will return the value unknown. Furthermore, the number of children per qubic position at level d equals SIZE-d. Therefore, the first 5 ply of the qubic game tree, using evaluation function db-search, has a uniform branching factor per level of the tree. Executing pn-search for the full game tree (with the root representing the empty board) will be ineffective, as pn-search relies on non-uniformity. For this reason, we decided to split the game tree into subtrees.

Selecting a minimal set of subtrees

The subtrees each represent positions 4-ply into the game. A depth of four was selected since it was deep enough to overcome the uniformity problem for pn-search mentioned above, while it required the selection of only 13 strategic moves by hand (i.e., one move for the initial position, and 12 moves in the twelve 2-ply positions of figure 4.3) thus leaving as much work to pn-search as possible.

Starting from the empty board, we suggested a move for white. Since there are only two distinct moves, one at a 4-cube, and one at a 7-cube, we selected the 7-cube move as white's best chance for winning.

As shown in figure 4.3, black has 12 different first moves. Thus, at ply two we have 12 positions to solve. In each of these positions we suggested a move for white. In Patashnik (1980), moves at 7-cubes were selected, such that the number of different resultant positions (after applying automorphisms) was as small as possible. There, 7 three-ply positions are presented. To obtain the 7 three-ply positions, in each of the 12 two-ply positions, white played in a one-dimensional group containing white's first move. Since white's first move at a 7-cube is an element of 3 one-dimensional groups, it is possible to select such a move with the extra constraint that black's first move is not an element of the same group.

Using this approach, it turns out that there are eight different ways in which the 12 two-ply positions are reduced to 7 three-ply positions. We represent a three-ply position by a three-tuple $< w_1, w_2, b_1 >$, with w_1 and w_2 the cube number of the white stones, and b_1 the cube number of the black stone. The cube numbers for each of the 64 cubes of the qubic board are shown in figure 4.4. The eight ways to create 7 three-ply positions is as

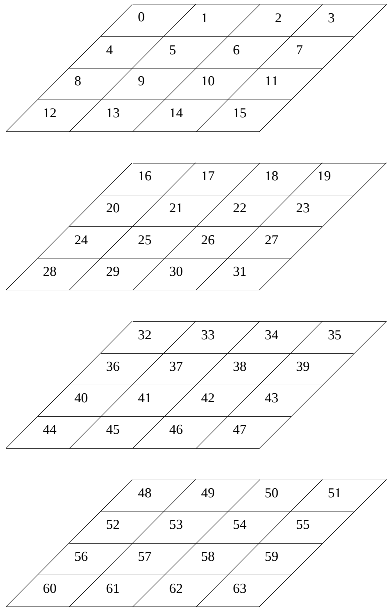


Figure 4.4: Cube numbers on the qubic board.

follows.

$$< 0, 3, 12 >, < 0, 3, 21 >, < 0, 3, 60 >,$$

$$((< 0, 3, 5 >, < 0, 3, 29 >) \vee (< 0, 3, 20 >, < 0, 3, 24 >)),$$

$$((< 0, 12, 1 >, (< 0, 3, 28 > \vee < 0, 3, 61 >))\vee$$

$$(< 0, 3, 28 >, (< 0, 3, 1 > \vee < 0, 12, 1 >)))$$

For each of these eight groups of 7 three-ply positions, we have created the set of all four-ply positions. Since there are 61 legal moves per position, initially 427 four-ply positions were created. After applying automorphisms, however, 195, 195, 217, 217, 226, 226, 241 or 241 positions remain, depending on the group of three-ply positions. The 3-ply position $< 0, 3, 1 >$ looks bad for white, since black has blocked the potential white threats. Therefore, the group expanding to 195 four-ply positions of which $< 0, 3, 1 >$ is an element is ignored. The remaining group of 7 three-ply positions expanding to 195 four-ply positions is listed below.

$$< 0, 3, 12 >, < 0, 3, 20 >, < 0, 3, 21 >, < 0, 3, 24 >,$$

$$< 0,3,60 >, < 0,3,61 >, < 0,12,1 >$$

The same group of three-ply positions was selected by (Patashnik, 1980).

Since each 7-cube move is also an element of three two-dimensional groups, we could instead try moves at 7-cubes in the same two-dimensional group as the first white move. Again, the 12 two-ply positions can be reduced to 7 three-ply positions, this time in seven different ways, all of which have been listed below.

$$< 0,15,51 >, < 0,15,21 >, < 0,60,3 >, < 0,51,6 >, < 0,51,5 >,$$

$$((< 0,15,1 >, (< 0,60,1 > \vee < 0,60,7 >))\vee$$
$$(< 0,60,7 >, (< 0,15,1 > \vee < 0,60,1 >))\vee$$
$$(< 0,60,1 >, (< 0,60,7 > \vee < 0,51,7 > \vee < 0,15,1 >)))$$

The number of four-ply positions grown from each of these groups is 219, 229, 229, 229, 229, 240 and 240.

We have selected the same set of three-ply positions as Patashnik (1980), since it yields the smallest set of four-ply positions. This choice also allows us to compare his results with ours.

Investigating the subtrees

Pn-search has been applied to all but two of the 195 four-ply positions. The two remaining positions have the property that the two black stones lie within a group G_1 which intersects the group G_2 containing the two white stones. By playing at the intersection c of G_1 and G_2, either player can create a threat and counter a potential threat by the opponent at the same time. Therefore, move c is regarded as a strong move for white. However, in pn-search we explicitly forbid white to create threats. In these two positions, this heuristic deprives white of her best move, and allows black to gain counterplay. Therefore, in these two positions we played white's third move at c and countered the threat with black's third move before applying pn-search.

Tests on these two four-ply positions showed that one position was quickly solved through an alternative third move for white, while the pn-search for the other position was terminated after a DAG of a million nodes had been created. In the latter case, these tests suggest that playing at intersection c may be white's only path to a win.

4.5.2 Statistics

In this section we present the statistics of running pn-search on the 195 positions described in the previous section. We distinguish between execution time, pn-search DAG size, db-search evaluations and solution size. We also present an example winning line.

Execution time

All experiments were run on a SPARCstation 2 at the Vrije Universiteit in Amsterdam. The machine has 128 Megabytes of internal memory, allowing pn-search trees of up to 1 million nodes to fit in internal memory, without slowing down the search by swapping to disk. The SPARCstation 2 is estimated to have an execution speed of 28 MIPS.

The CPU time needed for the 195 positions (193 four-ply positions and 2 six-ply positions) was 55,700 seconds, or roughly 15.5 hours.

Pn-search DAG size

The pn-search tree size is the number of nodes created during the search. Since no nodes are removed from the DAG once created, this equals the size at termination. We remark that terminal nodes solved by db-search are not included in the DAG, as described in section 4.4.2.

The smallest pn-search DAG consisted of 884 nodes, while the largest consisted of 310,000 nodes, with the median at 4,000 nodes and the average at 10,000 nodes. Only one other DAG was larger than 60,000 nodes, at 118,000. These two difficult positions are $< 0, 3, 1, 7 >$ (118,845 nodes) and $< 0, 3, 21, 22 >$ (310,424 nodes). (The positions are described by the two cubes containing white stones, followed by two cubes containing black stones.)

Db-search evaluations

A total of 3.5 million positions were evaluated with db-search, for white to move, and 0.9 million positions for black to move. Comparing the total number of evaluations, 4.4 million, with the sum of the sizes of the pn-search DAG's created, 2.0 million, it follows that not creating nodes for the solved positions in the tree shrinks the tree to be held in memory by a factor of almost 3.2.

Each time db-search created a DAG of 500 nodes or more it was reported. This occurred just 241 times, out of over 4 million evaluations. Among these,

depth	positions
0	1
2	12
4	195
6	2000
8	1426
10	1074
12	772
14	573
16	345
18	142
20	62
22	36
24	8
26	4
28	4
total	6654

Table 4.1: Number of positions in the qubic solution

31% were successful evaluations. The largest successful evaluation took 2,008 nodes, while the largest failed evaluation took 3,153 nodes. Creating the db-search DAG of 3,153 nodes took less than 5 seconds CPU time.

Solution size

The solution tree we found for qubic consists of a set of positions with white to move, and a winning move for each of these positions. The number of positions at each depth of the tree is shown in table 4.1.

A deep winning line

Our approach to solving qubic makes it difficult to determine the length of the winning line constituting optimal play by both sides. First, db-search does not search for the shortest winning threat sequence, but terminates as soon as any winning sequence is found. Second, pn-search does not search for the shallowest solution, but for one which reduces the work still to be done

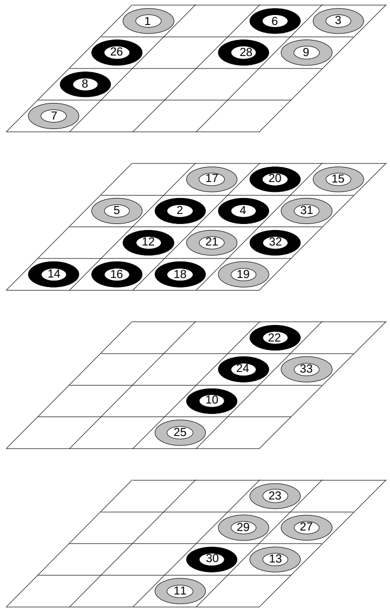


Figure 4.5: A deep winning line.

to complete the proof.

Therefore, the 4 lines of depth 28, as shown table 4.1, followed by the winning threat sequence found by db-search are not necessarily the longest lines with optimal play by both sides. Nevertheless, the winning line shown in figure 4.5, consisting of 33 ply, is one of these four.

Below follows a short analysis of the game. The first four ply consist of white and black occupying 7-cubes. White 5 comes somewhat as a surprise: white occupies a 4-cube to block the potential threat created by black. Black 6 similarly blocks white's potential threat. With white 7 two more potential threats are created, of which one is countered by black 8. White 9, again at a 4-cube, creates several opportunities for white to win through a threat sequence. Black then starts creating threats up to black 28, each of which is followed by a forced move by white. White 29, countering black 28, regains the initiative for white by creating a threat. After black's forced reply, white creates a double threat with white 31 and wins with white 33.

We remark that while this may be the line of play where black postpones the end as long as possible, after white 9 all white had to do is counter threats created by black. The first time white had to select a move again, she had many options to win, of which white 31 is the simplest way. Therefore, from the point of view of human players, playing white in this line only requires skill up to white 9. We remark that other lines exist in the solution to qubic which require more strategic moves by white, although the winning line is shorter.

4.5.3 Comparison with Patashnik

In this section we compare our solution with that of Patashnik. This comparison is not meant to criticize Patashnik's work in any way. On the contrary: his ability to solve qubic in the late 1970s constrained by the computing resources of that time should be regarded as one of the more impressive achievements in games research. The goal of our comparison is only to obtain information on the performance of db-search and pn-search.

First, we compare the performance of pn-search in selecting strategic moves with that of Patashnik as strong human player. Second, we compare the performance of db-search with that of the forced-sequence searcher used by Patashnik. Third, we summarize the results.

Pn-search vs. human expert

As stated in section 4.5.1, we have researched the same 195 four-ply positions as Patashnik (1980). Patashnik defines a *strategic move* as a non-obvious move for white, thus excluding moves suggested by the tactical search, and excluding forced moves for white when countering threats made by black. To compare our results with Patashnik's we must exclude all forced moves for white from the 6,654 moves in our solution to qubic. The number of strategic moves per depth, for both Patashnik and our solution, are shown in table 4.2.

From table 4.2 it follows that Patashnik made 10% fewer strategic moves than pn-search. For Patashnik, making the strategic moves was a bottleneck in solving qubic, as each strategic move was made by hand. Consequently, minimizing the number of strategic moves was a major concern in his research. Therefore, we feel that pn-search, while not explicitly trying to minimize the number of strategic moves in our solution to qubic, has performed well.

level	Patashnik	pn-search
0	1	1
2	12	12
4	195	195
6	1448	1960
8	788	668
10	309	248
12	110	113
14	51	41
16	15	14
18	0	2
total	2929	3254

Table 4.2: Number of positions in qubic solution per depth.

Db-search vs. forced-sequence search

Before we can compare the total amount of CPU time spent by Patashnik with our results, we must allow for the different types of machines used. Although it is difficult to compare such vastly different machines, an expert indicated that if the performance had to be expressed in MIPS, his best estimate for the DEC-10 would be between 2 and 3 MIPS (Witmans, 1994). Compared with the approximately 28 MIPS of the SPARCstation 2, we assume that our machine was between 10 and 20 times faster than the hardware used by Patashnik. In our comparison we disregard the fact that today's computers are equipped with much larger memories than 15 years ago.

Our first comparison is based on the total solution time. Patashnik's solution took approximately 1500 hours, not counting time wasted on backtracking due to bad strategic moves, and computer failure. We compare this figure with our 15.5 hours of CPU time. Factoring out the difference in machine speed, our solution is between 5 and 10 times faster than the solution found by Patashnik. As almost all CPU time is spent on searching forced sequences, for both Patashnik's solution and ours, this is a first indication that db-search may be 5 to 10 times more efficient than a conventional forced-sequence search as implemented by Patashnik.

Comparing the execution time of individual instances of Patashnik's forced-sequence search and our db-search is slightly more difficult. Patashnik remarks that typically his forced-sequence search took about two seconds, but occasionally as long as half an hour. He also remarks that if his strategic moves had been slightly worse, an uncontrollable combinatorial explosion would have occurred in some positions.

For a second comparison, we will assume an average of two seconds per forced-sequence search for Patashnik. To simplify matters for db-search, we assume that all 55,700 seconds of CPU time were spent on db-search evaluations (disregarding the time necessary to perform pn-search, to check for automorphisms, and to find transpositions in the pn-search DAG). During this time over 4.4 million evaluations were performed, for an average of almost 80 evaluations per second. Given the difference in machine speed, we find that db-search is between 8 and 16 times faster than Patashnik's forced-sequence search.

As a third comparison, we look at the slowest evaluation of db-search (less than 5 seconds) and the slowest forced-sequence search of Patashnik (approximately 30 minutes). This difference implies a gain factor for db-search of 20 to 40 on the difficult positions.

Summarizing comparison with Patashnik

We conclude that applying expert knowledge (Patashnik) to solve qubic, results in a marginally smaller solution compared to applying the knowledge-free search technique pn-search. On qubic, db-search performs between 5 and 40 times better than a conventional search algorithm. In our opinion, the qubic results illustrate the strengths of both pn-search and db-search.

4.5.4 Reliability

There are many sayings concerning the number of errors made by programmers, among which one of the most famous is: *There is always one more bug.* These bugs may vary from uninitialized variables to serious programming-logic errors. For a program the size of our qubic implementation (over 6,000 lines of C code), there may thus be some doubts about the reliability of our results. In this section we present some measures taken to ensure their correctness.

The most complicated part of the program consists of the db-search implementation. During the implementation errors were made, and corrected but, of course, ensuring that this code is error free is a difficult task. Therefore, the products of db-search, viz. winning threat sequences, were independently examined for their correctness. Once a potential winning threat sequence was found, the program started from the initial search position and played the sequence move by move. After each move by the attacker it is investigated whether (1) the defender has a threat, and (2) the attacker has a threat at the cube suggested as next move for the defender. After the last move by the attacker, it is investigated if indeed a group of four cubes has been occupied by the attacker. If any of these investigations show that db-search made an error, this is reported. No errors have been discovered in db-search during the process of solving qubic. We conclude that the product of the most complicated part of the program is independently verified.

The second most complicated part of the program consists of the pn-search implementation. Fortunately, pn-search has been implemented for several different games, ensuring that the chances of implementation errors are much lower than for new code. Still, the search process is too complicated to monitor fully, and thus errors may go unnoticed. To examine our results, a successful pn-search produces a small database consisting of the positions in the solution tree. After we solved all 195 four-ply positions, we merged these databases. Next, we created a database-checking module. For each position in the database with white to move a successor should be contained in the database. For each position in the database with black to move, all successors are generated. A successor should either be contained in the database, or white should have a winning threat sequence as found by db-search. We have thus checked the database and found it complete.

Third, our solution is consistent with the results of Patashnik (1980), but arrived at independently. In conclusion, we believe that our implementation may be regarded as reliable.

Chapter 5

Go-Moku

In this chapter we discuss the application of pn-search and db-search to go-moku. In the previous chapter we stated two goals for chapters 4 and 5, which we repeat here. The first goal is to explain in detail how pn-search and db-search have been applied to two combinatorially complex problems. The second goal is to show that qubic and go-moku can be solved, thereby positively answering our first research question (cf. section 1.4) for two specific games.

In several ways, qubic and go-moku are related games, with go-moku being the more complex one. The relationship between qubic and go-moku is expressed in the organization of this chapter: almost every section has a corresponding section in chapter 4. We mention this relationship for readers who are particularly interested in the application of db-search or pn-search. Comparing corresponding sections on qubic and go-moku may provide additional insight in these algorithms.

The chapter is organized as follows. In section 5.1 we provide a background to investigations in go-moku. The rules of go-moku and common strategies are presented in section 5.2. The application of db-search to go-moku is described in section 5.3. The role of pn-search in the solution of go-moku is explained in section 5.4. The results of our investigations are presented in section 5.5.

5.1 Background

Among the games of the Olympic List, go-moku has the simplest rules: two players (black and white) alternate placing stones on a 15×15 square lattice with the goal of obtaining a line of exactly five consecutive stones of the player's color. While its roots lie in China and Japan, it is also popular in several countries of Europe and the former Soviet-Union. Part of go-moku's popularity must be ascribed to the fact that it can be played with pencil and paper, allowing it to be played virtually everywhere (including classrooms) by virtually everyone (including bored students).

In Japan professional renju players (renju being a complicated variant of go-moku) have studied go-moku in detail and have stated that the player to move first (black) has an assured win (Sakata and Ikawa, 1981). These statements are sometimes accompanied by a list of main variations, such as the 32-page analysis in Sakata and Ikawa (1981). Close examination of these analyses reveals that in each position only a small number of white moves are analyzed. For example, after black's first move at the center of a 15×15 board, white has 35 distinct moves, of which 2 are adjacent to black's first move, ignoring symmetrically equivalent moves. In Sakata and Ikawa (1981) only the variations after the 2 moves adjacent to black's first move are discussed. As far as we know, prior to this work no complete proof of black's win in go-moku has been published.

Until this study, all go-moku programs have been defeated at least once or been in a lost position when playing black. As an example of the latter we mention the game between the go-moku 1991 world-champion program *Vertex* (black) and the program *Polygon* (white). *Vertex* maneuvered itself into a position provably lost for black (Uiterwijk, 1992a). As an aside we note that *Polygon* played its first move non-adjacent to the first black stone, indicating that finding a win in such a variation may not be entirely obvious.

Summarizing, go-moku is assumed to be a first-player win but, as far as we know, no complete proof has been published nor has any go-moku program ever been shown to be unbeatable when playing black.

At this point we reiterate our remark of section 4.1. In earlier publications we have used the term *threat-space search* for the application of db-search to qubic and go-moku. In this text we only use the term db-search.

