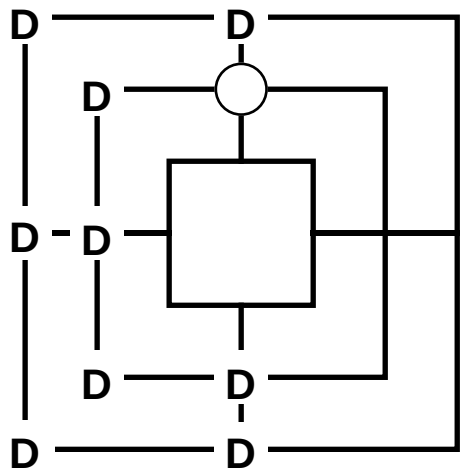
Perfect Play

using Nine Men's Morris as an example



Diploma Thesis
Department of Computer Science
ETH Zürich

Thomas Lincke

Supervising Professor: Prof. Dr. J. Nievergelt

Assistant: Ralph Gasser

Contents

Introduction

Exhaustive search of large spaces is a prominent research topic in computer science and artificial intelligence.
Board games with their large state spaces provide a suitable means to compare the performance of search algorithms. An important milestone in the developement of search techniques is reached when a new game can be solved.
Using SearchBench [2], a tool for exhaustive search in game trees, Ralph Gasser has recently proved that the game of Nine Men's Morris is drawn. He stored all position values of the mid- and endgame phase in databases and used an additional opening search program (the opening prover) to show that the initial position is a draw.

The task of this thesis is to improve SearchBench and the opening search program in three ways.

- None of the two programs allowed the user to play a whole game of Nine Men's Morris. The first goal of the thesis was to merge the programs together and to provide a comfortable user interface for game playing and database analysis. The user interface is described in appendix B.

- The opening prover could not find perfect moves, because it did not distinguish between drawn and won moves for efficiency. In the context of this thesis, a perfect move is a move which attains the value of the position.
  The second goal of the thesis was to rewrite the opening search program to play perfect without requiring excessive search time. The section 'Perfect Play' describes the technical aspects of improving the opening search algorithm.

- A perfect play Nine Men's Morris program can never lose. But it can try to win, even when the actual position is a proven draw, because it can assume that its opponent is fallible and will blunder now and then.
  The third goal of this thesis was to implement an algorithm for traps and swindles. In section 'Traps and Swindles', a model for the difficulty of a position is introduced and a number of experimental results are presented.

Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Diplomarbeit für Thomas Lincke, Abt. IIIC

3. Mai 1994 bis 2. September 1994

Perfektes Spielen am Beispiel von Mühle

Einleitung

perfektes Spielen: Traditionelle Spielprogramme für Brettspiele verwenden bei der Suche nach guten Zügen heuristische Suchalgorithmen. Bei ausreichender Rechenzeit ist es aber auch möglich, den spieltheoretischen Wert zu berechnen, d.h. zu entscheiden, ob ein Spiel gewonnen, verloren oder unentschieden ist für den Anziehenden. Mit Hilfe dieses spieltheoretischen Wertes ist es möglich, ein Spiel perfekt zu spielen.

Datenbanken für Mühle: Ralph Gasser hat für Mühle bewiesen, dass das Spiel unentschieden ist. Dazu hat er Datenbanken für alle Stellungen nach dem Setzen sowie eine Datenbank für Eröffnungen berechnet. Mit einem einfachen Suchprogramm lässt sich so für jede Stellung ein Zug finden, der mindestens das Unentschieden erreicht.

Ausbaumöglichkeiten: Im jetzigen Zustand kann das Programm während der Eröffnung nicht perfekt spielen, da sonst die Antwortzeit und der Speicherbedarf zu gross wären. Aber auch ein perfekt spielendes Programm lässt sich noch gefährlicher machen, wenn es versucht, aus einer Reihe von perfekten Zügen denjenigen zu wählen, der für den Gegner am ‘schwierigsten’ zu beantworten ist. Man kann zum Beispiel für alle perfekten Züge untersuchen, bei welchem von diesen man eine Stellung erhält, welche dem Gegner einen scheinbar guten Zug lässt, der jedoch zum Verlust führt.

Aufgabenstellung

Der erste Teil der Arbeit soll der Programmierung der Benutzeroberfläche gewidmet werden. Dazu gehört das Einarbeiten ins Smart Game Board bzw. SearchBench sowie die anschliessende Entwicklung der grafischen Benutzerschnittstelle.
Im zweiten Teil soll zuerst ein Programm entwickelt werden, das in jeder Stellung in sinnvoller Antwortzeit einen perfekten Zug findet. Dabei sollen die vorhandenen Endspieldatenbanken verwendet und eine neue Datenbank für die Eröffnung berechnet werden. Danach sollen die Möglichkeiten für eine Suche nach ‘schwierigen’ Varianten untersucht werden (traps & swindles). Eventuell müssen zur Verringerung der Anwortzeiten Datenkompressionsverfahren untersucht werden.

Diplomand:	Thomas Lincke
Betreuer:	Ralph Gasser
Leitung:	Prof. J. Nievergelt

Perfect Play

This section describes how the opening search algorithm was improved to return better values and to be faster.
In the first chapter, SearchBench and the opening prover are presented as the starting points of this thesis.
The following chapters describe the steps performed to improve the opening search algorithm. Each chapter concludes with a table showing the performance of the program after the new features were implemented. Chapter 8 summarizes the results.

1 Starting point

As explained in the introduction, two programs already existed when work started, one for the opening and one for the mid- and endgame phase. As a first step, these two programs were merged together so that a complete game could be played with one program.

1.1 SearchBench: An endgame solver

SearchBench [2] is a powerful tool for the computation and management of endgame databases. It provides all necessary routines for computation, access, compression and decompression of database files. To implement a game, only a number of game specific routines, e.g. move generation and board display, must be written. Games implemented on the SearchBench include Chess, the 15 puzzle and Nine Men's Morris.

The state space of all mid- and endgame positions of Nine Men's Morris comprises about 10^{10} positions. SearchBench stores both the game-theoretic value and a distance to win and a distance to loss respectively for every position. In the case of Nine Men's Morris this requires one byte per position. The total size of the databases is about 10GByte, which can be reduced to about 1.7GByte using the internal compression routines.

To make the Nine Men's Morris state space easier to handle, it was split into 28 databases. (Figure 1.1)

For Nine Men's Morris, SearchBench already provided a user interface to allow board editing and game playing in the mid- and endgame. The program could carry out a single perfect move or a sequence of perfect moves.

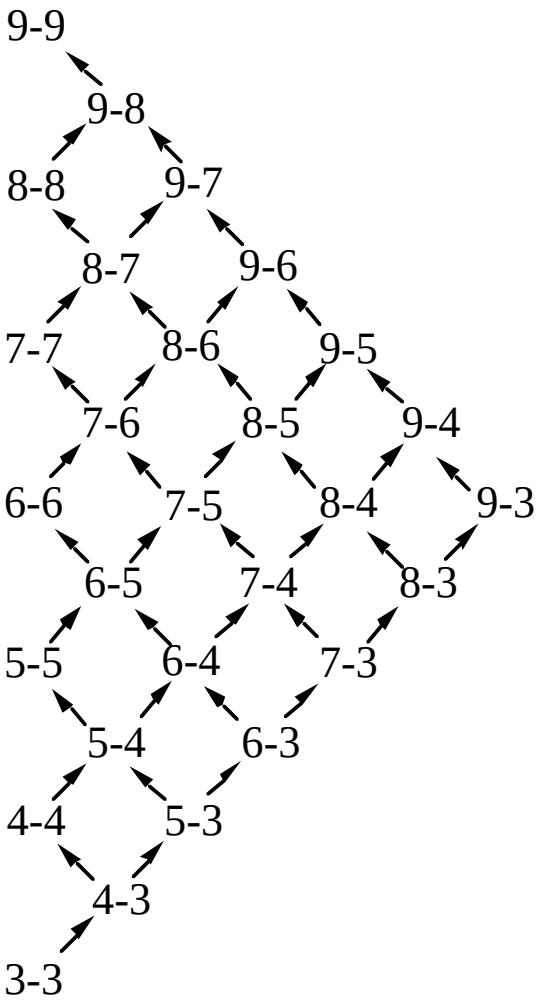


Figure 1.1

1.2 Opening prover

The opening prover was a special purpose program to solve the opening part of the game (Figure 1.2). This chapter describes the techniques and trade-offs used for this program as far as they are necessary for the understanding of the rest of the thesis.

