seit 1558

Randomized Evaluation Functions in Single Agent Search

DIPLOMARBEIT
zur Erlangung des akademischen Grades
Diplom-Mathematikerin

FRIEDRICH SCHILLER UNIVERSITÄT JENA
Fakultät für Mathematik und Informatik

eingereicht von Susanne Heuser
geb. am 25.09.1978 in Stuttgart/Bad Cannstadt

Betreuer: Prof. Dr. Ingo Althöfer

Jena, den 13.08.2003

Contents

1 Introduction **6**
- 1.1 Related Research . . . 6
- 1.2 Thesis Structure . . . 7

2 Overview **8**
- 2.1 The Puzzle *SlideThree* . . . 8
- 2.2 Search Algorithm . . . 9
- 2.3 Evaluation Functions . . . 10
- 2.4 Randomization . . . 11
- 2.5 Experiments . . . 13
  - 2.5.1 Choice of Moves Experiments . . . 13
  - 2.5.2 Autoplay Experiments . . . 14

3 The Puzzle *SlideThree* **15**
- 3.1 Rules . . . 15
- 3.2 Properties . . . 16
  - 3.2.1 Movement of the Blanks . . . 16
  - 3.2.2 Successors . . . 17
  - 3.2.3 Number of Positions/Solvability . . . 17
- 3.3 Building a *SlideThree* Database . . . 22
  - 3.3.1 Backward Analysis . . . 22
  - 3.3.2 Making the Database Random Access . . . 23

4 Methods of Empirical Research **28**
- 4.1 Search Algorithm . . . 28
  - 4.1.1 Depth-t-Search . . . 28
  - 4.1.2 Including a Transposition Table . . . 29
- 4.2 Evaluation Functions . . . 31
  - 4.2.1 Definitions . . . 32
  - 4.2.2 Examples and Remarks . . . 33
  - 4.2.3 Properties . . . 36
- 4.3 Randomization . . . 37

5 Experiments **39**
- 5.1 Choice of Moves Experiments . . . 39
  - 5.1.1 Experiment Design . . . 39
  - 5.1.2 Conducted CoM Experiments . . . 42
- 5.2 Autoplay Experiments . . . 43
  - 5.2.1 Previous Results . . . 43
  - 5.2.2 Statistical Model . . . 43
  - 5.2.3 Experiment Setup . . . 45

6 Results **48**
6.1 Choice of Moves 48
6.1.1 Ranking of Evaluation Functions 48
6.1.2 Randomization Experiments 49
6.2 Autoplay 49
6.2.1 Ranking of Evaluation Functions 49
6.2.2 Intensity of Randomization 50
6.2.3 Different DTWs 52
6.3 Summary 53

7 Bug Report **54**

8 Conclusions **57**
8.1 Conclusions from Experiments 57
8.2 Future Research 57

Bibliography **59**

A CoM Experiments **62**

B Autoplay, DTWs 14 and 24 **64**

C Autoplay, Start Sample **67**

D Autoplay, DTWs 8 to 18 **69**

E TileTab **71**

F Rounded Evaluation Functions **72**

G Program Descriptions **73**
G.1 makesample.exe 73
G.2 comexp.exe 73
G.3 autoplay.exe 75
G.4 autoplayIA.exe 77

H SlideThree **78**

I *Zillions* **playing** *SlideThree* **80**

Zusammenfassung

Diese Diplomarbeit beschäftigt sich mit der Frage, ob deterministische heuristische Bewertungsfunktionen durch Randomisierung verbessert werden können. Heuristische Bewertungsfunktionen werden eingesetzt, um bei einer begrenzten Suche in Spiel- und Entscheidungsbäumen mögliche Alternativen zu bewerten.
Die im folgenden dargestellte Forschung wurde anhand des Ein-Personen-Spiels *SlideThree* durchgeführt, eines Schiebepuzzles ähnlich dem bekannten 15er-Puzzle. Das Puzzle und seine Eigenschaften werden in Kapitel 3 ausführlich beschrieben. Als zentraler Punkt werden die Lösbarkeitseigenschaften von Stellungen diskutiert.
Die weiteren Kapitel befassen sich mit den Details der verwendeten Methoden und der durchgeführten Experimente. Der verwendete Algorithmus für die Spielbaumsuche ist heuristische Tiefe-t-Suche. Dabei wird der Baum bis zu einer festen Tiefe t durchsucht.
Zur Bestimmung des Einflusses von Randomisierung auf die Spielstärke deterministischer Bewertungsfunktionen wurden zwei Ansätze entwickelt und im Experiment umgesetzt. Im ersten Ansatz wird die Spielstärke anhand der Zugauswahl gemessen, im zweiten anhand der Gewinnquote einer Anzahl von Spielen mit begrenzter Zuganzahl.
Aufgrund der durchgeführten Experimente konnte nachgewiesen werden, dass sich Randomisierung als Mittel zur Verbesserung von heuristischen Bewertungsfunktionen in Ein-Personen-Spielen eignet. Allerdings muss die deterministische Bewertungsfunktion bereits eine gewisse Spielstärke besitzen, damit das Verfahren erfolgreich ist.

Acknowledgements

First, I would like to thank my supervisor, Prof. Ingo Althöfer for his guidance and help. He managed to get me excited about my work while keeping me on track at the same time.
Furthermore, I owe thanks to the Institute for Knowledge and Agent Technology (IKAT) in Maastricht and its head, Prof. Jaap van den Herik, for accepting me as an exchange student. The three months I spent there were most beneficial for my work on this thesis. Special thanks to my supervisor there, Jos Uiterwijk, for his support and help, and to Levente Kocsis and Jeroen Donkers for many expert comments on my work and fruitful discussions.
I want to thank also Dr. Günter Schorr and the Fakultätsrechenzentrum of the Friedrich Schiller Universität Jena for providing the computing power without which the extent of the empirical research presented in this thesis could have never been achieved.
Finally, I thank my family, especially my parents, for their support and Iman for his continuous love and understanding.

Chapter 1

Introduction

Many people like to play games. And lots of software is written in order to enable computers to play games. The standard method of these programs is to generate as many possibilities for the next few moves of the game as possible. In other words, a part as large as possible of the game tree rooted at the current position is built up. Then, the best move is identified by evaluating the possible future positions. This evaluation function is usually heuristic, that means it only estimates the value of a position by some properties. Generally, there are two ways to improve a game playing program. First, the search algorithm can be improved. That is, the tree is searched deeper or more efficiently by identifying and pruning unpromising alternatives. Second, the evaluation function can be improved. If it can be made a better estimator of the quality of a position, surely the program will play better.
In this thesis, the second way is investigated. We will answer the question whether randomization can improve heuristic evaluation functions in single agent search.

1.1 Related Research

Up to now, research in randomized evaluation functions has been conducted for two player games.
D.F. Beal and M.C. Smith investigated this topic for chess (cf. [4]). They used random values as an evaluation function in a minimax search of fixed depth. They discovered that this plays better than random move selection. Moreover, simple evaluation functions for chess positions could be improved by adding a random value to them. They suggested that the reason for this behaviour is, that randomization tends to prefer the positions with the most possible moves.

T. Rolle describes in his diploma thesis the effects of random increments to deterministic evaluation functions for several simple two player board games (cf. [12]). His results varied depending on the game and the nature of the search that was used. Rolle suggests in his thesis, that randomization leads to less alpha-beta cutoffs during the search and thus makes the tree search more inefficient.

This thesis intends to extend the research in this field to single agent search.

1.2 Thesis Structure

The thesis is divided into 8 chapters and an appendix.
The block consisting of Chapters 2, 6, 7, and 8 can be read independently from the rest and forms a "short version" of the thesis. It gives a general overview of the work done without providing every specific detail.

Chapter 2 gives a brief overview of the research conducted.
Chapters 3, 4, and 5 represent the main body of the thesis. Here, the subject of research and the methods that were used are lined out. Chapter 3 refers to the Puzzle *SlideThree* and its properties. Especially the solvability properties of the game are discussed. In addition, it is explained how the puzzle was solved with the help of the computer. In Chapter 4 the methods used for the empirical research are explained. The search algorithm, evaluation functions and randomization techniques are presented. The experiments are described in Chapter 5. Two approaches to answer the research question and their realization as experiments are explained. Also, a discussion about the statistical significance of the results is contained in this chapter.
Chapter 6 summarizes the results that answer the research question.
Unfortunately, a bug occurred in the programs written for running the experiments. Due to a lack of time, it was not possible to redo all affected experiments. Chapter 7 explains the nature of the bug and how it influenced the results presented in Chapter 6. Finally, Chapter 8 presents the conclusions that can be drawn from the conducted research. It also relates what future research in this area should be done.
In the appendix the handling of the programs for running the experiments is explained. All programs are written in C++ and run on Windows. Furthermore, some more results of the experiments are shown in graphs and tables. Finally, a view is taken on how well Zillions of Games, a software playing multiple games and puzzles, plays *SlideThree*.
The thesis also includes a CD-ROM. On this CD, all the data from the experiments can be found as put out by the programs and assembled into tables and graphs. It contains the programs, database, and samples that were used for running the experiments. The programs are given as executable files and C++ source code. Also included on the CD is the thesis in PS and PDF format.

Chapter 2

Overview

This chapter gives a brief overview of the conducted research. It starts with a section on *SlideThree*, the game chosen for research. Sections 2.2 and 2.3 describe the algorithm and evaluation functions used for the empirical investigations. Section 2.4 refers to the why and how of randomization. The experiments are explained briefly in Section 2.5.

2.1 The Puzzle *SlideThree*

As subject for research, a single player game was chosen. That is why the game tree search is called *single agent search*. The game is called *SlideThree* and is a sliding tile puzzle similar to the well known *15 Puzzle* (cf. [1]).
It was invented and programmed for *Zillions of Games* by W. D. Troyka (cf. [2]). The Puzzle's properties are outlined in detail in Chapter 3.

Setting

SlideThree is played on a 4 × 4 board with nine tiles which are numbered from 1 to 9. The remaining 7 squares of the playing board are blanks.

Figure 2.1 shows the Start and Win positions of *SlideThree*. The task is

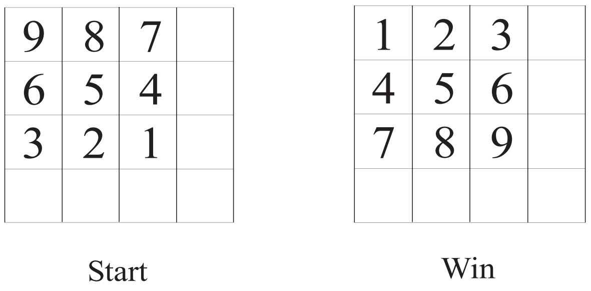


Figure 2.1: Start and Win Positions

to reverse the order of the tiles by sliding blocks of *three* tiles horizontally or vertically. A block can be chosen from a row or column and can only move within the row or column it was chosen from. Figures 2.2 and 2.3 illustrate the admissible moves. The tiles of the block must not have blanks between them

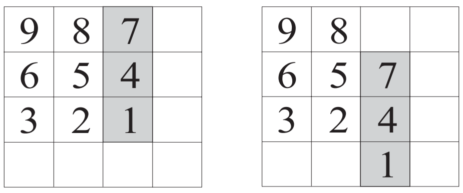


Figure 2.2: Admissible Column Slide

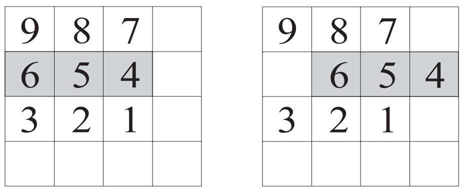


Figure 2.3: Admissible Row Slide

and there has to be a blank at one of the ends of the block. The sliding is done in direction of this blank.

Properties

In this introduction, we only give a short list of properties of the puzzle. For further details, see Chapter 3.

- Every position has 4, 5, or 6 successor positions.
- There are 11,612,160 different positions in *SlideThree*.
- The quickest solution for the Start position needs 24 moves.
- The most complicated positions need 26 moves to be solved.

At this point, we introduce the notion of the *DTW* of a position.

Definition 2.1.1 *The DTW (Distance To Win) of a position P is the minimal number of moves from P to the Win position.*

2.2 Search Algorithm

The algorithm used for the research is called *depth-t-search*. The levels of the built up game tree are also called *plies*. Searching a good move for the current game position works in the following way:

1. Generate the first t plies of the game tree, the current position is the root position in ply 0.

2. Evaluate the leaves of this subtree, i.e. the nodes in ply t. For this, a heuristic evaluation function is used. If the Win position is found before ply t is reached, this is recognized. The Win always gets the value 0.
3. Pass the values v up the tree by minimizing.
   $\mathrm{v(predecessor)}=\min\{\mathrm{v(successor}_1),\dots,\mathrm{v(successor}_n)\}$
4. Execute a one-step move leading from the root position in direction of the depth-t-successor with the minimal value.

The playing strength of depth-t-search varies, depending on the heuristic evaluation function and search depth.
In the following, we will talk of the playing strengths of evaluation functions. This always refers to the playing strength of the respective evaluation function used in combination with depth-t-search.

2.3 Evaluation Functions

For the research, six heuristic evaluation functions were built and their playing strengths were measured. For convenience, the word "heuristic" is left out in the following.

Definitions

The evaluation functions are defined in detail and explained in examples in Chapter 4, Section 4.2. In short, the six evaluation functions are:

1. Euclidian On Board (Eu)
   Let r(i) be the distance of Tile i to its goal square in rows, c(i) the distance in columns, respectively.
   $\mathrm{Eu} = \sqrt{\sum_{i=1}^{9} r(i)^2 + c(i)^2}$
2. Manhattan Metric (Man)
   Sum of the Manhattan distances of the single tiles to their goal square.
3. Neighbour Distance (Nb)
   The neighbourhood of each tile is examined. For every wrong, missing, or additional neighbour of the tile, a malus of +1 is assigned. Also, a malus of +1 is added for a wrong position of the tile. All malus are assigned with respect to the goal position of each tile. The sum of all malus is the value of the *SlideThree* position.
4. Permutation Distance (Perm)
   The *SlideThree* position is transformed to a permutation π of the numbers 1 to 9.
   $\mathrm{Perm} = \sqrt{0.5 + \sum_{i=1}^{9} (\pi(i) - i)^2}$
5. Inversions Distance (Inv)
   The *SlideThree* position is transformed to a permutation π like in 4.
   The Inversions Distance is the number of inversions in π.
6. L2 Distance (L2)
   The position is converted into a vector of $\mathbf{R}^{16}$. Then its L_2 distance (euclidian distance) to the vector corresponding to the Win position is calculated.

Also, Random Evaluation and Random Play were tested. Random Evaluation means, that the value of a position is a random number from (0, 1). In Random Play, a move is chosen randomly, without making any evaluation.

Discussion

The above evaluation functions are very different. Some of them are derived from the game setting and carry much information, others are rather distant from the actual structures in *SlideThree* and general. Thus, we obtain evaluation functions that differ in structure and playing strength. Both factors may influence the effect of randomization. Therefore it is important to also include weak and strange evaluation functions in our investigations.

2.4 Randomization

The goal of our research is, to find out whether randomization can improve deterministic heuristic evaluation functions. By randomization we understand a random alteration of the heuristic values assigned to the leaves of the searched tree.

Motivation

There are several reasons, why randomization might be able to improve evaluation functions.

Cycles Deterministic evaluation functions behave always identically in the same situation. If the evaluation function chooses one move in position P, it will choose the same move every time, position P is reencountered during the tree search. Therefore, deterministic evaluation functions often get stuck in cycles. Randomization offers a possibility to leave such cycles, thus solving this problem.

Tiebreaks In *SlideThree* we often have the situation that after a depth-t-search several successors of the root share the minimal value. In such a tie situation, randomization might help to choose a good move. Imagine the situation shown in Figure 2.4. When using randomization, the leaf with the minimal value will be in the left subtree with a higher probability than in the right subtree. The conjecture is that randomization tends to descend in the subtree with the most minimal nodes in the case of a tie. This might lead to better play.

K-Best Mode If the randomization is intense enough, it is able to switch the values of the successors. Thus not only the best, but also the second best, third best, or k-best moves become candidates for execution.

Summary

The above mentioned problems of deterministic evaluation functions are well known. In many programs, they are solved by additional routines for cycle detection, k-best modes, or counting of the nodes. Randomizing the evaluation function is much less complicated and might deal with several of these problems at the same time.

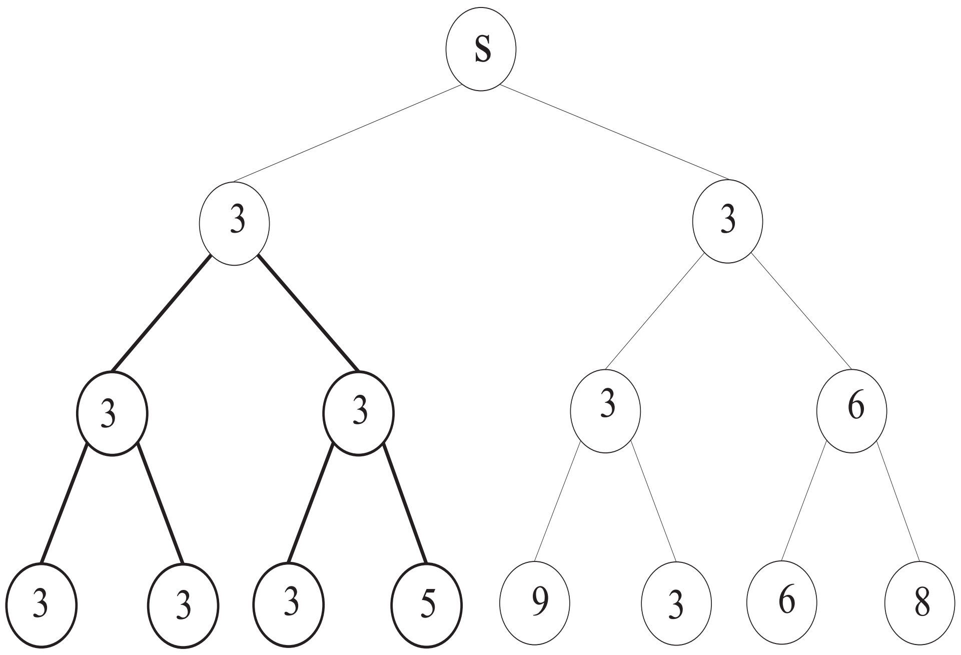


Figure 2.4: Example for Tie Situation

Randomization Techniques

The method that is used for randomization surely has a big influence on the behaviour of the randomized evaluation functions. Therefore, several types of randomization were attempted.