5.2 Rules and strategies

Go-moku is a two-player game, related to the well-known trivial game of tic-tac-toe. While in tic-tac-toe players must create a line of three consecutive markers of their color on a restricted 3×3 board, in go-moku players must create a line of five on a practically unrestricted lattice. Through the years, several variants of go-moku have been developed, which are described in detail in section 5.2.1. Next, threats and threat trees are discussed in section 5.2.2. Finally, in section 5.2.3 some insight is given into the way human go-moku experts think.

5.2.1 Rules

In go-moku, simple rules lead to a highly complex game, played on the 225 intersections of 15 horizontal and 15 vertical lines. Going from left to right the vertical lines are lettered from a to o; going from the bottom to the top the horizontal lines are numbered from 1 to 15. Two players, black and white, move in turn by placing a stone of their own color on an empty intersection, henceforth called a *square*. Black starts the game. The player who first makes a line of five consecutive stones of her color (horizontally, vertically or diagonally) wins the game. The stones once placed on the board during the game never move again nor can they be captured. If the board is completely filled, and no one has five-in-a-row, the game is drawn.

Go-moku variants

Many variants of go-moku exist; they all restrict the players in some sense, mainly reducing the advantage of black's first move. We mention four variants.

Non-standard boards In the early days the game was played on a 19×19 board, since go boards have that size. Some people prefer to think of go-moku as being played on an infinite board. However, a larger board increases black's advantage (Sakata and Ikawa, 1981), which resulted in the standard board size of 15×15.

Free-style go-moku An overline is a line of six or more consecutive stones of the same color. In this variant, an overline is regarded as a win.

Standard go-moku In the variant of go-moku played most often today, an overline does not win (this restriction applies to both players). Only a line of exactly five stones is considered as a winning pattern.

Renju A professional variant of go-moku is renju. White is not restricted in any way, e.g., an overline wins the game for white. However, black is not allowed to make an overline, nor a so-called *double three* or *double four* (cf. Sakata and Ikawa (1981)). If black makes any of these patterns, she is declared to be the loser. Renju is not a symmetric game: to play it well requires different strategies for black and for white. Even though black's advantage is severely reduced, she still seems to have the upper hand.

We have investigated both free-style go-moku and standard go-moku. We remark that in this chapter we discuss free-style go-moku unless it is explicitly stated otherwise.

Opening restrictions

In an attempt to make the game less unbalanced, opening restrictions have been imposed on black. We mention two such restrictions.

Professional go-moku In professional go-moku, black is forced to make her first move in the center of the board. White must play her first move at one of the eight squares adjacent to black's first move. Black's second move must be outside the set of 5×5 squares centered by black's first stone.

Professional renju In professional renju, the game starts with two players which are named *temporary black* and *temporary white*. Temporary black plays her first move at the center of the board, while temporary white plays her first move adjacent to the black stone on the board. Due to symmetry, there are only two distinct first moves for temporary white. For each of these two, there are 12 selected squares where temporary black is allowed to play her second move. Thus, there are 24 possible 3-ply sequences in this variant. Next, temporary white may choose between playing black or white for the remainder of the game. Temporary black automatically plays the other color. Then, white plays her second move. Finally, black selects two squares for her third move and gives white the choice between these two. From there, the game continues according to the rules of standard renju.

In our research we have investigated variants of go-moku without any opening restrictions.

5.2.2 Threats and threat trees

We describe the four types of threats in go-moku, followed by a discussion of threat trees and winning threat trees.

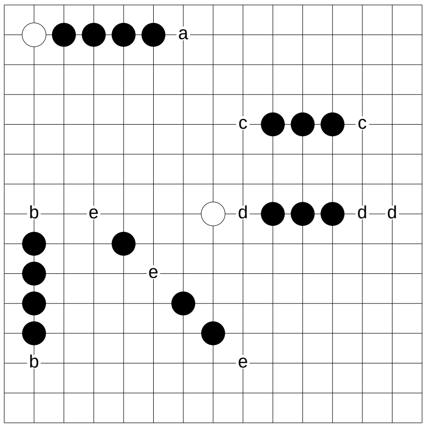


Figure 5.1: Threats in go-moku.

Threats

In go-moku a threat is an important notion; the main types have descriptive names: the *four* (figure 5.1a) is defined as a line of five squares, of which the attacker has occupied any four, with the fifth square empty; the *straight four* (figure 5.1b) is a line of six squares, of which the attacker has occupied the four center squares, while the two outer squares are empty; the *three* (figure 5.1c and 5.1d) is either a line of seven squares of which the three center squares are occupied by the attacker and the remaining four squares are empty, or a line of six squares with three consecutive squares of the four center squares occupied by the attacker and the remaining three squares empty; the *broken three* (figure 5.1e) is a line of six squares of which the attacker has occupied three non-consecutive squares of the four center squares, while the other three squares are empty. A winning pattern, i.e., a line of five squares, all occupied by one player, is named a *five*.

If a player constructs a four, she threatens to win on the next move. Therefore, the threat must be countered immediately at the empty square of the four. If a straight four is constructed, the defender is too late, since there are two squares where the attacker can create a five at her next move (unless, of course, the defender has the opportunity to win at her next move). With a three, the attacker threatens to create a straight four at her next move. Thus, even though the threat has a depth of two moves, it must be countered immediately. If an extension at both sides is possible (figure 5.1c), then there

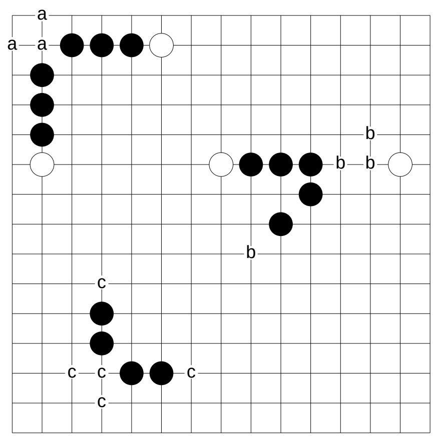
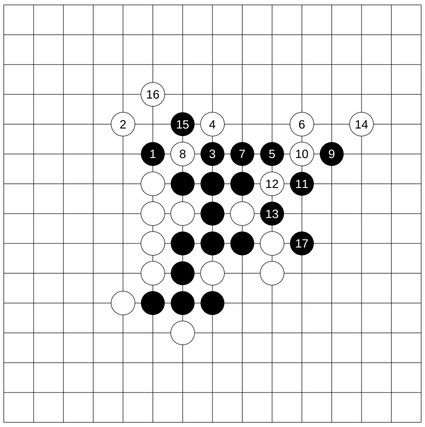


Figure 5.2: Complicated threats.

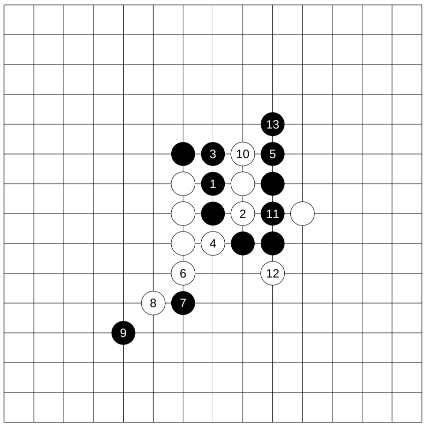
are two defensive moves: both directly adjacent to the attacking stones. If only one extension is possible then three defensive moves are available (figure 5.1d). Moreover, against a broken three, three defensive moves exist (figure 5.1e).

We remark that more complicated threats exist, which threaten to win in two or more moves. Three examples are shown in figure 5.2, in each of which black threatens to play at the intersection of the two lines of black stones. In figure 5.2a, black threatens to create a double four, in figure 5.2b, black threatens to create a four-three, and in figure 5.2c, black threatens to create a double three. Each of these is a winning pattern. White can counter the threats of figure 5.2 in 3, 4 and 5 possible ways, respectively.

In our research we have not included the patterns of figure 5.2 as threats for three reasons. First, the large number of defensive moves per threat does not combine well with our transformation of the winning threat-tree search to a single-agent search, as described in section 5.3.1. Second, recognizing threats which consist of a single line on the board can be performed more efficiently than recognizing threats which consist of combinations of lines. Third, the threats shown in figure 5.2 are only a small sample of the complete set of more complicated threat patterns, making inclusion of all possible threats of go-moku a complex task. In Uiterwijk (1992b) a program based on a large set of threat patterns is described.



(a) Fours only.



(b) Threes only.

Figure 5.3: Winning threat variations

Threat trees

To win the game against any opposition a player needs to create a *double threat* (either two threes, two fours, or a three and a four). In most cases, several threats are executed before a double threat occurs. A tree in which each attacker move is a threat is called a *threat tree*. A threat tree leading to a (winning) double threat in each variation is called a *winning threat tree*. A variation in a winning threat tree is called a *winning threat variation*. Each threat in the tree forces the defender to play a move countering the threat. Hence, the defender's possibilities are limited.

In figure 5.3a a position is shown in which black can win through a winning threat variation consisting of fours only. Since a four must be countered immediately, the whole sequence of moves is forced for white.

In figure 5.3b a position is shown in which black wins through a winning threat variation consisting of threes, twice interrupted by a white four. As mentioned earlier, white has at each turn a limited choice. During the play, she can create fours as is shown in figure 5.3b. Still, her loss is inevitable.

5.2.3 Human strategies

During the second and third Computer Olympiad (Levy and Beal, 1991; Van den Herik and Allis, 1992), we observed two human expert go-moku players (A. Nosovsky, 5th dan and N. Alexandrov, 5th dan). These Russian players are involved in two of the world's strongest go-moku playing programs (*Vertex* and *Stone System*). While observing the experts, it became clear that they are able to find quickly sections on the board where a winning threat tree can be created, regardless of the number of threes which are part of the winning threat tree. The depth of these winning threat trees are typically in the range of 5 to 20 ply.

The way a human expert finds winning threat trees so quickly can be broken down into the following four steps.

1. A section of the board is chosen where the configuration of stones seems favorable for the attacking player. It is then decided whether enough attacking stones can collaborate making it useful to search for a winning threat tree. This decision is based on a "feeling", which comes from a long experience in judging patterns of stones (cf. De Groot (1965)).

2. Threats are considered, and in particular the threats related to other attacking stones already on the board. Defensive moves by the opponent are mostly disregarded.

3. As soon as a variation is found in which the attacker can combine her stones to form a double threat, it is investigated how the defender can refute the potential winning threat variation. Whenever the opponent has more than one defensive move, an examination is made to see whether the same threats work in all variations of the threat tree. Moreover, it is investigated whether the opponent can insert one or more fours, effectively neutralizing the attack.

4. If only a few variations of the tree do not lead to a win via the same threat variation, an examination is made to see whether the remaining positions can be won via other winning threat trees.

In practical play, a winning threat tree often consists of a single set of attacking moves applicable to each variation of the tree, independent of the defensive moves.

We remark that the size of the state space is considerably reduced by first searching for one side (the attacker). Only if a potential winning threat tree is found is the impact of defensive moves investigated. This approach is supported by the analyses given in (Sakata and Ikawa, 1981). When presenting a winning threat tree, they only provide the moves for the attacker, thus indicating that the set of attacking moves works irrespective of the defensive moves. Possible fours which the defender can create without refuting the threat tree can be neglected altogether

In positions without winning threat trees, the moves to be played preferably increase the potential for creating threats or, whenever defensive moves are called for, the moves chosen will reduce the opponents potential for creating threats. The human evaluation of the potential of a configuration is based on two aspects: (1) direct calculations of the possibilities, (e.g., if the opponent does not answer in that section of the board) and (2) a so-called good shape (i.e., configurations of which it is known that stones collaborate well).

In section 5.3 we model the above thinking process in our application of db-search to go-moku.

5.3 Applying db-search

As mentioned before, threat trees play a dominant role in go-moku. To play go-moku well, it would be advantageous to have a module which determines whether a winning threat tree exists. Our application of db-search to go-moku is restricted to searching for winning threat trees.

This section consists of four parts. First, in section 5.3.1 we describe how the adversary-agent state space, if restricted to a subset of all possible threat trees, can be transformed into a single-agent state space. Second, in section 5.3.2 we illustrate how the single-agent state space thus created for go-moku fits in the framework for db-search as presented in chapter 3. Third, in section 5.3.3 we discuss properties of the single-agent state space for go-moku which have not been included in the framework of section 5.3.2. For each of these properties it is explained how our implementation of db-search handles them. Fourth, in section 5.3.4 heuristics are described which lead to a significantly improved efficiency, at the cost of a slightly reduced efficacy. Fourth, in section 5.3.5 we describe the additional requirements necessary to apply db-search to standard go-moku instead of free-style go-moku.

5.3.1 A single-agent search in go-moku

Our description of the single-agent state space in go-moku consists of a set of definitions, an interpretation of the definitions, and the transformation of the adversary-agent state space to a single-agent state space.

Definitions

In the previous sections we have introduced the concept of threats, threat trees and winning threat trees. For our application of db-search to go-moku, we formally define the notions *threat* (definition 5.1), *reply* (definition 5.2), *threat sequence* (definition 5.3), *potential winning threat sequence* (definition 5.4) and *winning threat sequence* (definition 5.5).

Definition 5.1 *A* threat *in* go-moku *is a move by the attacker creating a five, a straight four, a four, a three or a broken three. A five and a straight four are called* double threats*, while a four, three and broken three are called* single threats. *The* squares related to a threat *are the 5 (five and four), 6 (straight four, three, broken three) or 7 (three) squares in the line of the threat (cf. section 5.2.2).*

Definition 5.2 *A* reply *to a threat T in* go-moku *is the set of defender moves R, such that each element of R counters T. Against a five and a straight four, R is empty, against a four, R consists of one move, against a three R consists of two or three moves, and against a broken three, R consists of three moves.*

Definition 5.3 *A* threat sequence *in* go-moku *is any sequence of moves $(a_1, d_1, a_2, d_2, \ldots, a_n, d_n)$, with $n \geq 1$, such that each a_i, $1 \leq i \leq n$ is a single threat, and each d_i is the reply to a_i.*

Definition 5.4 *A* potential winning threat sequence *in* go-moku *is any sequence $(a_1, d_1, \ldots, a_n, d_n, a_{n+1}, d_{n+1})$, such that $(a_1, d_1, \ldots, a_n, d_n)$ is a threat sequence, a_{n+1} is a double threat and d_{n+1} is the reply to a_{n+1}.*

Definition 5.5 *A* winning threat sequence *in* go-moku *is a potential winning threat sequence $(a_1, d_1, \ldots, a_n, d_n, a_{n+1}, d_{n+1})$, for which it has been checked that the defender cannot counter the threat sequence by:*

1. *interjecting a sequence of threats the attacker must respond to, leading to a win for the defender*

2. *interjecting a sequence of threats the attacker must respond to, leading to occupation of a square related to a threat a_i, before the defender has played the reply to d_i.*

Interpretation

Here we elaborate on the definitions presented above. Definition 5.1 defines threats in accordance with the definitions of section 5.2.2. The only difference is our inclusion of the five as a threat, and naming the straight four and the five double threats. The reason for doing so is explained below.

When a double three is created, it is assumed that the defender counters one of them, allowing the attacker to convert the remaining three into a straight four at the next move. When a double four is created, it is assumed that the defender counters one of them, allowing the attacker to convert the remaining four into a five at the next move. When a four-three is created, depending on the threat countered by the defender, the attacker can create either a five or a straight four. Thus, we may recognize double threats one move after they appear in the form of straight fours or fives.

The definition of a reply forms a crucial step in our conversion of the adversary-agent state space of go-moku into a single-agent state space. Human strategies imply that often threat trees are found such that in each variation the same attacking moves are played. In other words, the choice between defensive moves in such threat trees is irrelevant. We convert these threat trees to threat sequences by allowing the defender to play *all* defensive moves as a single reply. In figure 5.4, we have depicted such a winning threat sequence, consisting of four threats. After black 1, white has the three-move reply 2. After black 3, white has the two-move reply 4. After black 5, white has the three-move reply 6. Black 7 creates a straight four, to which the reply set is empty.

Clearly, in free-style go-moku, having extra stones on the board is never a disadvantage. Thus, if a variation wins for the attacker when the defender is allowed to play replies consisting of multiple stones, then the variation wins also if the defender is forced to select one stone from each multiple-stone reply.

Positions exist in which the multiple-stone reply leads to counter play for the defender, while the attacker would win in all variations through the same attacking moves if the defender were restricted to playing one stone per reply, but these are rare.

A potential winning threat sequence as defined in definition 5.4 has investigated only *local* defensive moves, i.e., after each threat, it is assumed that the defender must immediately counter the threat. A winning threat sequence has also been checked for *global* defensive moves, i.e, that the squares not related to the threat sequence have been investigated for their influence

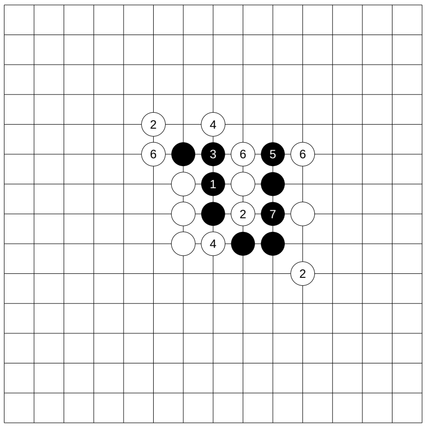


Figure 5.4: White defending with multiple-stone replies

on the success of the threat sequence.

Adversary-agent vs. single-agent

As we have seen, in (winning) threat sequences, each reply by the defender is implied by the previous attacker move. Therefore, we may conceptually merge these two moves into a single meta-move.

The state space created by these meta-moves is no longer an adversary-agent state space, but instead a single-agent state space. In the remainder of this section, when discussing meta-moves, we assume that the attacker move and defender move in a meta-move are made simultaneously.

5.3.2 A db-search framework for go-moku

In this section we define a db-search framework for the single-agent state space of go-moku, defined in the previous section. We mention that the framework only involves *local* defensive moves, while ignoring *global* defensive moves. Global defensive moves of a position will be discussed in section 5.3.3. The terminology introduced in chapter 3 is used throughout this chapter.

Notation

Lines of five, six and seven squares play an important role in go-moku. For notational purposes, we define the following sets.

$$\begin{aligned} G_5 &= \{\{s_1, s_2, \ldots, s_5\} \mid s_1, \ldots, s_5 \text{ form a line of five squares}\} \\ G_6 &= \{\{s_1, s_2, \ldots, s_6\} \mid s_1, \ldots, s_6 \text{ form a line of six squares}\} \\ G_7 &= \{\{s_1, s_2, \ldots, s_7\} \mid s_1, \ldots, s_7 \text{ form a line of seven squares}\} \end{aligned}$$

We mention that on a 15×15 board, G_5 has 572 elements, G_6 has 500 elements and G_7 has 432 elements.

Furthermore, we define a linear order on the squares of the go-moku board, such that $a1 < a2 < \ldots < a15 < b1 < \ldots < o15$. Clearly, the outer squares of a line are always minimal and maximal within the line, with respect to this ordering.