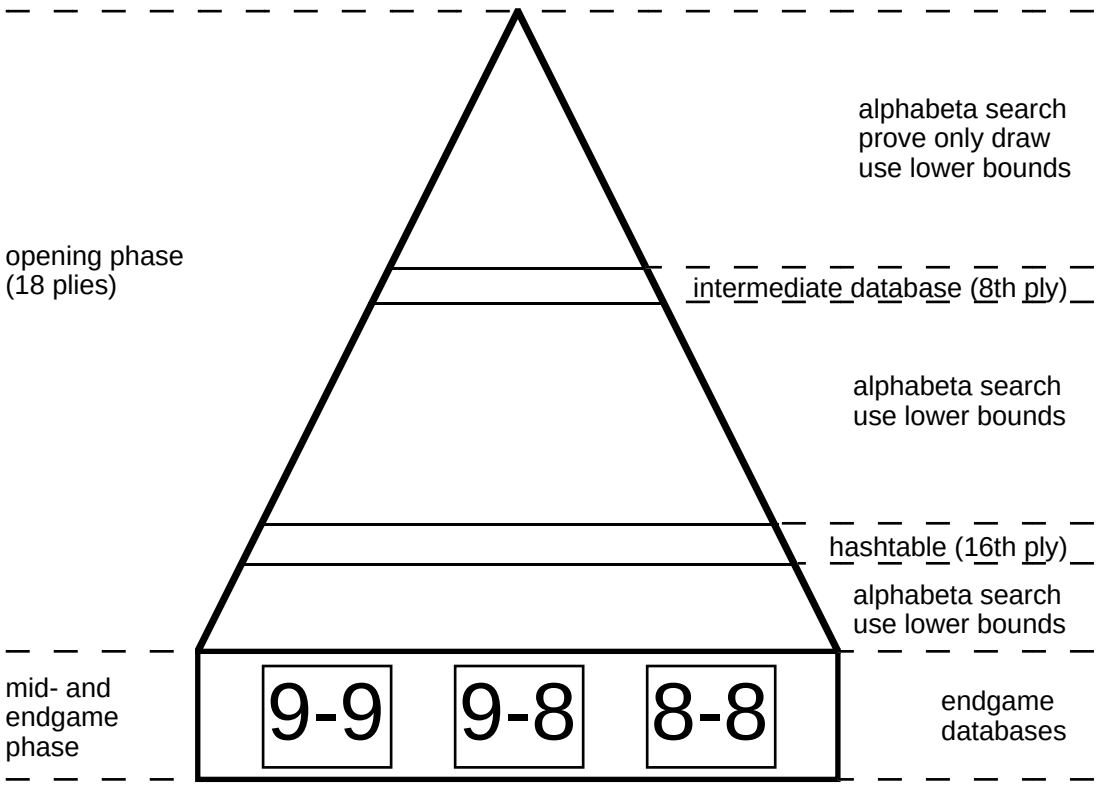


Figure 1.2: organisation of the opening search program. Alpha-beta search is used for value propagation. Where the search tree reaches the 18th ply (endgame phase), values are loaded from the databases. At the 16th ply, values are hashed to be reused in case of move transpositions. At the eighth ply, values are stored into an intermediate database.

store values in an intermediate database: The search time for positions in the first few plies is much too high for real-time evaluation. An intermediate database was introduced to store all necessary values of positions at the eighth ply.
In this database one bit of information was stored per position, it was only differentiated between loss and at least drawn. Again, additional speed is traded off against lower quality of position values, because a position can not be proved to be a win during the first eight plies of the game. Computation of the database could be done relatively quickly, as the alpha-beta search algorithm generates a cutoff as soon as a draw is found. Because the state space of Nine Men's Morris is small enough after eight plies (about $3.5 * 10^6$), this database can be loaded into main memory for faster access.

hash table: During the opening phase, many move transpositions can occur in Nine Men's Morris. To avoid repeated access to the same endgame database value, a hash table was introduced to store values at the 16th ply (Figure 1.2).

special purpose databases: The most straightforward way of accessing endgame values is to call the proper SearchBench function and wait for the value. But this makes the search algorithm slow since SearchBench may have to decompress parts of the database to read a single value. Additionally, the databases contain more information than necessary.

SearchBench does not only compute the game-theoretic values of endgame positions, but also stores the number of moves to win or loss, assuming both Black and White play optimally. But to compute the value of the game, it is sufficient to know the game-theoretic values of endgame positions. Also, not every position in the endgame databases is reachable from opening positions, e.g. an endgame position with nine black and seven white stones must contain exactly two black mills to be reachable (Figure 1.3).
Therefore, smaller special purpose databases are used where only the game-theoretic values are stored and where non-reachable positions are excluded. Storing five values per byte results in databases which are smaller and faster to access than the original databases of SearchBench.

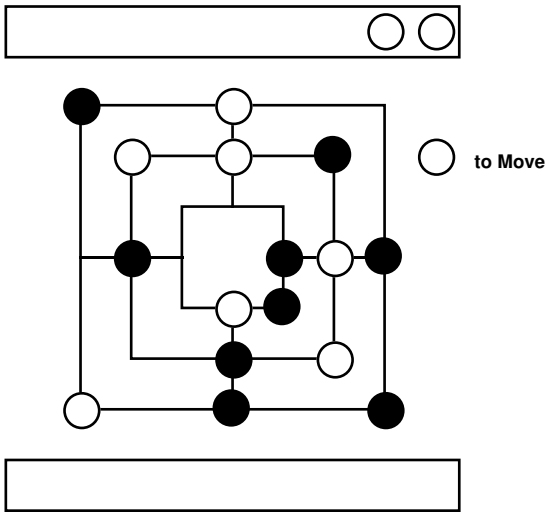


Figure 1.3: A non-reachable position. Black has captured two stones, but has no mills on the board

use small number of databases: Theoretically, positions in any endgame database can be reached from the opening phase. Thus for every SearchBench database a special purpose database must be computed for the opening prover.
But, to prove that Nine Men's Morris is a draw, it turned out that the 9-9, 9-8 and 8-8 databases were sufficient, because in 'reasonable' games none or only a few stones are captured during the opening. In case a database is required which is not available, a lower bound for the value of the position is propagated (see '*worst case values*' below). This technique makes the opening search fast, because less disk accesses are needed, but reduces the quality of the values returned by the alpha-beta search. For example, it is no longer possible to play perfectly from all opening positions.

worst case values: The root node of a search tree is either a Black or a White node. When the search algorithm expands a node it must always check whether the new position is still within the domain of available databases. If the value of the new position is not available, a worst case value for this position is generated. This worst case value depends on the player at the root node and on the player who expanded the new position. If the root player and the expanding player are the same, then the worst case value is a loss, otherwise it is a win. This is just an application of the rule where, for example, if it's Black's move at the root node, any losing move from any Black node in the tree is bad for Black.

2 Performance measurements

In the following chapters, the opening prover will be improved step by step. To give an idea about the effectiveness of the new techniques, every chapter concludes with table showing some profiling information. This chapter explains the recorded data, how they were computed and their values for the opening prover.

hardware: The measurements were performed on a Macintosh Quadra 840AV using a 68040 processor running at 33 MHz. The system has 24MByte RAM and an internal harddisk of 500MByte.

the test position: Positions at the eighth ply are the most difficult to compute, because search must go deeper to reach the databases. The test position was, therefore, chosen at the eighth ply (Figure 2.1).

Note that the values are valid only for this position. The intention is not to show the performance of the algorithm in general, but to demonstrate the strengths and weaknesses of the techniques used.

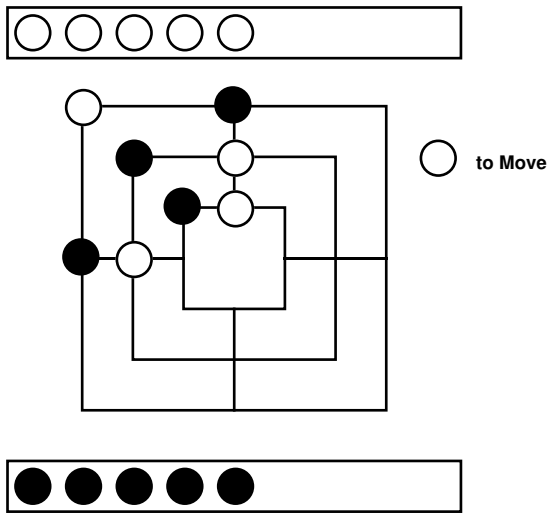


Figure 2.1: the test position

the recorded data:

- Total search time: time required to compute the move
- Disk access time: time spent waiting for a disk access to complete
- Search tree size: number of nodes in the search tree that had to be expanded to compute the value
- Database misses: number of accesses to databases which are not available
- Database hits: number of read accesses to the endgame values
- File accesses: number of load accesses to the database files
- Hashtable accesses: number of accesses to the hashtable
- Hashtable hits: number of values found in the hashtable

Total search time [s]	2468
Disk access time [s]	2012
Relative to total [%]	81
Search tree size	2017853
Database misses	379776
Database hits	1069072
File accesses	1069072
Hashtable accesses	340775
Hashtable hits	249069

opening prover performance: The table on the right shows the profiling data for the old opening prover. For the test position, the total search time was about 20 minutes, of which 81% was spent during disk accesses. The number of database hits and file accesses is equal because values are read one by one from the database files.

3 Buffering

During an opening search from a position at the eighth ply, the number of positions accessed in the endgame databases is in the order of magnitude of 10^6 to 10^7. Profiling of the search routine showed that about 80% of the time is spent accessing the disk to load position values into memory. This in spite of the fact that a hashtable had been introduced at the 16th ply (Figure 1.2) to provide values in case of move transpositions.

3.1 Locality

Even when the repeated access to the value of the same position is avoided, the number of disk accesses can be reduced significantly. Analysis of the access pattern showed, that the loaded values tended to be close together in the database files. This makes it very effective to load whole blocks of values. The locality in the access pattern is a result of the way positions are numbered in the endgame databases.
In databases where the number of white stones is higher or equal to the number of black stones, all positions with the same configuration of white stones are stored in a continuous block. The size of these blocks depends on the number of possible black configurations and is different for every database.

Why is the alpha-beta search algorithm likely to access several values in the same block? This is because black has the last move in the opening. At the 17th ply, White has made his last move in the opening and the white configuration is fixed. All positions generated for the last black move belong to the same white configuration and are stored in the same block. Consequently, a major increase in performance can be achieved by loading whole blocks of values at once and by keeping as many of them as possible in main memory.