1. Add a random value (rv) to the deterministic evaluation function (evf). (Add)
   This is the easiest way of randomizing a deterministic evaluation function. The intensity of the randomization is scalable by changing the range of the random value, but depends on the absolute values of the heuristic evaluation function.

2. Multiply the evaluation function by (1+rv). (Mult)
   This is a relative randomization of the evaluation function. Its intensity no longer depends on the range of the deterministic evaluation function.

3. $\frac{\text{evf+rv}}{\text{evf}}$. (Norm+)
   This randomization method was designed in such a way that it switches the order of the moves for low intensity already. In a way, it rejects the values from the deterministic evaluation function. Here, also moves with a high heuristic value have a chance to be executed.

4. $\frac{\text{evf}-\text{rv}}{\text{evf}}$. (Norm–)
   Norm– is the mirrored version of Norm+. Whenever Norm+ orders two positions in one way, Norm– reverses the order.

5. Add (rv · Perm) to the deterministic evaluation function. (Comb)
   Perm is the Permutation Distance, which is one of our deterministic evaluation functions. So this type of randomization combines the knowledge of two deterministic evaluation functions in a random way.

2.5 Experiments

During the research, two classes of experiments were run. Each class uses a different approach to measure the playing strength of an evaluation function.

2.5.1 Choice of Moves Experiments

Preparation

In preparation of the experiments, the puzzle *SlideThree* was solved. That is, a database was built containing all *SlideThree* positions with their respective DTW (Distance To Win). The technical details of how the database was built are described in Chapter 3, Section 3.3.

Setup

In the choice of moves experiments (*CoM experiments*) it is determined how well the different evaluation functions choose their moves. During a depth-t-search for a *SlideThree* position, its successors get heuristic values. All moves leading to a successor with minimal heuristic value are considered for execution. These moves will be called *candidate moves* in the following. The set of candidate moves differs depending on the evaluation function used for the search. Among the candidate moves there are some good moves and some bad moves. A move leads from the current position to a successor position. Every move that leads us closer to the Win position is a good move. A bad move leads us away from the Win position. Therefore, good and bad moves can be distinguished by the DTW of the successor positions. A less DTW than that of the current position indicates a good move, a greater DTW a bad move, respectively. In the experiment, the DTW of all positions is retrieved from the database mentioned above.
In order to measure the playing strength of an evaluation function in the CoM experiments, the measure *Hits* is used.

Definition 2.5.1 *Let GM be the number of good moves among the candidate moves in a SlideThree position. Then, for this position we define:*

$$\text{Hits} := \frac{\text{GM}}{\#\,\text{candidate moves}}$$

As only the candidate moves are considered for execution, Hits is a measure for the playing strength of an evaluation function in the respective game situation. In the CoM experiments, the arithmetic mean of the Hits is calculated over a set of sample positions in a fixed DTW. The sample size is 1068 for all DTWs in order to obtain statistical significance (cf. Subsection 5.2.2). Only for the smallest DTW in the experiments, DTW=8, a sample of 1000 *SlideThree* positions was used. A full account of how the CoM experiments were set up with all technical details can be found in Section 5.1. For the results, see Chapter 6.

2.5.2 Autoplay Experiments

Setup

In the autoplay experiments, depth-t-search is used to play games of *SlideThree* for up to 100 moves. The ratio

$$\frac{\text{Games won within 100 moves}}{\text{Games played}}$$

is called *winning quota*. The winning quota is used as a measure for the playing strength of an evaluation function.
Typically, one autoplay experiment included about 1000 games, starting from *SlideThree* positions in a fixed DTW. Not only the type, but also the intensity of the randomization was varied. This was done by changing the size of the intervals from which the random values for the randomization were taken. The experiments were run for search depths 4, 5, and 6. Their results are explained in Chapter 6.

Chapter 3

The Puzzle *SlideThree*

The puzzle *SlideThree* was invented and implemented for *Zillions of Games* by W. D. Troyka (cf. [2]). It is a sliding tile puzzle similar to the well known *15-Puzzle* (cf. [1]). In this chapter, the rules of the game are explained and its properties are presented. The properties are both deduced theoretically and by computer aid. As a major result, the solvability properties of *SlideThree* positions are described. The last section refers to how the puzzle has been solved by building a *SlideThree* database.

3.1 Rules

SlideThree is played with nine tiles on a 4×4 board. The tiles are numbered from 1 to 9. At the start of the game, they are positioned in the first three rows and columns of the playing board in reverse order. The task is to order the tiles according to their numbers (Figure 3.1).

9	8	7	
6	5	4	
3	2	1	

Start

1	2	3	
4	5	6	
7	8	9	

Win

Figure 3.1: Start and Win Positions

Only blocks of three tiles can be slid. The blocks must be dense, meaning that there must not be blanks between the tiles. There must be a blank at one end of the block in direction of which the block is slid. A block is chosen from one row or column and can only slide within this row or column. Examples for admissible moves are shown in Figures 3.2 and 3.3.

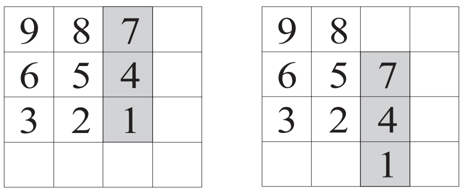


Figure 3.2: Admissible Column Slide

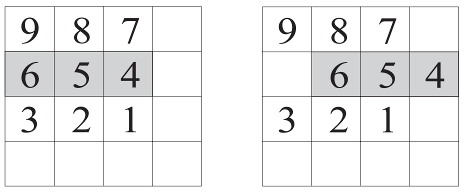


Figure 3.3: Admissible Row Slide

3.2 Properties

For easier reference we number the rows and columns of the playing board as shown in Figure 3.4. We will refer to each square by the two dimensional vector of its row- and column number.

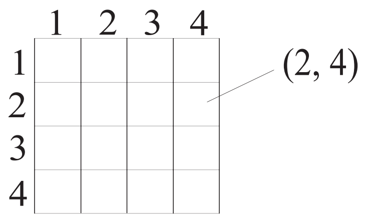


Figure 3.4: Numbering of Rows and Columns

3.2.1 Movement of the Blanks

There are seven blanks on the *SlideThree* playing board. In the Start position, three are located in the corners (squares (1, 4), (4, 1) and (4, 4)) and four on the edge of the board (squares (2, 4), (3, 4), (4, 2) and (4, 3)).

Corollary 3.2.1 *The three blanks starting in the corners move only to other corners of the playing board.*

Proof: The blanks in the corners are moved, when a block adjacent to them is slid. They switch their position from one end of the block to the other, which is again a corner.

■

Corollary 3.2.2 *The blanks on squares (2, 4) and (3, 4) move only within their respective rows. The blanks on squares (4, 2) and (4, 3) move only within their respective columns.*

Proof: The proof is given for the blank located on square (2, 4). This blank only moves when a block adjacent to it is slid. Such a block must cover squares (2, 1), (2, 2) and (2, 3). If it is slid, the blank moves from square (2, 4) to square (2, 1), staying within its row. For the other blanks the proof is analogous.

■

3.2.2 Successors

Every *SlideThree* position has either four, five or six successors. An example for a position with six successors is the position Start (Fig. 3.1). With nine tiles, this is the maximum number of successors. The minimum of four successors follows from the two corollaries of the previous section. They imply that, in any game position, each of the rows and columns 2 and 3 hold exactly one blank. This blank is at one end of the row or column (Corollary 3.2.1). Therefore each of these four rows and columns contains a block that can be slid. An example for a position with 5 successors are the successors of the Start position (see e.g. Figure 3.2).

3.2.3 Number of Positions/Solvability

Number of Patterns

To compute the number of positions occurring in *SlideThree* we introduce the notion of a *pattern.*

Definition 3.2.1 *A pattern is the distribution of the tiles on the board without taking into account the numbers on them.*

We will now determine the number of patterns occurring in *SlideThree.* This is derived from the possible positions of the blanks on the board. According to Corollary 3.2.1 there are always three blanks in the corners. So one out of four corners is covered with a tile. Furthermore, the blanks in rows 2 and 3 and in columns 2 and 3 can be at either end of their respective rows or columns. In summary, we have a total of $4 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 64$ patterns occurring in *SlideThree.* Within each of the patterns we theoretically have 9! possibilities of arranging the tiles.

Block Positions

In the following we will show that only half of the 9! permutations occur in each pattern. For this, the notion of a *block position* is introduced.

Definition 3.2.2 *A block position is a SlideThree position with six successors.*

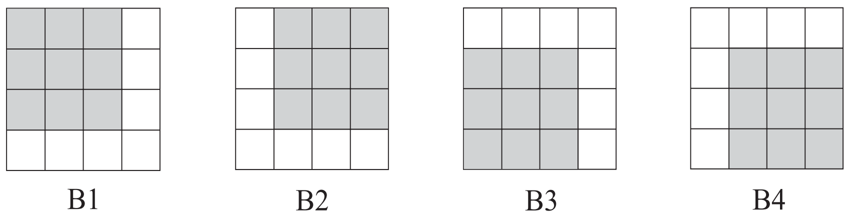


Figure 3.5: Block Positions

The four patterns of block positions are shown in Figure 3.5.

Lemma 3.2.3 *Any sequence of moves from a block position into the same block position consists of an even number of moves.*

Proof: The proof is given for the block position B1 from Figure 3.5. It is analogous for the other block positions.

We colour the squares of the playing board as shown in Figure 3.6.

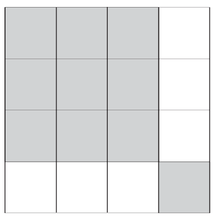


Figure 3.6: Colouring of the Squares

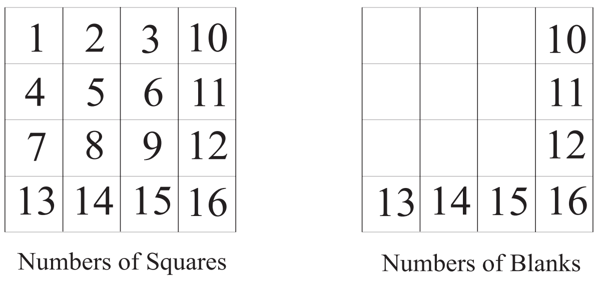
When playing from block position B1, each move either covers or uncovers exactly one of the coloured squares. Therefore we have to perform the same number of covering and uncovering moves in order to return to B1. So the total number of moves is even.

■

Remark 3.2.1 *From the above lemma we derive the following:*
Let A and B be two SlideThree positions and let a and b be the number of coloured squares covered by each position. If a and b are of the same parity, any sequence of moves which starts from A and ends in B consists of an even number of moves. If a and b are of different parity, the number of moves in any sequence leading from A to B is odd. Note that the Win position covers an odd number of coloured squares. Therefore the number of moves in any sequence from the Start position to the Win position must be even.

Conversion to Permutations

For the next step we introduce numbers for the blanks and squares of the playing board as shown in Figure 3.7. We now assign to each *SlideThree* position a corresponding permutation of the numbers 1 to 16. For an introduction to



1	2	3	10
4	5	6	11
7	8	9	12
13	14	15	16

Numbers of Squares

			10
			11
			12
13	14	15	16

Numbers of Blanks

Figure 3.7: Numbering of the Blanks and Squares

permutations, especially the concept of even and odd permutations, see [5]. The numbers of the squares provide the domain of the permutation. The image is the number of the tile or blank covering the respective square.

$$\pi(i) = \text{number of tile or blank on square } i$$

For example, the Start position is assigned to the permutation

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \end{pmatrix}$$

For block position B1, the corresponding permutation π can be written as the product of two permutations ζ and η.

$$\pi = \zeta \cdot \eta \tag{3.1}$$

ζ represents the order of the tiles. In B1, it has fixpoints at numbers 10 to 16. η represents the order of the blanks. It has fixpoints at numbers 1 to 9.

Main Theorem

We now want to prove that every permutation corresponding to a B1 position is even. For this, we first take a closer look at η.

Corollary 3.2.4 *The permutation η from Equation 3.1 is even.*

Proof: We reduce the *SlideThree* playing board to an undirected graph with four nodes. The nodes represent the corners of the board. For the Start position, the graph is shown in Figure 3.8 on the left. The numbers denoting the blanks are in their start positions. T denotes a tile. It can switch places with a neighbouring blank. Therefore, the order of the corner blanks is never changed during the game. I.e. in clockwise direction the order of the corner blanks is always 10, 16, 13. Thus, there are only three possible configurations of the corner blanks in B1. They are shown as graph representations in Figure 3.8. Their corresponding permutations in cycle notation are:
$\alpha = id$, $\beta = (10, 13, 16)$, $\gamma = (10, 16, 13)$, which are all even.

■

The above corollary implies that the Win position corresponds to an even permutation. Now we are ready to take the final step.

Theorem 3.2.5 *Only the even permutations of the tiles in block position B1 are solvable, i.e. there exists a sequence of moves to the Win position.*

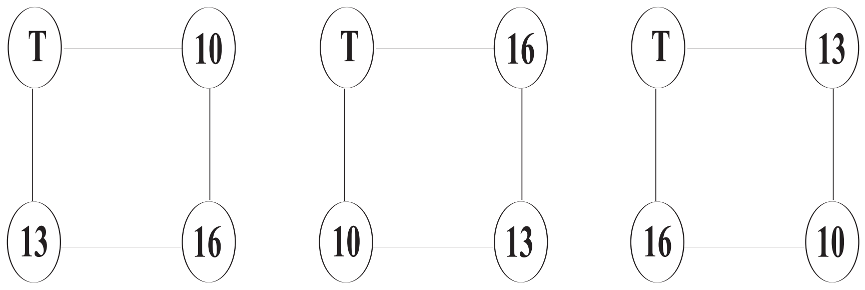


Figure 3.8: Order of Corner Blanks in B1

Proof: Let π be the permutation corresponding to a solvable B1 block position. It suffices to show that ζ in Equation 3.1 is even. From Lemma 3.2.3 we already know that we need an even number of moves to reach the Win position. Making a move in the game corresponds to multiplying two permutations. So the sequence of moves corresponds to a product of permutations, the first factor being π. Let ω denote the permutation corresponding to the Win position and μ_i the permutation corresponding to move i. Then the situation is represented by the following equation:

$$\pi \cdot \mu_1 \cdot \mu_2 \cdots \mu_{2n} = \omega \tag{3.2}$$

We now determine the permutations corresponding to the moves. If we start

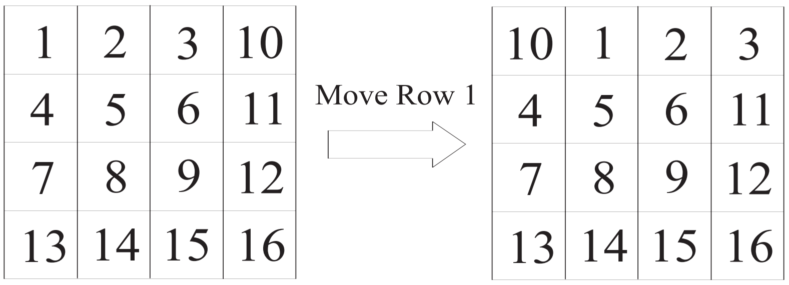


Figure 3.9: Permutation for Row 1-Slide

with the identical permutation and slide Row 1 to the right, we obtain (in cycle notation) the permutation $\rho = (1, 2, 3, 10)$, as illustrated in Figure 3.9. This permutation is odd, as can be seen when decomposing it to a product of transpositions:

$$\rho = (1, 2, 3, 10) = (1, 2)(2, 3)(3, 10)$$

In every move, three tiles and one blank change their positions in a circular way. Therefore they can all be represented as a cycle of length 4 in the cycle notation for permutations. A cycle of length 4 can be decomposed to a product of three transpositions. So all permutations corresponding to *SlideThree* moves are odd. Therefore, the product of the μ_i in Equation 3.2 is even. Also, ω is even. This implies that π must be even. We already know that in the decomposition $\pi = \zeta \cdot \eta$, η is even. Therefore ζ must be even.

■

Generalization:
SlideThree can be viewed as a special case of the game *SlideK*, which is played on a $K+1 \times K+1$ board ($K \geq 2$). In *SlideK* there are K^2 tiles and $2K+1$ blanks. Blocks can be slid if they contain K tiles without a blank in between. Theorem 3.2.3 holds for all these variants:
With an analogous numbering of the blanks and squares, we obtain a similar decomposition $\pi = \zeta \cdot \eta$ for each solvable B1 position π in *SlideK*. π here is a permutation of the numbers 1 to $(K+1)^2$. ζ represents the permutation of the tiles, η the permutation of the blanks. The movement of the corner blanks is the same as in *SlideThree*. Therefore, η is even. π is a product of an even number of permutations which represent the moves. Each of the factors can be written as a product of K transpositions because it is a cycle of length $K+1$. Applying the same reasoning as in 3.2.3 we conclude that π is even and consequently ζ is even.

Number of Positions

We know that only half of the 9! possible permutations of the tiles in B1 are solvable. It is easy to see that the same applies to each *SlideThree* pattern. In the following, we leave out the blanks and narrow our view to the tiles only. We introduce the notion of the *permutation within a pattern*. The numbers on the tiles are counted rowwise from the upper left corner to the lower right corner of the playing board. An example for how the permutation within a pattern is derived is given in Figure 3.10.

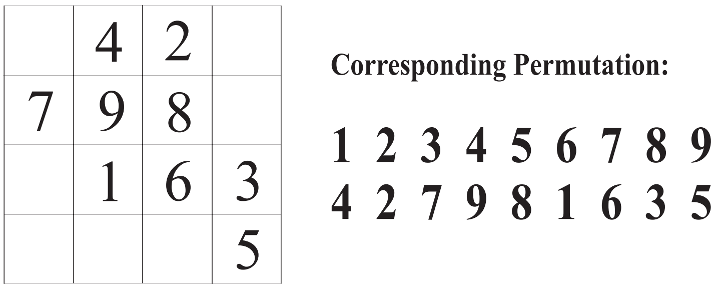


Figure 3.10: Permutation for a *SlideThree* Position

Because of the way of counting, sliding a block within a row does not change the permutation. Sliding a block within a column multiplies the permutation by either an even or an odd permutation. In *SlideThree*, there are only 16 patterns that are solvable for odd permutations. They are shown in the appendix in Figure H.1. All other patterns are solvable for even permutations.

Summarizing the results, the total number of *SlideThree* positions amounts to

$$64 \cdot \frac{9!}{2} = 11612160 \tag{3.3}$$

Generalization: As remarked earlier, Theorem 3.2.3 holds for *SlideK*. For these variants, the total number of positions is

$$4 \cdot 2^{2(K-1)} \cdot \frac{K^2!}{2}. \tag{3.4}$$

3.3 Building a *SlideThree* Database

In order to have a means to measure the quality of the evaluation functions in the later experiments, first a database of all *SlideThree* positions was created. Each entry in the database consists of three parts: the pattern of the position, the permutation of the tiles within the pattern, and its distance to win (DTW). The DTW of a position is the minimal number of moves that are needed to solve it (cf. Subsection 2.1.1). The pattern and permutation identify the position uniquely.