Attributes

The set U of all attributes is defined as follows. $U = \{S(i, x) \mid a1 \leq i \leq o15 \wedge x \in \{\circ, \bullet, \cdot\}\}$. Attribute $S(i, x)$ represents the fact that square i is occupied by the attacker ($\circ$), occupied by the defender ($\bullet$), or empty ($\cdot$). It can easily be checked that U has 675 elements.

Operators

The operator f_{FI,g_5} (five), for $g_5 = \{s_1, \ldots, s_5\}$ and $g_5 \in G_5$, is defined as follows.

$$\begin{aligned} f^{pre}_{FI,g_5} &= \{S(s_1, \circ), S(s_2, \circ), S(s_3, \circ), S(s_4, \circ), S(s_5, \cdot)\} \\ f^{del}_{FI,g_5} &= \{S(s_5, \cdot)\} \\ f^{add}_{FI,g_5} &= \{S(s_5, \circ)\} \end{aligned}$$

The operator f_{SF,g_6} (straight four), for $g_6 = \{s_1, \ldots, s_6\}$ and $g_6 \in G_6$, and $s_1 < s_2, s_3, s_4, s_5 < s_6$, is defined as follows.

$$\begin{aligned} f^{pre}_{SF,g_6} &= \{S(s_1, \cdot), S(s_2, \circ), S(s_3, \circ), S(s_4, \circ), S(s_5, \cdot), S(s_6, \cdot)\} \\ f^{del}_{SF,g_6} &= \{S(s_5, \cdot)\} \\ f^{add}_{SF,g_6} &= \{S(s_5, \circ)\} \end{aligned}$$

The operator f_{FO,g_5} (four), for $g_5 = \{s_1, \ldots, s_5\}$ and $g_5 \in G_5$, is defined as follows.

$$\begin{aligned}
f^{pre}_{FO,g_5} &= \{S(s_1,\circ), S(s_2,\circ), S(s_3,\circ), S(s_4,\cdot), S(s_5,\cdot)\} \\
f^{del}_{FO,g_5} &= \{S(s_4,\cdot), S(s_5,\cdot)\} \\
f^{add}_{FO,g_5} &= \{S(s_4,\circ), S(s_5,\bullet)\}
\end{aligned}$$

The operator f_{BT,g_6} (broken three), for $g_6 = \{s_1, \ldots, s_6\}$ and $g_6 \in G_6$, and $s_1 < s_2, s_3, s_4, s_5 < s_6$ and s_4 neither minimum nor maximum in $\{s_2, s_3, s_4, s_5\}$, is defined as follows.

$$\begin{aligned}
f^{pre}_{BT,g_6} &= \{S(s_1,\cdot), S(s_2,\circ), S(s_3,\circ), S(s_4,\cdot), S(s_5,\cdot), S(s_6,\cdot)\} \\
f^{del}_{BT,g_6} &= \{S(s_1,\cdot), S(s_4,\cdot), S(s_5,\cdot), S(s_6,\cdot)\} \\
f^{add}_{BT,g_6} &= \{S(s_1,\bullet), S(s_4,\bullet), S(s_5,\circ), S(s_6,\bullet)\}
\end{aligned}$$

The operator f_{T2,g_7} (three with 2 reply moves), for $g_7 = \{s_1, \ldots, s_7\}$ and $g_7 \in G_7$, and $s_1 < s_2 < s_3, s_4, s_5 < s_6 < s_7$, is defined as follows.

$$\begin{aligned}
f^{pre}_{T2,g_7} &= \{S(s_1,\cdot), S(s_2,\cdot), S(s_3,\circ), S(s_4,\circ), S(s_5,\cdot), S(s_6,\cdot), S(s_7,\cdot)\} \\
f^{del}_{T2,g_7} &= \{S(s_2,\cdot), S(s_5,\cdot), S(s_6,\cdot)\} \\
f^{add}_{T2,g_7} &= \{S(s_2,\bullet), S(s_5,\circ), S(s_6,\bullet)\}
\end{aligned}$$

The operator f_{T3,g_6} (three with 3 reply moves), for $g_6 = \{s_1, \ldots, s_6\}$ and $g_6 \in G_6$, and $s_1 < s_2, s_3, s_4, s_5 < s_6$ and s_2 either minimum or maximum in $\{s_2, s_3, s_4, s_5\}$, is defined as follows.

$$\begin{aligned}
f^{pre}_{T3,g_6} &= \{S(s_1,\cdot), S(s_2,\cdot), S(s_3,\circ), S(s_4,\circ), S(s_5,\cdot), S(s_6,\cdot)\} \\
f^{del}_{T3,g_6} &= \{S(s_1,\cdot), S(s_2,\cdot), S(s_5,\cdot), S(s_6,\cdot)\} \\
f^{add}_{T3,g_6} &= \{S(s_1,\bullet), S(s_2,\bullet), S(s_5,\circ), S(s_6,\bullet)\}
\end{aligned}$$

The set of all operators U_f is defined as follows.

$$\begin{aligned}
U_f = {} & \{f_{FI,g_5} \mid g_5 \in G_5\} \cup \{f_{SF,g_6} \mid g_6 \in G_6\} \cup \{f_{FO,g_5} \mid g_5 \in G_5\} \cup \\
& \{f_{BT,g_6} \mid g_6 \in G_6\} \cup \{f_{T2,g_7} \mid g_7 \in G_7\} \cup \{f_{T3,g_6} \mid g_6 \in G_6\}
\end{aligned}$$

We mention that on a 15×15 board, U_f contains 3076 operators, of which each can be applied in more than one way, resulting in a total number of 23596 possible applications of operators.

Initial state and goal states

The initial state consists of exactly 225 attributes, one per square indicating the contents of the square. Each possible configuration of black, white and empty squares in which neither player has occupied a line of five can serve as initial state. The set U_g of goal states is independent of the initial state, and is defined as follows.

$$\begin{aligned} U_g = \quad \{ \quad & \{S(s_1,\circ), S(s_2,\circ), S(s_3,\circ), S(s_4,\circ), S(s_5,\circ)\} \mid \\ & \{s_1,s_2,s_3,s_4,s_5\} \in G_5\} \cup \\ \{ \quad & \{S(s_1,\cdot), S(s_2,\circ), S(s_3,\circ), S(s_4,\circ), S(s_5,\circ), S(s_6,\cdot)\} \mid \\ & \{s_1,s_2,s_3,s_4,s_5,s_6\} \in G_6 \wedge s_1 < s_2,s_3,s_4,s_5 < s_6\} \end{aligned}$$

In other words, each state containing a five or straight four is a goal state. U_g is not singular.

Properties of the go-moku framework

The framework we have described above is monotonous. Furthermore, we can easily restrict ourselves to non-redundant paths. If U_g were singular, our U_k would be complete.

We can create a singular $U_{g'}$, by defining a special goal attribute G and operators which transform any element of U_g into G, which would result in a complete U_k. A discussion of the completeness of U_k would be premature, however, since so far we have ignored global defensive moves.

5.3.3 Go-moku specific enhancements to db-search

The db-search framework for go-moku presented in the previous section focuses only on the local defensive moves. For those moves we defined replies such that each defender move was forced, allowing us to transform the search into a single-agent search.

A search for global defensive strategies is only necessary to investigate whether a potential winning threat sequence is correct. Thus, given such a threat sequence, it should be investigated whether the defender has alternatives to the local reply to refute the threat sequence. To investigate the global defensive strategies, we perform single-agent searches, this time fixing the *attacker* choices. After each attacker move specified in the threat sequence, the resultant position is investigated for a global defensive strategy by the defender. We describe the investigations in four steps.

First, we define the *threat categories*, which play an important role in determining for each position the types of global defensive moves available. Second, we describe two ways in which global defensive moves may successfully counter a potential threat sequence. Third, we describe a set of parameters for db-search. Fourth, we describe how the module searching for winning threat sequences is composed of a series of db-searches.

Threat categories

The operators defined in section 5.3.2 can be divided in three categories. Category 0 consists of the five, category 1 of the straight four and four, and category 2 consists of the three and the broken three. Using these categories we can state exactly what kind of global defensive moves may be interjected by the defender while countering a threat sequence. Against a threat from category i, only threats from categories j can be used as global defensive moves, with $j < i$. Thus, against a five no global defensive moves exist, against a (straight) four only a five can serve as global defensive move, while against a three or broken three, both fives, straight fours and fours may serve as global defensive moves.

The above relation between global defensive moves and threat categories can easily be verified by noting that each threat in category i threatens to win in exactly i moves.

Global defensive strategies

In section 5.3.1 we have listed two ways in which the defender may successfully counter a threat by interjecting global defensive moves. First, she may create a sequence of threats leading to a win. Second, she may create a sequence of threats leading to the occupation of a square in the threat sequence.

Here we describe how db-search can be used to determine whether such a global defensive strategy exists. Our application of db-search for this purpose is such that we may erroneously decide that a defensive strategy exists, thus rejecting a winning threat sequence for the attacker, but that we will never overlook the existence of a defensive strategy.

To prevent confusion arising from the terms attacker and defender in this context, we assume here that player A has found a potential winning threat sequence, and we investigate whether player B has a global defensive strategy after move a_i by A. Three remarks concerning the application of db-search to search for global defensive strategies for player B are in order.

1. The goal set U_g for player B should be extended with singleton goals for occupying any square in threat a_j or reply d_j, with $j \geq i$.

2. If B finds a potential winning threat sequence (i.e., a global defensive strategy against the potential winning threat sequence of A), this threat sequence is not investigated for counter play of player A. Instead, in such a case we always assume that A's potential winning threat sequence has been refuted.

3. In the application of db-search for player B, only threats of categories less than the category of the threat played by A may be applied. Thus, in a db-search for player B, only threats having replies consisting of a single move are applied.

If we examine the description of db-search for B, we may find that the search is monotonous and contains no redundant paths. As argued before, U_g can be easily transformed into a singular $U_{g'}$, without a conceptual difference in the resulting U_k. Since any sequence found for player B is accepted as refutation of the potential winning threat sequence of A, we claim that if application of db-search does not find a global defensive strategy, such a strategy does not exist for player B.

We stress this point as it is a vital element in the process of solving go-moku: we must ensure that in no position we accept a threat sequence as winning, if the threat sequence could be refuted.

Parameters to db-search

Above, we have seen that db-search is used to find potential winning threat sequences as well as to investigate whether the defender has a global defensive strategy refuting a potential winning threat sequence. These searches are all performed by the same module, whose parameters are listed below.

1. The *position* to which db-search is to be applied.

2. The *attacker*, i.e., the player for whom a db-search is applied.

3. The *goal squares*, i.e., the set of squares, which, if one is occupied by the attacker, terminates the search.

4. The *defensive check option*. This is a Boolean value indicating whether a potential winning threat sequence should be investigated for counter play.

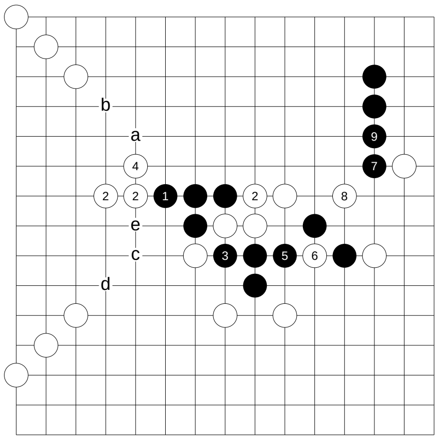


Figure 5.5: White refutes a potential winning threat sequence.

5. The *maximum category*, i.e., only threats of this category and lower categories may be applied.

The winning threat sequence module

Here we present a step by step description of the winning threat sequence module, with the aid of the position in figure 5.5.

To find the winning threat sequence for black in the position before black 1 of figure 5.5, db-search may be called with (1) that position; (2) attacker black; (3) the empty set of goal squares; (4) the defensive check option at value `true`; (5) maximum category 2. If the potential winning threat sequence shown in figure 5.5 is found, db-search will be called five more times, after black 1, black 3, black 5, black 7 and black 9. The parameters to db-search after, for instance, black 1 are: (1) the position after black 1; (2) attacker white; (3) the set consisting of the 28 squares related to the threats black 1 (7 squares), black 3 (5 squares), black 5 (5 squares), black 7 (5 squares) and black 9 (6 squares); (4) the defensive check option at value `false`; (5) maximum category 1.

After black 5, which is of category 1, black can only use a defensive strategy involving threats of category 0, i.e., fives only. However, to create a five after black 5, white should have created several fours after black 1 (of category 2), followed by the local defensive reply white 2. Therefore, we need to try threats of category 0 after black 5, for all positions which could arise after sequences of fours by white, in earlier global defensive strategy searches.

Indeed, if white, instead of playing 2 immediately after 1, interjects move a (followed by black's forced reply b) and move c (followed by black's forced d), then after white 2, black 3, white 4 and black 5, white can create a five at e.

Summarizing, to find the global defensive strategies, after each attacker move of category 2, a search for category 1 for the defender should be performed, while after each attacker move of category 1, a search for category 0 for the defender should be performed, from every position which could be reached by interjecting defender fours after previous threats of category 2 by the attacker.

5.3.4 Heuristically improving the efficiency of db-search

As we have argued before, the module which searches for winning threat sequences will only return a winning threat sequence if the winning threat sequence is guaranteed to lead to a win for the attacker. The opposite is not true: not all winning threat sequences will be found. This is caused by our acceptance of a global defensive strategy, without investigating whether the defensive strategy itself can be countered.

In the context of winning threat *trees* our search is far from complete, as we only find winning threat *sequences*, i.e., threat trees in which each variation leads to a win through the same attacking moves, in the same order.

In this section we present three heuristics which significantly increase the efficiency of our winning threat sequence module, at the cost of another (small) reduction in efficacy. Each of the heuristics, if at all applicable, is *not* applied during searches for global defensive strategies, in order to ensure that all existing refutations of potential winning threat sequences are found.

Global refutation

Our first heuristic for increasing the efficiency of db-search is based on the existence of *global refutations* in some positions. A global refutation is a configuration on the board which refutes all winning threat sequences of the attacker. An artificial example is depicted in figure 5.6.

Black to move has a large number of distinct potential winning lines at her disposal, each starting with a three. For instance, black 1 creates a double three immediately. White 2, however, creates a double four, thus successfully countering the three created by black 1. Alternative lines for black, such as

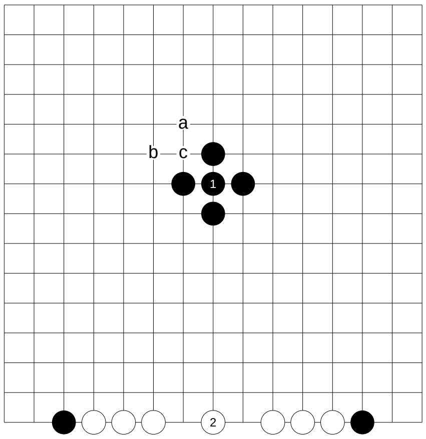


Figure 5.6: Global refutation of all potential winning lines.

black a, black b and black c, again creating a double three, are all also refuted by white 2.

Thus, while db-search, focusing on local defenses, finds many potential winning threat sequences, each of these is refuted by the search for global defensive strategies. Finding *all* several hundreds or thousands of potential winning threat sequences in such a position is clearly a waste of time.

As heuristic to recognize those positions, we check at the end of each db-search level the number of potential winning threat sequences investigated so far. If this number exceeds a preset threshold T, the search is terminated. Experiments showed that $T = 10$ leads to a largely increased efficiency, at a small cost in efficacy.

We remark that while searching for global defender strategies, the first potential winning threat sequence found is accepted as refutation. The search is therefore not influenced by this heuristic.

Category reduction

The category reduction heuristic is designed for a special type of global refutations. Let us suppose that the defender has a threat T_{c_1} of category c_1. If the attacker creates a threat T_{c_2} of category c_2, then either (1) $c_2 < c_1$, or (2) T_{c_2} should counter T_{c_1}, or (3) T_{c_1} is a refutation of T_{c_2}. As the search for potential winning lines does only consider local replies, countering T_{c_1} by T_{c_2} will only occur by accident.

Ignoring the option that this may happen, we obtain the category reduction heuristic: if in a node N of the db-search DAG, the defender has a threat of category c_1, for each descendent of N the attacker is restricted to threats of categories less than c_1.

We remark that this heuristic is switched off while searching for global defender strategies.

Restricted threes

The definitions of operator f_{T3,g_6} (three with 3 reply moves) and operator f_{T2,g_7} (three with 2 reply moves) imply that if the latter is applicable, the former is too. While in most positions where both are applicable they are interchangeable, operator f_{T2,g_7} is superior in that its reply consists only of 2 moves, thus diminishing the chances for counterplay. Only in rare occasions are both applicable, while only f_{T3,g_6} leads to a winning threat sequence.

To prevent the creation of threat sequences with as only difference the occurrence of f_{T3,g_6} instead of f_{T2,g_7}, we restrict application of f_{T3,g_6} to lines where f_{T2,g_7} is not applicable.

We remark that while searching for global defender strategies, only threats of categories 0 and 1 are applicable. The search is therefore not influenced by this heuristic.

5.3.5 Additional requirements for standard go-moku

Standard go-moku differs from free-style go-moku in the value of overlines: an overline is a win in free-style go-moku, while it is not in standard go-moku.

To apply our winning threat sequences module, as described in the previous sections, to standard go-moku, a few additional requirements are necessary. We discuss these requirements briefly.

First, we introduce the concept of a line extension. Second, we describe how a line extension influences a db-search for potential winning threat sequences. Third, we describe the influence of line extensions to the search for global defensive strategies.

Extensions

For each line $g \in G_5$, a square c is an *extension* of g, if $g \cup \{c\} \in G_6$. Similarly, for each line $g \in G_6$, a square c is an extension of g, if $g \cup \{c\} \in G_7$. We mention that the extension of a line $g \in G_7$ is defined analogously, after the set G_8 has been defined. The *extension set* of a line g, i.e., the set of all extensions of g consists of 0, 1 or 2 elements, depending on the position of g on the board, with respect to the board edge.

Line extensions and winning threat sequences

A winning threat sequence in standard go-moku must meet all the requirements for a winning threat sequence in free-style go-moku. An added requirement is that at the moment of execution of threat a_i, the squares in the extension set of a_i must not be occupied by an attacker stone.

An attacker stone may be placed at the extension of a threat in three distinct ways.

1. The stone was present in the initial position.
2. The stone is played while executing an earlier threat in the threat sequence.
3. The stone is played as forced response to a defender threat.

The first and second way of placing an attacker stone at a threat extension is checked during the db-search for potential winning threat sequences: an operator can only be applied if the extension squares are empty or occupied by the defender. During the combination stage of db-search, we ignore the occupation of extensions. Instead, after a potential winning threat sequence has been found, the extensions of all threats in the threat sequence are examined.

Line extensions and global defensive strategies

The third way of placing an attacker stone in a threat extension provides the defender with an extra global defensive strategy. This strategy fits as follows within the parameters provided to db-search. In addition to the set of goal squares provided for free-style go-moku, the set of extensions to the threats which have not yet been executed by the attacker is passed to db-search. A refutation of the potential winning threat sequence has been found, if one of the extensions has been occupied by the attacker (i.e., the player whose potential winning threat sequence is being examined).