3.2 Memory management

If the algorithm has to buffer loaded blocks, it must allocate and deallocate memory quite frequently. Because the system calls for memory allocation and deallocation tend to be slow (on a Macintosh), all memory used for buffering is allocated at initialisation and then managed by the program itself. This is a short description of the memory management algorithms used.

data structure: (see Figure 3.1)
The search algorithm must store and access blocks of different lengths as fast as possible and wants to keep a block in memory as long as possible. To make access fast a hashtable is used, containing a key and a pointer to the corresponding block.
A linear list of free blocks (freelist) is maintained for memory allocation, and a doubly linked list is provided to keep track of the used blocks. Every block is preceded by a header to store the pointers for the lists.

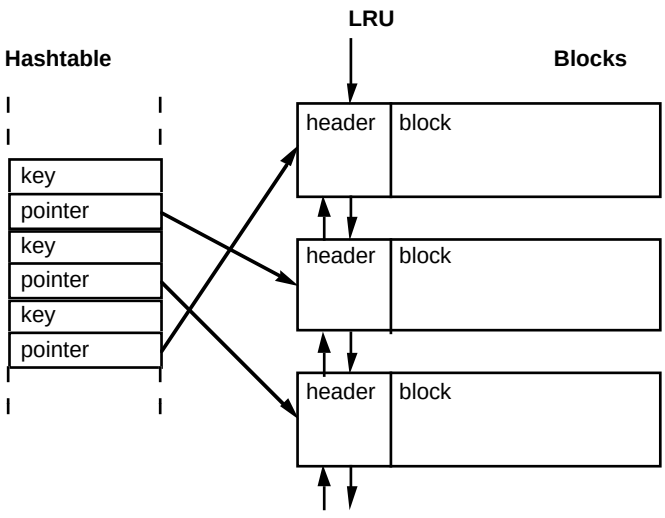


Figure 3.1: data structure

hashing: A hashtable with collision detection is used to search for loaded blocks. The time required for the access through the hashtable is negligible compared to the overall search time, so a simple linear probing algorithm is used.

first fit: To allocate memory, the freelist is searched linearly and the first block which is big enough is used. If no block is available in the free list, the least recently used block is deleted from the used list and appended to the free list, merging it with adjacent free blocks if possible.

LRU: The doubly linked list was implemented to keep track of which block is the least recently used at any time. On every access to a value, the corresponding block is moved to the end of the used list. Hence, the blocks are ordered according to the time of their last access, and the first entry in the list points to the least recently used block.

3.3 Result

The following table shows the profiling data of the buffering algorithm as compared to the old opening prover algorithm.
The number of file accesses was reduced by a factor of about 200, which resulted in a speedup of 4.41.

	old	buffering
Total search time [s]	2468	559
Speedup	1.0	4.41
Disk access time [s]	2012	130
Relative to total [%]	81	23
Search tree size	2017853	2017853
Database misses	379776	379776
Database hits	1069072	1069072
File accesses	1069072	5786
Hashtable accesses	340775	340775
Hashtable hits	249069	249069

4 Perfect values

To get perfect values for all positions in the opening, it would be necessary to make all endgame databases available. But this would increase search time, because values must be loaded instead of using just simply a lower bound, and is not reasonable because only a few databases are responsible for most accesses. Thus, another way of improving the quality of the returned values is needed.

4.1 Value ranges

To reduce the effect of not knowing the exact value, the search algorithm was rewritten to produce both a lower and an upper bound to the exact value, thus providing a means of judging the quality of a value.

Instead of computing the worst case for not available values, we can generate a lower and an upper bound to the exact value. Besides the usual perfect values of win, loss and draw we introduce three new value ranges, namely 'Unknown' ('loss to win'), 'maxDraw' ('loss to draw') and 'minDraw' ('draw to win'). Now the value of any position outside the domain of available databases is set to 'Unknown', and search continues.
range propagation: Figure 3.2 shows a simplified algorithm for alpha-beta search. The new algorithm differs only in the way values are propagated. Figures 3.3 and 3.4 show the two propagation functions, implemented as an array PropTable in Figure 3.2.

```
AlphaBeta(Position, Alpha,Beta)
  SuccessorList := GetSuccessors(Position)
  FOR I := all successors DO BEGIN
          NewValue:=AlphaBeta(SuccessorList[I], Inverse[Beta], Inverse[Alpha])
          Alpha:=PropTable[NewValue, Alpha]
          IF Alpha>=Beta THEN RETURN Alpha
  END
  RETURN Alpha
END
```

Figure 3.2

		Alpha		
		Win	Loss	Draw
NewValue	Win	Win	Win	Win
	Loss	Win	Loss	Draw
	Draw	Win	Draw	Draw

Figure 3.3: standard propagation function

		Alpha					
		Win	Loss	Draw	MinDraw	MaxDraw	Unknown
NewValue	Win	Win	Win	Win	Win	Win	Win
	Loss	Win	Loss	Draw	MinDraw	MaxDraw	Unknown
	Draw	Win	Draw	Draw	MinDraw	Draw	MinDraw
	MinDraw	Win	MinDraw	MinDraw	MinDraw	MinDraw	MinDraw
	MaxDraw	Win	MaxDraw	Draw	MinDraw	MaxDraw	Unknown
	Unknown	Win	Unknown	MinDraw	MinDraw	Unknown	Unknown

Figure 3.4

The new propagation function requires special handling in the root node. One problem arises when one successor is evaluated as 'Unknown' and an other as 'Drawn'. Which is better? Or, the first successor is evaluated as drawn, then search continues with Alpha=draw and Beta=win. If the next value is a 'MinDraw', this may be the result of an 'Unknown' where the lower part of the range has been clipped.
In the former case, the algorithm chooses a conservative strategy, and plays the drawn move. In the latter case, the same move is searched a second time with a full search range.

Discussion worst case values versus value ranges:

advantages of worst case values:

- many worst case values are wins, generating immediate cutoffs, thus speeding up search. Figure 3.5 shows an example where a worst case loss generates a cutoff

disadvantages of worst case values:

- the value resulting from search is only a lower bound for the exact value
- no information is available to tell whether the value is exact or only a lower bound

advantages of perfect ranges:

- the best value is found, given a certain domain of available databases
- if an exact value is returned, then it is a proven value

disadvantages of ranges:

- when propagating a value of 'Unknown', the alpha-beta search gains no new information, thus no cutoffs are generated (Figure 3.6)
- some special handling is necessary at the root node.

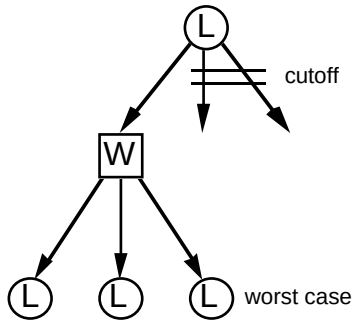


Figure 3.5: worst case values

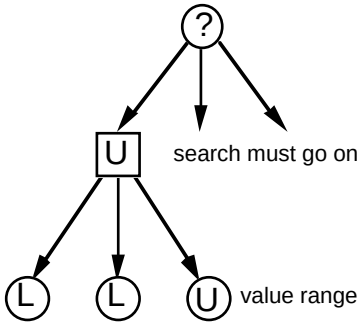


Figure 3.6: value ranges

4.2 Result

The effect of using value ranges instead of worst case values is, that the size of the search tree is increased 36 times, resulting in a disastrous slow down. When value ranges were implemented the first time, they were immediately judged as 'nice but not practical' and discarded. Only after further investigations means were found to avoid the large search tree.

	old	buffering	value ranges
Total search time [s]	2468	559	16568
Speedup	1.0	4.41	0.14
Disk access time [s]	2012	130	309
Relative to total [%]	81	23	1.9
Search tree size	2017853	2017853	73006119
Relative to 'old' [%]	100	100	3618
Database misses	379776	379776	17216242
Database hits	1069072	1069072	49876901
File accesses	1069072	5786	12775
Hashtable accesses	340775	340775	11696076
Hashtable hits	249069	249069	10507311

5 More endgame databases

To prove that Nine Men's Morris is a draw, only three endgame databases were required. When the alpha-beta search algorithm tried to access a database not available, a worst case value was returned and used for further value propagation. This made the search algorithm very fast, because less database values had to be read from the disk and because the worst case values tended to create lots of cutoffs in the search tree.

After the new algorithm for propagation of value ranges had been implemented the strategy of using as few databases as possible had to be dropped. Instead of a worst case value, a value range 'Unknown' is now returned for every access to a database not available, blowing up the search tree.

Because it is important to create as few unknowns as possible, more databases need to be provided for the search algorithm. In fact, the higher search speed resulting from more databases now must be balanced against the disk access delays resulting from more different database values loaded and the increased memory requirements of these databases.

The table shows the results after including three more databases (9-7, 9-6, 8-7). Note how the number of database misses and the search tree size was reduced.

	old	value ranges	databases
Total search time [s]	2468	16568	1148
Speedup	1.0	0.14	2.15
Disk access time [s]	2012	309	186
Relative to total [%]	81	1.9	16
Search tree size	2017853	73006119	4218225
Relative to 'old' [%]	100	3618	209
Database misses	379776	17216242	186808
Database hits	1069072	49876901	2749546
File accesses	1069072	12775	8115
Hashtable accesses	340775	11696076	529696
Hashtable hits	249069	10507311	385202

6 Move ordering

The move ordering strategy of the opening prover was optimized to compute lower bounds. If it was White's move at the top level, then in White's nodes the defensive successors were evaluated first, in Black's nodes the offensive, i.e. the capturing moves, were evaluated first. This makes the search algorithm fast, because every time a Black capturing move leaves the domain of available databases, a win for Black is returned (the worst case for the White player) which results in an immediate cutoff.