3.3.1 Backward Analysis

The database was created by making a backward analysis of the *SlideThree* game tree. This method has already been used and explained by Gasser in his PhD thesis [7]. In a backward analysis the game tree is expanded from the Win

```
ListRoot=Win;
ListPred=empty;
ListSucc=empty;
dtw=0;
new=1;
while(new!=0)
{
     new=0;
     for each in ListRoot
      {
           ComputeSuccessors;
           for each succ
           {
                if(succ not in ListPred)
                {
                     DTW(succ)=dtw+1;
                     write succ to ListSucc;
                     new=1;
                }
           }
      }
      save(ListSucc);
      ListPred=ListRoot;
      ListRoot=ListSucc;
      ListSucc=empty;
      dtw=dtw+1;
}
```

Table 3.1: Alg. 1 Backward Analysis

position until it contains all positions occurring in the game. The advantage of this backward method is, that each position occurs only once in the tree. The length of the path from the Win position to another position P is the DTW of P. Of course, only the positions that are solvable are included in the tree. Therefore, backward analysis is also a method of gaining knowledge about solvability. The terms *successor* and *predecessor* will be used in the following with respect to the structure of the tree that is built up for the backward analysis. When playing *SlideThree*, the Win position does not have successors, of course, but only predecessors. Table 3.1 shows the algorithm that was used in pseudo code.

The algorithm uses three lists. ListRoot contains the nodes which have to be expanded in the current step. ListPred contains the predecessors (the parent nodes in the backward tree) to the nodes in ListRoot. ListSucc is filled with the successors of the nodes in ListRoot (the child nodes in the backward tree). For the first step of the while loop, ListRoot is initialized with the Win position. The other lists are empty. The variable `dtw` holds the DTW of the nodes in ListRoot. The variable `new` indicates whether new successors were found in the last step. If there are no more new successors found, `new` is set to 0 and the while loop breaks. This happens when all solvable *SlideThree* positions have been processed. To enter the while loop, the variable `new` is initialized to 1 in Line 5. The algorithm works as follows: For all the nodes in ListRoot the successors are computed. For each of the successors it is checked, whether it appears in ListPred. This is necessary because the puzzle is reversible. At least one of the successors must appear as predecessor in ListPred. If the successor was not found in ListPred, its DTW is set to `dtw`+1 (it is one move further away from the Win than its predecessor). Then it is written to ListSucc (Lines 8–20). After each node of ListRoot has been processed in this way, ListSucc is saved to a file and the lists are reinitialized for the next step (Lines 21–23). The variable `dtw` is incremented to the DTW of the nodes of the new ListRoot. One can imagine the lists moving through the tree rooted at the Win position. ListRoot contains all nodes at ply `dtw`, ListPred holds the nodes at ply `dtw`–1, and ListSucc is filled with the nodes at ply `dtw`+1.

Before running the backward analysis for *SlideThree*, variables for counting the new found successors in each step were added. Thus, the distribution of the positions of *SlideThree* according to their DTWs was obtained (Figure 3.11). Further results from the backward analysis are:

- The Start position has DTW 24.
- The maximum DTW is 26.
- There are 184 positions with a DTW of 26.

The patterns of the 184 positions in DTW 26 are shown in Figure H.3 in the appendix. A complete list of them is included on the CD-ROM, filename *26Pos.txt*. For each position, its pattern and order codes are given. For the definition of these codes, see Subsection 3.3.2.

3.3.2 Making the Database Random Access

In the experiments, the DTW for *SlideThree* positions had to be retrieved very often. To reduce the time required for this, the files written during the backward analysis were reformatted to a random access database.

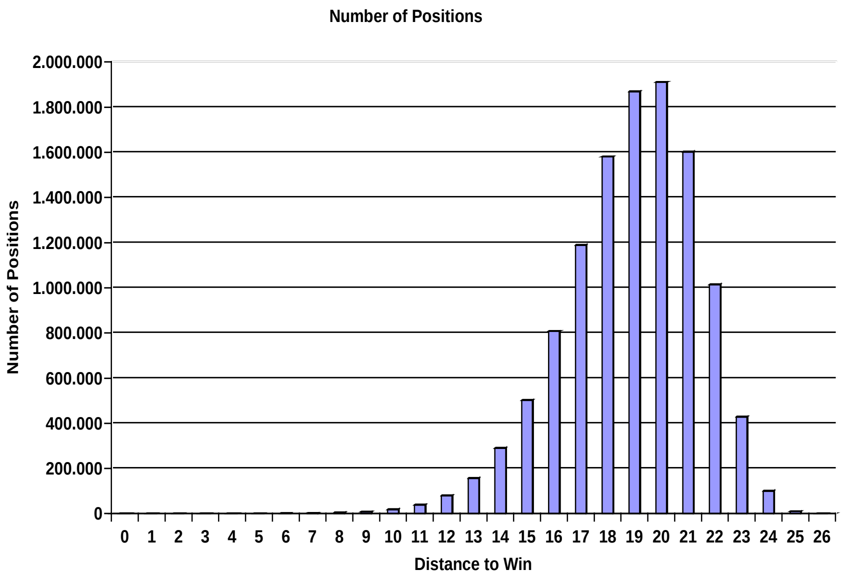


Figure 3.11: Distribution of *SlideThree* Positions

Gödelization

To have random access to a database means that each entry of the database can be reached directly, without performing expensive file searches. The first step towards this is to make a Gödelization of the *SlideThree* positions.

Definition 3.3.1 *Let P be the set of SlideThree positions.*
An injective function

$$g\colon\ P \longrightarrow \mathbf{N}$$

is called Gödelization of P.
The image g(P) is called Gödelnumber of the position P.

In case of *SlideThree* a Gödelization G was defined for each of the 64 patterns occurring in the game. Within each pattern, the permutation of the tiles identifies the position uniquely. So in defining a Gödelization for S_9, the symmetric group of the permutations of the numbers 1 to 9, we obtain a Gödelization for the positions within a pattern. The Gödelization is defined in a constructive way. An algorithm for computing the Gödelnumber for a position is given in pseudo code. Suppose the permutation is given in the form
$(\pi(1), \pi(2), \pi(3), \pi(4), \pi(5), \pi(6), \pi(7), \pi(8), \pi(9))$. Then the Gödelnumber is computed as explained in Table 3.2.

In other words, we start with subtracting 1 from the first entry of the permutation and multiply it with 8!. This is the value that will be added to in the following steps. It is called G in Table 3.2. Then we subtract 1 from each entry of the permutation that is greater than the first one. After that, the first entry is removed. Now a permutation of the numbers 1 to 8 is left. We subtract 1 from the first entry and multiply it by 7!. The result is added to G from the previous step. Then we proceed with subtracting 1 of every entry being greater than the first one and removing the first entry. Now we have a permutation of

```
G=0;
length=9;
while(length>0)
{
    G=G+($\pi(1)-1$)$\cdot$ (length-1)!;
    for($\pi(j)$ )
    {
        if( $\pi(j)$ >$\pi(1)$ )
            $\pi(j) = \pi(j) - 1$ ;
    }
    $\pi = \pi$ without $\pi(1)$
    length=length-1;
}
```

Table 3.2: Alg. 2 Gödelization G

the numbers 1 to 7 for which the whole procedure starts over again. The sum of all the G values is the Gödelnumber of the permutation or the corresponding *SlideThree* position, respectively.

Example: We take the *SlideThree* position from Figure 3.10 as an example. The corresponding permutation (in our notation) is
$\pi = (4, 2, 7, 9, 8, 1, 6, 3, 5)$.
First we calculate $(4 - 1) \cdot 8! = 120960$.
Next we subtract 1 from each entry that is greater than 4 and remove 4 from the permutation.
We obtain
$\pi' = (2, 6, 8, 7, 1, 5, 3, 4)$, a permutation of the numbers 1 to 8.
Now we proceed:
$120960 + (2 - 1) \cdot 7! = 126000$.
Reformatting π': $\pi'' = (5, 7, 6, 1, 4, 2, 3)$
$126000 + 4 \cdot 6! = 128880$
$\pi''' = (6, 5, 1, 4, 2, 3)$
$128880 + 5 \cdot 5! = 129480$
$\pi^4 = (5, 1, 4, 2, 3)$
$129480 + 4 \cdot 4! = 129576$
$\pi^5 = (1, 4, 2, 3)$
$129576 + 0 \cdot 3! = 129576$
$\pi^6 = (3, 1, 2)$
$129576 + 2 \cdot 2! = 129580$
$\pi^7 = (1, 2)$
$129580 + 0 \cdot 1! = 129580$
$\pi^8 = (1)$
$126580 + 0 \cdot 0! = 129580$
$\pi^9 = \{\}$
So the Gödelnumber for this position is G(Pos)=126580.

Properties: G has the following properties:

- The Gödelnumbers of the positions are unique (injectivity)
- The range of G is $\{0,1,2, \ldots, 9! - 1\} \subset \mathbf{N}$

As only half of the 9! *SlideThree* positions within each pattern are solvable, only half of the numbers in the range of G are Gödelnumbers of positions. We have to live with this slight disadvantage to keep the algorithm for calculating the Gödelnumbers simple and reasonably fast.

Structure of the Final Database

The structure of the final random access database is fairly simple. It consists of 64 files, one for each *SlideThree* pattern. Each of the files is a binary file containing 9! entries. Each entry represents a *SlideThree* position and is a record of the following structure:

- `short int Pattern` (code for the pattern of the position)
- `int Order` (permutation of the tiles within the pattern)
- `short int DTW` (DTW of the position)

The size of such a record is 12 Bytes. The entries in `Pattern` and `Order` identify the position uniquely.

Encoding of Positions Order holds the permutation of the tiles within the pattern encoded as an integer. E.g. the permutation (3, 7, 6, 1, 2, 9, 8, 4, 5) would be stored as the number 376129845. The code for the pattern is a little more complicated. It is a three digit integer, the digits having different meanings. Consider the corners of the playing field numbered as shown in Figure 3.12.

In each pattern exactly one of the corners is covered with a tile. The number

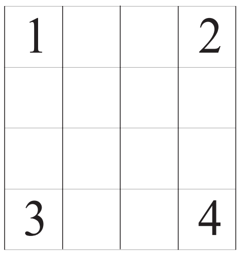


Figure 3.12: Numbering of the Corners

of this corner is the leftmost digit of the code. The positions of the blanks in rows 2 and 3 and columns 2 and 3 determine the other two digits. For each of the rows and columns, a binary number is computed. It is 1 if the first square of the row or column is covered with a tile and 0 if the last square is covered. The four bits are assembled to a binary four digit number in the following order:

$$\underbrace{0}_{\text{c2}}\underbrace{1}_{\text{c3}}\underbrace{1}_{\text{r2}}\underbrace{0}_{\text{r3}},$$

where c_i is column i and r_j row j, respectively. In this example, the last square in column 2, the first square in column 3, the first square in row 2 and the last square in row 3 are covered with tiles. The four digit binary number is converted to a decimal number which makes digits 2 and 3 of the pattern code. Thus every integer in the set
$\{100,\ 101\ldots\ 115,\ 200, 201\ldots\ 215,\ 300,\ 301\ldots\ 315,\ 400,\ 401\ldots\ 415\}$ identifies a pattern uniquely.

The advantage of this kind of encoding is that it is easy to compute. Even more important is the fact that a position can be easily encoded and decoded, even for a human being. In all programs that were written for this thesis, the *SlideThree* positions are represented by their Pattern and Order values.

Reformatting In order to build the random access database, a program was written which paces the text files that were created during the backward analysis. The codes for pattern and order and the DTW of each position are taken from the text file, put in the above described record and written to the database file corresponding to the pattern. Each record is included at position G(Order) in the database file. The entries in the database file containing no encoded *SlideThree* position are filled with zeroes. The filenames are 'DBxxx' , of which xxx is the 3 digit pattern code.
The size of the whole database is about 265 Mb. It is included on the CD-ROM.

Chapter 4

Methods of Empirical Research

This chapter describes the methods for the empirical research on the behaviour of randomized evaluation functions in single agent search. The first section explains the algorithm that was used for searching the *SlideThree* game tree. It is followed by the definitions of the heuristic evaluation functions and the different types of randomization.

4.1 Search Algorithm

The algorithm that was used is called *depth-t-search*. Note that the highest interest lies in comparing evaluation functions, not in finding a best search algorithm. Therefore the search algorithm was kept simple.

4.1.1 Depth-t-Search

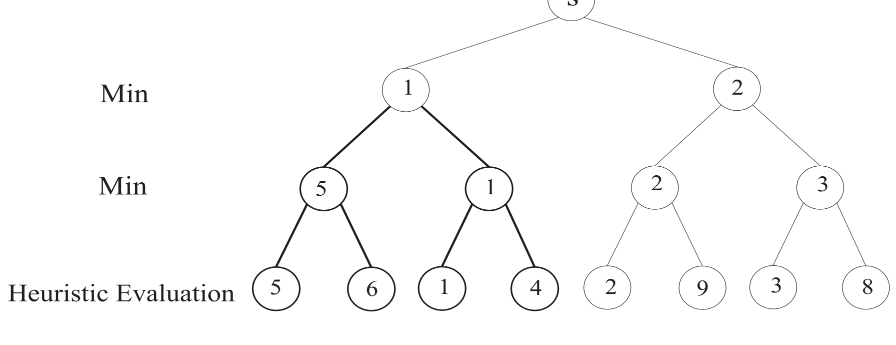


Figure 4.1: Depth-3-Search Example

The game tree of *SlideThree* is (as in most games) much too large to make an exhaustive tree search in reasonable time. Therefore, the search is limited to

```
float DepthTSearch(node n, int t, int depth)
1. {
2.       if(n==Win)
3.            Value(n)=0;
4.       else if(depth==t)
5.            Value(n)=Evaluate(n)
6.       else
7.       {
8.            {succ1,...,succm}=ComputeSucc(n)
9.            for(j=1 to m)
10.            {
11.                  Value(succj)=DepthTSearch(succj, t, depth+1);
12.            }
13.            Value(n)=min(Value(succ1),...,Value(succm));
14.       }
15.       return Value(n);
15. }
```

Table 4.1: Alg. 3 Depth-t-Search

a fixed depth t. If a node at ply t is reached during the search, it is evaluated by a heuristic evaluation function. The value of the parent node is the minimum of the values of its children. The search may be stopped before ply t, if the Win position is found. The value of the Win position is 0 for all evaluation functions considered. There is neither move ordering nor pruning done. In pseudo code, the algorithm is given in Table 4.1. It is a recursive function calling itself for every node that is expanded. The function takes the current node to expand (`n`), the maximum search depth (`t`), and the current search depth (`depth`), i.e. the depth of n in the tree. The depth of the root is 0. `DepthTSearch` is called with `depth`=0 for the starting position. In Lines 2 and 3, a check is made whether `n` is the Win position. If so, its value is set to 0. Lines 4 and 5 treat the case that `n` is in ply t. If so, `n` is evaluated by a heuristic evaluation function. Lines 6 to 14 describe the recursion. If `n` is neither in ply t, nor the Win position, it has to be expanded. That means that its successors are calculated and evaluated. To do this, `DepthTSearch` is called recursively on each of the successors (Lines 9 to 12). As already mentioned, the value of `n` is the minimum of its children's values (Line 15). It is the return value of the function. Figure 4.1 shows how depth-t-search works for t=3. In this example, a binary tree is searched. The nodes at ply 3 obtain their values from a heuristic evaluation function. These values are passed up the tree by minimizing.

4.1.2 Including a Transposition Table

Motivation

If several different sequences of moves have the same start- and end positions, this is called a transposition. An example for traspositions in *SlideThree* is the execution of two independent moves in different order.

Definition 4.1.1 *Two moves M_1 and M_2 are called independent, if the intersection of the tiles slid by M_1 and M_2 is the empty set.*

This happens very often in *SlideThree*, so many positions occur several times in the game tree. To avoid the redundant expansion of all these nodes, a transposition table is used. Positions are stored in the table after they have been expanded. They are retrieved from the table when they occur again in the game tree. This saves a lot of computing time.
In addition to speeding up the tree search, there was another reason for implementing a transposition table. Let P be a position that occurs several times in the game tree. When doing experiments with randomized evaluation functions, P will obtain a different value at each occurrence. With a transposition table this can be avoided to a certain extent.

Hashing Terminology

The transposition table is implemented as a large hash table. Each *SlideThree* position is written to the hash table after it has been expanded during the tree search. Its location in the hash table is called *hash index*. To determine the hash index, first the *hash value* of the position is computed:
Let P denote the set of all *SlideThree* positions. Furthermore, let k

$$\mathrm{k}\colon \mathrm{P} \longrightarrow \mathrm{H} \subset \mathbf{N}$$

be the function assigning the hash value to each position. Here, H is the set of hash values. Usually, its cardinality is much higher than the number of entries in the hash table. Therefore, the hash value has to be transformed to a valid entry number, the hash index. This is done by another function l:

$$\mathrm{l}\colon \mathrm{H} \longrightarrow \{0,\ 1, \ldots,\ \mathrm{size(TT)} - 1\},$$

size(TT) being the number of entries in the transposition table.
The concatenation

$$\mathrm{h} = \mathrm{l(k)} \qquad \mathrm{h}\colon \mathrm{P} \longrightarrow \{0,\ 1, \ldots,\ \mathrm{size(TT)} - 1\}$$

is called *hash function* h. A good hash function should distribute the positions uniformly in the hash table.
Whenever two positions compete for the same entry in the table, this is called a collision. In the following, we distinguish two types of collisions:

1. Two *different* positions compete for the same entry in the table (Type1).
2. A position competes with itself from two different places in the tree for the same entry (Type2).

Type1 collisions can be kept small by a good hash function and a sufficiently large table. Type2 collisions cannot be avoided. There are various possibilities for the treatment of collisions. How they are treated in our implementation is described in following.

Implementation

Structure In detail, the transposition table implemented for the experiments has 2^{19} entries. Each entry is a record of the following structure:

- `short int Pattern` (pattern code of position)
- `int Order` (permutation of the tiles within the pattern)

- `short int Depth` (depth of position in the search tree)
- `float Value` (value of the position)

The first two fields, Pattern and Order, are used to identify the position. Two positions are equal if and only if they have the same Pattern and Order codes. Storing the depth of the position in the search tree is important. A position is only retrieved from the table, if its depth is equal to the stored `Depth`. This guarantees that the uniformity of the tree search is not altered. The last field holds the value of the position according to the evaluation function.