Special attention must be paid to the multiple-stone replies. While having extra stones on the board does not harm a player in free-style go-moku, it may harm a player in standard go-moku. To ensure that each global defensive strategy is found, we perform the db-search for global defensive strategies as a free-style go-moku search. Thus, a potential winning threat sequence in standard go-moku may be refuted through a sequence of defender threats containing overlines.

5.4 Applying pn-search

To apply pn-search to go-moku, we need to convert the go-moku game tree into an AND/OR tree. This is described in section 5.4.1. Furthermore, we describe the enhancements to basic pn-search adopted for our go-moku implementation in section 5.4.2.

5.4.1 Go-moku as an AND/OR tree

Pn-search (as described in chapter 2) is an AND/OR-tree algorithm. To apply it to go-moku, we represent positions where black is to move as OR nodes, and positions where white is to move as AND nodes. A win for black is represented by the value true, while a draw and a win for white are represented by the value false. Thus, proving the pn-search tree means that black can win in the root positions, while disproving the pn-search tree means that white can achieve at least a draw.

In each OR node, black is to move. As evaluation function at such a node, we apply db-search with black as attacker. If db-search finds a winning threat sequence, the node evaluates to true, otherwise to the value unknown. In each AND node, white is to move. The same procedure as in OR nodes is applied, this time with white as attacker. If a winning threat sequence is found, the node evaluates to false, otherwise to the value unknown. A node representing a position with all 225 squares occupied and neither player having a winning configuration, is a draw, and therefore obtains value false, without applying db-search.

5.4.2 Enhancements

The above description explains how standard pn-search is applied to go-moku. However, five enhancements have been added to speed up the search. The enhancements are discussed in this section.

Transpositions

A DAG is created instead of a tree, using the algorithm described in section 2.3.3. This ensures that if a position has already occurred in the DAG, or if a position is equivalent through automorphisms to another position in the DAG, the position is not investigated again. We test for the 8 standard automorphisms of a square board.

Restricting black's moves

In go-moku, the average branching factor is more than 200. Most of these moves are unrelated to the battle at the center of the board and should be ignored. However, since we want to *prove* the value of the root position, we cannot simply ignore moves using heuristic selection functions.

A large reduction of the branching factor at the OR nodes can be made, however. Since we want to prove a win for black in the root position, it is sufficient to prove for each internal OR node that (at least) *one* child leads to a win for black. For each internal AND node *all* children must be proved.

Using these properties, we may at each OR node restrict black to, say, the N most-promising children, using a heuristic ordering function. If in the restricted game tree a proof of black's win is found, the same proof is valid in the full game tree. In our investigations presented in section 5.5 we have restricted black in each OR node to the 10 most-promising children. Before the ordering function is applied, we first restrict the set of all legal moves to the set of moves which counter the threats of the opponent, as described in the next section.

The heuristic ordering function used is rather simple: each square is assigned 4 points for each three with a two-stone reply, 3 points for each three with a three-stone reply, 2 points for each broken three, 2 points for each *open two*, which is defined as two black stones in the center of an otherwise empty line of 6 and 1 point for each broken two, which is defined as two black stones with a one-square gap in the center three squares of an otherwise empty line of 7. Among all children, the 10 children with the highest score are selected.

No points are given for the creation of a four. Creating a four is often only a strong move if it stops a threat of the opponent, or if it creates a winning threat sequence. Since a node is only expanded if no winning threat sequence exists and it is ensured that we select the 10 best moves among the moves which counter the existing threats of the opponent, there was no need to assign any points for creating fours.

Clearly, a thorough analysis of the strategic knowledge of experts would have led to a more refined move-ordering function. As we show in section 5.5, the function described here was sufficient for our purposes.

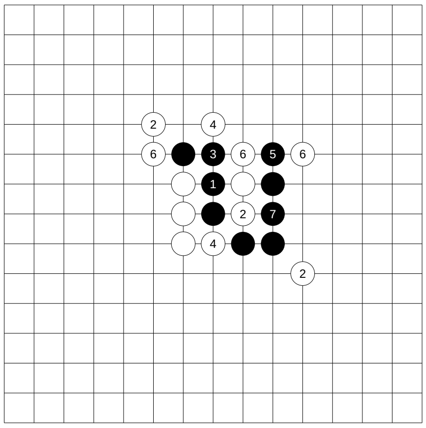


Figure 5.7: Black threatens to win by moves 1 through 7.

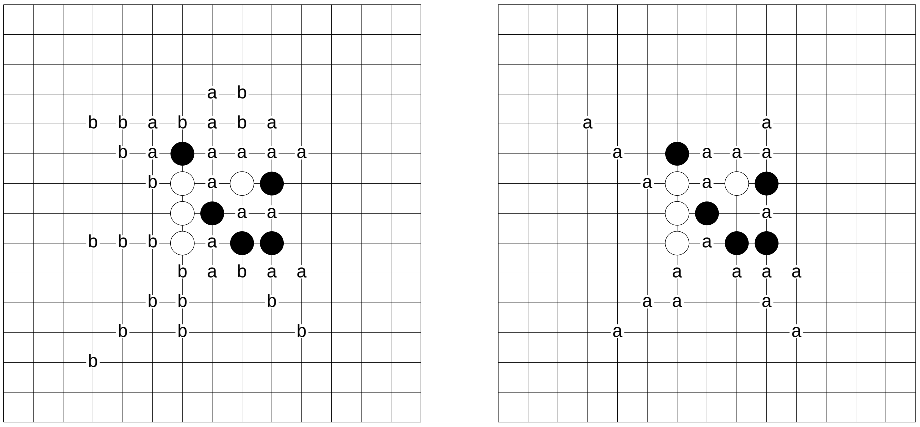
Related squares

As stated before, most of the approximately 200 legal moves per position are unrelated to the battle at the center of the board and should be ignored. Although we cannot ignore moves by white using heuristic selection functions, we may try to apply a winning threat sequence found as reply to one move to a large number of other moves. In this section we describe how this is done in a reliable way.

For each winning threat sequence of the attacker, we define the set of related squares as follows. An empty square c is *related* to a winning threat sequence in a given board position, if the threat sequence no longer wins, if c would have been occupied by the defender.

Before we use the notion of related squares, we introduce the term *implicit threat*, for any position where a player threatens to win through a winning threat sequence. In figure 5.7 black threatens to win through the threat sequence consisting of black 1 through 7. Therefore, the position is an implicit threat for black.

Now let us suppose that we have algorithms to determine whether a position is an implicit threat, and that we can determine for each winning threat sequence the set of related squares. Given a position with white to move, which is an implicit threat for black, we determine the set of squares related to the winning threat sequence. Then, it follows directly from the definition of related squares that we may restrict white to these related



(a) Superset of related squares (b) Related squares

Figure 5.8: Replies to the threat sequence of figure 5.7

squares.

Clearly, by determining implicit threats and sets of related squares in an efficient way, we could speed up our search. To determine an implicit threat, it suffices to make a null-move for the opponent (white in figure 5.7) and to apply db-search to find a winning threat sequence for black. Determining the exact set R of squares related to a winning threat sequence is computationally expensive. Instead, we determine a superset S of the set of related squares. The set consists of all squares meeting one of the following two conditions.

1. The square is in one of the lines of the threats in the winning threat sequence.

2. The square may be used in any counter threat by the opponent, in any of the global defensive strategy searches performed to investigate the winning threat sequence.

Using db-search S can be determined efficiently. Without proof we state that $R \subseteq S$. For empirical evidence of this claim we refer to section 5.5.4.

In figure 5.8a, we have shown the set S for the threat sequence of figure 5.7. The squares labeled a are part of the lines of the threats. The squares labeled b may, together with white stones on the board or the defensive moves in the threat sequence, form new defensive threats for white.

Iterated related squares

The related-squares concept can be used to even further reduce the set of white moves to be examined. After having determined the superset S of the set of related squares, an element s of S is selected. A white stone is placed at s, and the position is investigated with db-search. If no winning threat sequence is found, a child is added to the tree for s. Otherwise, the superset S_1 of squares related to the newly found winning threat sequence is determined. Only squares in $S \cap S_1$ need further be investigated, since all moves at other squares lead to a win through one of the two winning threat sequences found so far. This procedure is repeated until all moves have been examined.

In figure 5.8b we have marked the set of squares for which child nodes are grown. Of the 35 related squares of figure 5.8a (set S), only 19 squares (set R) remain in 5.8b.

The null-move heuristic and the related-squares heuristic are applied for both players in the pn-search DAG. For the attacker in the search (the player for which we select only 10 moves per node) we first determine the set of counter moves using the heuristics of this section, and then order the moves according to the move-ordering function of the previous section. Of course, if less than 10 counter moves exist, these are all selected.

The implicit-threat heuristic

The branching factor of go-moku is such that the search tree may become quickly intractable. To force black to select moves where white has a restricted number of moves, we evaluate a position which is not an implicit threat for black to false. Only early in the game tree (i.e., when there are less than 9 stones on the board), if no black move leads to an implicit threat, is the above restriction lifted.

We have found that no later than move 11 in the game, black can ensure that each move is an implicit threat. By enforcing this restriction, the size of the search tree is significantly reduced.

Heuristic (dis)proof number initialization

During our initial experiments, we have used the standard proof and disproof numbers initialization of 1 each. While studying the trees grown, it became apparent that pn-search tended to pursue some deep lines longer than desirable. This is mainly caused by continuously executing threats, without creating a potential for a winning threat sequence.

In qubic, as described in chapter 4, we chose to remove all threatening moves for the attacker from the search tree. We could safely do so, since our db-search implementation for qubic searched the full space of threatening sequences. The incompleteness of db-search in go-moku with respect to the space of all threat trees blocks a similar approach in go-moku. Instead, we have opted to attach a small penalty to all deep lines. At each frontier node the proof and disproof numbers are initialized to the number of full moves made from the root position. Thus, at depth d, the proof and disproof numbers are initialized to $1 + \lfloor d/2 \rfloor$.

This heuristic initialization ensures that forcing lines are not searched too deeply (before sufficient alternatives have been tried), without interfering with the essence of pn-search.

5.5 Solving go-moku

The program *Victoria* consists of the pn-search algorithm described in the previous section, using db-search as evaluation function. In this section we describe how *Victoria* solved both free-style go-moku and standard go-moku. First, we describe the I/O of *Victoria*. Second, it is explained how the game tree was split in several hundreds of subtrees. Third, we present statistics regarding the search process. Finally, we discuss the reliability of our results.

5.5.1 Victoria's I/O

The input to *Victoria* consists of (1) A go-moku position; (2) The game variant (free-style go-moku or standard go-moku); (3) The player to move; and (4) The maximum tree size for pn-search.

The output of *Victoria* consists of (1) the value upon termination of pn-search (true, false, unknown) (2) a database containing a record for each position in the solution tree. The database returned by *Victoria* is empty unless the value true was returned. For each record in the database representing a position with black to move, at least one child position will also be represented in the database. For each record in the database representing a position with white to move, only child positions are represented in the database in which black does not have a winning threat sequence.

The database created by *Victoria* served two purposes. First, the merged database of all subtrees investigated should provide us with a solution tree for the full go-moku game tree. Second, the databases created by solved subtrees were used as transposition table for pn-searches. We have seen several occasions where a search of several hundreds of thousands of nodes without transposition tables was reduced to a mere few thousands nodes, by hitting the database early during the search.

5.5.2 Subdividing the game tree

We have divided the go-moku game tree into several hundreds of smaller problems. The main reason for doing this is that the size of the go-moku game tree is such that we could not solve it through a single pn-search, due to the limits imposed on pn-search by the size of our computer's working storage.

We remark that by splitting the game tree into subtrees, part of the solution process has been performed by hand. Most of these moves have been made with the aid of Sakata and Ikawa (1981), while others where suggested by the proof and disproof numbers of failed pn-searches. The number of black moves selected by hand (several hundreds) is less than one percent of the total number of black moves in the solution tree (many tens of thousands).

5.5.3 Statistics

In this section we present the statistics of running pn-search on go-moku. As mentioned before, we have subdivided the problem in several hundreds of subtrees, each of which was individually solved. Since each completed search extended the database of solved positions, the number of positions searched partly depend on the order in which the subproblems were solved.

Execution time

Our calculations were performed in parallel on 11 SUN SPARCstations of the Vrije Universiteit in Amsterdam. Each machine was equipped with 64 or 128 megabytes internal memory, ensuring that pn-search trees of up to 1 million nodes would fit in internal memory, without slowing down the search by swapping to disk. The processor speed of the machines ranged from 16 to 28 MIPS. Our processes could only run outside office hours. As a result, sometimes processes which had not finished at 8am were killed, and had to be restarted at 6pm. Still, over 150 CPU hours per day were available for solving go-moku. In the figures below, we have not included CPU time spend on processes which were killed in the morning and restarted in the evening, nor have we included the CPU time spent on test runs during which we discovered bugs in our software (see also section 5.5.4). Thus, the time mentioned indicates the amount of time necessary to solve go-moku without interruptions, using the final version of *Victoria*.

Free-style go-moku was solved using 11.9 days of CPU time, while standard go-moku (thus banning wins through overlines) was solved with 15.1 days of CPU time.

Pn-search tree size

The summed size of all pn-search trees built during the calculations (again excluding terminated processes and runs of initial versions of the program) for free-style go-moku is 5.3 million. For standard go-moku, 6.3 million nodes were grown.

Comparing these figures with the execution time necessary for the solutions, we see that both variations ran at the speed of approximately 5 nodes per second. The rejection of potential winning lines involving overlines, resulted in the creation of a 20% larger search tree.

Db-search evaluations

For each internal node of the pn-search tree, 10-20 independent db-searches (excluding global defensive strategy searches) were performed on the average, resulting in, between 50 and 100 db-searches per CPU second. Multiplied by the total calculation time, the number of independent db-searches executed to solve go-moku lies between 50 million and 130 million.

Solution size

The solution tree found by *Victoria* for free-style go-moku is slightly smaller than the solution tree found for standard go-moku: 138,790 versus 153,284 database records. Comparing these numbers with the total size of the pn-search trees, we find that 1 out of every 40 nodes created participates in the solution. The deepest variation in both solution trees is 35 ply.

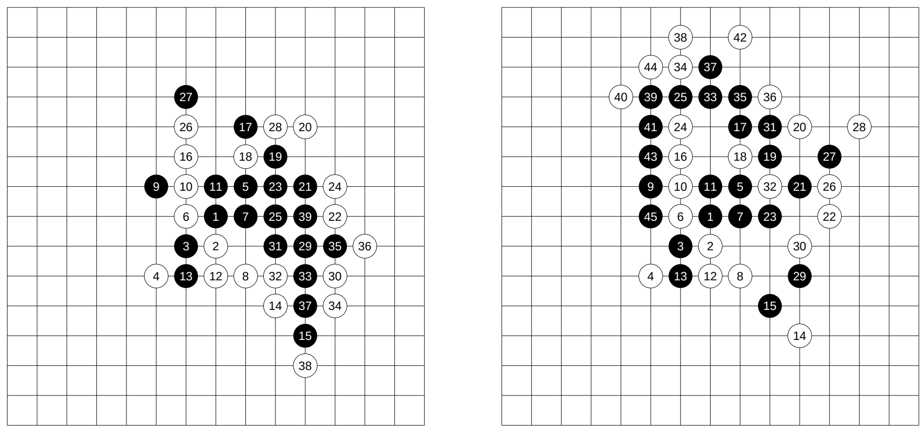
depth	free-style	standard	depth	free-style	standard
0	1	1	18	1351	1885
1	1	1	19	1094	1590
2	35	35	20	710	1125
3	35	35	21	594	954
4	7227	7242	22	408	641
5	6824	7251	23	327	506
6	20859	22749	24	193	296
7	20239	21078	25	154	241
8	20686	22056	26	85	159
9	20550	21898	27	74	128
10	8959	10015	28	40	67
11	8637	9570	29	35	54
12	5246	6015	30	7	19
13	4778	5492	31	7	18
14	2999	3663	32	1	8
15	2647	3282	33	1	6
16	2173	2810	34	1	1
17	1811	2392	35	1	1

Table 5.1: Nodes per tree depth in go-moku solutions.

In table 5.1 we have listed the number of nodes per depth for both solution trees. We remark that for each position with black to move, only one child position needs to be included. Due to transpositions, the number of nodes at each odd ply should therefore be less or equal to the number of nodes at the preceding even ply. The only exception in the table, ply 5 for standard go-moku, is caused by the fact that we have included several options for black for some opening positions in our set of positions to be checked by pn-search.

Deep winning lines

The combination of db-search and pn-search makes it difficult to determine the *maximin* of go-moku (i.e., the length of the game after optimal play of both players). Both db-search and pn-search do not aim at finding the shortest winning paths, while the longest path found by the combination of the algorithms may well be different from the game leading to the single



(a) Free-style go-moku (b) Standard go-moku

Figure 5.9: Deep variations

position at level 35 of the solution tree. Even though the games leading to positions at level 35 of the solution trees do not necessarily show optimal play of either side, we have depicted two of these games in figure 5.9.

5.5.4 Reliability

In section 4.5.4, we have explained the hazards of solving games through large computer programs. The same hazards mentioned there exist in go-moku in even greater form.

Our go-moku implementation consists of almost 20,000 lines of C-code. Approximately half is dedicated to the X-interface created by Loek Schoenmaker, while the other half consists of db-search, pn-search, database look-up and database creation, automorphism management, etc. Errors in programs this size are virtually unavoidable. Many errors have been created and corrected during implementation and testing of the program, but there is no guarantee that all bugs have been found.

A further source of error is the complexity of the calculation process. We used 11 SPARCstations in parallel to solve each of the several hundreds of subtrees. These 11 SPARCstations created their own databases when solving a position, while they all used one large database as transposition table. After solving a position, the transposition table should be extended with the newly created small databases. A locking mechanism was created to ensure that no databases would be corrupted. Still, computers going down at critical moments introduced the possibility that data would get lost. This, in fact, has happened during our calculations.

To ensure the completeness of the solution found, we have created a module which examines the final database created. For each position with black to move a successor position must be present in the database. For each position with white to move, for each legal move either a winning threat sequence must exist, or the successor position must be present in the database. The only common element with the solving process is db-search. Thus, an error in db-search may go unnoticed, while all other parts, including pn-search and the related-squares generator, are eliminated from the checking process. Using the database checking module, we have both located missing database parts, due to computers failing at critical moments and human error, and have found an error in our related-squares generator. The final investigations, however, for both the free-style go-moku and standard go-moku variants were successful.

The correctness of our db-search implementation is based on meticulously testing all possible types of counterplay, including intricate ways in which the opponent forces the attacker, after a sequence of fours, to occupy an extension square of a threat in the threat sequence. After the final database creation, which was checked and accepted by the database-checking module, no errors have been found in this part of the program. Therefore, go-moku should be considered a solved game.

Chapter 6

Which Games Will Survive?