When using perfect value ranges in the search algorithm, every capturing move which leaves the domain of available databases generates an 'Unknown'. This does not result in cutoffs, but increases the size of the search tree enormously.
It proved to be best to avoid the evaluation of capturing moves as long as possible, thus in the new move ordering, both players use a defensive evaluation strategy, independent of the player at the top level.

The search time is now reduced to about five minutes. Note the small number of database misses. In a position where stones already have been captured, there will still be many misses.

	old	databases	move ordering
Total search time [s]	2468	1148	328
Speedup	1.0	2.15	7.52
Disk access time [s]	2012	186	105
Relative to total [%]	81	16	32
Search tree size	2017853	4218225	1100258
Relative to 'old' [%]	100	209	55
Database misses	379776	186808	18
Database hits	1069072	2749546	560411
File accesses	1069072	8115	4968
Hashtable accesses	340775	529696	287248
Hashtable hits	249069	385202	235064

7 Minor improvements

7.1 Code optimization

Modern compilers provide options for code optimization. Alas, the compiler used for this thesis neither offered any means for automatic code optimization, nor did it make any of the simplest and most straightforward optimizations, as for example replace a 'DIV 2' operation by a 'shift right'.
After profiling the search algorithm, some procedures were optimized manually for better performance. This included the following steps:

- loop unrolling of loops with a constant number of steps
- storing values computed at high cost for later reuse
- storing values of simple functions in tables, thereby replacing a function call by a table access
- instruction strength reduction, especially replacements of 'DIV 2^n' by 'shift right n'

7.2 Hashtable

The old opening prover used a hashtable to store computed values at the 16th ply to avoid repeated access to the value of the same position (Figure 1.2).
A few measurements showed that better results can be achieved when hashing values on every ply level from 11 to 17.

7.3 Result

	old	move ordering	optimizations	hashtable
Total search time [s]	2468	328	281	232
Speedup	1.0	7.52	8.78	10.64
Disk access time [s]	2012	105	105	105
Relative to total [%]	81	32	37	45
Search tree size	2017853	1100258	1100258	646737
Relative to 'old' [%]	100	55	55	32
Database misses	379776	18	18	18
Database hits	1069072	560411	560411	361041
File accesses	1069072	4968	4968	4968
Hashtable accesses	340775	287248	287248	285514
Hashtable hits	249069	235064	235064	98523

8 Results

Figure 8.1 shows the new organisation of the opening search program, see Figure 1.2 for the organisation of the old search program. The most important changes are, that perfect values can now be computed in the whole search tree and that more databases are available.

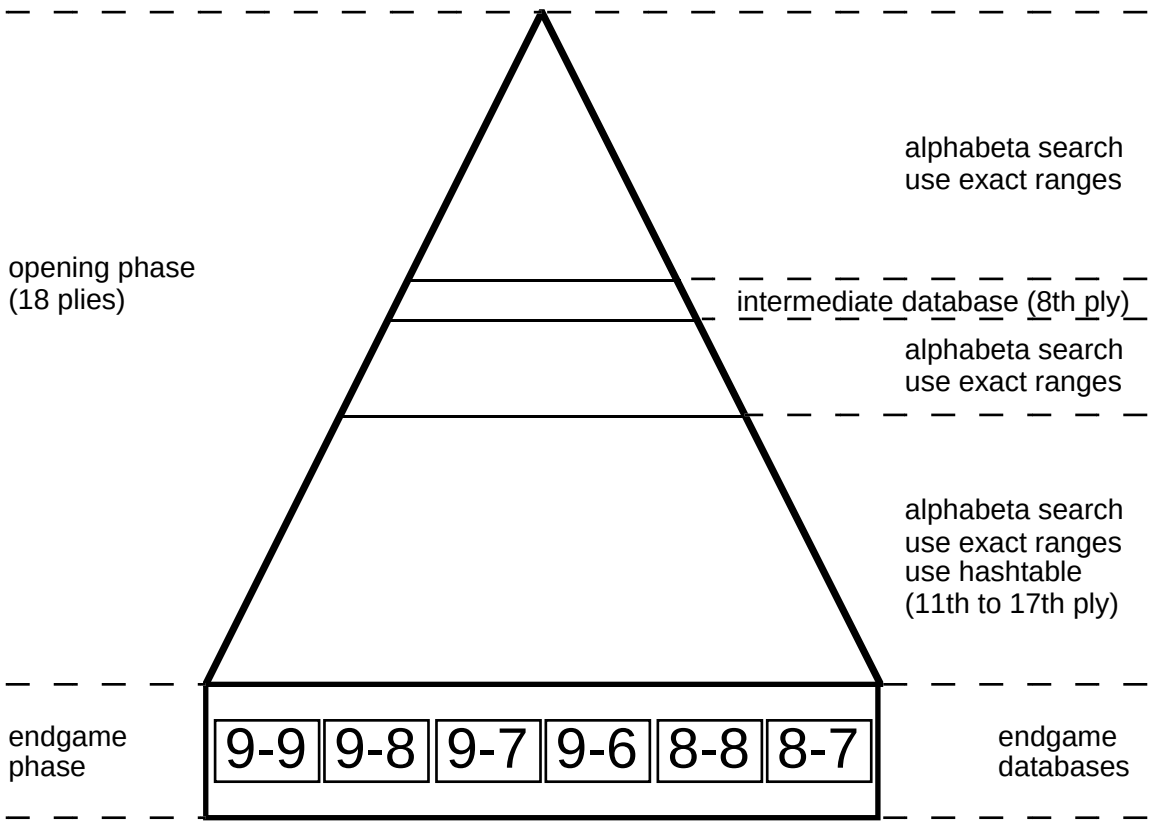


Figure 8.1: the new organisation of the opening search program (see Figure 1.2)

After the implementation of value ranges and after including additional databases, the intermediate database can be recomputed and filled up with more exact values ranges. The two original databases, which contained lower bounds for Black and lower bounds for White respectively, were merged together to use the old values where possible. Because the computation of all values in the database would take too long, a mechanism was implemented to allow incremental computation.

The required time for a search depends mostly on the ply number at which it is started. Figure 8.2 shows a qualitative plot of search time against ply number. The hardest moves are those from the eighth ply, because moves from the first to the seventh ply can use the intermediate database, and an eighth ply search must go down to the endgame databases.

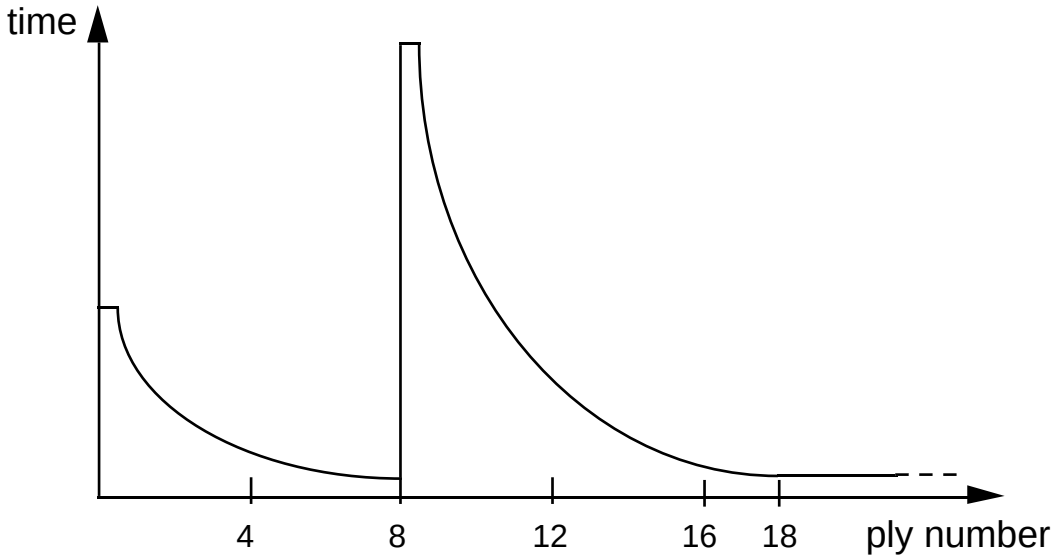


Figure 8.2: qualitative plot of search time versus ply number

The following table repeats the results of the measurements.

	old	buffering	value ranges	databases	move ordering	final
Total search time [s]	2468	559	16568	1148	328	232
Speedup	1.0	4.41	0.14	2.15	7.52	10.64
Disk access time [s]	2012	130	309	186	105	105
Relative to total [%]	81	23	1.9	16	32	45
Search tree size	2017853	2017853	73006119	4218225	1100258	646737
Relative to 'old' [%]	100	100	3618	209	55	32
Database misses	379776	379776	17216242	186808	18	18
Database hits	1069072	1069072	49876901	2749546	560411	361041
File accesses	1069072	5786	12775	8115	4968	4968
Hashtable accesses	340775	340775	11696076	529696	287248	285514
Hashtable hits	249069	249069	10507311	385202	235064	98523

Traps & Swindles

This section examines methods to improve perfect playing programs. A model to measure the difficulty of a position is introduced in chapter 10, and implementations of it are described and compared in chapters 11 and 12.