Treating Collisions After a position has been expanded, it is written to the transposition table. If a Type1 collision occurs, the old position in the hash table is overwritten. This is based on the assumption that transpositions occur locally in the game tree (cf. [6]). In case of a Type2 collision, the position from the higher ply is stored. The reason for this is, that the value obtained from a deeper search is believed to be more reliable.

Hashing Method The hash value for a *SlideThree* position is computed by the method suggested by Zobrist [13]. This method is widely used in game playing programs. It is based on random numbers for each tile and its location on the board. For *SlideThree*, $9 \cdot 16$ 32-Bit random numbers are used, one for each pair (tile, square). They are assembled to the hash value by the XOR operator. The advantage of this method is, that the hash value can be calculated incrementally for many positions. This means that only the changes from one position to the next have to be recalculated. The most parts of the hash value can be reused.
The complete hash value is a 32-Bit integer. To obtain a valid hash index, it is mapped to a number between 0 and $2^{19} - 1$ (2^{19} is the size of the transposition table) by the binary AND operator (&). If we denote the hash value by *hv* and the hash index by *hi*,we have for position p

$$\text{hi(p)} = \text{hv(p)} \ \& \ (2^{19} - 1).$$

Using AND rather than MOD allows the table size to be a power of 2.

Random Numbers All random numbers in the experiments are generated with a pseudo random numbers generator called "Mersenne Twister". It was developed and implemented by Makoto Matsumoto and Takuji Nishimura [11]. The code of this pseudo random numbers generator is included on the CD-ROM, file *mt.c*.

4.2 Evaluation Functions

Depth-t-search is a heuristic search algorithm. When a node at ply t is reached during the search, the node is evaluated by a heuristic evaluation function. If an exhaustive tree search is made, we can use the DTW of the positions as an evaluation function. This would be a perfect evaluation function. Assume the DTW of the current position is k. Then every successor has either DTW $k + 1$ or $k - 1$. A successor with DTW $k + 1$ is always a bad move, whereas one with DTW $k - 1$ is always a good move.
A heuristic evaluation function only *estimates* the quality of the position. The

evaluation is usually based on simple properties of the position that may indicate its quality. For example, a high number of possible moves might be an indicator for a good position in some games.
In general, the playing strength of a program depends on the search algorithm *and* the heuristic evaluation function. Frequently the evaluation function is a linear combination of several parts, each part estimating the quality of the position according to a different property.

4.2.1 Definitions

We define seven simple evaluation functions which estimate the distance to the Win position by different criteria.

Definition 4.2.1 *Let Pos be any SlideThree position.*
(1) Euclidian On Board
Let r(i) be the distance in rows, c(i) the distance in columns of tile i to its goal square, respectively. The value of Pos according to Euclidian On Board is

$$\textit{Eu(Pos)} = \sqrt{\sum_{i=1}^{9} r(i)^2 + c(i)^2} \tag{4.1}$$

(2) Manhattan Metric
Let d(i) be the Manhattan distance of tile i to its goal square (d(i)=distance in rows + distance in columns). Then the value of Pos according to the Manhattan Metric is

$$\textit{Man(Pos)} = \sum_{i=1}^{9} d(i) \tag{4.2}$$

(3) Neighbour Distance
The value of Pos according to the Neighbour Distance is the sum of various malus. We assign a malus of +1 to a tile in Pos for each wrong, additional, or missing neighbour. Furthermore a malus of +1, if the tile is not on its goal square. The neighbourhood structure is the four neighbourhood. Whether a neighbour is missing, additional, or wrong is derived with respect to the neighbours of the tile in the Win position.
Let m(i) be the malus for tile i. The Neighbour Distance for Pos is given by

$$\textit{Nb(Pos)} = \sum_{i=1}^{9} m(i) \tag{4.3}$$

(4) Permutation Distance
Let π be the permutation of the tiles within Pos. Then the value of Pos according to the Permutation Distance is

$$\textit{Perm(Pos)} = \begin{cases} \sqrt{0.5 + \sum_{i=1}^{9} (\pi(i) - i)^2} & \textit{if Pos} \neq \textit{Win} \\ 0 & \textit{if Pos} = \textit{Win} \end{cases} \tag{4.4}$$

(5) Inversions Distance
Let π be defined as in (4). The value of Pos according to the Inversion Distance is

$$\textit{Inv(Pos)} = \begin{cases} 0.5 + \#\textit{Inversions of } \pi & \textit{if Pos} \neq \textit{Win} \\ 0 & \textit{if Pos} = \textit{Win} \end{cases} \tag{4.5}$$

(6) L2 Metric
To define this evaluation function, we convert Pos into a vector v(Pos) in $\mathbf{R}^{16}$*. This is done rowwise, starting in the upper left corner. The first four entries of the vector is the first row of Pos, the second four entries is the second row of Pos, and so on. The blanks are represented by zeroes.*
The value of Pos according to the L2 Metric is the euclidian distance (also called L2 distance) of v(Pos) to v(Win)=(1, 2, 3, 0, 4, 5, 6, 0, 7, 8, 9, 0, 0, 0, 0, 0)*:*

$$L2(Pos) = \sqrt{\sum_{i=1}^{16} (v_i(Pos) - v_i(Win))^2} \tag{4.6}$$

(7) Random Evaluation
The value of Pos according to the Random Evaluation is

$$Ran(Pos) = \begin{cases} \textit{random value} \in (0,\ 1) & \textit{if } Pos \neq Win \\ 0 & \textit{if } Pos = Win \end{cases} \tag{4.7}$$

4.2.2 Examples and Remarks

Examples

We give examples for all evaluation functions defined in Subsection 4.2.1. Figure 4.2 shows an example position.

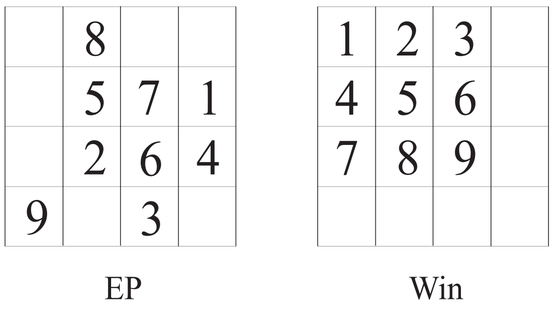


Figure 4.2: Example Position (EP)

Euclidian On Board Tile 1 needs to be slid one row up and three columns to the left to reach its goal square. Therefore we have $r(1) = 1$ and $c(1) = 3$. For the example position EP we obtain

$$\begin{aligned} \text{Eu(EP)} &= \sqrt{\textstyle\sum_{i=1}^{9} r(i)^2 + c(i)^2} \\ &\approx 6.928 \end{aligned}$$

Manhattan Metric For Tile 1 we have the Manhattan distance $d(1) = 1 + 3 = 4$, 1 being the distance in rows, 3 the distance in columns, respectively. The heuristic value of position EP according to the Manhattan Metric is

$$\begin{aligned}\text{Man(EP)} &= \textstyle\sum_{i=1}^{9} d(i)\\ &= 4+2+3+4+0+1+3+2+3\\ &= 22\end{aligned}$$

Neighbour Distance We first calculate the contribution of Tile 1 to the Neighbour Distance of EP.
In the Win position, Tile 1 is situated in the upper left corner of the playing board. It has two neighbours, Tile 2 on the right and Tile 4 below. In EP, Tile 1 is located on square (2, 4) of the playing board. The right neighbour is missing, which gives a malus of +1. Tile 4 is below Tile 1, which complies with the Win position, therefore no malus. On the left side and above Tile 1 there are two additional neighbours. Each of them contributes a malus of +1. It does not matter that one of them is a blank. Finally there is a malus of +1 because Tile 1 is not on its goal square. So altogether, Tile 1 contributes a malus of $m(1) = 4$. The malus for the other tiles are determined similarly. Thus we get

$$\begin{aligned}\text{Nb(EP)} &= \textstyle\sum_{i=1}^{9} m(i)\\ &= 4+5+4+4+4+5+5+5+4\\ &= 40\end{aligned}$$

Permutation Distance The permutation of the tiles within EP is

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9\\ 8 & 5 & 7 & 1 & 2 & 6 & 4 & 9 & 3 \end{pmatrix}$$

So the heuristic value of EP as determined by the Permutation Distance is

$$\begin{aligned}\text{Perm(EP)} &= \sqrt{0.5 + \textstyle\sum_{i=1}^{9} (\pi(i) - i)^2}\\ &= \sqrt{(8-1)^2 + (5-2)^2 + \cdots + (3-9)^2}\\ &\approx 11.769\end{aligned}$$

Inversions Distance To compute the Inversions Distance of EP, we start with the permutation π from the previous paragraph. Now we have to count the inversions in π, i.e. the number of entries that appear in wrong order. We start with the first entry, which is 8. In lexicographical order, 5 is before 8, so 8-5 is the first inversion. 8-7 is also an inversion. The right order would be 7-8. For the first entry we furthermore have the inversions 8-1, 8-2, 8-6, 8-4, and 8-3. That makes a total of 7 inversions for the first entry. For the second entry, which is 5, we have the inversions 5-1, 5-2, 5-4, and 5-3. Altogether we get

$$\begin{aligned}\text{Inv(EP)} &= 0.5 + 7 + 4 + \cdots + 1\\ &= 20.5\end{aligned}$$

L2 Metric In order to compute the value of EP according to the L2 Metric, we need the vector representation of EP. The position is written rowwise to a vector, the blanks being represented by 0.

$$v(EP) = (\underbrace{0,\ 8,\ 0,\ 0,}_{\text{row 1}}\underbrace{0,\ 5,\ 7,\ 1,}_{\text{row 2}}\underbrace{0,\ 2,\ 6,\ 4,}_{\text{row 3}}\underbrace{9,\ 0,\ 3,\ 0}_{\text{row 4}})$$

The vector corresponding to the Win position is

$$v(Win) = (1,\ 2,\ 3,\ 0,\ 4,\ 5,\ 6,\ 0,\ 7,\ 8,\ 9,\ 0,\ 0,\ 0,\ 0,\ 0).$$

The euclidian distance in $\mathbf{R}^{16}$ is the value of EP according to the L2 Metric:

$$\begin{aligned} \text{L2(EP)} &= \sqrt{\textstyle\sum_{i=1}^{16}(v_i(Pos) - v_i(Win))^2} \\ &\approx 16.248 \end{aligned}$$

Remarks

Euclidian On Board was suggested by Prof. Ingo Althöfer. It is a refinement of the Manhattan Metric. The intention was, to give credit to "diagonal" displacement of the tiles. When a tile is displaced diagonally, there are usually several shortest Manhattan paths to its goal square. An example of such a situation is shown in Figure 4.3. In the left situation, Tile 1 is displaced diagonally, i.e.

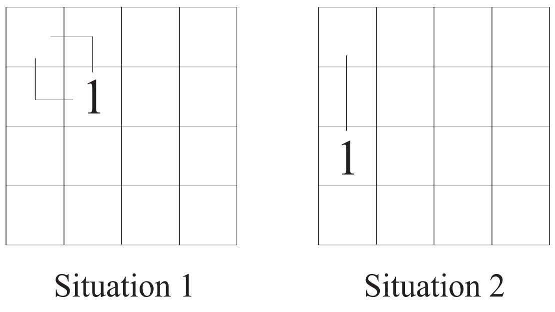


Figure 4.3: Motivation for Euclidian On Board

one square down and to the right with respect to its goal position. The right situation shows Tile 1 two squares below its goal position. In both cases, the Manhattan distance to the goal square is 2. But in Situation 1, there are two shortest paths, while in Situation 2, there is only one. The higher flexibility of Situation 1 is taken into account by Euclidian On Board. The contribution of Tile 1 in Situation 1 is +2 while in Situation 2 it contributes +4 to the total value of the position.

The values assigned to Pos by the different evaluation functions can be interpreted as a distance to the Win position. For each evaluation function E, we have E(Win)=0. To grant this property, the value 0.5 is added in (4) and (5). Some evaluation functions are not closely related to the game positions. Converting the position into a vector, as done for computing the L2 Metric, means giving up lots of information. This is also true for considering the permutation of the tiles only, as done for the Permutation- and Inversion Distance.
Others are more obviously related to the actual position of the tiles on the playing board. Especially the Neighbour Distance uses lots of information regarding the locations and relationships of the tiles. The choice of these very different evaluation functions was made on purpose. The research should comprise evaluation functions of different knowledge level and different origin. The hope is, that they might respond to different structures of *SlideThree*, so that

each of them can contribute to a linear combination later on. Furthermore it is important to investigate, whether different evaluation functions will respond to different methods of randomization.

4.2.3 Properties

In this subsection, we discuss some properties of the evaluation functions defined in Subsection 4.2.1.

Positivity

All heuristic evaluation functions defined in Subsection 4.2.1 are positive. As only the Win position gets value 0, all deterministic evaluation functions play perfectly once the Win lies in the scope of the tree search.

Coarseness

In order to get an idea of the coarseness of the different deterministic evaluation functions, it was determined how many different values were assigned to a number of different *SlideThree* positions. For our experiments with depth-t-search, it is interesting to know how the coarseness changes with respect to the search depth that is used to evaluate the respective position. Experiments were run for a mixed sample with 22458 different *SlideThree* positions (successor positions of 5000 randomly generated positions) from DTWs 6 to 24. Their results are shown in Table 4.2.

t	**Eu**	**Man**	**Nb**	**Perm**	**Inv**	**L2**	*Sum*
0	53	23	22	116	33	211	*458*
1	48	22	25	111	31	115	*352*
2	48	22	28	106	30	118	*352*
3	44	22	24	102	28	94	*314*
4	41	21	27	94	27	80	*290*
5	39	21	26	86	24	67	*263*
Sum	*273*	*134*	*152*	*615*	*173*	*685*	

Table 4.2: Number of Different Heuristic Values for 22458 Positions

Ties

Table 4.3 shows the percentage of ties occurring in 5000 depth-t-searches for *SlideThree* positions in DTWs 7 to 25.

t	**Eu**	**Man**	**Nb**	**Perm**	**Inv**	**L2**	*Av*
1	23.6	49.3	23.1	27.0	38.7	1.7	*27.2*
2	46.8	64.9	41.1	24.2	43.8	34.1	*42.5*
3	49.9	68.0	43.1	26.8	47.0	30.2	*44.2*
4	52.1	69.1	48.2	29.8	50.2	36.8	*47.7*
5	56.2	71.3	49.7	33.7	52.3	41.3	*50.8*
6	55.9	72.5	51.2	34.4	52.5	44.2	*51.8*
Av	*38.1*	*65.9*	*42.7*	*29.3*	*47.4*	*31.4*	

Table 4.3: Percentage of Tie Situations in 5000 Depth-t-Searches

Interpretation

Man is the coarsest, Perm and L2 are the finest evaluation functions. The coarseness increases with the search depth. There is an obvious relationship between the coarseness and number of ties. The coarser an evaluation function is, the more often tie situations are produced during the tree search. Therefore, also the number of ties increases with the search depth.

4.3 Randomization

The goal of this thesis is to determine whether randomization has a positive effect on the playing strength of evaluation functions. Of course, there are many possibilities of how randomization can be introduced. In the experiments, five different types of randomization were tested.

1. Add a random value (rv) to the deterministic evaluation function (evf). (Add)
   This is the easiest way of randomizing a deterministic evaluation function. The intensity of the randomization is scalable by changing the range of the random value, but depends on the absolute values of the heuristic evaluation function.

2. Multiply the evaluation function by (1+rv). (Mult)
   This is a relative randomization of the evaluation function. Its intensity no longer depends on the range of the deterministic evaluation function.

3. $\frac{\text{evf}+\text{rv}}{\text{evf}}$. (Norm+)
   This randomization method was designed in such a way that it switches the order of the moves for low intensity already. In a way, it rejects the values from the deterministic evaluation function. Here, also moves with a high heuristic value have a chance to be executed.

4. $\frac{\text{evf}-\text{rv}}{\text{evf}}$. (Norm–)
   Norm– is the mirrored version of Norm+. Whenever Norm+ orders two positions in one way, Norm– reverses the order.

5. Add (rv · Perm) to the deterministic evaluation function. (Comb)
   Perm is the Permutation Distance, which is one of our deterministic evaluation functions. So this type of randomization combines the knowledge of two deterministic evaluation functions in a random way.

The experiments were designed in such a way, that the type and intensity of the randomization could be varied easily. How to set the appropriate parameters in the programs is described in the appendix (cf. Appendix G).

0	[0, 1]	[0.5, 1.5]	[1, 2]
[0, 1]	[0.5, 1.5]	[1, 2]	[1.5, 2.5]
[0.5, 1.5]	[1, 2]	[1.5, 2.5]	[2, 3]
[1, 2]	[1.5, 2.5]	[2, 3]	[2.5, 3.5]

Figure 4.4: Ranges of Random Values for TileTab Randomization for Tile 1

As an additional randomization for the Manhattan Metric, TileTab was introduced. TileTab chooses random values for each tile and sums them up to the final value of the position. The range of the random values of the tiles depends on the Manhattan Distance of the tile to its goal square. An example for Tile 1 is shown in Figure 4.4. E.g. Tile 1 would get a random value from [1.5, 2.5], if it is located on square (2, 4). TileTab is *only* available for the Manhattan Metric. The Results for the TileTab randomization are shown in the appendix, (cf. Appendix E).

Chapter 5

Experiments

In this chapter, a full account of the conducted experiments is given. Two classes of experiments were run, each of them using a different approach to measure the playing strength of evaluation functions. In the following, the setup of the experiments is described. Also, a statistical model of the experiments is presented. For the results, see Chapter 6 and the appendix.

5.1 Choice of Moves Experiments

In the first class of experiments the focus is on the choice of moves of the evaluation functions. They were run for the deterministic evaluation functions and their randomized versions.

5.1.1 Experiment Design

In the Choice of Moves experiments (*CoM experiments*), a depth-t-search for a set of *SlideThree* positions is performed, varying t from 1 to 8. For each of the positions, the heuristic values of its direct successors are compared to their DTWs. The DTW for each position is retrieved from the database described in Section 3.3.

Measures

In the following, we distinguish several kinds of moves.

Definition 5.1.1 *Assume the current situation in the game is a position P with DTW(P)=k.*
By definition of DTW, each of its successors has either DTW=$k+1$
or DTW=$k-1$. We will call moves into successors with DTW=$k-1$ good moves *and successors with DTW=$k+1$* bad moves *in the following.*

We also divide the moves into two classes according to their heuristic values.