6.1 Scope

In chapters 2 and 3, we presented two new search techniques which have been applied to qubic and go-moku in chapters 4 and 5, thereby partly answering our first research question (see section 1.4). In this chapter, we broaden our scope to all three research questions and the problem statement. The chapter consists of four parts.

First, in section 6.2, we define four properties of games. These are *perfect information*, *convergence*, *sudden death*, and *complexity*.

Second, in section 6.3, we discuss four aspects of each of the games of the Olympic List.

1. The relation between the game and the four game properties introduced in section 6.2.

2. The state of the art in game-playing programs.

3. Techniques currently applied.

4. Obstacles to progress.

Third, in section 6.4 we review our three research questions on the basis of the discussion of individual games presented in section 6.3, leading to a review of the problem statement.

Finally, in section 6.5, we speculate about the future playing strength of computer game playing programs, as well as of the future of thinking games in our society.

For the rules of the games discussed in this chapter, we refer to Levy and Beal (1989), Levy and Beal (1991) and Van den Herik and Allis (1992).

6.2 Game properties

In this section we define four properties of games, viz. *perfect information* (section 6.2.1), *convergence* (section 6.2.2), *sudden death* (section 6.2.3) and *complexity* (section 6.2.4).

6.2.1 Perfect information

The perfect-information property divides the set of games into two disjoint subsets: the set of *perfect-information games* and the set of *imperfect-information games*. In a perfect-information game, all players, at any time during the game, have access to all information defining the game state and its possible continuations. Any game which is not a perfect-information game is defined to be an imperfect-information game.

For example, chess is a perfect-information game. Relevant information defining the game state in chess includes: (1) the configuration of chess pieces on the board; (2) the number of moves made since a pawn was moved, or a piece has been captured; (3) the en-passant capturing opportunities in the current game state; (4) the castling options left to both players; and (5) previous configurations with their en-passant capturing opportunities and castling options. The information described here allows each player to determine the game state and its possible continuations, including en-passant capturing moves, castling moves, repetition of positions, and the status with respect to N-move rules. In practice, a player needs only three pieces of information: (1) the configuration of chess pieces; (2) the game score, i.e., all moves played since the start of the game; and (3) the official rules of chess. The combination of these three pieces of information allows a player to deduce all necessary information during a game.

Bridge is an example of an imperfect-information game. During the bidding phase of bridge, each player sees only her own cards, leaving her unaware of the distribution of the remaining 39 cards over her partner and her opponents. During the playing phase, each player sees her cards, those of the dummy and the cards already played, still leaving her unaware of the distribution of the remaining cards over the undisclosed hands.

Optimal play in a perfect-information game always consists of a *pure strategy*, while in imperfect-information games optimal play may require a *mixed strategy.* In a pure strategy, for each game state a single move can be determined, which leads to the game-theoretic value of the position. In a mixed strategy, optimal play requires a player to play a move i with probability p_i, while at least two such p_i are non-zero. For a discussion of pure and mixed strategies, we refer to von Neumann and Morgenstern (1944).

6.2.2 Convergence

The convergence property labels games as either *converging*, *diverging* or *unchangeable.* Before we can define these classifications, we introduce *conversions* in definition 6.1.

Definition 6.1 *A move M from state A to state B is a* conversion*, if no configuration of pieces which could have occurred in any game leading to the configuration of pieces in A, can occur in a game continuing from state B.*

Examples of conversions in chess are moving a pawn, or capturing a piece. In checkers, any move except for a non-capture move by a king is a conversion.

For most games, the main conversions involve the addition (e.g., connect-four, go-moku, qubic and othello) or removal (e.g., chess, checkers, bridge) of pieces from play. We may divide the state space of all legal positions of a game into disjoint classes, where each class contains all positions with the same number of pieces on the board. Let us define a directed graph G in which each class is a node, and an arc exists between class A and class B if and only if a position P exists in A such that a move exists from P which leads to a position Q in B. We can now define convergence using this notion of classes of positions. A game *converges* if for the majority of edges from A to B in G, the cardinality of A is larger than the cardinality of B. A game *diverges* if for the majority of edges from A to B in G, the cardinality of B is larger than the cardinality of A. A game is *unchangeable* if the game does not have conversions, or if it neither converges nor diverges.

An example of a converging game is checkers. The initial position in checkers consists of 24 men, while during the game the number of men decreases. After the first few captures, the number of legal checkers positions decreases as the number of pieces on the board decreases.

An example of a diverging game is othello. Each move in othello adds a piece to the board. Except for the endgame, the number of legal positions increases as the number of stones on the board increases.

An example of an unchangeable game is shogi. Although shogi contains captures, there are no conversions in shogi. Captured pieces may be brought into play again by the player who captured the piece. As a result, the total number of pieces participating in a shogi game does not increase or decrease. Thus, shogi is an unchangeable game.

The relevance of the convergence property is that for converging games endgame databases (Thompson, 1986) can be created, while this is generally unfeasible for diverging or unchangeable games.

6.2.3 Sudden death

The sudden-death property labels games as either *sudden-death* or *fixed-termination*. A sudden-death game may end abruptly by the creation of one of a prespecified set of patterns. A fixed-termination game lacks sudden-death patterns.

An example of a sudden-death game is go-moku: the game is terminated if one of the players has created a line of five stones in her color. Sudden-death games need not always terminate through the creation of a sudden-death pattern: go-moku is declared a draw when all 225 squares have been occupied without either player creating a winning pattern.

An example of a fixed-termination game is othello. Othello lasts until both players run out of moves or one of the players has no discs left on the board. In practice, games last between 55 and 60 moves. Even though a game might be decided within 15 moves by one player capturing all the discs of the opponent, such an anomaly is only of marginal relevance.

The sudden-death property often is an important property in restricting the search tree of a game. For games of high complexity (see section 6.2.4) the sudden-death element in combination with a clear advantage for one of the players may be the main property that allows the game to be solved. Examples are qubic and go-moku (both sudden-death games) described in chapters 4 and 5.

6.2.4 Complexity

The property *complexity* in relation to games is used to denote two different measures, which we name the *state-space complexity* and the *game-tree complexity*.

State-space complexity

The *state-space complexity* of a game is defined as the number of *legal* game positions reachable from the initial position of the game. While calculating the exact state-space complexity of games such as chess is hardly feasible, we present a method for calculating an approximation, using tic-tac-toe as an example.

A crude approximation to tic-tac-toe's state-space complexity is obtained through the notion that each of the nine squares can be occupied by cross, nought, or be empty. Thus, an upper bound to the state-space complexity is $3^9 = 19,683$. A sharper upper bound is obtained by noting that the number of crosses should equal the number of noughts, or exceed it by one. This results in an upper bound of $6,046$. The exact state-space complexity, however, is obtained by observing that a position is illegal if a move has been added *after* a player has created three-in-row. Thus, positions containing a line of three noughts with nought to move, or a line of three crosses with cross to move must be excluded. The resulting $5,478$ legal positions determine the state-space complexity of tic-tac-toe. The definition of the state-space complexity could be refined so that symmetrically equivalent positions are counted only once. We refrain from such a refinement.

Let us assume that we have established a superset of all legal positions of the game and the cardinality of that superset. Let us also assume that for each individual position of the superset we have an evaluation function which determines whether the position is legal. Using the combination of these two and a Monte-Carlo simulation, we may obtain an estimate of the true state-space complexity. We performed 10 Monte-Carlo simulations, with a thousand samples per simulation, chosen from the superset of 3^9 configurations mentioned above. For each simulation we determined the fraction of the positions which were legal. Multiplication of this fraction by the size of the superset, 3^9, gave an estimated state-space complexity in our 10 simulations ranging from $4,920$ to $5,983$ with an average of $5,479$, surprisingly close to the true state-space complexity.

The main application of the state-space complexity of a game is that it provides a bound to the complexity of games which can be solved through complete enumeration. With today's (1994) technology, where computer networks have access to Gigabytes of disk storage, the boundary of solvability by exhaustive enumeration lies at a state-space complexity of approximately 10^{11}.

Game-tree complexity

Before we are able to define the *game-tree complexity* of a game, two auxiliary definitions are needed.

Definition 6.2 *The* solution depth *of a node J is the minimal depth (in ply) of a full-width search sufficient to determine the game-theoretic value of J.*

According to definition 6.2, the solution depth of a mate-in-n position in chess, $n \geq 1$, is $2n - 1$ ply.

Definition 6.3 *The* solution search tree *of a node J is the full-width search tree with a depth equal to the solution depth of J.*

As an example we consider a chess position J with white to move. White has 30 legal moves. For simplicity's sake, we assume that after each legal white move, black has 20 legal moves of which at least one mates white. Then, the solution search tree of J consists of J, the 30 children of J, and the 600 grandchildren of J.

Definition 6.4 *The* game-tree complexity *of a game is the number of leaf nodes in the solution search tree of the initial position(s) of the game.*

If J were the initial position of a game, its game-tree complexity would be 600.

While calculation of the exact game-tree complexity of games such as chess is hardly feasible, we can calculate a crude approximation as follows. Using tournament games, we can observe the average game length. Also, we may determine the average branching factor, either as a constant, or as a function of the depth in the game tree. The game-tree complexity can be approximated by the number of leaf nodes of the search tree with as depth the average game length (in ply), and as branching factor the average branching factor (per depth).

For instance, in tic-tac-toe, the average game length is close to nine ply, since most games end in a draw, which always takes exactly nine half-moves. The branching factor at level i in the game tree equals $9 - i$. Thus, the minimax search tree with depth 9 and branching factor $9 - i$ at level i consists of $9! = 362880$ terminal nodes, which is an estimate of the game-tree complexity of tic-tac-toe. Note that the game-tree complexity of a game may be larger than the state-space complexity, as the same position may occur at several different places in the game tree.

The game-tree complexity is an estimate of the size of a minimax search tree which must be built to solve the game. Thus, using optimally-ordered α-β search, we may expect to search a number of positions in the order of the square root of the game-tree complexity (Knuth and Moore, 1975).

As a guide to the perplexed, anticipating results duely credited in the following section, we present a graphical overview of the two complexities we

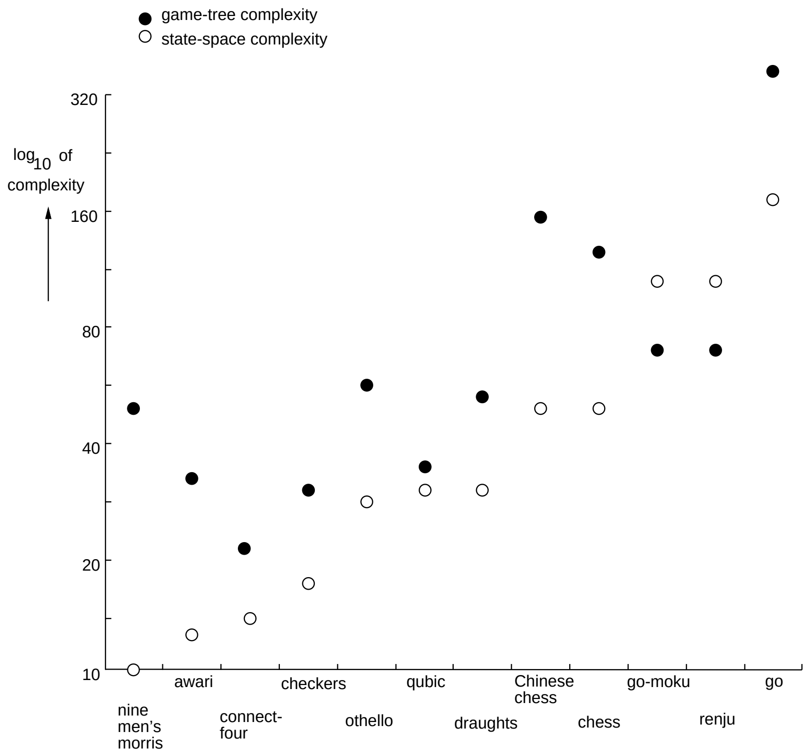


Figure 6.1: Estimated game complexities.

distinguish in figure 6.1. For credits and sources see the discussions of the individual games.

6.3 The games of the Olympic List

In this section we discuss each of the games of the Olympic List individually. For each game, we describe (1) its properties, as introduced in section 6.2; (2) the currently strongest computer programs; (3) the techniques applied in these programs; and (4) the obstacles to progress in the game.

We have ordered the games of the Olympic List as follows. First, we discuss four solved games (qubic, connect-four, go-moku and nine men's morris) in the order in which they were solved. Second, we discuss the eight unsolved perfect-information games, in an order depending on the strengths of the currently strongest game-playing program: (1) stronger than the current world champion (awari and othello) (2) Grand Master strength or stronger (checkers, draughts and chess) (3) below Grand Master strength (Chinese chess, renju and go). Third, we discuss the three imperfect-information games of the Olympic List (scrabble, backgammon and bridge).

6.3.1 Qubic

Game properties

Qubic is a diverging, perfect-information game with sudden death. An upper bound to the state-space complexity of qubic is $3^{64} \approx 10^{30}$. To estimate the game-tree complexity, we assume an average game length of 20 ply. With $64 - i$ legal moves in a position at ply i, the game-tree complexity of qubic is approximately $\frac{64!}{44!} \approx 10^{34}$.

The state of the art

Qubic was the first game of the Olympic List to be solved. It was proved that the game is a win for the player to move first (Patashnik, 1980). The main game property responsible for qubic being solvable is sudden-death. For details on the solution of qubic, we refer to chapter 4.

Techniques currently applied

Qubic was solved by Patashnik using a standard α-β search for determining the existence of winning threat sequences. All non-forced moves leading to the solution were made by hand, using expert knowledge. Qubic has been solved again using db-search for determining the existence of winning threat sequences, and pn-search for making the non-forced moves, as described in chapter 4.

Obstacles to progress

It could be argued that qubic provides additional challenges beyond solving the game. For instance, one might want to determine the game-theoretic value of *every* legal position, or determine the shortest winning threat sequence from each position. However, we believe that with respect to human performance on qubic, all interesting problems within qubic have been solved. During the solution processes, no obstacles to progress have been discovered.

6.3.2 Connect-Four

Game properties

Connect-four is a diverging, perfect-information game with sudden death. Although at first sight the sudden death in connect-four may seem as important as in qubic, most games in connect-four are decided between moves 37 and 42, i.e., while filling the last column of the board.

The state-space complexity of connect-four has been estimated at 10^{14} (Allis, 1988). With an estimated average game length of 36 ply, and an average branching factor of 4, the game-tree complexity of connect-four is approximately $4^{36} \approx 10^{21}$.

The state of the art

In September 1988, James Allen determined the game-theoretic value through a brute-force search (Allen, 1989): a win for the player to move first. A few weeks later, in October 1988, connect-four was solved through a knowledge-based approach, resulting in the tournament program VICTOR (Allis, 1988; Uiterwijk *et al.*, 1989a; Uiterwijk *et al.*, 1989b). Recently John Tromp has calculated the game-theoretic value for all 8-ply connect-four positions (Tromp, 1993).

Techniques currently applied

Both Allen and Tromp used a sophisticated implementation of α-β search. While Allen spent 300 hours of CPU time to determine the game-theoretic value of the position after 1. $d1$, Tromp's calculations took some 40,000 hours CPU time for his (vastly) more complex task. Our knowledge-based solution initially took 350 hours of CPU time. However, adding a knowledge rule in combination with changing the search algorithm to pn-search has resulted in a program which solves connect-four in less than 25 CPU hours. All these experiments were performed on comparable hardware.

Obstacles to progress

The current version of VICTOR, in combination with the 8-ply database created by Tromp, can be used to determine the game-theoretic value of almost any connect-four position within minutes. Furthermore, VICTOR's knowledge-based component is able to provide us with an explanation *why* a position is won. Therefore, we believe that no challenges remain within connect-four and no obstacles to progress have been discovered.

6.3.3 Go-moku

Game properties

Go-moku is a diverging, perfect-information game with sudden death. An upper bound to the state-space complexity is $3^{225} \approx 10^{105}$. To estimate the game-tree complexity, we assume an average game length of 30 ply. With $225 - i$ legal moves in a position at ply i, the game-tree complexity of go-moku is approximately 10^{70}. For the professional variant of go-moku, with opening restrictions for black, the average game length will be somewhat larger, resulting in a higher game-tree complexity.

The state of-the-art

As described in chapter 5, two variants of go-moku without opening restrictions have been solved in August 1992, proving that the game-theoretic value is a win for the player to move first. The current computer go-moku world champion (according to the rules of professional go-moku) is the program *Vertex* written by Shaposhnikov (Uiterwijk, 1992a). It is unclear at what performance level *Vertex* plays in relation to the strongest human players.

Techniques currently applied

As described in chapter 5, the two variants of go-moku without opening restrictions were solved using a combination of db-search and pn-search. world champion *Vertex* is based on standard game-tree search techniques: a fixed-depth (16-ply) α-β search for the most-promising 14 moves in each position. *Vertex* has been provided with expert pattern knowledge and opening knowledge of two-fold world correspondence Renju champion Nosovsky .

Obstacles to progress

During the solution process of go-moku, it became apparent that through its tactical knowledge *Victoria* was able to suggest strong positional moves in many positions. In other words, many positionally strong moves could be explained through tactical calculations. We believe that a combination of db-search and pn-search, without the multi-move reply and other efficiency measures, can be implemented to outperform all human players in the search for deep winning threat trees. With similar positional benefits as encountered during the solution process of the free-style and standard go-moku, we conjecture that the best human players can be defeated at any variant of go-moku. It is also possible that standard techniques as applied in *Vertex* would prove sufficient for the task. Therefore, we conclude that no obstacles have been discovered in go-moku.

6.3.4 Nine men's morris

Game properties

Nine men's morris is a converging, perfect-information game. The game has a sudden-death element: if a player is unable to make a move, she loses. Even though this plays a role in practice, its influence on the game is much less than that of the main feature: closing mills and thereby capturing men of the opponent. Therefore, it seems more appropriate to classify nine men's morris as a fixed-termination game than as a sudded-death game.

The state-space complexity of nine men's morris, calculated by Gasser (1990), is the smallest of all games of the Olympic List: 10^{10}. Nine men's morris' game-tree complexity is much larger. During the opening phase of the game, the branching factor is 16 on the average In the middle game and end game, the branching factor ranges from 1 to over 50, resulting in our conservative estimate of the average branching factor of 10. Setting the average game length at 50 ply (again a conservative estimate), the game-tree complexity of nine men's morris is calculated to be at least 10^{50}.

The state of the art

Nine men's morris has been solved in October 1993 by Ralph Gasser, proving that the game-theoretic value is a draw. In the years preceding the solution of the game, the program *Bushy*, also by Gasser, has shown itself to be stronger than the best human players, as illustrated by defeating the British champion by 5 to 1 in an exhibition match during the 2nd Computer Olympiad (Levy and Beal, 1991).

Techniques currently applied

Nine men's morris has been solved through the creation of databases by retrograde analysis for all positions which may occur during the middle game or endgame (Gasser, 1993). For the opening phase, which takes exactly 18 ply, a forward search using α-β search was applied.

Obstacles to progress

During the solution of nine men's morris through the application of standard search techniques, no obstacles to progress on the game have been discovered.