9 Overview

Much effort has gone into solving games. After a game is solved, it is possible to play the game perfectly. This means that the computer player is always able to attain the game-theoretic value of the actual position. In a stronger sense this can mean that the computer player can move so that he can win with a minimum number of moves, or, in lost positions, lose with a maximum number of moves. There is still some potential for improvement assuming that the opponent is a human player or a heuristic program. Because of their limited knowledge of the values of the positions, they may blunder now and then. How this can be exploited by a perfect player depends on the value of the position he has to move from.
If the perfect player already has achieved a won position, a blunder by his opponent will result in a win within a lower number of moves. This can be decisive in games where limits in the number of moves are given, as for example the 50 moves rule in chess.
If the perfect player is in a drawn position, a blunder of his opponent would increase the game-theoretic value to a win. Note that there may not be any distance information associated with drawn positions, because opponents agree on a draw only after repetition of positions already taken up.
If the perfect player is in a lost position, there are two kinds of blunders his opponent may make, he may increase the game theoretic value to a draw or even to a win for the perfect player.

9.1 Definition traps and swindles

How can the perfect player increase the probability that his opponent blunders? A number of general strategies have been proposed to exploit additional knowledge from databases, or additional knowledge resulting from a deeper search horizon [3][5].
For example, two strategies are proposed for a player who can search deeper than his opponent [5], said strategies are called 'traps' and 'swindles'.
A trap is a position where the opponent has a move that looks good at shallow depth, but is revealed as bad at greater depth. Whether a position is a trap or not is strongly related to the difficulty of a position (as seen from the viewpoint of the heuristic player). The perfect player has to identify such positions and must try to steer the game in their direction.
A swindle is a move that looks good for the perfect player at shallow depths, but turns out bad at greater depth (against the opponent's perfect play). By definition, a swindle is a move of the perfect player which does not attain the game-theoretic value, i.e. it is not a perfect move.

9.2 Implications of a proven draw in Nine Men's Morris

In Nine Men's Morris, the perfect player will never lose a game, unless he swindles. This thesis concentrates on strategies for drawn positions, assuming that the perfect player will not take any risks. A lost position for the perfect player will therefore not be encountered, and in case he achieves a win, there is no need for using traps or swindles because no limit on the number of moves exists for Nine Men's Morris.

10 General model

The definition of traps suggests that the difficulty of a position must be defined. This chapter introduces a general mathematical model to measure the difficulty of a position.

10.1 The perfect player

Let PP be the perfect player and HP his opponent, the heuristic player. Assuming PP is to move next and the value of the position is a draw, PP has u perfect moves (u>=1), and an arbitrary number of losing moves. As PP will not take any risks, the losing moves can be discarded immediately.
PP now needs some means to decide which of his moves produces the most difficult position for HP. Assuming we know the probability m_i HP will make a perfect move for the position after PP's i-th move (1<=i<=u), then PP must choose the move k so that $m_k <= m_i$ for all i.

Rule for PP: choose move k so that $m_k = \min m_i$ (1<=i<=u)

10.2 The heuristic player

How can we compute the m_i's? HP has a total of v moves, where at least one is drawn and some may lose, but he does not know the exact value of them. Let's introduce a variable d_{ij} = 1 if the j-th move of HP is perfect and d_{ij}=0 if not. Assuming we know the probability p_{ij} that HP plays the j-th move after PP has played the i-th move, then m_i becomes

$$m_i := \sum_{j:=1}^{v} p_{ij}\, d_{ij}$$

Figure 10.1 shows a subgraph of the search tree and the propagation functions.

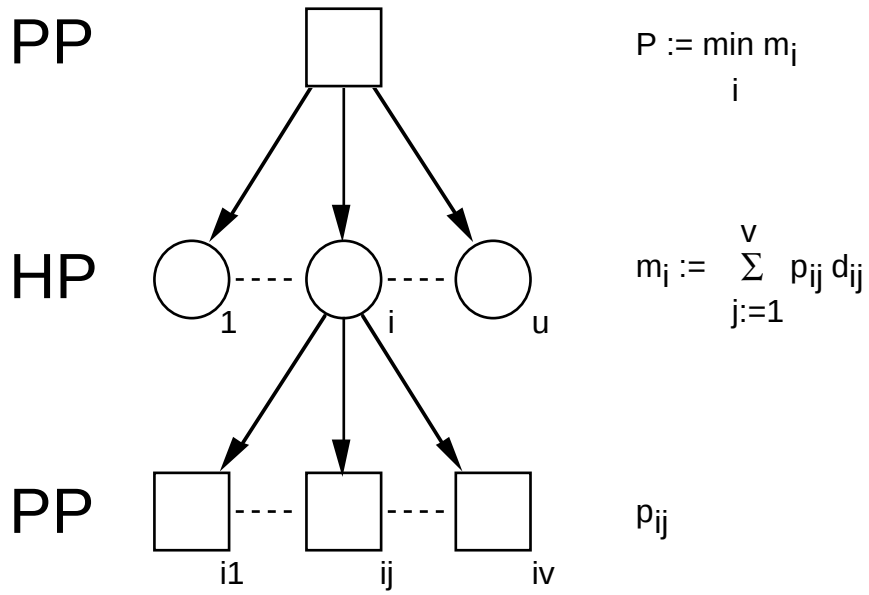


Figure 10.1
The overall evaluation function for PP becomes

$$P := \min_{i} \left(\sum_{j:=1}^{v} p_{ij}\, d_{ij} \right)$$

Applying this function to PP's nodes in the next deeper level of the search tree (Figure 10.1), we can write the evaluation function recursively as

$$P^{(0)} := 1$$

$$P^{(n)} := \min_{i} \left(\sum_{j:=1}^{v} p_{ij}\, d_{ij}\, P_{ij}^{(n-2)} \right) \qquad ; n= 2, 4, 6, ...$$

10.3 Implications

The values of the d_{ij}'s are known to PP, because they can be computed from the endgame databases.
The values of the pij's are known only if the heuristic function of HP is known. In this special case, PP must evaluate P(2), P(4), P(6), ... until he finds a P(n)=0, which means that he found a move sequence where he knows HP will blunder at the n-th ply.

In general, the heuristic function of HP is not known, so PP in turn needs a heuristic function to compute approximate values for the pij's.

11 Implementations of the model

11.1 Random player

For the first implementation we assume that HP is a random player. This implies that the move probabilities are all equal, if HP has v moves then $p_{ij}=1/v$ (Figure 11.1).

m_i = number of perfect moves/total number of moves

The only information PP needs to compute m_i are the game-theoretic values of the positions of the successor nodes.

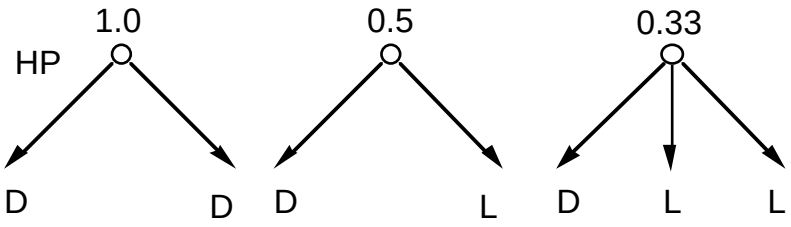


Figure 11.1: numerical examples for m_i in nodes with two and three successors

11.2 Distance dependent move selection

Two kinds of information are available from the endgame databases as computed by SearchBench. In the 'Random' implementation, only the game-theoretic value was used to generate traps for the opponent. But for positions with a game-theoretic value of win or loss, also the maximum distance to win and the minimum distance to loss are stored, where distance means the number of moves. Positions with a game-theoretic value of draw have no distance information stored, because a drawn game will go on infinitly, or until a repetition of positions occurs.
Can this additional information be exploited to find better moves? We assume that HP, to find his next move, uses some kind of a search function with a limited horizon. The search depth of the search function is unknown, but we assume to have a lower bound L and an upper bound U for the search depth of the opponent. Then the probability that HP plays a move which would be a loss in L or fewer moves is zero. The probabilities of moves with a distance equal or larger than U are assumed to be equal to the probability to play a drawn move (or a drawn move can be looked at as losing in an infinite number of moves).

Figure 11.2 shows an algorithm for computing m_i, where D means distance to loss. For drawn moves, p_{ij} is set equal 1, the last step scales the sum of the p_{ij}'s to 1.
Figure 11.3 gives some numeric examples, assuming L=2 and U=12.

```
FOR j:=1 TO v DO
        IF dij=0 THEN
                IF D>=U THEN
                        Sum:=Sum+1
                ELSE IF D>L THEN
                        Sum:=Sum+(D-L)/(U-L)
        ELSE BEGIN
                Sum:=Sum+1
                CountPerfect:=CountPerfect+1
        END
mi:=CountPerfect/Sum
```

Figure 11.2

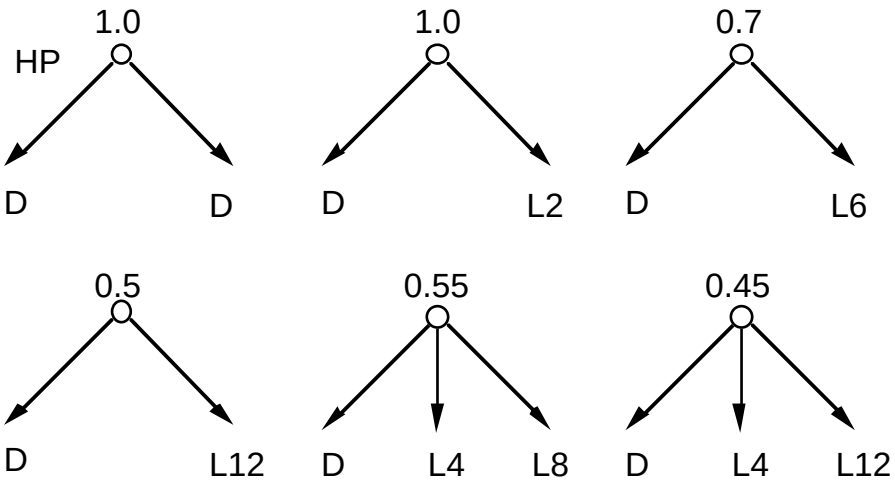


Figure 11.3: numerical examples for L=2 and U=12

12 Experimental tests

To compare the two implementations of the model, a set of test positions with a value of draw was randomly selected from the databases. A sequence of four moves was played from these positions, PP making the first move.
The performance of the implementation is then measured as the number of wins achieved against a heuristic player.