Definition 5.1.2 *Let again P denote the current position in the game. Its successors are $S_1, \dots, S_n$, their heuristic values $Hv(S_i)$, $i = 1, \dots, n$.*

Candidate move:
A move to a successor S_j with $Hv(S_j) = \min\{Hv(S_1), \dots, Hv(S_n)\}$ is called candidate move.
Rejected move:
A move is a rejected move*, if it is not a candidate move.*

The two classes of moves from Definition 5.1.1 are natural. They are based on the definition of the DTW. Distinguishing only two classes of moves resulting from the heuristic tree search is somewhat artificial. Usually, the heuristic method makes a refined distinction, like best-, second best-, or worst move. The restriction to only two classes is done for two reasons:

1. There are only two classes of moves for the game theoretical value DTW.
2. We consider in our experiments only the moves with the best (i.e. minimal) heuristic value for execution.

We introduce some further notions that will be helpful when defining the measures for the playing strength of the evaluation functions.

Definition 5.1.3
A Hit *is a move that is both a candidate move and a good move.*
A Fake *is a move that is both a candidate move and a bad move.*
A Miss *is a move that is both a rejected move and a good move.*
A Keep *is a move that is both a rejected move and a bad move.*

In order to measure the playing strength of the evaluation functions, the following ratios are computed for each position.

(1) $\text{Hits} = \frac{\#\ \text{Hit}}{\#\ \text{candidate move}}$

(2) $\text{Fakes} = \frac{\#\ \text{Fake}}{\#\ \text{candidate move}}$ $\quad$ Fakes = 1−Hits

(3) $\text{Misses} = \frac{\#\ \text{Miss}}{\#\ \text{rejected move}}$

(4) $\text{Keeps} = \frac{\#\ \text{Keep}}{\#\ \text{rejected move}}$ $\quad$ Keeps = 1−Misses

An example for a possible situation after the depth-t-search is shown in Figure 5.1.

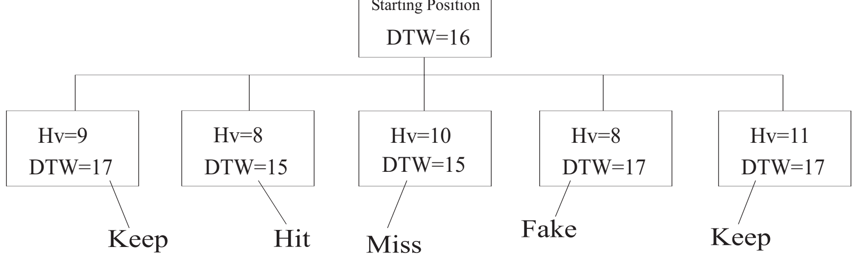


Figure 5.1: Example Situation after Depth-t-Search

Analysis of Measures

Figure 3.11 implies that the ratios (1) to (4) from Definition 5.1.1 depend on the DTW of the starting position. For example, the Hits are high if the DTW of the starting position is 24, because these positions tend to have many good moves and only a few bad ones. In other words, it is difficult to choose a bad move if the DTW is higher than 20. Therefore, even if the evaluation function has problems to recognize good moves, the Hits will be high for such positions. As the DTW decreases, it becomes increasingly difficult to select a good move. So the Hits of the evaluation functions will be much lower for small DTWs. Assuming that all positions in a fixed DTW are similar, the ratios (1) to (4) are estimators of the conditional probabilities

(1) P(good move|candidate move)

(2) P(bad move|candidate move)

(3) P(good move|rejected move)

(4) P(bad move|rejected move)

The conditional probabilities are linked by the total probability formula.

Lemma 5.1.1 *Let p be the probability that a move is a candidate move when using evaluation function E on a position in DTW=k. The probability that a move from a position in DTW=k is a good move is denoted by P(gm). For convenience we set*

$$\begin{aligned} H &= \text{P(good move} \mid \text{candidate move)}, \\ M &= \text{P(good move} \mid \text{rejected move)}. \end{aligned}$$

Then

$$H \leq M \Longleftrightarrow H \leq P(gm)$$

Proof: The proof is based on the total probability formula:

$$\text{P(gm)} = H \cdot p + M \cdot (1 - p)$$

$\Rightarrow$:
Assume that H $\leq$ M.

$$\begin{aligned} \Rightarrow \text{P(gm)} &\leq H \cdot p + H \cdot (1-p) \\ \Leftrightarrow \text{P(gm)} &\leq H \end{aligned}$$

$\Leftarrow$:
Assume that H $\leq$ P(gm).

$$\begin{array}{rrl} \Rightarrow & H & \leq H \cdot p + M \cdot (1-p) \\ \Leftrightarrow & H \cdot (1-p) & \leq M \cdot (1-p) \\ \Leftrightarrow & H & \leq M \end{array}$$

■

This connection shows in the results of the CoM experiments.

Suitability of Measures

There are many possibilities how the quality of an evaluation function can be measured. The criterion that is used in this thesis is its playing strength in connection with the depth-t-search algorithm. As already mentioned, only candidate moves are considered for execution when playing *SlideThree* with heuristic depth-t-search. Therefore, the Hits are a suitable measure for the playing strength of an evaluation function.

5.1.2 Conducted CoM Experiments

For the CoM experiments, two series were run.

100 Series

The first series includes experiments on 100 sample positions for search depths 1 to 10. Three Samples were taken randomly from each of the DTWs 5, 8, 9, 10, 11, 14, 15, 18, 19, 20, 21, 23, and 24. The arithmetic means of the Hits over all sample positions were computed for each sample. The results of these experiments can be found on the CD-ROM, file *ProbHitTab.doc*. The experiments were repeated using randomized versions of the evaluation functions on the samples in DTW=11. Their results are included on the CD, files *Randomizations Compared 1.doc* and *Randomizations Compared 2.doc*
This was the first series of experiments that was run for this thesis. It gives a good overview on how the Hits of the deterministic versions of the evaluation functions vary depending on the DTW and search depth. The randomization experiments encouraged the conjecture that randomization can raise the playing strength of evaluation functions, but were not statistically significant. Therefore, a second series of randomization experiments was run.

1000 Series

In this series, experiments were run on samples containing 1000 *SlideThree* positions or more. As the running time increases with the sample size, the search depth was only varied from 1 to 8. For each evaluation function, all types of randomization except Norm+ were used. Norm+ was known to pull the Hits down to the level of random play from the previous experiments of the 100 series. The intensity of the randomization was varied and the Hits compared to the Hits of the deterministic evaluation function. The samples were taken from DTWs 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20. The results are summarized in Chapter 6. More details are provided in the appendix. The full results are included on the CD, files *Randomization Tab DTW8.doc* to *Randomization Tab DTW20.doc.*

5.2 Autoplay Experiments

The autoplay experiments are probably the most interesting ones regarding the impact of randomization on deterministic evaluation functions. In the autoplay experiments, whole games of *SlideThree* are played. Each move is determined by a depth-t-search with a heuristic evaluation function. The interest here is on the winning quota. A high winning quota indicates a good evaluation function.

5.2.1 Previous Results

Similar experiments and their results have already been described by D.F. Beal and M.C. Smith for chess [4] and in the Diploma Thesis of T. Rolle [12] for several simple two player board games. D.F. Beal and M.C. Smith used random values as an evaluation function in a minimax search of fixed depth. They discovered that this plays better than random move selection. Moreover, simple evaluation functions for chess positions could be improved by adding a random value to them. They suggested that the reason for this behaviour is, that randomization tends to prefer the positions with the most possible moves.
T. Rolle's results varied depending on the game and nature of the search used. The search algorithm he used was alpha-beta search with and without pre sorting of the moves. In most cases there was no improvement of play when randomization was used. Rolle suggests in his thesis, that randomization leads to less alpha-beta cutoffs during the search and thus makes the tree search more inefficient. An improvement was only achieved in one game and with an alpha-beta search that did not pre sort the moves.
He furthermore gives an explanation for why randomization could have an improving effect at all. He states, that randomization picks the moves with the most optimal nodes in their subtree with higher probability than others (see also Section 2.4). Intuitively one might think that this results in descending a subtree containing many good positions. The autoplay experiments were designed to also investigate this conjecture.

5.2.2 Statistical Model

How important it is to achieve statistic significance in experiments was shown by E.A. Heinz [9]. Heinz suggests a Bernoulli experiment as a statistical model for self- or autoplay experiments. Bernoulli random variables take only values in {0,1}. In order to obtain statistically significant results from the experiments, the following analysis was made (cf. [8], [9]):

Significance Level

We introduce a Bernoulli random variable $X \in \{0, 1\}$.

$$X = \begin{cases} 1 & \text{if Win was found within U moves} \\ 0 & \text{else} \end{cases}$$

We assume furthermore, that for each evaluation function we have

$$P(X = 1) = p, \quad P(X = 0) = 1 - p,$$

where p also depends on the search depth t, DTW of starting position, and maximum number of moves per game U. When running n such experiments, we get a vector $\vec{X} = \{X_1, X_2 \dots X_n\} \in \{0, 1\}^n$ of i.i.d Bernoulli random variables.

The number of won games W, i.e. $W = \sum_{j=1}^{n} X_j$, is binomially distributed with parameters n, p. Taking the Winning Quota $\hat{p} = \frac{W}{n}$ as an estimator for p, we get $\sqrt{\frac{\hat{p}\cdot(1-\hat{p})}{n}}$ as an estimator of the standard deviation. The random value $Y = \frac{\hat{p}-p}{\sqrt{\frac{\hat{p}\cdot(1-\hat{p})}{n}}}$ is then approximately standard normal distributed. Now we can determine confidence intervals for p. For a $(1-\alpha)$ confidence interval, we need the $\alpha/2$ quantiles $q_{\alpha/2}$ of the standard normal distribution. The $(1-\alpha)$ confidence interval is then

$$\left[\hat{p} - q_{\alpha/2} \cdot \sqrt{\frac{\hat{p}\cdot(1-\hat{p})}{n}}, \quad \hat{p} + q_{\alpha/2} \cdot \sqrt{\frac{\hat{p}\cdot(1-\hat{p})}{n}}\right].$$

The probability that the true value of p is within the $(1-\alpha)$ confidence interval is $1-\alpha$. For $\alpha = 0.05$ we have $q_{\alpha/2} = 1.96$.

Sample Design

In order to obtain statistically significant results, we need to design the autoplay experiments such that the confidence intervals will be sufficiently small. The easiest way to do this, is to choose a sufficiently large sample. On the other hand, the sample should not be too big, with regard to the running time of the experiments. One can make a worst case estimation of the size of the confidence interval (cf. [8]):
The confidence interval is largest, if $\hat{p} = 0.5$. If we specify for this case a maximal deviation of 3% for a $(1-\alpha)$ confidence level, we get

$$\begin{aligned}
0.03 &= q_{\alpha/2} \cdot \sqrt{\tfrac{0.5^2}{n}} \\
0.0009 &= q_{\alpha/2}^2 \cdot \tfrac{0.25}{n} \\
n &= \tfrac{q_{\alpha/2}^2 \cdot 0.25}{0.0009} \\
&\text{For } \alpha = 5 \text{ we get} \\
n &= 1067.1
\end{aligned}$$

So for a deviation of 3% on a 95% confidence level, a sample size of 1068 suffices. For $\hat{p} \neq 0.5$, the confidence interval narrows.

Application to Choice of Moves Experiments

We can apply the same statistical model for the Choice of Moves experiments. Here, the Bernoulli random variable is defined as follows:

$$X = \begin{cases} 1 & \text{if candidate move is a Hit} \\ 0 & \text{if candidate move is a Fake} \end{cases}$$

The Hits are then an estimator for the probability of executing a good move in the current position.

5.2.3 Experiment Setup

Algorithm

In the following, the algorithm for the autoplay experiments is described.

One after another, the starting positions are read from a sample file. The sample file holds 1068 randomly chosen *SlideThree* positions in a fixed DTW. For each of the starting positions, a game of up to U moves is played. We refer to the position in which the game currently is as the *current* position. The game is played in the following way:

```
 1. Read current position from file
 2.   Perform depth-t-search for current position
 3.   If several minimal successors, choose one randomly
 4.   Else choose successor with minimal value
 5.   Make chosen successor the new current position
 6.   Increment number of moves made
 7.   If current position is Win, print out number of moves made
    and **quit game**
 8.   If number of moves made reaches upper bound U, print out message
    and **quit game**
 9.   If Win found in search tree, reduce search depth ($t = t - 1$)
10.   Goto 2
11. If game was quit and still positions in file, goto 1
```

In the above listing, Line 2 comprises evaluating the nodes with a specific heuristic evaluation function, with or without randomization. The number of moves made in Line 6 is initialized to 0 before the start of each game. The game is quit, if either the upper bound of moves is reached or a move into the Win position was made (Lines 7 and 8). Line 9 is very important for enabling the algorithm to "play till the end". Otherwise, the algorithm would lose the Win again: Imagine, that in a ply t search for the current position the Win appears for the first time. Surely, it will be a leaf of the search tree. All nodes on a path to the Win will be evaluated to 0. As the next current position, we get a node on the straight path to the Win. What happens, if we continue with the same search depth? For the next step, everything works out. The Win will only appear in the paths where it appeared before. Therefore, we also get a node on the straight path as the new current position. If we continue now with the same search depth, things become devastating. The Win will now appear two plies above leaf level, that is on ply $t - 2$. But if the shortest path to the Win is of length $t - 2$, we can build paths of length t to the Win via each of the successors of our current position: The first move is to the successor, then we move back to the current position and moves 3 to t are the shortest $t - 2$ path to the Win. Thus, all successors will be evaluated to 0 and no distinction between good and bad moves is possible. In decrementing t whenever the Win appears at leaf level, we assert that it stays at leaf level. We can then be sure of our good moves and really win the game.

Random Play D.F. Beal and M.C. Smith remarked in their article [4] that pure Random Play has an immanent disadvantage compared to the other evaluation functions. For these, a game tree search is conducted. If the Win lies within the scope of the tree search, this results in perfect play. The solution offered in [4] is to also make a tree search for Random Play. In this search, detection of the Win takes place, but no distinction between the other nodes is made. This solution for Random Play was adapted for the autoplay experiments. Using the evaluation function RP,

$$\mathrm{RP}(\mathrm{Pos}) = \begin{cases} 0 & \text{if Pos} = \text{Win} \\ 1 & \text{else} \end{cases} \quad ,$$

leads to the following behaviour: We get perfect play, if the Win is found in the tree. Otherwise, the move is chosen randomly among all possibilities.

Computed Values

For an empirical investigation of the conjecture of Rolle [12], a look was taken on how ties are broken during the game. A tie occurs, if two or more successors share the minimal value after a depth-t-search with the deterministic evaluation function. It is counted how many minimal nodes with respect to the deterministic evaluation function are contained in each of the subtrees rooted at the successors. If the tie is broken in favour of one of the successors with the most minimal nodes in its subtree, this is called a *fat move.* The *fat quota* computed during the experiment is the ratio $\frac{\#\text{ fat moves}}{\#\text{ ties}}$. In addition to the winning quota and fat quota, several other values are computed during an autoplay experiment.

1. Maximum number of moves needed to win a game
2. Minimum number of moves needed to win a game
3. Mean number of moves needed to win a game
4. Number of ties (sum over all games)

The number of ties is needed to compute the confidence interval for the fat quota. It follows the same statistical model as the winning quota. We interpret it as a Bernoulli experiment with

$$X = \begin{cases} 1 & \text{if fat move} \\ 0 & \text{else} \end{cases}$$

When calculating the standard deviance, we have to divide by the number of experiments, which is in this case the number of ties. Note that the ties are counted with respect to the deterministic evaluation function. When using randomization, not the ties produced by the randomized heuristic evaluation function are counted, but the ties from the deterministic evaluation function.

Conducted Experiments

Autoplay experiments were run for all evaluation functions and all randomizations on a sample in DTW=14 and a sample in DTW=24. Each of the samples contains 1068 positions. The intensity of the different types of randomization was varied. Experiments for search depths 4, 5, and 6 were run. Their Results are summarized in Chapter 6. A more detailed overview is given in the appendix, Chapter B. The full results are included on the CD-ROM, file *Autoplay.doc*.

The different kinds of randomization were not only compared to the deterministic evaluation function, but also to the winning quota when ties are always broken by fat moves. The winning quota of this evaluation proved to be much lower than the deterministic evaluation function and most of the randomizations. The reason for this might be that this method leads to much more cycles than the others. It is the most deterministic method that was tested. When conducting depth-t-searches with deterministic evaluation functions, the move to execute was randomly chosen among the minimal successors.

For the Manhattan Metric and Euclidian On Board, a series of autoplay experiments on a sample containing 1000 times the Start position was run. The serach depth used in these experiments was 5. Different randomizations were tested. An excerpt of the results is shown in the appendix, Chapter C. The full results can be found in file *Autoplay.doc* on the CD.

A series for samples in DTWs 8 to 18 was run for several evaluation functions with search depth 5. An excerpt of the results is presented in Chapter 6 and the appendix, Chapter D. The full results are included on the CD, file *Autoplay.doc*.

Chapter 6

Results

This chapter gives a full overview of the results of the research conducted. Section 6.1 refers to the CoM experiments, Section 6.2 relates the results of the autoplay experiments. Section 6.3 summarizes the results into some "rules of thumb".

6.1 Choice of Moves

The results of the CoM experiments are summarized in the following two sections. Section 6.1.1 gives a ranking of the evaluation functions according to their Hits. The results from the experiments with randomized evaluation functions are presented in Section 6.1.2.

6.1.1 Ranking of Evaluation Functions

The arithmetic mean of the Hits over all experiments on the samples of size 100 was used to rank the evaluation functions. Three samples of size 100 were taken from each of the DTWs 5, 8, 9, 10, 11, 14, 15, 18, 19, 20, 21, 23, and 24. The search depths vary from 1 to 10. From best to worst, the evaluation functions and their average Hits are:

1. Euclidian On Board (0.693)
2. Permutation Distance (0.651)
3. Inversions Distance (0.633)
4. Manhattan Metric (0.624)
5. Neighbour Distance (0.618)
6. L2 Metric (0.562)
7. Random Evaluation (0.503)

The average Hits for Random Play are 0.500. Random Play and Random Evaluation do not differ in playing strength.

6.1.2 Randomization Experiments

Experiments with randomized evaluation functions were conducted on samples of size 1000 and higher in DTWs 8 to 20. DTWs less than 8 are close to the endgame and the deterministic evaluation functions play rather well there. They also play very well in positions with DTW higher than 20, because these positions have only few bad moves. The search depth was varied from 1 to 8. Also, different intensities of randomization were tested. Table 6.1 shows which of the randomizations led to an improvement of the playing strength of the respective evaluation function. Here, an improvement of the playing strength is assumed if the following is true: For at least one search depth and at least one intensity of the randomization, a statistically significant increase of the Hits was observed for at least one sample. For all other samples, no significant decrease of the Hits occurred for that search depth and intensity of randomization. No experiments

	Add	**Mult**	**Norm–**	**Comb**
Eu	+	+	+	
Man	+	+	+	+
Nb				
Perm	+	+	+	
Inv	+	+	+	+
L2			+	+

Table 6.1: Randomization in CoM Experiments

were conducted with the Permutation Distance and the randomization Comb. Using Comb for randomizing the Permutation Distance is equivalent to Mult. Norm+ is not listed in the table. The pilot experiments with randomized evaluation functions showed that Norm+ pulls down the Hits to the level of Random Play for all evaluation functions.