6.3.5 Awari

Game properties

Awari is a converging, perfect-information game, with fixed termination. Only in rare circumstances may a player run out of moves early in the game, terminating it. The chances of this happening, however, are quite remote, which is why awari is not a sudden-death game.

The state-space complexity of awari is calculated by Allis *et al.* (1991c) to be 10^{12}. The game-tree complexity of awari, based on an average branching factor of 3.5 and an average game length of 60 ply, is estimated at 10^{32}.

The state of the art

Although lack of official human awari champions makes it difficult to prove, empirical evidence suggests that today's strongest awari program, *Lithidion* (Allis *et al.*, 1991c), outperforms the strongest human players. Lithidion has lost games against human opponents, but in each of these cases the game revealed a serious bug in the program. All other games against human opponents were won, most by large margins.

We believe that awari will be the next game to be solved. Its state-space complexity is such that, using 2 terabyte of disk space, awari can be solved. It is only because solving awari is not a high-priority project, that it will take several years and advances in technology before the hardware becomes available to solve the game through full enumeration.

A similar approach as applied to nine men's morris, i.e. endgame-database construction in combination with a forward search, may reduce the memory requirements for solving awari.

Techniques currently applied

For a detailed description of the techniques applied to today's strongest awari programs, we refer to section 2.4.3.

Obstacles to progress

Given the current strength of awari programs, and the impending solution of the game, no obstacles have been found on awari.

6.3.6 Othello

Game properties

Othello is a diverging, perfect-information game with fixed termination. The state-space complexity of othello has an upper bound of $3^{64} \approx 10^{30}$. Several legality tests, such as that the four center squares should not be empty and that the occupied squares must form a connected set, reduced the upper bound in a Monte-Carlo analysis to approximately 10^{28}.

To calculate the game-tree complexity of othello, we assume an average game length of 58 ply. With a conservative estimate of the average number of moves per position set at 10, we obtain a game-tree complexity of 10^{58}.

The state of the art

Othello programs have played at the level of the human world champion since 1980. In that year the program *The Moor* won a game against the reigning world champion. Since then, programs have continued to improve. Currently, rating lists for othello players show that several programs clearly exceed the strongest human players in playing strength. Today's strongest program is *Logistello* by Michael Buro, which, among other tournaments, has won the 1st Paderborn othello tournament.

Techniques currently applied

All strong othello programs are based on standard game-playing techniques: (1) a deep α-β search; (2) a large opening database; (3) an endgame search determining the outcome of a game after approximately 36 ply; and (4) a finely-tuned evaluation function.

The chances that othello will be solved in the near future are extremely remote. The state-space complexity rules out the option of full enumeration, while the game-tree complexity renders a full-depth forward search impossible. The diverging nature of othello makes creation of endgame databases unfeasible. Finally, the property of fixed termination of othello renders solving the game in similar fashion to the solution of qubic and go-moku impossible. Only if a so far unidentified structure in the game is discovered, resulting in knowledge rules which prove the value of nodes early in the game tree, may othello be solved in the coming decades.

Obstacles to progress

The strongest othello programs have already surpassed their human opponents. Even though solving the game is out of reach, human players do not possess knowledge or skill not shown by their artificial opponents. We conclude that no obstacles have been found in the research on othello.

6.3.7 Checkers

Game properties

Checkers is a converging, perfect-information game with fixed termination. In checkers a game is lost by a player who runs out of moves. Although in exceptional cases this may happen while both players still have most of their pieces, in practice to win a game, (almost) all of the opponent's pieces must be captured. The state-space complexity of checkers is estimated at 10^{18} (Schaeffer *et al.*, 1991). The average branching factor is surprisingly low: 2.8, which is mostly due to the forced-capture rule (Schaeffer, 1993a). With an estimated average game length of 70 ply, we obtain a game-tree complexity of 10^{31}.

The state of the art

As stated in section 1.1, Samuel's learning checkers program has, at least by some, been wrongfully credited with solving the game, which has clouded the history of the performance of checkers programs. Recent efforts by Schaeffer *et al.* (1992) have led to the development of a true world-championship level checkers program, named *Chinook*. *Chinook* has challenged the human world champion, Marion Tinsley, for his title. In a rather close match, 4 wins, 2 losses and 33 draws, Tinsley successfully defended his title.

A rematch is scheduled for August 1994 in Boston. With the extra efforts spent on *Chinook* (see below), it is not unlikely that 1994 will see a computer program become the strongest checkers entity in the world.

Techniques currently applied

Chinook consists of (1) a deep α-β searcher (averaging approximately 20 ply); (2) a fine-tuned evaluation function; (3) a large, man-made, computer checked opening book; and (3) endgame databases comprising all endgame positions of 7 pieces or less, and all endgame positions of 4 pieces against 4.

The influence of the endgame databases in checkers should not be underestimated. Due to forced captures in checkers, removing 16 men off the board may happen rather quickly.

With regard to solving checkers, we mention that full enumeration of the game is ruled out by the size of the state-space complexity. A complete forward search, even if the game-tree is perfectly ordered, is also out of reach of current technology. However, convergence in checkers has allowed the creation of large endgame databases, which decrease the size of the game-tree significantly. Therefore, we do not rule out that the combination of forward search (either pn-search or α-β search) and endgame databases may prove sufficient to solve (some of the openings of) checkers, as stated by Schaeffer (1993a).

Obstacles to progress

While *Chinook*'s strength is its deep tactical searches, combined with perfect-endgame knowledge, its main weakness is that the value of each pattern not available in the evaluation function must be compensated for by search. In contrast, Tinsley's pattern knowledge is such, that he knows of many positions for which a search of 50 or more plies is necessary to reveal the value of the position. Each of such patterns corresponds to a weakness of the program with respect to human players.

Although *Chinook*'s tactical and endgame ability make up for most of the lack of pattern knowledge, it reveals traces of an obstacle to progress in checkers: the inability to gain *experience* from previous plays. The suitability of checkers to alternative approaches, such as the brute-force approach applied by *Chinook* shows that this experience obstacle has not prevented checkers programs from successfully challenging the strongest human players.

6.3.8 Draughts

Game properties

Draughts is a converging, perfect-information game with fixed termination, in many ways similar to checkers. The state-space complexity of draughts is significantly larger than that of checkers, and we have calculated an upper bound of 10^{30}. The game-tree complexity of draughts is also larger than that of checkers. Conservatively estimating the average branching factor at 4, and the average game length at 90 ply, we obtain an estimated game-tree complexity of $4^{90} \approx 10^{54}$.

The state of the art

The strongest draughts program is *Truus* written by Stef Keetman (Keetman, 1993). *Truus*' current level of play at tournament speed is ranked around the 40th position in the world. In speed draughts, *Truus* has beaten reigning world champion Alexei Tsjizjow once, and reached the 9th position in a tournament entered by almost all strong human players.

Currently, Keetman works towards the goal of creating a tournament program able to defeat the human world champion. These efforts may improve *Truus*' level of play in the near future.

Techniques currently applied

Truus consists of (1) a deep α-β searcher (averaging a nominal depth of approximately 10 ply); (2) a fine-tuned evaluation function; (3) a large, man-made, computer-checked opening book; and (4) a set of about 1,000 tactical patterns, which *Truus* learned through automatic generalization.

According to its author, *Truus*' undefeated record amongst draughts programs since 1990, is mostly due to its learning of tactical patterns (Keetman, 1993). In the near future, *Truus*' learning abilities will be extended to positional patterns, which have so far been hand-coded by the author.

The large state-space complexity, in combination with the large game-tree complexity, make draughts unsolvable in the foreseeable future.

Obstacles to progress

Truus' strength is mostly based on its knowledge of tactical patterns and deep tactical searches. Although it has been argued by Keetman (1993) that tactical knowledge in draughts enhances positional play, positional knowledge is *Truus*' main weakness in comparison with human experts. Like in checkers, each pattern not available in the evaluation function must be compensated for by search revealing similar traces of an obstacle to progress as in checkers: the inability to gain *experience* while playing the game.

6.3.9 Chess

Game properties

Chess is a converging, perfect-information game with sudden-death. While convergence and sudden-death are major contributors to high-level play in games like qubic, go-moku and checkers, there is only a slight influence on tournament play in chess. Convergence in chess is slow, and a large majority of all chess games are decided long before endgame databases come into play. In chess practice the subgoal of obtaining material superiority often dominates the sudden-death goal of checkmate. Thus, both convergence and sudden-death are less pronounced in chess, than in games like checkers and draughts, or qubic and go-moku, respectively.

In our calculation of the state-space complexity of chess, we have included all states obtained through various minor promotions. Using rules to determine the number of possible promotions, given the number of pieces and pawns captured by either side, an upper bound of $5 \cdot 10^{52}$ was calculated. Not all of these positions will be legal, due to the king of the player who just moved being in check, or due to the position being unreachable through a series of legal moves. Therefore, we assume the true state-space complexity to be close to 10^{50}. A state-space complexity of 10^{43}, as mentioned by various authors (Schaeffer *et al.*, 1991), is in our opinion too low an estimate.

The game-tree complexity of chess, 10^{123} is based on an average branching factor of 35 and an average game length of 80 ply.

The state of the art

Today's strongest chess program is *Deep Thought* (Hsu, 1990). Its estimated ELO rating of 2550 ranks it between positions 100 and 150 on the world rating list. Current efforts to create *Deep Blue*, a parallel program consisting of 1000 *Deep Thoughts*, aim at surpassing the human world champion.

Possibly as early as 1994 a new match with today's strongest computer chess player and one of the reigning world champions, Garry Kasparov, will be held. So far, all previous games between Kasparov and Deep Thought have been won by the human Grand Master (Van den Herik and Herschberg, 1989).

Techniques currently applied

Most AI research on games has focused on chess. Several different approaches have been tried, ranging from purely knowledge-based (Reznitsky and

Chudakoff, 1990) to purely brute-force (Hsu, 1990).

Deep Thought consists of (1) a deep α-β searcher (averaging approximately 10 ply); (2) a fine-tuned evaluation function; (3) a move-generator embedded in hardware; and (4) a large, man-made, computer-checked opening book.

Even though deep searches have had a large impact on the strength of today's chess programs, we should not ignore the contribution of the improved evaluation functions developed alongside the deeper searches. A strong example is Ed Schröder's 1992 world champion program, which compensates for one or more plies of search through a highly sophisticated evaluation function, manually fine-tuned through years of development and testing.

Obstacles to progress

In chess, just as in checkers, many strategic concepts known to human Grand Masters are based on gains achieved after a large number of moves. For many of these patterns, programs cannot compensate for their lack of knowledge by simply searching a few ply deeper.

Again, but more clearly than in checkers and draughts, the contours of lack of *experience* as obstacle to progress in chess becomes visible.

The extent to which this obstacle prevents programs from attaining dominance over their human counterparts through brute-force alone is unclear. While some believe that it will still take decades before computers will defeat the human world champion, others have stated that this event will occur before the year 2000 (Van den Herik, 1983).

6.3.10 Chinese chess

Chinese chess is similar to (Western) chess in many ways: (1) it is a converging, perfect-information game with sudden-death; (2) its state-space complexity, at 10^{48}, is similar to that of chess (at 10^{50}). (3) the approaches to creating computer programs for playing Chinese chess have been similar to that of chess. Its game-tree complexity, estimated at $38^{95} \approx 10^{150}$ (Tsao *et al.*, 1991), is somewhat larger than the game-tree complexity of chess, at 10^{123}.

In our opinion, the main reason why Chinese chess programs fall somewhat behind in their challenge of the stronger human players is the lesser amount of effort invested in Chinese-chess research.

6.3.11 Renju

Game properties

Renju (see also section 5.2.1) is a variant of go-moku, played by professional players. It is a diverging, perfect-information game with sudden death. Its state-space complexity and game-tree complexity are similar to that of go-moku.

The state of the art

In its purest form, without special opening rules restricting black (see section 5.2.1), we believe renju can be proved a first-player win, in the same way as go-moku has been solved. The main extension needed consists of the definition of special types of threats white can create, using squares forbidden to black (squares where black would create a double three, a double four or an overline). Using these extra threat types, white may be able to counter threat sequences which cannot be countered otherwise. Furthermore, potential winning threat sequences by black must be checked for the occupation by black of forbidden squares. Despite the extra complications in the program, and the somewhat enlarged solution complexity, we believe that renju should be solvable in at most ten times the effort required for the go-moku solution.

Professional renju, as described in section 5.2.1, is a game with virtually equal chances for both players. As go-moku could only be solved through black's opening advantage, we believe that professional renju will be unsolvable in the foreseeable future. Today's strongest renju programs, such as *Vertex* by Shaposhnikov, are estimated to play at a level of 2 or 3 kyu (Ohta, 1993), which is the level of intermediate to strong club players.

Techniques currently applied

World champion *Vertex* is based on standard game-tree search techniques: a fixed-depth (16 ply) α-β search for the most-promising 14 moves in each position. *Vertex* has been provided with expert pattern knowledge and opening knowledge by two-fold world correspondence Renju champion Nosovsky.

Obstacles to progress

In go-moku we have seen that the availability of a strong tactical module allows a program to determine positionally strong moves: through refutation of positionally weak moves by tactically forced sequences, the positionally strong moves automatically emerge as the only options. In renju, a strong tactical module can be created using the same principles as applied to go-moku, albeit somewhat more complex. So far, it has not been shown that it is necessary to master deep positional knowledge as applied by human master players. In other words, so far no obstacles to progress in renju have been discovered.

6.3.12 Go

Game properties

Go is a diverging, perfect-information game with fixed termination, We remark that, in theory, go should be regarded as an *unchangeable* game, instead of a diverging game, as any legal state can be reached from any other legal state, if both players cooperate to this end. However, in practice, the board is slowly filled with stones until the board is divided into territories for both players. For practical purposes, therefore, go is a diverging game..

Go's state-space complexity, bounded by $3^{361} \approx 10^{172}$, is far larger than that of any of the other perfect-information games of the Olympic List. Its game-tree complexity, with an average branching factor of 250, and average game length of 150 ply, is approximately 10^{360}.

The state of the art

The strongest programs, such as *Goliath* by Mark Boon and *Go-Intellect* by Ken Chen, have achieved ratings roughly between 8 and 10 kyu (Boon, 1991; Chen, 1992), a level equivalent to weak club players. The low playing strength in comparison to human players cannot be attributed to the lack of interest by strong players or by financiers: both Mark Boon and Ken Chen have a go-rating of 5 dan, while large sums of money can be won by the strongest go programs.

The explanation for the low playing strength of current go programs is found in the nature of the game. While the potential branching factor averages 250, human players only consider a small number of these, through extensive knowledge of patterns relevant to go. Similarly, while evaluating a position, humans determine the strengths and weaknesses of groups on the board with pattern knowledge. Thus, either programs must obtain pattern knowledge similar to human experts, or compensate for a lack of such knowledge through search or other means.

Techniques currently applied

We restrict our description to two-fold computer world champion *Goliath*, written by Mark Boon. *Goliath*'s main strength is its evaluation function. As part of the evaluation function, heuristics determine the value of groups under attack, as well as the result of many forcing sequences, without having to analyze these sequences in detail. The evaluation function is used in a selective search, where moves are generated using pattern knowledge indicating candidate moves.

A future version of *Goliath*, aimed at achieving a playing level of 5 kyu, is currently being developed.

Obstacles to progress

The main progress made by human go novices can be attributed to learning important patterns, in go terminology called *good shape* and *bad shape*. Furthermore, after each life-and-death attack, their pattern knowledge regarding the liveliness of each group on the go board is enhanced. After playing a few hundred games, a novice go player will have acquired sufficient pattern knowledge to defeat today's strongest go programs.

While lack of pattern knowledge is not unique to go (cf. checkers, draughts and chess), the main reason why it stands out in go is that deep search fails to mask the lack of pattern knowledge. As a result, in go, the *experience* obstacle is clearly visible.

6.3.13 Scrabble

Game properties

Scrabble is a diverging imperfect-information game with fixed termination. The imperfect information in scrabble consists of not knowing the contents of the rack of the opponent and of the chance element in drawing tiles from the heap.

The state of the art

During our investigations we have not been able to determine the current level of the strongest scrabble programs. While some people stated that scrabble programs such as *TSP* by Jim Homan and *Tyler* by Alan Frank (the two competitors at the third Computer Olympiad) (Uljee, 1992) are stronger than the best human players, others believe that human players still have the edge.

Techniques currently applied

Scrabble as a family game may be best known for its potential of family disputes: while one player maintains that a word is valid, another may dispute it. At official scrabble tournaments, the set of legal words is strictly defined. Either the British *Official Scrabble Words* or the American *Official Scrabble Players Dictionary* determine the legal words. For words of nine or more letters, *Webster's Ninth Collegiate* is decisive. All strong scrabble programs have these dictionaries in memory.

Generally, a set of legal moves is selected, of which each move is evaluated according to (1) the number of points scored; (2) the remaining board position (i.e., the average score the opponent may obtain after the move); and (3) the potential of the letters remaining on the rack, in combination with the letters likely to be drawn from the heap.

The endgame of scrabble (i.e., once all letters from the heap have been drawn) is a perfect-information game. A standard forward search can be applied to such positions to determine optimal play for both sides.

Obstacles to progress

Scrabble programs have shown to be capable of high-level play, even though relatively little research has been performed in this area. We believe that using existing techniques, scrabble programs will surpass their human opponents, if this is not already the case. Summarizing, no obstacles to progress in scrabble have been encountered.

6.3.14 Backgammon

Game properties

Backgammon is a converging, imperfect-information game of fixed termination. Although both players have access to all information determining the current state, dice determine the legal continuations. Not until a player is bearing her stones off or until the game has converted into a running game, are conversion moves made.

The state of the art

In 1980, the human world champion in backgammon, Luigi Villa, was beaten in a short match by the backgammon program BKG (Berliner, 1980). However, both the length of the match, and the fact that Villa seems not to have taken the match as seriously as he should have done, suggest that BKG may not have been truly stronger than the top human players of that time.

Recently, Gerald Tesauro created the program *TD-gammon*, which narrowly lost a match against former world champion Bill Robertie: 40-39. Tesauro's investigations suggest that *TD-gammon* is significantly stronger than BKG (approximately 0.35 points per game), while being close to current human world-champion level (Tesauro, 1993).

Techniques currently applied

While BKG has been created through expert knowledge, *TD-gammon* is a three-layer neural network, which is trained through the unsupervised TD(λ) learning algorithm. The input to *TD-gammon* consists of the board position in combination with some fairly basic backgammon knowledge. From the input and a random initialized network, *TD-gammon* has trained itself on 1.5 million games of self play, resulting in world-class level play (Tesauro, 1993).

Using the neural network as the evaluation function, *TD-gammon* performs a 3-ply search. Doubling is handled by a separate algorithm, as well as part of bearing off, for which an endgame database is used.

Obstacles to progress

Tesauro's work on *TD-gammon* indicates that a neural network is capable of capturing pattern knowledge in backgammon as well as the strongest human players. Therefore, we do not see obstacles which have become apparent through research on backgammon.