12.1 The perfect players

The two implementations of the general model described in the previous chapter are compared with each other and with a perfect player using no traps, i.e. choosing the first perfect move found in the databases.

12.2 The heuristic player

The opponent of the perfect players is a heuristic Nine Men's Morris program called Bushy[1]. It is implemented on the Smart Game Board [4], a general graphical user interface for games.

Bushy is the best heuristic Nine Men's Morris program I am aware of. In 1990 it challenged the British champion in a six game match and won 5-1 (4 wins, 2 draws). In 1991 it was the winner of the Nine Men's Morris tournament at the Computer Olympiad.

Bushy is also incorporated into a Nine Men's Morris playing robot, which is on display at the Technorama in Winterthur, Switzerland. Bushy is reported to have a good score against the museum visitors.

12.3 Selection of test positions

From each of the databases 3-3, 4-3, 4-4, 5-4 and 5-3, twenty random positions with a game-theoretic value of draw were selected. Note that the average number of losing successor nodes is not the same for different databases Figure (12.1). For some databases, the probability of hitting a position with losing successors is extremely low.

	won	lost	drawn
3-3	47167	9659	96
4-3	177678	3095	1340023
4-4	159	29	3225409
5-3	586961	9677	4564142
5-4	9940	1518	20609534

Figure 12.1

12.4 Results

SearchBench - Bushy	no Traps	Random (2)	Distance (2)
3 - 3	55	55	75
4 - 3	0	0	0
3 - 4	30	80	70
4 - 4	0	0	0
5 - 3	0	0	0
3 - 5	0	20	30
5 - 4	0	0	0
4 - 5	0	0	0

Figure 12.2

Figure 12.2 shows the result of the tests. The column on the left gives the number of stones of the perfect player and Bushy respectively at the beginning of the test. The other three colums contain the percentage of games the perfect players won against the heuristic player.

The perfect players achieved wins only when they had three stones, i.e. in the endgame phase. Within the other databases, the probability of finding losing successors seems to be too small to make the heuristic player blunder.

Nevertheless, it seems to be better to search for traps than to play just any perfect move, since the Random and Distance players always performed at least as well as the perfect player using no traps. The difference between Random and Distance is not so clear, because none of them seems to be absolutely superior to the other.

The results are not very useful, since they depend more on the chosen databases than on the chosen implementation of the perfect player.

Conclusions

In addition to what SearchBench provides, the new Nine Men's Morris user interface allows the user to play games both in the opening and the endgame phase, to set up arbitrary positions in the opening and the endgame, and to analyse positions by displaying the values of the moves on the board.

The quality of the values returned from the opening search program has been improved by introducing value ranges. For most positions reached with 'reasonable' moves the computed value will be the perfect value, otherwise a lower and an upper bound are returned.
Speed and quality of the opening search program can be improved by making additional databases available. I strongly recommend to add at least one level of databases (9-5, 8-6, 7-7).

A model was introduced to measure the difficulty of positions. After making a few assumptions about the heuristic player (random player, player using search function with limited search depth), two implementations of the model and the standard move selection were compared in play against a heuristic program. For positions in some endgame databases the new move selection performed better while it made no difference in others.
A better way to compare the performance of the implementations would be to measure the correlation between the computed difficulties of positions and the error probability of the heuristic program. For example, all positions of the endgame could be classified by their difficulty, and then the positions of a certain difficulty range, say 0.8 to 0.89, could be presented to a heuristic player. The better the values for the computed difficulties of the positions are, the closer to 0.875 will the average error probability of the heuristic player be.
Because of the large size of the databases, this kind of measurements could not be done within the given time limits.

Appendices

A Rules of Nine Men's Morris

background: Nine Men's Morris (NMM) is a board game for two persons. It belongs to the family of 'three in a row' games, where both players aim to arrange three of their stones in a row on adjacent fields. The board contains 24 fields, represented as the crosspoints in Figure A.1. For details on game history, strategies and rule variants see [6].

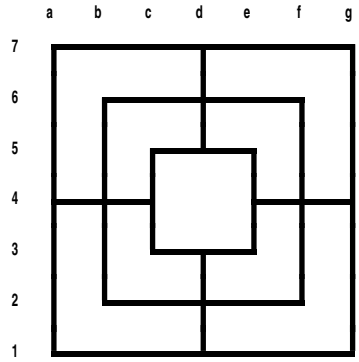


Figure A.1: The empty board with the coordinate system for game annotation

terminal positions: The game ends with either a draw or a loss for the player to move. A game is drawn by repetition of positions, a player loses when he has fewer than three stones or no valid move.

closing a mill: A configuration of three stones is called a mill (Figure A.2). The white stones do not represent a mill, because they are not connected by edges. The act of putting the third stone into the row is called 'closing a mill'. The player who closes a mill captures one stone of the opponent. If possible, a stone which is not part of a mill must be removed. If two mills are closed at once, only one stone is captured.

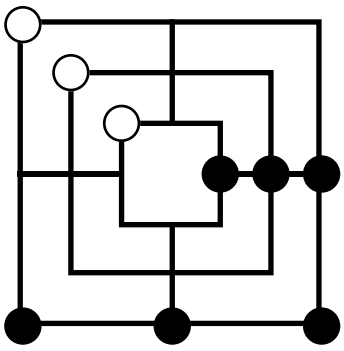


Figure A.2: Two black mills

opening: The game starts with an empty board, and both players have nine stones. White begins and then both players move alternatingly. During the opening phase, a move consists of putting a stone onto an empty field on the board. Since both players start with nine stones, the opening phase lasts exactly 18 plies.

midgame: All positions after the opening where both players have more than three stones belong to the midgame phase. Both players move by sliding a stone of their color to an adjacent, empty field.

endgame: Endgame starts as soon as one player has only three stones on the board. The player who has still more than three stones moves as in the midgame phase, while players with exactly three stones are allowed to 'jump' with one of their stones to any empty field on the board.

B Nine Men's Morris user interface

This appendix describes the Nine Men's Morris user interface. The program is implemented on top of SearchBench [2], a tool for endgame database computation and handling. Here only the part of the user interface specific to Nine Men's Morris is described.

B.1 The NMM menu

Edit Mode / Play Mode: Here the user can choose between edit and play mode. Edit mode is for position setup and allows the user to move the stones in arbitrary ways. In play mode, only valid moves according to the rules in appendix A may be carried out.

No Search / Search All / Search First: These three commands tell the search algorithm what to do if a value needed from the intermediate database is not yet evaluated.
Since the computation of all values in the intermediate database takes very long, a mechanism for incremental evaluation was implemented.
With 'No Search' selected, the values of all positions not yet evaluated are set to 'Unknown'.
With 'Search All' selected, the algorithm starts evaluation of the needed values not found in the database, adding the new values for future use.
With 'Search First' selected, only the first unknown value in the database is computed and updated.
The evaluation of a single position at the eighth ply may take a few minutes, so the user will prefer 'No Search' or 'Search First' to play the game.

No Traps / Random / Distance: These commands affect the way search is performed in mid- and endgame positions.
With 'No Traps' selected, the move command returns the first move which attains the game-theoretic value of the position.
With 'Random' selected, the search algorithm assumes that its opponent is a random player.
With 'Distance' selected, the search algorithm assumes that its opponent uses a search function with limited search horizon. A more detailed description of the 'Random' and the 'Distance' algorithm can be found in chapter 11.

Full Search / Draw Only: There still are positions for which the opening search algorithm takes too long to find a perfect move. With 'Draw Only' selected, the search algorithm only tries to prove a draw. This results in a considerable speedup, but the program may miss a possible win. For perfect play, 'Full Search' must be selected.

B.2 The Problem Display

The 'Problem display' window is responsible for the representation of the game board and for user interaction. It provides buttons for program control and displays the state of the game.