6.2 Autoplay

The results of the autoplay experiments are split into two sections, each of them providing different insights into the behaviour of randomized evaluation functions. Section 6.2.2 shows how the intensity of the randomization influences the winning quota. The effect of randomization of a fixed intensity for different DTWs is investigated in Section 6.2.3. But first we take a look at the ranking of the evaluation functions according to their winning quotas.

6.2.1 Ranking of Evaluation Functions

From best to worst, the deterministic evaluation functions have the following ranking:

1. Euclidian On Board (0.360)

2. Manhattan Metric (0.174)

3. Neighbour Distance (0.050)

4. Inversions Distance (0.047)

5. Permutation Distance (0.045)

6. L2 Metric (0.003)

7. Random Evaluation (0.001)

The numbers given in the above listing is the average winning quota over all experiments on the samples in DTW 14 and 24. The winning quotas of the L2 Metric are not significantly higher than those of Random Evaluation. Also in the autoplay experiments, Random Evaluation did not perform better than Random Play.

6.2.2 Intensity of Randomization

The results in this section are based on a series of autoplay experiments for two samples of *SlideThree* positions. The samples were taken randomly from DTWs 14 and 24, each containing 1068 positions. In summary, the results can be classified in three different types. When using randomization we observe, with respect to the winning quota of the deterministic evaluation function,

1. an increase of the winning quota (+).

2. no significant change of the winning quota (≈).

3. a decrease of the winning quota (–).

Table 6.2 shows which of the effects was observed for which evaluation function and randomization. Here, a + means that a significant increase of the winning quota was observed for a special intensity of the randomization in most of the experiments. Of course, we observe a decrease of the winning quota for every evaluation function when we make the randomization intense enough.

	Add	**Mult**	**Norm+**	**Norm–**	**Comb**
Eu	+	+	–	≈	≈
Man	+	+	–	+	+
Nb	+	+	–	+	+
Perm	≈	≈	–	≈	
Inv	≈	≈	–	≈	≈
L2	≈	≈	–	≈	≈

Table 6.2: Effects of Randomization in Autoplay

When taking a closer look at the results, we can give a refined distinction of the behaviour of the winning quota in dependency on the randomization intensity.

1. The winning quota increases as soon as randomization is introduced. It stays at a high level until the randomization becomes too intense, then it decreases.

2. The winning quota increases only after a certain intensity of randomization has been reached. When the randomization becomes too intense it decreases again.

3. The winning quota does not change when randomization is used. Once the randomization reaches a certain intensity, the winning quota decreases.

4. The winning quota drops to 0 as soon as randomization is introduced. It does not recover.

5. None of the behaviours 1 to 4 was recognizable.

	Add	**Mult**	**Norm+**	**Norm−**	**Comb**
Eu	1	1	4	5 ≈	5 ≈
Man	1	1	4	5 +	5 +
Nb	2	2	4	5 +	5 +
Perm	3	3	4	3	
Inv	3	3	4	3	3
L2	5 ≈	5 ≈	4	5 ≈	5 ≈

Table 6.3: Behaviour of Winning Quota

Table 6.3 lists the evaluation functions and their type of behaviour. The entries with a 5 are additionally marked with a +, −, or ≈ sign, to indicate the change in the winning quota when using randomization. The classification of the behaviour of the L2 Metric fails because of too low winning quotas.
Figure 6.1 shows some results of the experiments, to give an impression on how the randomization intensity was varied. The data point 0 is the winning quota

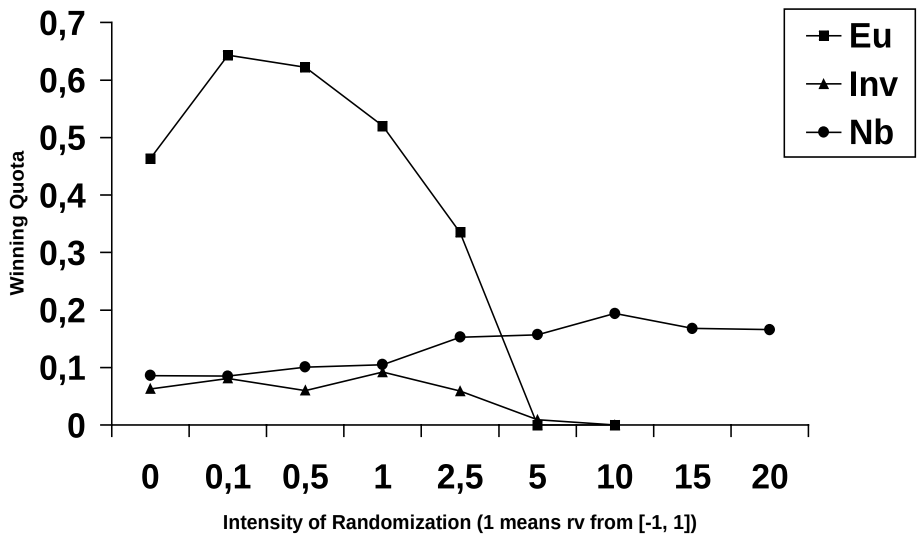


Figure 6.1: Results for Add, DTW=14, t=5

of the deterministic evaluation function without randomization. As we move to the right on the x-axis, the intensity of randomization is increased. The numbers on the x-axis are abbreviations for the interval from which the random number is taken. E.g. 5 means the random value is taken from [-5, 5]. The y-axis gives the winning quota. The sample consists of 1068 *SlideThree* positions in DTW=14.

Fat Quota

Randomizing deterministic evaluation functions should have an effect on the way ties are broken. As described in Section 2.4, the probability of descending into a subtree with the maximum number of optimal nodes should increase when randomization is used. We call a move to a successor with the most minimal nodes in its subtree a *fat move.* The percentage of ties broken by fat moves is called *fat quota.* One would expect that a weak randomization increases the fat quota. This was not always observed in the experiments. Table 6.4 shows the behaviour of the fat quota for the different evaluation functions and weak randomization.

	Add	**Mult**	**Norm+**	**Norm–**	**Comb**
Eu	+	+	−	+	+
Man	+	+	−	+	+
Nb	+	−	−	−	−
Perm	+	+	+	+	
Inv	+	+	−	+	+
L2	+	−	−	−	−

Table 6.4: Behaviour of Fat Quota

Interpretation

The evaluation functions that are closely related to the game (Eu, Man, Nb) react positively to randomization. For these, every randomization that does not switch the heuristic values of the possible successor positions immediately is able to improve the play of the evaluation functions. In case of Eu and Man, the improvement is a result of the tiebreaks performed by the randomization. Nb is improved when a k-best mode is reached.
The three weaker evaluation functions (Perm, Inv, L2) could not be improved by any randomization [1]. For Perm and Inv a more favourable tiebreak behaviour is reached by randomization. Nevertheless, a weak randomization can only choose a node among the minimal nodes as obtained by the deterministic evaluation function. If the deterministic evaluation function does not find a good move, a weak randomization cannot produce one, either.

6.2.3 Different DTWs

To see how the winning quota changes with the DTW of the starting position, a series of experiments on samples from DTWs 8 to 18 was conducted for several evaluation functions. The experiments were run with the deterministic and a randomized version of the evaluation function. Figure 6.2 shows the results for Euclidian on Board. The randomization used is Add with the random value taken from [–0.1, 0.1], the search depth is t=5.

[1]This result comes from a buggy program.
The Bug and its effects are described in Chapter 7

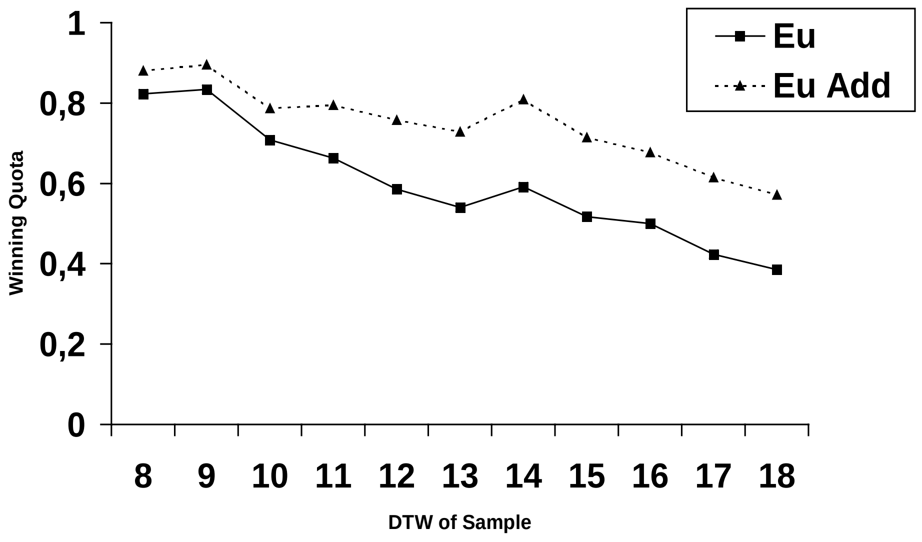


Figure 6.2: Euclidian on Board for several DTWs

6.3 Summary

The results of the experiments can be summarized in the following rules of thumb:

- Randomization is able to improve the play of deterministic evaluation functions.
- Randomization works best for deep searches and small intensity of randomization.
- Randomization worsens the play of any evaluation function if it becomes too intense or neglects the knowledge from the deterministic evaluation completely.
- To achieve an improvement by randomization, the deterministic evaluation function has to play sufficiently well.

Chapter 7

Bug Report

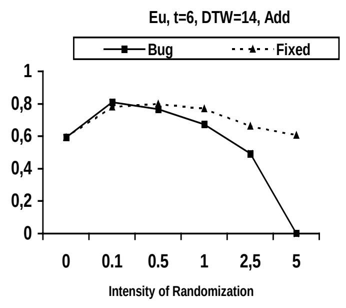
The programs for the CoM- and autoplay experiments as they were used for the empirical research have a bug. As stated in Subsection 4.2.3, the Win is evaluated to 0 by each deterministic evaluation function, which results in a perfect endgame. For intense randomization, it happens that some positions get negative heuristic values. In that case, the endgame is no longer perfect because the Win has no longer minimal value.
The result is, that in a CoM experiment with a search depth equal to the DTW of the sample positions, the 100% Hits are not achieved. In the autoplay experiments, the imperfect endgame leads to a faster decrease of the winning quota when making the randomization more intense.
To fix the bug, it has to be granted that the value of the Win position is always minimal. This was achieved by simply setting the value of the Win to –1000. The fixed versions of the programs, *comexpfixed.exe* and *autoplayfixed.exe*, are included on the CD-ROM.
The Bug effects only those randomizations that can lead to negative heuristic values. These are Add and Norm–. The following Figures illustrate the impact of the bug on the winning quota in the autoplay experiments for the different evaluation functions.

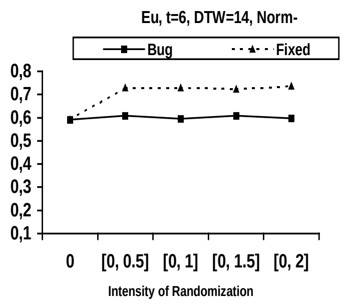
As can be seen, the bug has no influence on the general answer to our research question. Fixing the bug leads to even better results, i.e. higher winning quotas for the randomized evaluation functions. It is especially remarkable, that the formerly observed indifference of Perm and L2 to randomization was actually an effect of the bug. When the bug is fixed, these evaluation functions are also improved significantly by randomization.

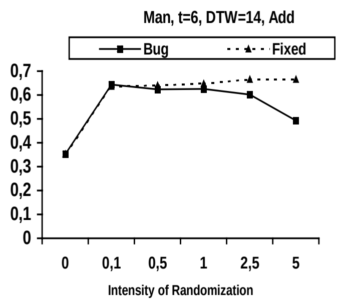
In conclusion, we adjust Table 6.2 according to the results from the bug fixed programs.

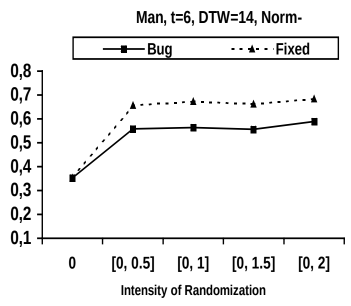
	Add	**Mult**	**Norm+**	**Norm–**	**Comb**
Eu	+	+	−	+	≈
Man	+	+	−	+	+
Nb	+	+	−	+	+
Perm	+	≈	−	+	
Inv	+	≈	−	+	≈
L2	+	≈	−	+	≈

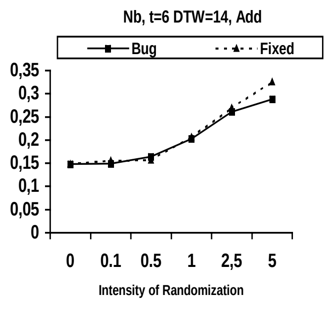
Table 7.1: Effects of Randomization in Autoplay (Corrected)

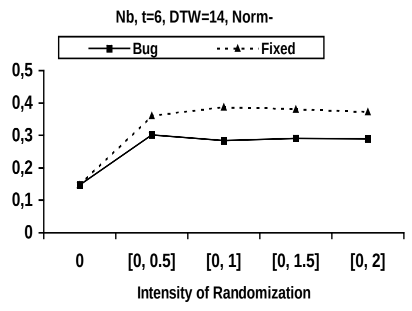

Eu, t=6, DTW=14, Add
Bug
Fixed
1
0,8
0,6
0,4
0,2
0
0
0.1
0.5
1
2,5
5
Intensity of Randomization

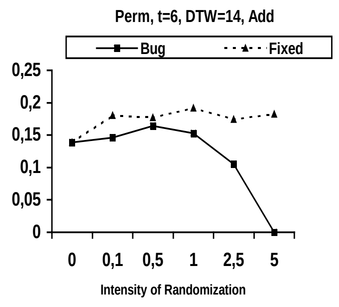

Eu, t=6, DTW=14, Norm-
Bug
Fixed
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

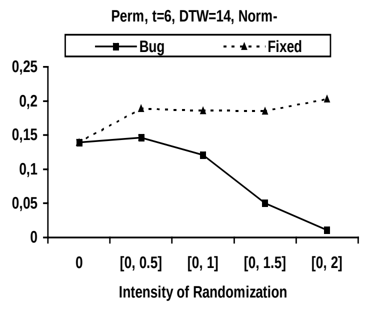

Man, t=6, DTW=14, Add
Bug
Fixed
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0
0,1
0,5
1
2,5
5
Intensity of Randomization

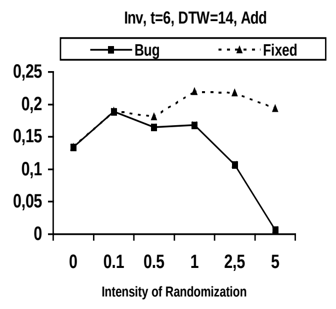

Man, t=6, DTW=14, Norm-
Bug
Fixed
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

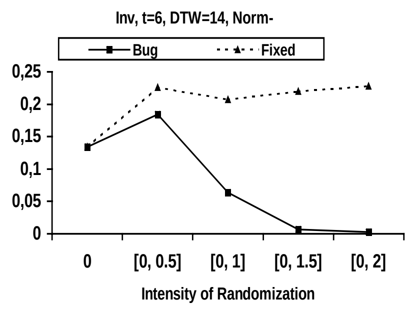

Nb, t=6 DTW=14, Add
Bug
Fixed
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
0
0.1
0.5
1
2,5
5
Intensity of Randomization

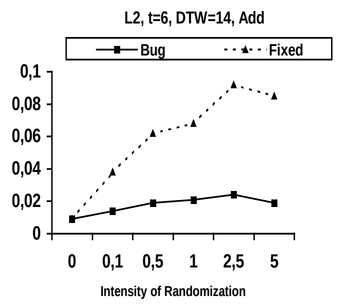

Nb, t=6, DTW=14, Norm-
Bug
Fixed
0,5
0,4
0,3
0,2
0,1
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

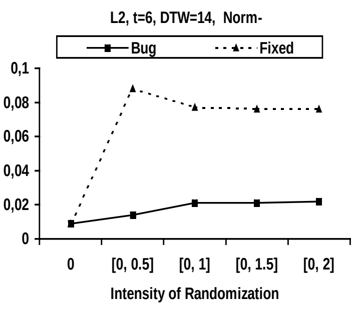

Perm, t=6, DTW=14, Add
Bug
Fixed
0,25
0,2
0,15
0,1
0,05
0
0
0,1
0,5
1
2,5
5
Intensity of Randomization


Perm, t=6, DTW=14, Norm-
Bug
Fixed
0,25
0,2
0,15
0,1
0,05
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization


Inv, t=6, DTW=14, Add
Bug
Fixed
0,25
0,2
0,15
0,1
0,05
0
0
0.1
0.5
1
2,5
5
Intensity of Randomization


Inv, t=6, DTW=14, Norm-
Bug
Fixed
0,25
0,2
0,15
0,1
0,05
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization


L2, t=6, DTW=14, Add
Bug
Fixed
0,1
0,08
0,06
0,04
0,02
0
0
0,1
0,5
1
2,5
5
Intensity of Randomization


L2, t=6, DTW=14, Norm-
Bug
Fixed
0,1
0,08
0,06
0,04
0,02
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

Chapter 8

Conclusions

From the outcome of the experiments described in this thesis, we draw the following conclusions.

8.1 Conclusions from Experiments

Randomization

Randomization is able to improve the playing strength of deterministic evaluation functions in connection with heuristic depth-t-search in single player games.

In order to achieve an improvement, the deterministic evaluation function should include some knowledge about the game position. It should be closely related to the game.
The easiest randomization techniques for the heuristic values, adding a random value or multiplication by a factor (1+random value), lead to the most obvious improvement. The latter is better to handle, because it adapts to the absolute value of the evaluation function.
Using a randomization that switches the order of the heuristic values of the positions completely results in Random Play.

Empirical Methods

To measure the playing strength of evaluation functions, the autoplay experiments are more suitable than the CoM experiments. In the CoM experiments, a too local view of the behaviour of the game playing algorithm is taken. The changes in the Hits are mostly below significance level, which makes it difficult to distinguish between random effects and real improvement. To achieve a satisfying significance, the 95% confidence intervals should be narrowed from $\pm3\%$ to $\pm1.5\%$. This requires samples that include at least $4272 = 4 \cdot 1068$ positions (cf. Subsection 5.2.2).