6.3.15 Bridge

Game properties

By declaring bridge to be a two-player game, it was possible to include it in the Olympic List. Arguments can be adduced for bridge being a two-player, three-player or four-player game. During the bidding phase, four players participate in the bidding. During the playing phase, three players participate, while the fourth player becomes the *dummy*. On the score card, two partnerships are recognized as the players in bridge. Like Blair *et al.* (1993), we have chosen to regard bridge as a two-player game.

We remark that double-dummy bridge problems are two-player perfect-information games, while bridge problems assuming optimal counterplay can be regarded as two-player imperfect-information games. Finally, we mention that Blair *et al.* (1993) call the three and four player phases in bridge, two-player games without *perfect recall*.

Restricting ourselves to the playing phase of bridge, it is a converging, imperfect-information game with fixed termination.

The state of the art

Instead of trying to master the whole game at once, several researchers have concentrated on single aspects, such as Lindelof (1983), who developed a special bidding system for computer programs and Berlekamp (1963), who created a double-dummy analyzer. Recently, Schoo (1992) has created a program which determines optimal play in single suits.

Despite progress on parts of bridge, the strength of today's best bridge programs may at best be called amateur level. An example of leading bridge programs is *Bridge Baron* by Tom Throop and Tony Guilfoyle, winner of the bridge tournament at the second and third Computer Olympiads.

Techniques currently applied

Bridge Baron consists of knowledge rules which determine what to bid and the information each bid contains. A major problem not yet solved is interpreting the bids of the opponents when they are using vastly different bidding systems.

Knowledge rules containing standard playing patterns form the basis for the playing phase, in combination with search. The heuristic nature of the patterns is the source of errors, as shown in a deciding hand in the final of the third Computer Olympiad (Throop and Guilfoyle, 1992).

Except for double-dummy problems and single-suit problems, exhaustive search has so far not been successful, predictions by Levy (1989) notwithstanding that a world-champion level program based on brute-force search could be created with today's technology.

Obstacles to progress

The main reason for the slow progress on bridge seems the inability of programs to truly understand the vague information they are processing. Instead, programs are taught a bidding system by specifying for each bid the hands for which the bid may be applicable, and the information transferred by the bid. The creation of a bidding program in this way suffers from the knowledge-acquisition bottleneck (Feigenbaum, 1979). Furthermore, extracting information from the bidding phase for use during the playing phase has proved to be rather difficult. Novice human players learning to play bridge experience similar problems. However, through experience, they learn to interpret bids, judge hands, and transfer information gained during bidding to the playing phase. We believe that the experience obstacle blocks progress in bridge.

6.4 Reviewing the problem statement

In section 1.4, we have formulated the problem statement consisting of two questions. To find an answer to the questions in the problem statement, we formulated three research questions. In this section we summarize the answers found to the three research questions (section 6.4.1) and review the problem statement (section 6.4.2).

6.4.1 The research questions

In this section, we summarize the answers found to the three research questions of section 1.4. We discuss each of the questions separately.

Solvable games

The first research question reads: 'Which games can be solved and what techniques may contribute to the solution. With respect to the first part of the question, solvable games, we have found the following answer.

1. Four games (qubic, connect-four, go-moku and nine men's morris) have been solved.

2. Awari and renju without opening restrictions will be solved in the near future.

3. Checkers is a likely candidate for solution in the future.

With respect to the second part of the question, contributing techniques, we have found the following answer.

1. For qubic, go-moku and renju, db-search has been, or will be, a contributor to finding winning threat sequences.

2. For qubic, connect-four, go-moku, renju and checkers, pn-search has been, or may be, a contributor to performing a forward search to solve the game.

3. For nine men's morris, awari and checkers, retrograde analysis has been, or will be, a contributor to create endgame databases which reduce the size of the search tree necessary to solve the game.

4. In connect-four applying knowledge rules to determine the game-theoretic value of game positions has proved to be successful.

5. Variants of α-β search have proved effective as contributors to the solution of qubic, connect-four and nine men's morris, while they may aid in solving checkers.

Outperforming the best human players

The second research question reads: 'For which games can we create programs outperforming the best human players in the near future, and what techniques contribute to their performance.' With respect to the first part of the question, outperforming the best human players, we have found the following answers (we ignore the games listed in the answers to the first research question.)

1. Today's othello programs are stronger than the best human players.

2. Today's draughts, backgammon and scrabble programs are close to world champion level. Expected progress in the near future, possibly just by technological advances, seem sufficient to outperform the best human players.

3. For chess, Chinese chess and (professional) renju, current techniques may prove sufficient to obtain world-champion level, although it is rather difficult to predict when the last human hurdle will be taken.

With respect to the second part of the question, contributing techniques, we have found the following answer.

1. The most important techniques for obtaining high-level tournament programs have been sophisticated variants of α-β search, with fine-tuned static evaluation function. It is a contributing factor in othello, draughts, chess, Chinese chess and professional renju.

2. Db-search in combination with pn-search may prove a contributing factor for professional renju.

3. Neural networks are the basis for the high performance level in backgammon.

Human superiority

The third research question reads: 'In which games will humans continue to reign in the near future (say, at least the next decade) and what are the main obstacles to progress for computer programs?' With respect to the first part of the question, human superiority, we have found the following answer.

1. For chess, Chinese chess and (professional) renju it is unclear whether the, seemingly inevitable, defeat of the strongest human players will take place within the coming decade.

2. For bridge and go the current performance level as well as the obstacles to progress suggest that humans will remain superior for at least the coming decade, if not for much longer.

With respect to the second part of the question, we have found that the main obstacle to progress apparent in several games, but most clearly in bridge and go, is the lacking ability to gain *experience*.

6.4.2 The problem statement

Through the answers to the three research questions, as presented in section 6.4.1, we are now able to discuss the questions raised in the problem statement.

As an answer to the first question, concerning new AI techniques applicable to other domains, we have found in the course of our research two new search techniques, pn-search and db-search. Pn-search is applicable to AND/OR trees (see chapter 2), and can thus be applied outside the area of games. Db-search is a single-agent search (see chapter 3), for which we have presented examples including production systems. The applicability of db-search to problems outside the domains discussed in this thesis needs to be

Predicted program strengths in the year 2000				
Solved	Over Champion	World Champion	Grand Master	Amateur
connect-four	checkers			
qubic	renju	chess	Chinese chess	
nine men's morris	othello			go
go-moku	scrabble	draughts	bridge	
awari	backgammon			

Table 6.1: Predictions for the Olympic Games in the year 2000

investigated in the future. Clearly, as challenges remain within the domain of games, with as specific examples bridge and go, new AI techniques may be developed through further investigation of these games.

As answer to the second question, concerning obstacles emerging through investigation of games, we have found a single obstacle, apparent in several games, but most pronounced in bridge and go: the lack of an ability to gain *experience*. The ability to gain experience is based on *learning* and *flexibility*. Flexibility is necessary to generalize while learning, and to recognize the applicability of patterns learned. While these concepts are not at all new revelations, we believe that their importance in relation to our research consists of showing that even without other interfering obstacles, such as common-sense knowledge, gaining experience is an obstacle in itself. We believe that to overcome such an obstacle, a recommended approach is to research it in separation from other known obstacles. Stated differently, we believe that bridge and go are suitable test beds for investigating the nature of the experience obstacle.

In conclusion, we state that our research has contributed two new search techniques which may be applied in AI, as well as some additional insight in the importance of one obstacle to game research.

6.5 Predictions

6.5.1 Future playing strength

In 1990 we have predicted the strength of computer programs in the year 2000 for each of the games of the Olympic List (Allis *et al.*, 1991a). These predictions have been reproduced in table 6.1. In 1990, we were only aware of the solution to connect-four even though qubic had been solved over a decade before. Currently, four of the five games listed as predicted to be solved in

2000 are solved. In the *Over Champion* category (i.e., significantly stronger than the human world champion), renju is listed. If we were to recreate table 6.1 today, we would put renju without opening restrictions in the *Solved* category, while we would put professional renju at the *Grand Master* category. The *Over Champion* entry should thus be regarded as a compromise between these two. Of the five games in the *Over Champion* category, currently only othello is known to have achieved true Over Champion level. To be at world champion level means having a rating close to that of the human world champion. For both games mentioned (chess and draughts), an official rating system exists, which makes it possible to check such a claim. Equivalent to such a rating would be a close match over a large number of games. Thus, *Chinook* is considered by us to be of world-champion level in checkers. The main reason for listing Chinese chess at Grand Master level, instead of at world-champion level, is the little effort invested in comparison with chess. Therefore, we believe that progress in Chinese chess will keep trailing several years behind that of chess.

The bridge entry at Grand Master level in retrospect seems somewhat optimistic. Had we introduced a Master level, this is where we would categorize it with our 1994 knowledge. However, having to choose between amateur level and Grand Master level, we opted for the latter.

Finally, the go entry speaks for itself. In go terminology, the term amateur may be ambiguous. To be clear, any dan rating in the year 2000 for computer programs (even amateur dan ratings) would be above our current expectations.

6.5.2 The future of games

Even where computers have failed to achieve perfection, which we see as solving the game, they may succeed at the simpler task of outwitting human beings. In table 6.1, we predict that for the majority of the games of the Olympic List computers will have the advantage over their human opponents before the turn of the century.

This being so, we nevertheless argue that all games will continue to be played at all levels, from youngsters enjoying tic-tac-toe to Grand Masters competing in chess tournaments for titles and money. Neither known game-theoretic values nor the availability of silicon opponents of superior strength will extinguish man's urge to compete.

It has also been argued that, once a program of over-champion strength exists, programs will cease to improve. Not so: while human beings construct programs, competition among programmers will see to it that programs will continue to rise in strength. We therefore conclude: *all* games will survive at *all* levels.

Appendix A

Domain-specific solution to DLP

In this appendix we describe the algorithm TRIANGLE to determine the solution to an instance of the double-letter puzzle. TRIANGLE has storage complexity in the order of n^2 and time complexity in the order of n^3.

To simplify the description of TRIANGLE, we index the letters in the axiom of DLP from 0 to $n-1$, where n is the length of the axiom. We define a *substring* of the axiom as any range of letters from a start index i to an end index j, with $0 \leq i \leq j \leq n-1$.

TRIANGLE uses a triangular array of $\frac{1}{2}n(n+1)$ entries, where each entry can store any subset of $\{a, b, c, d, e\}$. Rows in the array represent start indices, and columns represent end indices, i.e., each row i consists of column entries i to $n-1$. In the triangular array, TRIANGLE stores for each *substring* of the axiom, the single letters to which that substring can be reduced. After finishing this task for all substrings, the solution to DLP is found in the entry representing the substring with start index 0 and end index $n-1$, which represents the whole axiom. The triangular array is filled in n steps.

First, the n entries with the start index equal to the end index (entries $[i, i]$, for $0 \leq i \leq n-1$) are initialized to the singleton set containing the letter at position i in the axiom. The other $\frac{1}{2}n(n-1)$ entries are not initialized.

Second, we concentrate on entries representing substrings of two letters (i.e., entries $[i, i+1]$, for $0 \leq i \leq n-2$). In general, the value of $[i, i+1]$ can be determined by looking at the sets at table entries $[i, i]$ and $[i+1, i+1]$. The intersection S of these sets indicates pairs of equal letters which can be replaced by the predecessor or successor of the letters in S. These

	0	1	2	3	4	5	6	7	8	9	10
0	a	be	ac	–	–	–	bd	ce	bd	–	abce
1		a	–	–	–	–	–	–	–	–	–
2			b	–	–	–	–	–	–	–	–
3				d	–	–	ce	–	ce	–	ad
4					c	–	bd	ce	bd	–	ac
5						b	ac	–	–	–	–
6							b	–	–	–	–
7								d	–	–	–
8									c	–	–
9										a	be
10											a

Figure A.1: Solution to instance *aabdcbbdcaa* of DLP.

predecessors and successors are then stored at the entry $[i, i+1]$.

Third, we determine the value of the entries representing substrings of three letters (i.e., entries $[i, i+2]$, for $0 \leq i \leq n-3$). To determine the value of $[i, i+2]$ we must look at the intersection S_1 of the sets at entry $[i, i]$ and $[i+1, i+2]$, and at the intersection S_2 of the sets at entry $[i, i+1]$ and $[i+2, i+2]$. The union of S_1 and S_2 determines the letters from which the predecessors and successors are included in entry $[i, i+2]$.

In general, entry $[p, q]$ is the set of predecessors and successors of the letters in $\bigcup_{i=p}^{q-1}([p, i] \cap [i+1, q])$.

Figure A.1 depicts the array of entries created to solve the instance *aabdcbbdcaa* of DLP (the example of section 3.2). The set of letters stored in entry $[0, n-1]$ yields the solution. As mentioned in section 3.2, only d is not a solution.

Summary

In this thesis "intelligent" games are investigated from the perspective of Artificial Intelligence (AI) research. Games were selected in which, at least partially, human expert players outperformed their artificial opponents. By investigating a game, we envision at least two possible outcomes.

- If we achieve a playing strength sufficient to defeat the best human players, analysis of the means which led to this improvement may uncover new AI techniques.
- If the playing strength keeps falling short, even after prolonged attempts, of that of the best human players a better understanding of the problems inherent in playing the game at a high level may be acquired.

We remark that there is a possibility that the results do not lead to progress (i.e., no new AI techniques and no better understanding of the inherent problems). In the first case, the improvement may be due to entirely domain-specific techniques which cannot be generalized to AI techniques. In the second case, we may find that we have difficulty in isolating the problems from our failed attempts. By investigating a representative set of games, the probability increases that new AI techniques are developed or insight into problems hindering progress is obtained. For our investigations, we have selected a set of games called the *Olympic List*, consisting of: awari, backgammon, bridge, chess, Chinese chess, checkers, connect-four, draughts, go, go-moku, nine men's morris, othello, qubic, renju and scrabble.

The research is in two parts. First, we have investigated three games which we believed could be *solved*: awari, qubic and go-moku. Games can be solved if it is possible to determine strategies leading to the best possible result for both players. For qubic and go-moku we have been able to find strategies which guarantee a win for the first player. For awari this has not yet been achieved, but we did create a program that outperforms the strongest human players. Analysis and generalization of the methods used in solving qubic and go-moku resulted in two new AI techniques: the search techniques *proof-number search* (pn-search) and *dependency-based search* (db-search). Awari is close to its solution, indeed so close that we believe that extant techniques suffice to solve it.

Second, for each game of the Olympic List we have investigated whether the difference in playing skill of human beings and computer programs gives us reason to believe that there is an intrinsic obstacle to progress. We have found that, based on insufficient flexibility and learning ability, an *experience* obstacle exists. This obstacle is particularly conspicuous in bridge and go. We conjecture that, while such obstacles exist in the games domain, these same obstacles will stand in the way of progress in other domains.

This thesis consists of six chapters. In chapter 1, the relevance of investigating games is discussed, leading to the formulation of a problem statement and three research questions. In chapter 2, pn-search is defined. It is shown that pn-search traverses a set of state spaces much more efficiently than alternative search algorithms; awari serves to provide an example. In chapter 3, db-search is defined, a search algorithm that traverses a state space significantly reduced when compared to traditional search algorithms. It is shown that under clearly defined conditions the reduced state space is *complete*, which means that it contains all solutions present in the original state space. The potential of db-search is demonstrated on an example domain. In chapter 4, it is demonstrated how pn-search and db-search solved qubic. Similarly, in chapter 5 it is demonstrated how pn-search and db-search combined solved go-moku. In chapter 6 all games of the Olympic List are investigated, resulting in, among others, a prediction of the playing strengths of the strongest computer programs in the year 2000 and a discussion of the future of games in our society.

Samenvatting

Dit proefschrift beschrijft onderzoek naar denkspelen in het kader van de Kunstmatige Intelligentie. Uitgegaan is van denkspelen waarin de sterkste menselijke spelers hun kunstmatige opponenten, in elk geval op onderdelen, nog de baas waren. Dergelijke onderzoekingen kunnen leiden tot tenminste twee nuttige uitkomsten.

- Wanneer de achterstand op de menselijke topspelers geheel wordt ingelopen, dan leidt analyse van de wijze waarop dit bereikt wordt mogelijk tot het vinden van nieuwe AI-technieken.

- Wanneer ook na langdurige pogingen het niveau van de mens onhaalbaar blijkt, kan analyse van de gevonden problemen leiden tot het ontdekken van algemene obstakels voor vooruitgang in de Kunstmatige Intelligentie.

Het is natuurlijk ook mogelijk dat de achterstand op de mens in een bepaald denkspel wordt ingehaald, maar dat reeds bestaande technieken gebruikt kunnen worden, of dat de gebruikte technieken geheel specifiek zijn voor dat spel en geen algemenere toepassing zullen vinden. Ook zou het zich kunnen voordoen dat langdurige pogingen tot analyse van de gevonden problemen tot niets leiden. Door een representatieve verzameling denkspelen te onderzoeken, achten wij de kans groot dat onderzoek bij een aantal daarvan tot nieuwe inzichten zal leiden. Deze verzameling, die der *Olympische Denkspelen*, bestaat uit: awari, backgammon, bridge, Chinees schaken, checkers, dammen, go, go-moku, molenspel, othello, qubic, renju, schaken, scrabble en vier-op-een-rij.

In het onderzoek hebben we ons allereerst geconcentreerd op drie denkspelen die mogelijk opgelost konden worden: awari, qubic, en go-moku. Dit zijn denkspelen waarvoor het mogelijk lijkt uitspraken te bewijzen over strategieën die tot het best bereikbare resultaat leiden voor beide spelers.

Voor qubic en go-moku hebben we een strategie kunnen vaststellen die de eerste speler winst garandeert. Voor awari zijn we nog niet zover; wel is een programma gecreëerd dat sterker speelt dan menselijke topspelers. Analyse en generalisatie van de methoden die tot de oplossing van qubic en go-moku leidden, hebben twee nieuwe AI technieken opgeleverd, namelijk de zoektechnieken *proof-number search* (pn-search) en *dependency-based search* (db-search). Awari staat op het punt opgelost te worden. We geloven dan ook dat bestaande technieken hiervoor afdoende zullen blijken te zijn.

Vervolgens is voor elk van de Olympische Denkspelen nagegaan in hoeverre de afwijking tussen de speelniveau's van mensen en computers aanleiding geeft te veronderstellen dat een belangrijk obstakel de vooruitgang in de weg staat. Wij hebben gevonden dat met name het feit dat computerprogramma's onvoldoende in staat zijn relevante *ervaring* op te doen, door gebrek aan flexibiliteit en lerend vermogen, dit bij sommige spelen leidt tot een wezenlijke achterstand ten opzichte van menselijke spelers. Het duidelijkst wordt dit gebrek bij bridge en go. We vermoeden dat zolang bij begrensde onderzoeksgebieden, zoals denkspelen, dergelijke obstakels vooruitgang in de weg staan, diezelfde obstakels een hindernis vormen bij vooruitgang in andere onderzoeksgebieden.