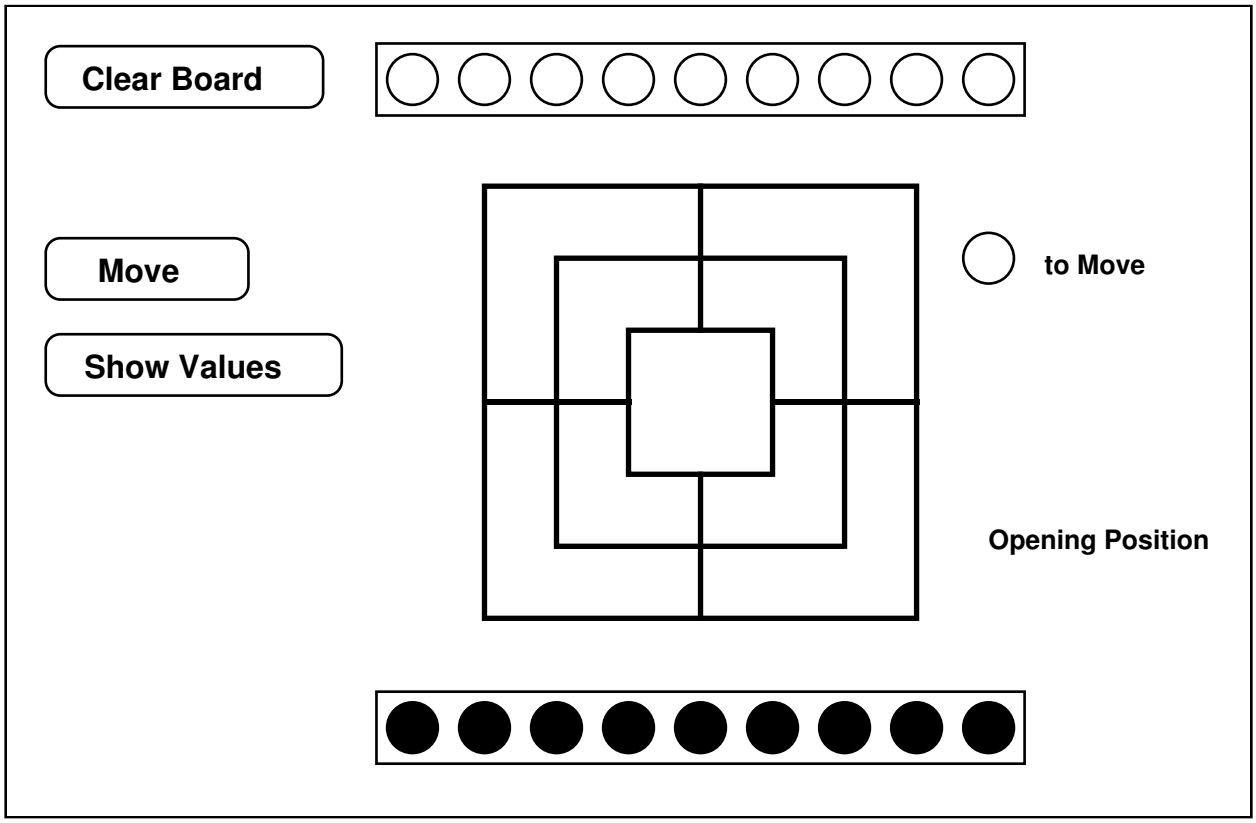


Figure B.1: The 'Problem Display' window

the buttons: the 'Clear Board' button empties the board and sets White to move. This can also be done by hand in edit mode. The 'Move' and the 'Show Values' buttons start a game tree search with the actual position as an initial position. The 'Move' button searches for a move either by starting the opening search or by starting a trap search in the endgame. After the successful completion of the search the best move is automatically carried out on the board. The 'Show Values' button also starts the proper search routine, but instead of carriying out a move, the values of all moves are displayed on the board. This command is a powerful means for game analysis and learning.
Chapter B.3 gives some example diagrams for the 'Show Values' command and explains how to interpret the results.

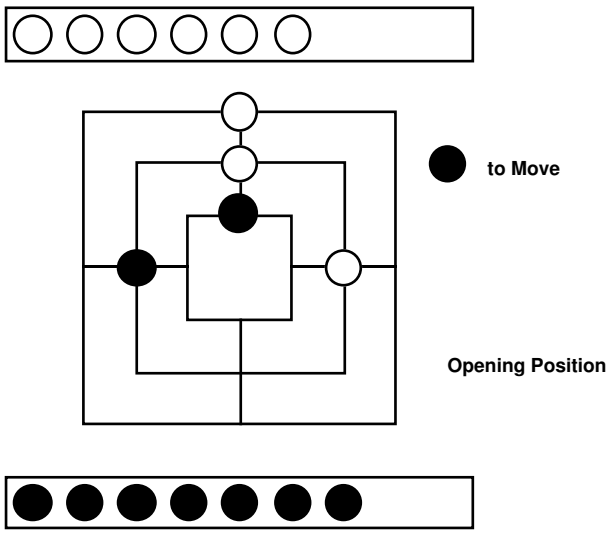
the board: besides a graphical representation of the board, the pits of black and white stones, the player to move and, if available, the game-theoretic value of the actual position are displayed. The game-theoretic value is known only after a search has been performed and is lost as soon as a move is made or the board is edited. The 'To Move' icon is toggled when a move is made, it can be edited by the user in edit mode. The pits show how many stones are still to be played (opening phase only) and how many stones have been captured. The captured stones are aligned to the right while the stones which are still to be played are aligned to the left.
(On the empty board, the stones in the pits are aligned both to the left and the right side. But they were all captured in the last game, and they are all to be played in the next game, so they just go round and round and round...).

how to move: the computer generated moves are carried out automatically. The user moves by clicking on a stone and moving it to the new field. When a mill is closed, an opponent's stone can be captured by clicking on it. Moves that are against the rules are ignored.

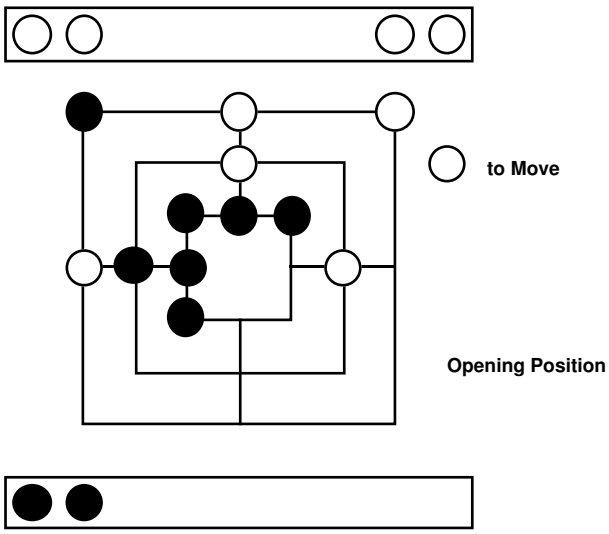
how to set up a position: to analyse specific positions, a setup function is provided. After selecting 'Edit' command in the 'NMM' menu, the stones on the board and in the pits can be moved to arbitrary positons by clicking on them and dragging them to the desired place. To move a stone to the pits, the user must drag the stone either to the left or the right end of the pit to distinguish between captured stones and stones which are still to play. To change the color of the next player to move, just click on the 'To Move' icon.

B.3 Examples

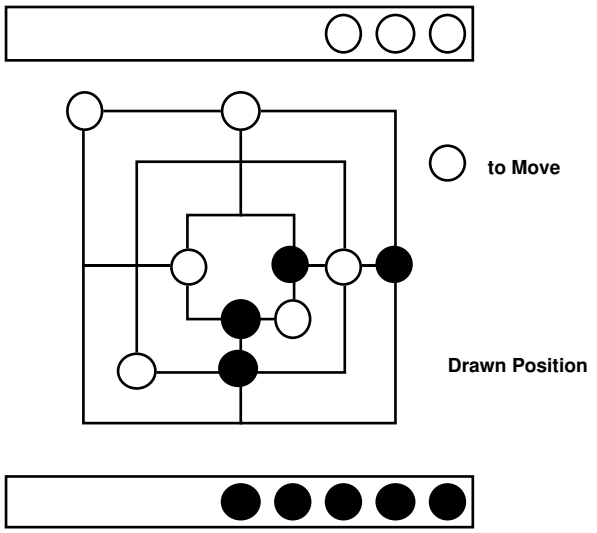
The following diagrams demonstrate the user interface and how it can be used as a powerful game analysis tool.



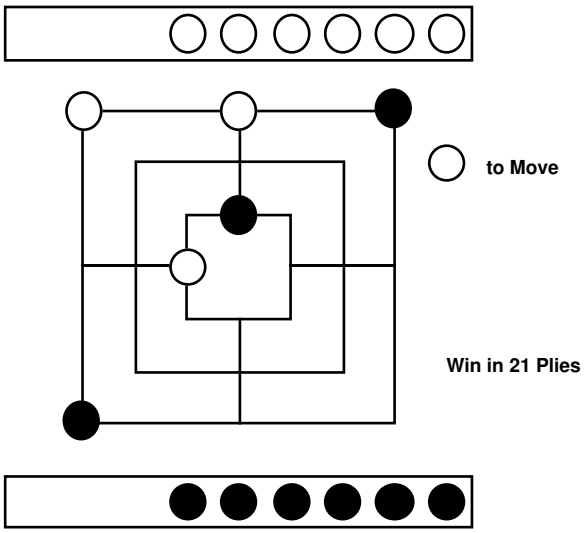
A position in the opening after the third move of White. White has six and Black has seven stones to play with Black to move.



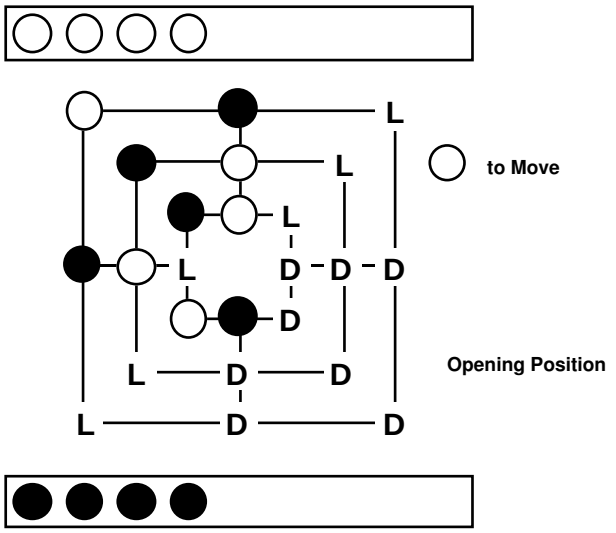
Another opening position. Black has two mills and has captured two stones.