8.2 Future Research

A quite impressive increase of the winning quota was observed in many cases. Therefore it is worthwhile to continue the research on randomized evaluation functions in game playing programs.

For *SlideThree*, it might be interesting to determine even better evaluation functions than Euclidian On Board and test whether randomization can improve them. The conjecture is that once the deterministic evaluation function reaches a certain playing strength, it cannot be improved by randomization. Also, the research should be continued for other single player games.

Depth-t-search is not the standard algorithm used for single agent search. State of the art are algorithms like A* or IDA* with various extensions (cf. [10]). Repeating the experiments with different search algorithms would lead to a more general view on randomized evaluation functions.

An open question that was raised by Prof. Althöfer and might be worthwhile to answer is to what extend not only the number of optimal nodes in the different subtrees, as considered for the fat quota, plays a role, but also their distribution in their respective subtrees. E.g. is it better to descend a subtree with uniformly spread optimal nodes or a subtree with clustered optimal nodes? There is a conjecture that a clustered distribution of optimal nodes is of advantage because one does not reject many alternatives when descending the first few steps into that subtree.

Bibliography

[1] http://www.cut-the-knot.com/pythagoras/fifteen.shtml

[2] http://www.zillionsofgames.com/games/slidethree.html

[3] Althöfer, I.: On Telescoping Linear Evaluation Functions. ICCA Journal 1993, Vol. 16, pp. 91–94.

[4] Beal, D.F.; Smith, M.C.: Random Evaluations in Chess. ICCA Journal 1994, Vol. 17, No. 1, pp. 3–9.

[5] Bosch, S.: Algebra. Springer-Verlag, überarbeitete Auflage 1996. Kapitel 5.3.

[6] Breuker, D.M.; Uiterwijk, J.W.H.M.; van den Herik, H.J.: Replacement Schemes for Transposition Tables. ICCA Journal 1994, Vol. 17, No.4, pp. 183–193.

[7] Gasser, R.U.: Harnessing Computational Resources for Efficient Exhaustive Search. Dissertation. Swiss Federal Institute of Technology Zürich. 1995. Chapter 2.3.

[8] Harnett, D.L.: Statistical Methods, 3rd Edition, pp. 326–331. Addidson Wesley 1982.

[9] Heinz, E.A.: Scalable Search in Computer Chess, Vieweg 2000. Chapter 9.

[10] Junghanns, H.: Pushing the Limits: New Developments in Single-Agent Search. PhD Thesis, University of Alberta, Department of Computer Science, 1999.

[11] Matsumoto, M. and Nishimura, T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACMTMCS: ACM Transactions on Modeling and Computer Simulation 1998, Vol.8, pp. 3–30. http://www.math.keio.ac.jp/matumoto/MT2002/emt19937ar.html.

[12] Rolle, T.: Development of a multi-game engine, Diploma Thesis, Friedrich-Schiller-University Jena, Faculty of Matehmatics and Computer Science, May 2003.

[13] Zobrist, A.L.: A New Hashing Method with Applications for Game Playing. ICCA Journal 1990. Vol. 13, No. 2, pp. 69–73.

List of Tables

3.1 Alg. 1 Backward Analysis 22
3.2 Alg. 2 Gödelization G 25

4.1 Alg. 3 Depth-t-Search 29
4.2 Number of Different Heuristic Values for 22458 Positions 36
4.3 Percentage of Tie Situations in 5000 Depth-t-Searches 36

6.1 Randomization in CoM Experiments 49
6.2 Effects of Randomization in Autoplay 50
6.3 Behaviour of Winning Quota 51
6.4 Behaviour of Fat Quota 52

7.1 Effects of Randomization in Autoplay (Corrected) 54

E.1 Winning Quota of TileTab Randomization 71

I.1 SlideTwo, Average Over 6 Games Each 80
I.2 SlideThree, Average Moves Over 3 Games Each 80
I.3 SlideFour, Average Moves Over 3 Games Each 81

List of Figures

2.1 Start and Win Positions 8
2.2 Admissible Column Slide 9
2.3 Admissible Row Slide 9
2.4 Example for Tie Situation 12

3.1 Start and Win Positions 15
3.2 Admissible Column Slide 16
3.3 Admissible Row Slide 16
3.4 Numbering of Rows and Columns 16
3.5 Block Positions 18
3.6 Colouring of the Squares 18
3.7 Numbering of the Blanks and Squares 19
3.8 Order of Corner Blanks in B1 20
3.9 Permutation for Row 1-Slide 20
3.10 Permutation for a *SlideThree* Position 21
3.11 Distribution of *SlideThree* Positions 24
3.12 Numbering of the Corners 26

4.1 Depth-3-Search Example 28
4.2 Example Position (EP) 33
4.3 Motivation for Euclidian On Board 35
4.4 Ranges of Random Values for TileTab Randomization for Tile 1 38

5.1 Example Situation after Depth-t-Search 40

6.1 Results for Add, DTW=14, t=5 51
6.2 Euclidian on Board for several DTWs 53

A.1 Hits in Dependency on the Randomization Intensity 62
A.2 Hits in Dependency on the Search Depth 63
A.3 Hits in Dependency on the DTW of the Sample 63

H.1 Patterns that are Solvable for Odd Permutations 78
H.2 Nice Starting Positions in DTW 26 79
H.3 Patterns of Positions in DTW=26 with Code 79

Appendix A

CoM Experiments

The following figures illustrate three aspects of the Hits. First, the behaviour of the Hits for different intensities of randomization is shown. Search depth and DTW of the sample are fixed. The second set of figures are examples of how the Hits change when the search depth is varied. Here, the randomization intensity and DTW of the sample set are fixed. Finally, the behaviour of the Hits in dependency on the DTW of the sample is shown. In this set of figures, the search depth and randomization intensity are fixed.

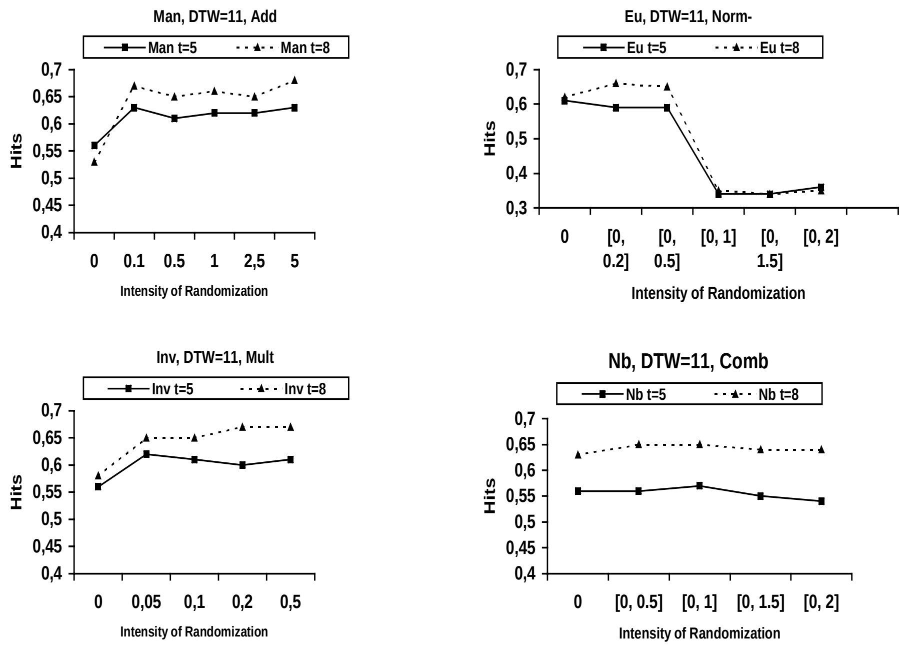


Figure A.1: Hits in Dependency on the Randomization Intensity

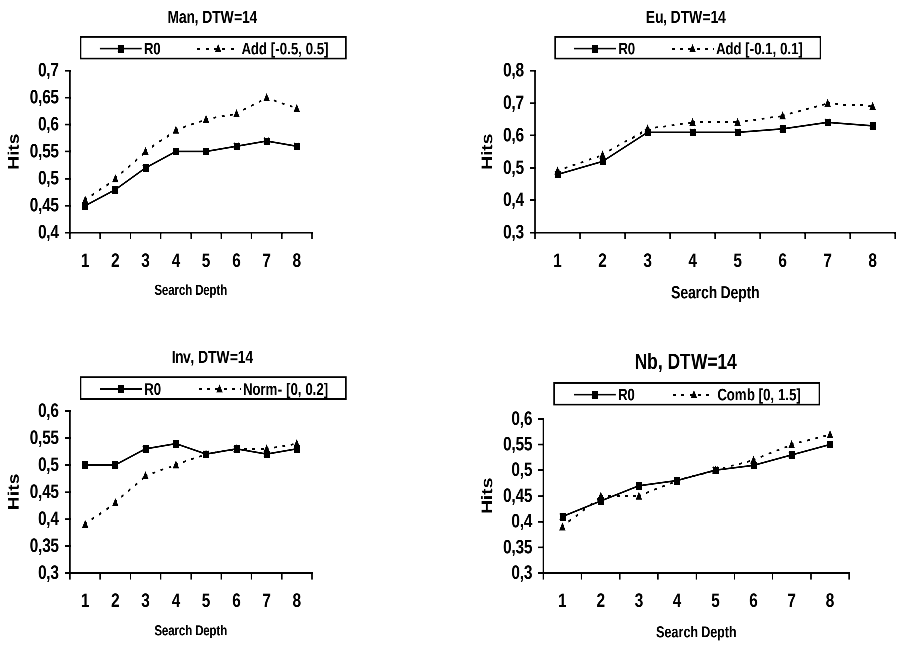


Figure A.2: Hits in Dependency on the Search Depth

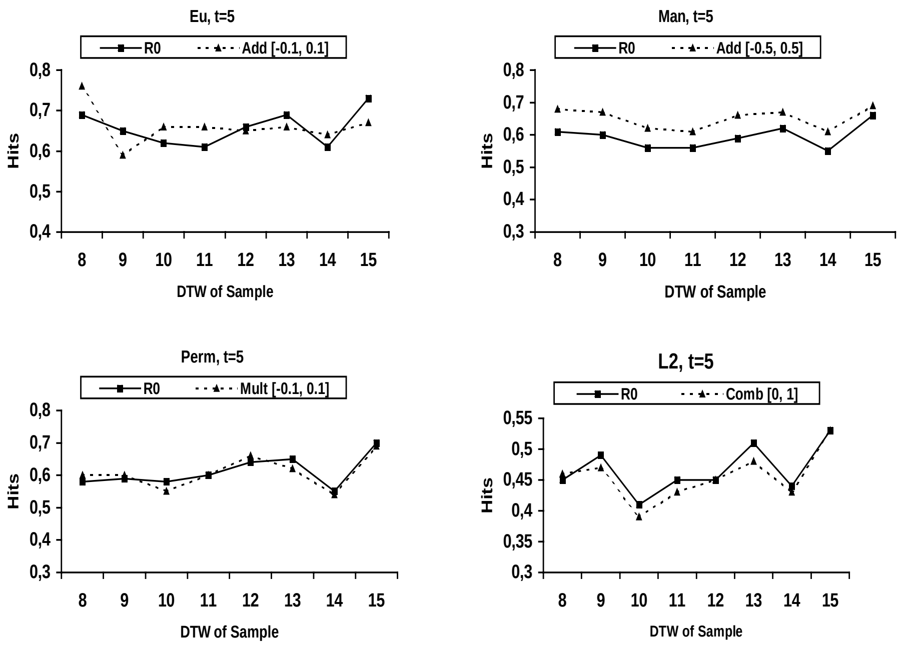


Figure A.3: Hits in Dependency on the DTW of the Sample

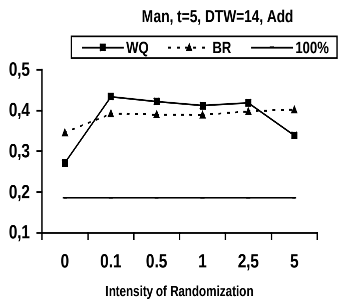
Appendix B

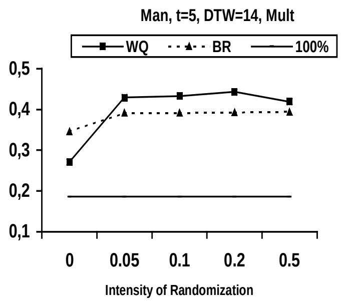
Autoplay, DTWs 14 and 24

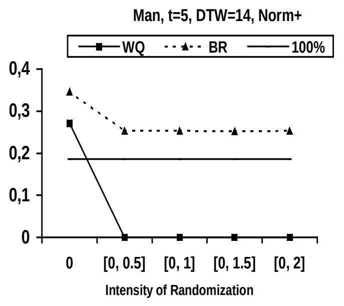
The following figures show some more results from the autoplay experiments for the samples in DTWs 14 and 24. In each figure, there are three datasets:

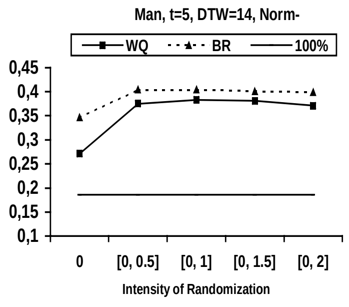
- WQ, the development of the winning quota when the randomization intensity is increased.
- BR, the development of the fat quota with increasing intensity of the randomization.
- 100%, the winning quota that is reached when ties are always broken in the direction of the successor with the most minimal nodes in its subtree.

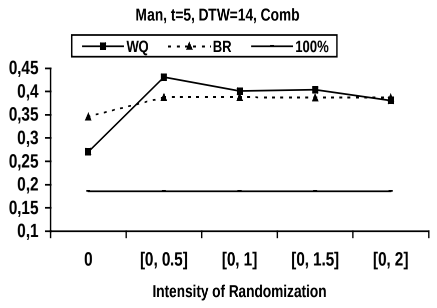
The numbers on the category axis indicate the intensity of the randomization. For Norm+, Norm-, and Comb, the intervals from which the random number is taken is given. For Add and Mult, only the right border of the interval is provided. The interval is symmetric with respect to 0, i.e. 2.5 on the category axis is an abbreviation for the interval [–2.5, 2.5]. The title of each figure provides the evaluation function, search depth, DTW of the sample randomization that was used in the experiment. All winning quotas and fat quotas with their respective confidence intervals are included on the CD-ROM, file *Autoplay.doc*.

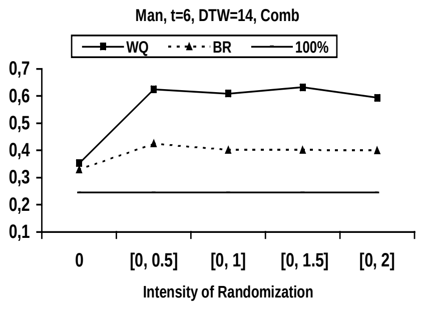

Man, t=5, DTW=14, Add
WQ
BR
100%
0,5
0,4
0,3
0,2
0,1
0
0.1
0.5
1
2,5
5
Intensity of Randomization

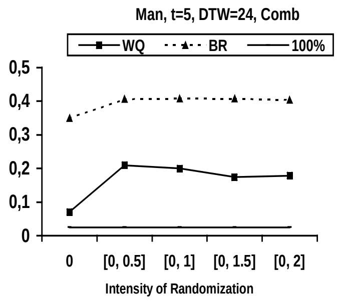

Man, t=5, DTW=14, Mult
WQ
BR
100%
0,5
0,4
0,3
0,2
0,1
0
0.05
0.1
0.2
0.5
Intensity of Randomization

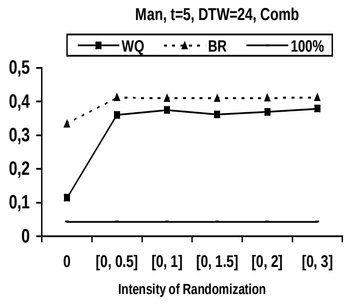

Man, t=5, DTW=14, Norm+
WQ
BR
100%
0,4
0,3
0,2
0,1
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

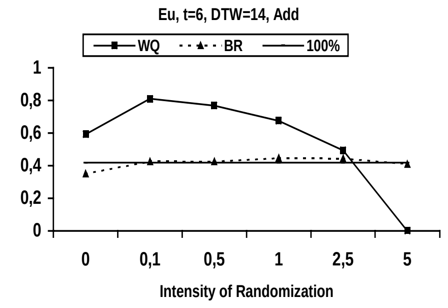

Man, t=5, DTW=14, Norm-
WQ
BR
100%
0,45
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

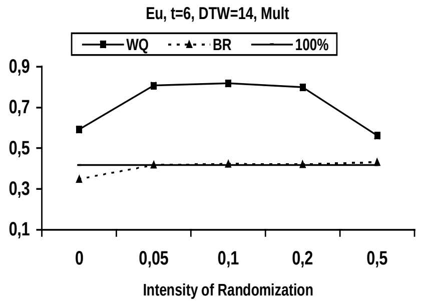

Man, t=5, DTW=14, Comb
WQ
BR
100%
0,45
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

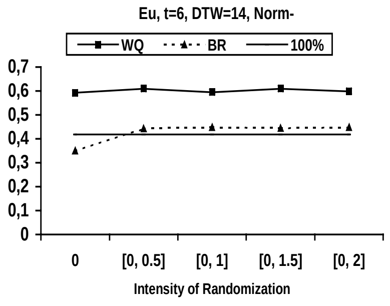

Man, t=6, DTW=14, Comb
WQ
BR
100%
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

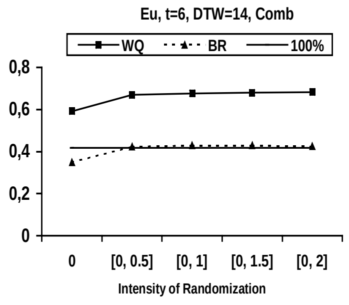

Man, t=5, DTW=24, Comb
WQ
BR
100%
0,5
0,4
0,3
0,2
0,1
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

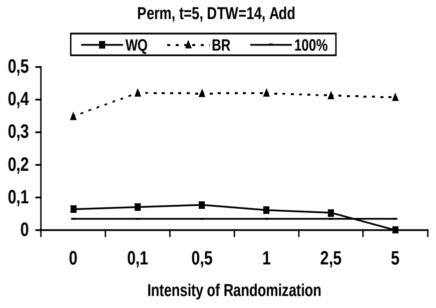

Man, t=5, DTW=24, Comb
WQ
BR
100%
0,5
0,4
0,3
0,2
0,1
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
[0, 3]
Intensity of Randomization