Het proefschrift bestaat uit zes hoofdstukken. In hoofdstuk 1 worden de mogelijke produkten van onderzoek naar denkspelen beschreven. Er wordt een probleemstelling geformuleerd, evenals drie onderzoeksvragen. In hoofdstuk 2 wordt pn-search gedefinieerd. Aan de hand van experimenten op awari wordt aangetoond dat pn-search een bepaald type zoekruimte aanzienlijk efficienter onderzoekt dan alternatieve zoekalgoritmen. In hoofdstuk 3 wordt db-search gedefinieerd, een zoekalgoritme dat de zoekruimte die door traditionele zoektechnieken wordt onderzocht aanzienlijk verkleint. Er wordt aangetoond dat onder nauwkeurig gedefinieerde omstandigheden de door db-search verkleinde zoekruimte *volledig* is, wat wil zeggen dat zij alle oplossingen van de oorspronkelijke ruimte bevat. Aan de hand van een voorbeeld wordt de potentie van db-search geïllustreerd. In hoofdstuk 4 wordt gedemonstreerd hoe pn-search en db-search qubic hebben opgelost, terwijl in hoofdstuk 5 het oplossen van go-moku met pn-search en db-search wordt beschreven. In hoofdstuk 6 worden alle Olympische Denkspelen onder de loep genomen, resulterend in, onder andere, een voorspelling van de speelsterkte van de beste computerprogramma's in het jaar 2000 en van de toekomst van denkspelen in onze samenleving.

Curriculum Vitae

Name:	L. Victor Allis
Date of birth:	May 19, 1965
Place of birth:	Gemert, The Netherlands
Nationality:	Dutch
Married to:	Petra Allis-Meinsma
Daughter:	Cindy
Email:	victor@cs.vu.nl

Education

Sept '77–Aug '83	Hermann Wesselink College, Amstelveen.
Sept '83–Oct '88	Vrije Universiteit, Amsterdam, Master degree (with honors)
Jan '90 –Aug '93	University of Limburg, Maastricht. Ph.D. student in Artificial Intelligence. Supervisor: H.J. van den Herik.

Work Experience

Sept '85 –June '87	Teaching assistant at the Vrije Universiteit, Amsterdam.
April '88–Sept '88	Free-lance Computer Science Teacher, NOVI, Maarssen.
Jan '89 –Nov '89	Programmer, Analist, Project Leader at Advanced Management Systems, Takapuna, New Zealand.
Jan '90 –Aug '93	Free-lance Computer Science Teacher, NOVI.
Sept '93 –	Assistant professor of Artificial Intelligence at the Vrije Universiteit, Amsterdam.

Bibliography

[1] Allen J. D. (1989). A Note on the Computer Solution of Connect-Four. *Heuristic Programming in Artificial Intelligence 1: the first computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 134–135. Ellis Horwood, Chichester, England. (163)

[2] 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, Chichester, England. (95, 97)

[3] Allis L.V. (1988). *A Knowledge-Based Approach of Connect-Four. The Game is Solved: White wins.* M.Sc. Thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam. (9, 60, 95, 163)

[4] Allis L.V., Van den Herik H.J., and Herschberg I.S. (1991a). 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, Chichester, England. (182)

[5] Allis L.V., Van der Meulen M., and Van den Herik H.J. (1991b). $\alpha\beta$ Conspiracy-Number Search. *Advances in Computer Chess 6* (ed. D.F. Beal), pp. 73–95. Ellis Horwood, Chichester, England. (60, 62)

[6] Allis L.V., Van der Meulen M., and Van den Herik H.J. (1991c). Databases in Awari. *Heuristic Programming in Artificial Intelligence 2: the second computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 73–86. Ellis Horwood, Chichester, England. (46, 47, 166)

[7] Allis L.V., Van den Herik H.J., 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 Technical Report FS93-02, Menlo Park, CA. (95, 97)

[8] Allis L.V., Van der Meulen M., and Van den Herik H.J. (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91–124. (62, 95)

[9] Anantharaman T.S., Campbell M.S., and Hsu F.-h. (1989). Singular Extensions: Adding Selectivity to Brute-Force Searching. *Artificial Intelligence*, Vol. 43, No. 1, pp. 99–109. (47)

[10] Beal D.F. (1984). Mixing Heuristic and Perfect Evaluations: Nested Minimax. *ICCA Journal*, Vol. 7, No. 1, pp. 10–15. (46)

[11] Beasley J.D. (1985). *The Ins & Outs of Peg Solitaire*. Oxford University Press, Oxford. (6)

[12] Berlekamp Elwyn R. (1963). Programs for Double-Dummy Bridge Problems - A New Strategy for Mechanical Game Playing. *Journal of the Association for Computing Machinery*, Vol. 10, No. 4, pp. 357–364. (178)

[13] Berlekamp E.R., Conway J.H., and Guy R.K. (1982). *Winning Ways for your mathematical plays II*. Academic Press, London. (5)

[14] Berliner H.J. (1979). The B* Tree Search Algorithm: A Best-First Proof Procedure. *Artificial Intelligence*, Vol. 12, pp. 23–40. (16, 62)

[15] Berliner H.J. (1980). Backgammon Computer Program Beats World Champion. *Artificial Intelligence*, Vol. 14, pp. 205–220. (177)

[16] Blair J.R.S., Mutchler D., and Liu C. (1993). Games with Imperfect Information. *Proceedings of the 1993 AAAI Fall Symposium on Games: Planning and Learning*, pp. 59–67. AAAI Press Technical Report FS93-02, Menlo Park, CA. (178)

[17] Boon M. (1991). Overzicht van de ontwikkeling van een go spelend programma. M.Sc. Thesis, University of Amsterdam, The Netherlands. (174)

[18] Breuker D.M., Allis L.V., and Herik H.J. van den (1994). How to Mate: Applying Proof-Number Search. *Advances in Computer Chess 7*, pp. 251–272. (60, 61)

[19] Buchanan B.C. and Shortliffe E.H. (1984). *Rule-Based Expert Programs: the MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, Reading, MA. (2)

[20] Campbell M.S. and Marsland T.A. (1983). A Comparison of Minimax Tree Search Algorithms. *Artificial Intelligence*, Vol. 20, No. 4, pp. 347–367. (15, 61)

[21] Carroll C.M. (1975). *The Great Chess Automaton*. Dover Publications, Inc., New York. (1)

[22] Charniak E. (1978). On the Use of Framed Knowledge in Language Comprehension. *Artificial Intelligence*, Vol. 11, pp. 225–265. (3)

[23] Chen K. (1992). Attack and Defense. *Heuristic Programming in Artificial Intelligence 3: the third computer olympiad* (eds. H.J. Van den Herik and L.V. Allis), pp. 146–156. Ellis Horwood Ltd, Chichester. (174)

[24] De Groot A.D. (1965). *Thought and Choice in Chess*. Mouton Publishers, The Hague-Paris-New York. Translation, with additions, of a Dutch Ph.D. thesis from 1946. Second edition 1978. (3, 128)

[25] Deledicq A. and Popova A. (1977). *Wari et Solo: le jeu de calcul African*. CEDIC, Paris. (43)

[26] Dreyfus H.L. (1980). Why Computers Can't Be Intelligent. *Creative Computing*, Vol. 6, No. 3, pp. 72–78. (4)

[27] Feigenbaum E.A. (1979). Themes and Case Studies of Knowledge Engineering. *Expert Systems in the Micro-Electronic Age* (ed. D. Michie), pp. 3–25. Edinburgh University Press, Edinburgh, Scotland. (3, 179)

[28] Fikes R.E. and Nilsson N.J. (1971). STRIPS: A new approach to the application of theorem proving to artificial intelligence. *Artificial Intelligence*, Vol. 1, No. 2. (65)

[29] Gasser R. (1990). Heuristic Search and Retrograde Analysis: their application to Nine Men's Morris. Diploma thesis, Swiss Federal Institute of Technology, Zürich. (165)

[30] Gasser R. (1991). Endgame Database Compression for Humans and Machines. *Heuristic Programming in Artificial Intelligence 3: the third computer olympiad* (eds. H.J. Van den Herik and L.V. Allis), pp. 180–191. Ellis Horwood, Chichester, England. (9)

[31] Gasser R. (1993). Personal Communication. (166)

[32] Gnodde J. (1993). Aïda, New Search Techniques Applied to Othello. M.Sc. Thesis, University of Leiden, The Netherlands. (39, 60)

[33] Greenblatt R.D., Eastlake III D.E., and Crocker S.D. (1967). The Greenblatt Chess Program. *Proceedings of the Fall Joint Computing Conference*, pp. 801–810. San Francisco. (39, 75)

[34] Hall M.R. and Loeb D.E. (1992). Thoughts on Programming a Diplomat. *Heuristic Programming in Artificial Intelligence 3: the third computer olympiad* (eds. H.J. Van den Herik and L.V. Allis), pp. 123–145. Ellis Horwood Ltd, Chichester. (6)

[35] Hart P.E., Nilsson N.J., and Raphael B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on SSC*, Vol. 4. (17, 63)

[36] Hart P.E., Nilsson N.J., and Raphael B. (1972). Correction to 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths'. *SIGART Newsletter*, Vol. 37. (63)

[37] Hofstadter D.R. (1979). *Gödel, Escher, Bach: an Eternal Golden Braid.* Basic Books, New York. (13)

[38] Hofstadter D.R. (1985). *Metamagical Themas.* Bantam Books, Toronto. (6)

[39] Hsu F.H. (1990). Large Scale Parallelization of Alpha-Beta Search: An Algorithmic Architectural Study with Computer Chess. PhD thesis, Carnegie Mellon University, Pittsburgh, USA. (171, 172)

[40] Keetman S. (1993). Personal Communication. (170)

[41] Klingbeil N. and Schaeffer J. (1988). Search Strategies for Conspiracy Numbers. *Canadian Artificial Intelligence Conference*, pp. 133–139. (60)

[42] Klingbeil N. (1989). Search Strategies for Conspiracy Numbers. M.Sc. Thesis, University of Alberta, Edmonton, Alberta, Canada. (31, 60)

[43] Knuth D.E. and Moore R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. (15, 160)

[44] Knuth D.E. (1969). *The Art of Computer Programming*, Vol. 2. Addison-Wesley Publishing Company. (p. 192 in the second (1981) edition). (8)

[45] Korf R.E. (1985). Depth-First Iterative-Deepening: an Optimal Admissable Tree Search. *Artificial Intelligence*, Vol. 27, pp. 97–109. (6)

[46] Levy D.N.L. and Beal D.F. (eds.) (1989). *Heuristic Programming in Artificial Intelligence: the first computer olympiad.* Ellis Horwood, Chichester, England. (6, 156)

[47] Levy D.N.L. and Beal D.F. (eds.) (1991). *Heuristic Programming in Artificial Intelligence 2: the second computer olympiad.* Ellis Horwood, Chichester, England. (6, 46, 128, 156, 165)

[48] Levy D.N.L. (1989). The Million Pound Bridge Program. *Heuristic Programming in Artificial Intelligence: the first computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 95–103. Ellis Horwood, Chichester, England. (178)

[49] Lindelof E.T. (1983). *COBRA - The Computer Designed Bidding System.* London, Gollancz. (178)

[50] Lister L. and Schaeffer J. (1994). An Analysis of the Conspiracy Numbers Algorithm. *Computers and Mathematics with Applications*, Vol. 27, No. 1, pp. 41–64. (60)

[51] Marr D. (1977). Aritificial Intelligence - A Personal View. *Artificial Intelligence*, Vol. 9, pp. 37–48. (3)

[52] McAllester D.A. (1988). Conspiracy Numbers for Min-Max Search. *Artificial Intelligence*, Vol. 35, pp. 287–310. (16, 60)

[53] Michalski R.S., Carbonell J.G., and Mitchell T.M. (1983). *Machine Learning: An Artificial Intelligence Approach*, Vol. 1. Tioga, Palo Alto, CA. (3)

[54] Michalski R.S., Carbonell J.G., and Mitchell T.M. (1986). *Machine Learning: An Artificial Intelligence Approach*, Vol. 2. Morgan Kaufmann, Los Altos, CA. (3)

[55] Michie D. (1982). Information and Complexity in Chess. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp. 139–143. Pergamon Press, Oxford. (4)

[56] Newell A., Shaw J.C., and Simon H.A. (1957). Preliminary Description of General Problem Solving Program-I (GPS-I). Report CIP Working Paper 7. (2)

[57] Nilsson N.J. (1971). *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill, New York. (15)

[58] Nilsson N.J. (1980). *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA. (14, 63)

[59] Ohta T. (1993). Personal communication. (173)

[60] Palay A.J. (1982). The B* tree search algorithm - new results. *Artificial Intelligence*, Vol. 19, pp. 145–163. (63)

[61] Patashnik O. (1980). Qubic: 4x4x4 Tic-Tac-Toe. *Mathematics Magazine*, Vol. 53, pp. 202–216. (95, 96, 109, 110, 112, 116, 119, 162)

[62] Reinefeld A. (1994). A Minimax Algorithm Faster than Alpha-Beta. *Advances in Computer Chess 7* (eds. H.J. Van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 237–250. University of Limburg, Maastricht. (15)

[63] Reznitsky A. and Chudakoff M. (1990). Pioneer: A Chess Program Modelling a Chess Master's Mind. *International Computer Chess Association Journal*, Vol. 13, No. 4, pp. 175–195. (171)

[64] Sakata G. and Ikawa W. (1981). *Five-In-A-Row. Renju*. The Ishi Press, Inc., Tokyo. (122, 123, 129, 149)

[65] Samuel A.L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, Vol. 3, No. 3. (2)

[66] 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. (2)

[67] Schaeffer J. (1989). Conspiracy Numbers. *Artificial Intelligence*, Vol. 43, No. 1, pp. 67–84. (16, 60, 61)

[68] Schaeffer J., Culberson J., Treloar N., Knight B., Lu P., and Szafron D. (1991). Reviving the Game of Checkers. *Heuristic Programming in Artificial Intelligence 2: the second computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 119–136. Ellis Horwood Ltd., Chichester, England. (2, 168, 171)

[69] Schaeffer J., Culberson J., Treloar N., Knight B., Lu P., and Szafron D. (1992). A World Championship Caliber Checkers Program. *Artificial Intelligence*, Vol. 53, pp. 273–289. (2, 168)

[70] Schaeffer J. (1993a). Personal Communication. (168, 169)

[71] Schaeffer J. (1993b). A Re-Examination of Brute-Force Search. *Proceedings of AAAI Fall Symposium on Games: Planning and Learning*, pp. 51–58. AAAI Press Technical Report FS93-02, Menlo Park, CA. (5)

[72] Schaeffer J. (1994). Personal Communication. (30)

[73] Schijf M. (1993). Proof-Number Search and Transpositions. M.Sc. Thesis, University of Leiden, The Netherlands. (39, 40, 41, 42)

[74] 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. (39)

[75] Schoo P.N.A. (1992). Optimal Play in a Single Bridge Suit. Personal Communication. (178)

[76] Shortliffe E.H. (1976). MYCIN: Computer-based Medical Consultations. Based on a PhD thesis, Stanford University, Stanford, CA, 1974. (2)

[77] Stiller L. (1989). Parallel Analysis of Certain Endgames. *ICCA Journal*, Vol. 12, No. 2, pp. 55–64. (9)

[78] Stockman G. (1979). A Minimax Algorithm Better than Alpha-beta? *Artificial Intelligence*, Vol. 12, pp. 179–196. (15, 61)

[79] Tesauro G. (1993). TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play. *AAAI Technical report FS93-02 Games: Planning and Learning*, pp. 19–23. AAAI Press Technical Report FS93-02, Menlo Park, CA. (177)

[80] Thompson K. (1982). Computer Chess Strength. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp. 55–56. Pergamon Press. (5)

[81] Thompson K. (1986). Retrograde Analysis of Certain Endgames. *ICCA Journal*, Vol. 9, No. 3, pp. 131–139. (9, 158)

[82] Throop T. and Guilfoyle T. (1992). A Thrilling Hand. *Heuristic Programming in Artificial Intelligence 3: the third computer olympiad* (eds. H.J. Van den Herik and L.V. Allis), pp. 27–28. Ellis Horwood Ltd, Chichester. (178)

[83] Tromp J.T. (1993). *Aspects of Algorithms and Complexity*. University of Amsterdam. Ph.D. Thesis. (163)

[84] Tsao Kuo-Ming, Li Horng, and Hsu Shun-Chin (1991). Design and Implementation of a Chinese Chess Program. *Heuristic Programming in Artificial Intelligence 2: the second computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 108–118. Ellis Horwood Ltd, Chichester. (172)

[85] Uiterwijk J.W.H.M., Van den Herik H.J., and Allis L.V. (1989a). 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. (9, 163)

[86] Uiterwijk J.W.H.M., Van den Herik H.J., and Allis L.V. (1989b). A Knowledge-Based Approach to Connect-Four. The Game is Over: White to Move Wins! Report CS 89-04, Department of Computer Science, Faculty of General Sciences, University of Limburg. (163)

[87] Uiterwijk J.W.H.M. (1992a). Go-Moku still far from Optimality. *Heuristic Programming in Artificial Intelligence 3: the third computer*

olympiad (eds. H.J. Van den Herik and L.V. Allis), pp. 47–50. Ellis Horwood Ltd, Chichester. (122, 164)

[88] Uiterwijk J.W.H.M. (1992b). Knowledge and Strategies in Go-Moku. *Heuristic Programming in Artificial Intelligence 3: the third computer olympiad* (eds. H.J. Van den Herik and L.V. Allis), pp. 165–179. Ellis Horwood Ltd, Chichester. (126)

[89] Uljee I.H. (1992). Letters beyond Numbers. *Heuristic Programming in Artificial Intelligence 3: the third computer olympiad* (eds. H.J. Van den Herik and L.V. Allis), pp. 63–66. Ellis Horwood Ltd, Chichester. (175)

[90] Van den Herik H.J. and Allis L.V. (eds.) (1992). *Heuristic Programming in Artificial Intelligence 3: the third computer olympiad*. Ellis Horwood, Chichester, England. (6, 46, 128, 156)

[91] Van den Herik H.J. and Herschberg I.S. (1985). The Construction of an Omniscient Endgame Data Base. *ICCA Journal*, Vol. 8, No. 2, pp. 66–87. (9)

[92] Van den Herik H.J. and Herschberg I.S. (1989). Champ meets Champ. *ICCA Journal*, Vol. 12, No. 4. (171)

[93] Van den Herik H.J. (1983). *Computerschaak, Schaakwereld en Kunstmatige Intelligentie*. Academic Service, 's-Gravenhage. (4, 172)

[94] Van den Herik H.J. (1991). *Kunnen computers rechtspreken?* Gouda Quint BV, Arnhem. (2)

[95] Van der Meulen M. (1990). Conspiracy-Number Search. *ICCA Journal*, Vol. 13, No. 1, pp. 3–14. (16, 60, 61)

[96] von Neumann J. and Morgenstern O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton. (157)

[97] Winston P.H. (1992). *Artificial Intelligence*. Addison Wesley, Reading, MA. 3rd edition. (5)

[98] Witmans P.A. (1994). Personal communication. (117)

Index

I

K

L

M

N

O

P

Q

R

S

T

U

W