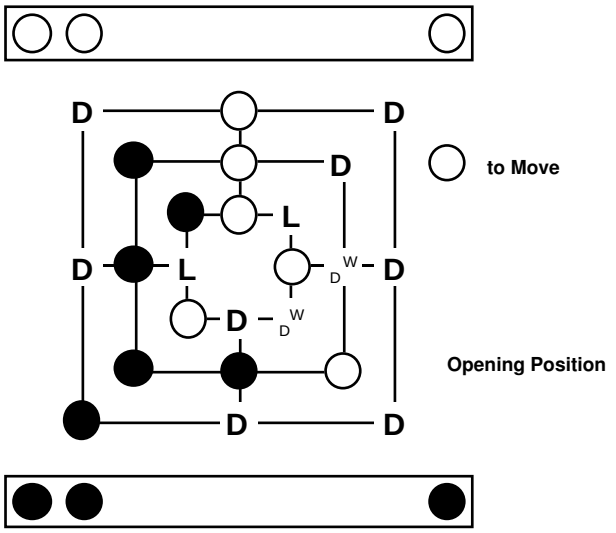
A midgame position where White is two stones ahead. The game-theoretic value is a draw.



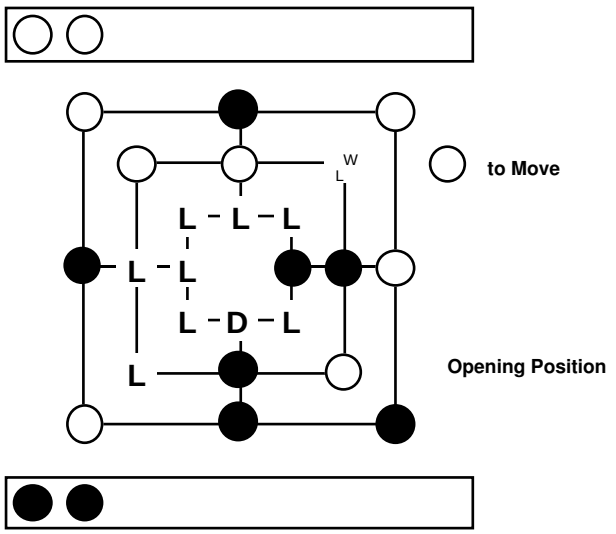
An endgame position where White has a forced win in 21 plies.



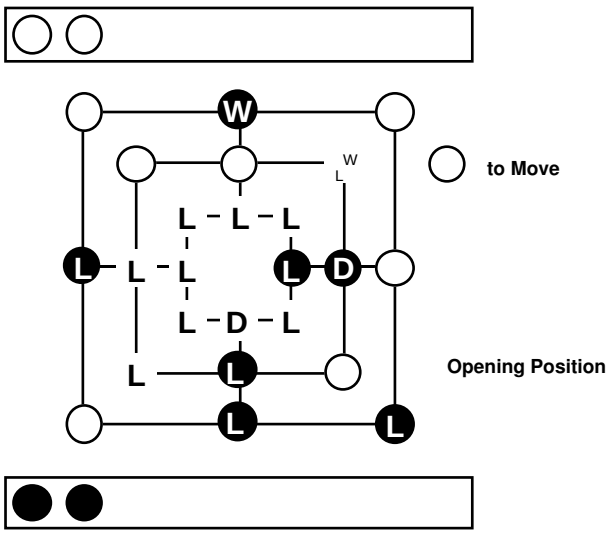
An opening position after the execution of the command 'Show Values'. A move to a field marked with 'D' is a proven draw for White, an 'L' is a proven loss and a 'W' a proven win.



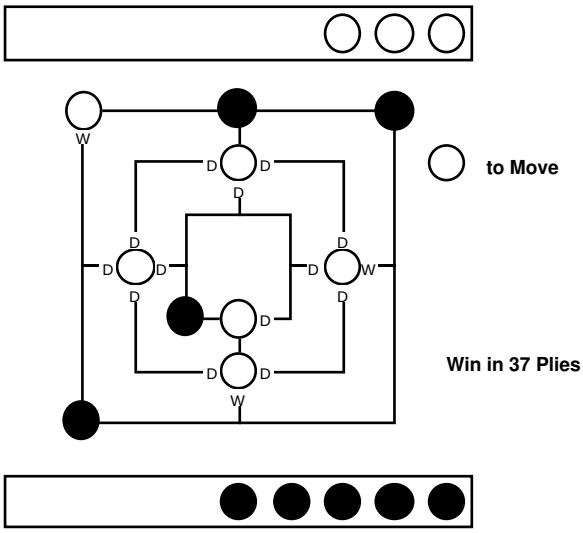
Another opening position. For two moves the exact value could not be found, because not all endgame databases were accessible. In such a case, a value range is displayed within which the exact value must be. Both moves are 'at least drawn'. Other possible value ranges are 'at most drawn' and 'unknown'.



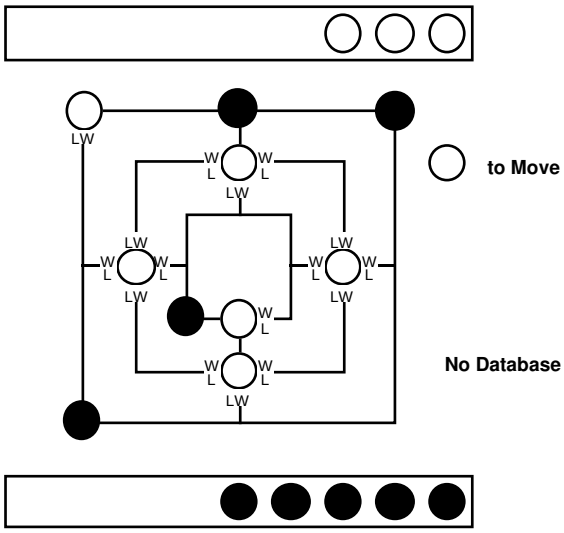
An opening position where White has a move which closes a mill. Because the values for capturing moves may depend on which stone is captured, these moves are marked with value ranges. The following example explains how to display the values of the captured stones.



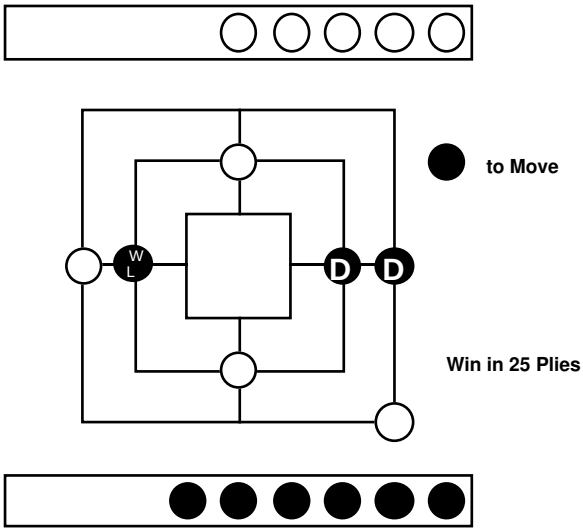
The same position as in the previous example. To display the values of the captured stones, click the mouse button as if to carry out the capturing move and keep the mouse button pressed. If the mouse button is released, the move is carried out, if the mouse pointer is dragged away and then released, the position remains unchanged.



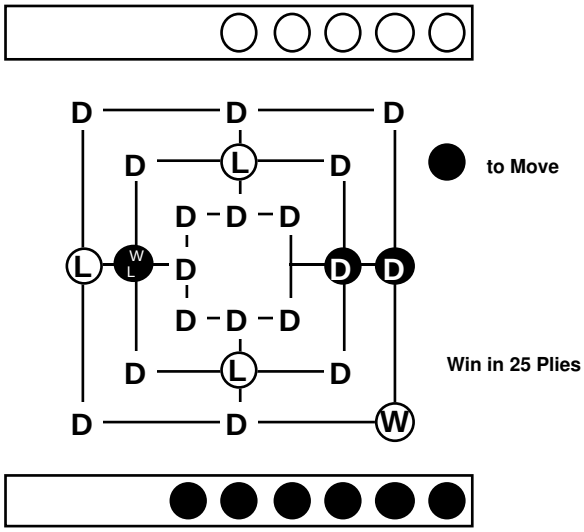
A midgame position after the execution of 'Show Values'. The move values are displayed in the direction the stones have to be slid.



For mid- and endgame positions the user must explicitly load the necessary databases by executing the 'Set File' command in the 'View' menu. Otherwise the values are 'Unknown'.



An endgame position. Because the player to move can 'jump' with any stone to any position, the exact values can only be seen after clicking on the desired stone.



The same position after clicking on the black stone marked as 'unknown'. The values are shown for every field it can be moved to. The values for the capturing move are shown on the white stones.

Bibliography

[1] R.Gasser, Heuristic Search and Retrograde Analysis: their application to Nine Men's Morris, Diploma thesis, 1990

[2] R.Gasser, PhD, to appear

[3] P.J.Jansen, Using Knowledge about the Opponent in Game-Tree Search, Ph.D. Thesis, Carnegie Mellon University, 1992

[4] A.Kierulf, Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies, PhD. Thesis, ETH Zürich, 1990

[5] J.Nievergelt, Heuristisches Wissen und Suchen am Beispiel: Computerspiele, Vorlesungsskript, SS1992

[6] H.Schürman, M.Nüscheler, So gewinnt man Mühle, Otto Meier Verlag Ravensburg, 1980