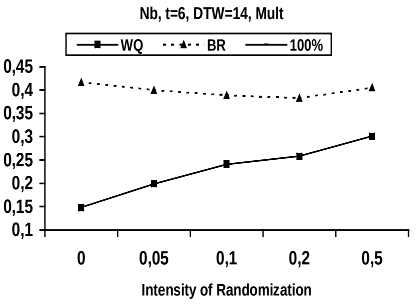

Eu, t=6, DTW=14, Add
WQ
BR
100%
1
0,8
0,6
0,4
0,2
0
0
0,1
0,5
1
2,5
5
Intensity of Randomization

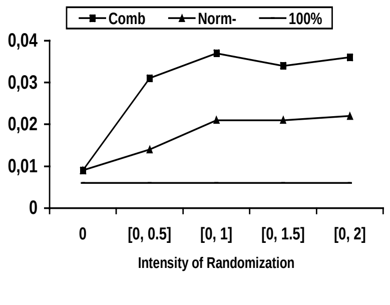

Eu, t=6, DTW=14, Mult
WQ
BR
100%
0,9
0,7
0,5
0,3
0,1
0
0,05
0,1
0,2
0,5
Intensity of Randomization

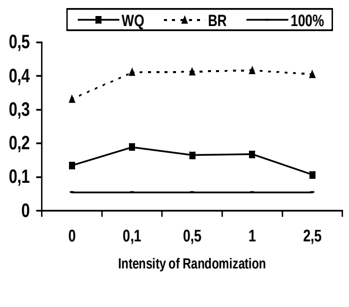

Eu, t=6, DTW=14, Norm-
WQ
BR
100%
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization


Eu, t=6, DTW=14, Comb
WQ
BR
100%
0,8
0,6
0,4
0,2
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization


Perm, t=5, DTW=14, Add
WQ
BR
100%
0,5
0,4
0,3
0,2
0,1
0
0
0,1
0,5
1
2,5
5
Intensity of Randomization


Nb, t=6, DTW=14, Mult
WQ
BR
100%
0,45
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0
0,05
0,1
0,2
0,5
Intensity of Randomization

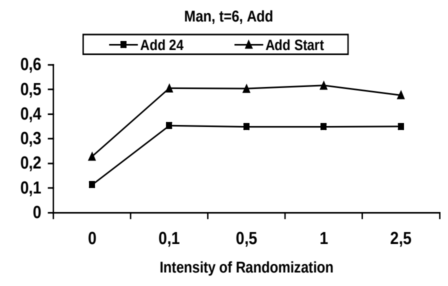

L2, t=6, DTW=14, Comb & Norm-
Comb
Norm-
100%
0,04
0,03
0,02
0,01
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

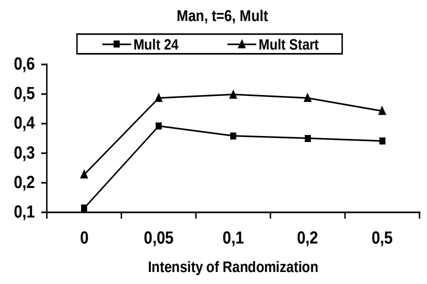

Inv, t=6, DTW=14, Add
WQ
BR
100%
0,5
0,4
0,3
0,2
0,1
0
0
0,1
0,5
1
2,5
Intensity of Randomization

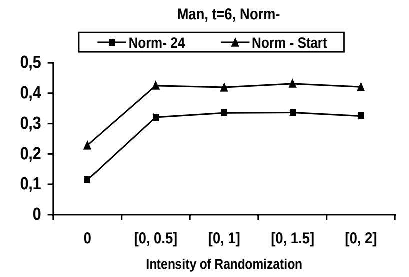
Appendix C

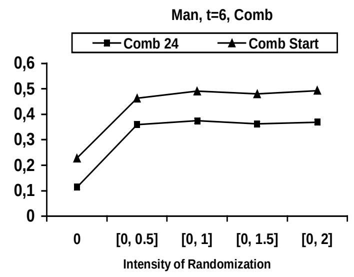
Autoplay, Start Sample

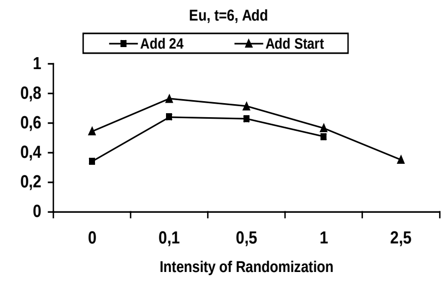
The following figures provide a comparison between the winning quotas for the sample containing 1000 times the Start position and the sample from all *SlideThree* positions in DTW=24.

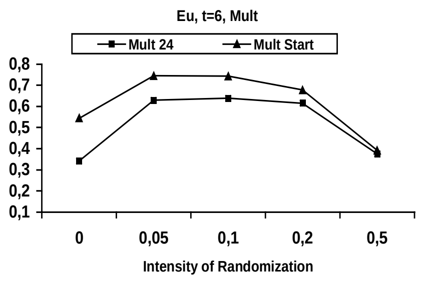


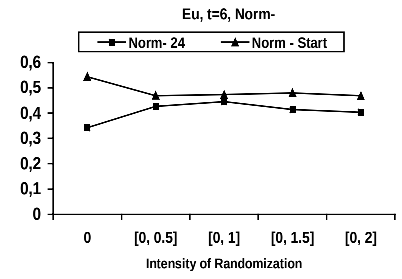


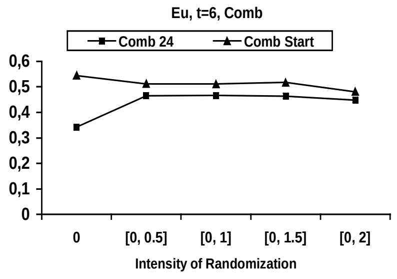





Eu, t=6, Add
Add 24
Add Start
1
0,8
0,6
0,4
0,2
0
0
0,1
0,5
1
2,5
Intensity of Randomization


Eu, t=6, Mult
Mult 24
Mult Start
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0,05
0,1
0,2
0,5
Intensity of Randomization

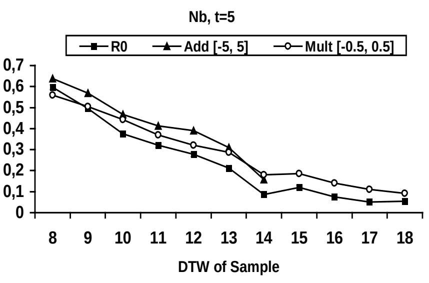

Eu, t=6, Norm-
Norm- 24
Norm - Start
0,6
0,5
0,4
0,3
0,2
0,1
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

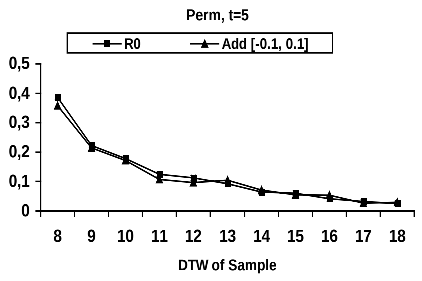

Eu, t=6, Comb
Comb 24
Comb Start
0,6
0,5
0,4
0,3
0,2
0,1
0
0
[0, 0.5]
[0, 1]
[0, 1.5]
[0, 2]
Intensity of Randomization

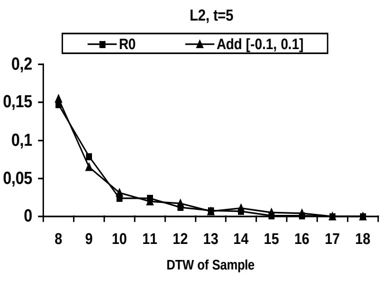
Appendix D

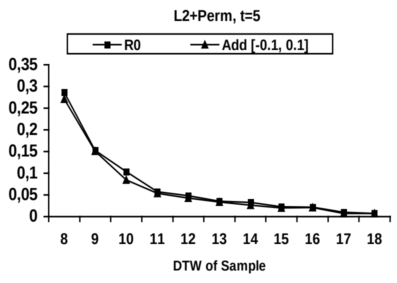
Autoplay, DTWs 8 to 18

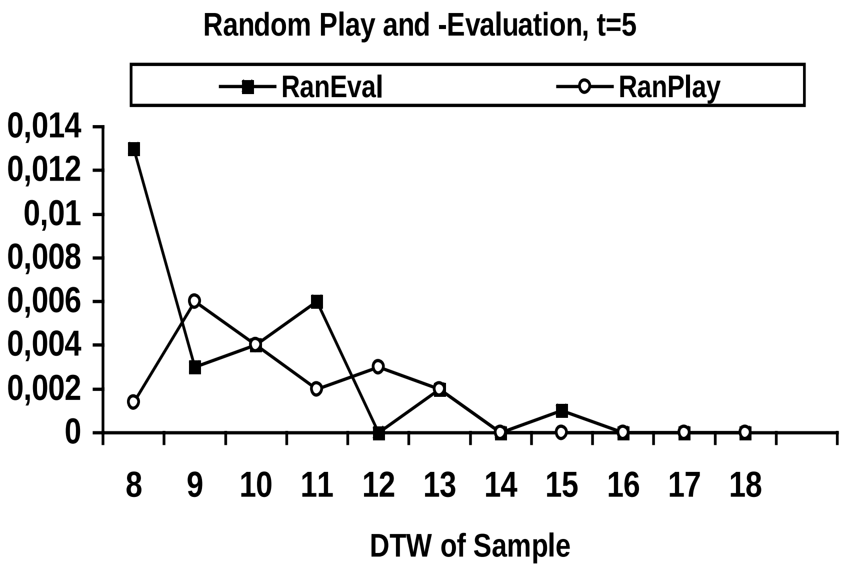
In this chapter the results of the autoplay experiments on samples from DTW=8 to DTW=18 are presented. In the following figures, R0 denotes the deterministic evaluation function (no randomization). The used randomization and the interval from which the random numbers were taken is indicated. Finally, two figures give the comparison of Random Play and Random Evaluation.










Random Play and -Evaluation, t=5
RanEval
RanPlay
0,014
0,012
0,01
0,008
0,006
0,004
0,002
0
8
9
10
11
12
13
14
15
16
17
18
DTW of Sample

Appendix E

TileTab

In the following table, a comparison between the deterministic Manhattan Metric and the TileTab randomization is made. The winning quota of TileTab is

DTW	**t**	**Man**	**TileTab**
24	4	0.063	0.075
24	5	0.071	0.187
24	6	0.114	0.347
14	4	0.169	0.232
14	5	0.271	0.406
14	6	0.353	0.579

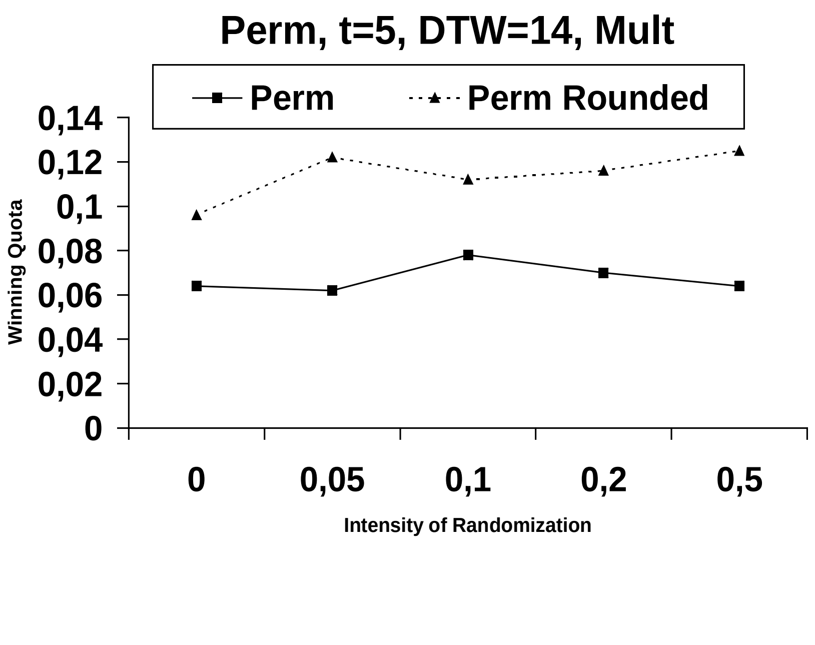
Table E.1: Winning Quota of TileTab Randomization

always significantly better, except for DTW=24, search depth 4.

Appendix F

Rounded Evaluation Functions

In Subsection 4.2.3 we discussed the coarseness of the deterministic evaluation functions and illustrated that it influences the number of ties. A coarse grained evaluation function leads to more tie situations than a fine grained one. Tie situations have the advantage that they offer possibilities for randomized play, and thus a way to leave cycles. Therefore the attempt was made to round the two finest evaluation functions, L2 and Perm, in order to produce more tie situations. Autoplay experiments were run for the rounded versions of L2 and Perm on a sample in DTW=14, with search depth t=5 and all types of randomization. In all experiments, an increase of the winning quota was observed, which was most of the time below significance level. Only the results for the Permutation Distance with randomization Mult, show a significant improvement. This is illustrated in Figure F. It was proved by I. Althöfer in [3] that it is possible to increase the coarseness of a linear evaluation function without losing important information regarding the game positions.



Appendix G

Program Descriptions

G.1 makesample.exe

The program *makesample.exe* creates samples of *SlideThree* positions that can be processed by *comexp.exe* and *autoplay.exe*.

When running the program be sure that the files *DTWxx* (xx=0, 1, ..., 26) and *PermTab1* and *PermTab2* are in the same directory as the executable file *makesample.exe*.

At the start of the program, the user is asked to specify the following parameters:

- `int` DTW of sample
- `int` sample size
- `int` seed for random numbers generator
- `char[30]` name of output file

The DTW of the sample can be any number between 5 and 26, as well as 0. When DTW 0 is chosen, the program creates a sample file from all *SlideThree* positions.
The sample size can be chosen between 1 and 5000. If the sample size is higher than there are positions in a certain DTW, the user is informed and asked to enter a suitable sample size.
One integer is required to initialize the random numbers generator.
At last, the name of the output file has to be entered. The program writes the sample positions to this file. It is created in the same directory where *makesample.exe* is, unless a different path has been specified by the user.

G.2 comexp.exe

Input

In one run of the program, up to 4 experiments can be performed. The number of experiments to be run is chosen by the user at the start of the program. For each of the experiments, the user is asked to specify the following parameters:

1. `float` Weights for the different evaluation functions:
   `Man, L2, Eu, Perm, Nb, Tile, Inv`

2. `int` Type of randomization to use
3. `int` Type of random numbers to use
4. `float` Weight for random numbers
5. `int` First search depth
6. `int` Last search depth
7. `int` Seed for random numbers generator
8. `char[30]` Name of input file
9. `char[30]` Name of output file

In the experiment, the linear combination $w_1 \cdot \text{Man} + w_2 \cdot \text{L2} + \cdots + w_7 \cdot \text{Inv}$ is used as the deterministic evaluation function, w_j being the weights entered by the user in Point 1.

Important: `Tile` is the weight for the TileTab randomization. It will not occur as part of the linear combination. It is implemented as an independent evaluation function, which is why it is listed with the evaluation functions rather than among the randomizations. To use TileTab, all other weights for the evaluation functions must be set to 0. The experiment must be run without randomization. The randomization in TileTab is implicit and will be done without command.

In Point 2, the user can choose between the different kinds of randomization explained in 4.3, no randomization, and Random Play.
The user is then asked to choose the kind of interval, where the random numbers should be taken from (Point 3). Here the structure of the randomization is determined (one- or two-sided, open or closed intervals).
Finally, the user can influence the intensity of the randomization by setting the weight for the random numbers.

Example: If we choose (Add) as randomization with a random number taken from the interval $[-0.5, 0.5]$ and set the weight for the random numbers to 0.2, a random number will be taken out of the interval $[-0.5 \cdot 0.2, 0.5 \cdot 0.2] = [-0.1, 0.1]$ and added to the deterministic evaluation function.

In one experiment, several depth-t-searches can be run on the same sample set. The user chooses the first and last search depths in Points 5 and 6. For each of the search depths *first*$\leq t \leq$ *last*, a depth-t-search will be performed and the results collected.
After the search depths are set, the user is asked to enter a seed to initialize the pseudo random numbers generator.
Finally the names of the input and output files are required. The input file must be a text file containing sample *SlideThree* positions stored in the following structure: Each position takes up two lines in the file. The first line holds the pattern code, the second line the order code of the position. The positions must be stored in the file without gaps, i.e. no empty lines or "garbage" between them. Note that the program needs the path to the input file, if it is not in the same directory as *comexp.exe*.
Processable input files can be created with the program *makesample.exe* (cf. G.1.

The program *comexp* will create a text file of the name specified in Point 9, where all the output is written to.

Random Evaluation and Random Play Experiments with Random Evaluation can be run by setting all weights for the evaluation functions to zero and then choosing (Add) as randomization with a random number from (0, 1). The open interval guarantees, that the Win is found if it occurs in the searched tree.

The Hits for Random Play can be computed by setting all weights of the evaluation functions to 0 and choosing no randomization. In this case, all nodes obtain 0 as their heuristic value and the Hits then estimate the probability of choosing a good move randomly, as done in Random Play.

Output

The program calculates for each of the sample positions the Hits, Fakes, Misses, and Keeps (cf. 5.1.1). Their arithmetic means over all positions in the sample are written to the output file. The output file also contains the parameters that were specified by the user when starting the program. Thus, typing mistakes can be detected and the experiment reproduced. As additional information, the number of sample positions with "all best successors" is provided for each search depth. "All best successors" means, that all successors obtained the same heuristic value in the depth-t-search.

G.3 autoplay.exe

Input

The program for running an autoplay experiment is included on the CD-ROM, file *autoplay.exe*. It takes as input the following parameters:

1. `float` Weights for the different evaluation functions:
   `Man, L2, Eu, Perm, Nb, Tile, Inv`
2. `int` Type of randomization
3. `int` Type of random numbers
4. `float` Weight for random numbers
5. `int` Search depth
6. `int` Upper bound for the moves in one game
7. `int` Number of positions to be read from file and played
8. `int` Seed for random numbers generator
9. `char[30]` Name of input file
10. `char[30]` Name of output file

The parameters 1 to 4 are the same as in G.2. The only difference is, that also Random Play can be chosen in Point 2. Random Play works as described in 5.2.3.
The search depth specified in Point 5 is used for the depth-t-searches to determine the next move.
In autoplay, the user must set upper bounds for the number of moves to play in one game, and the number of games to play (Points 5 and 6). Playing will be stopped automatically, if the end of the input file is read.
Finally a seed to initialize the random numbers generator is required and the file names of the input and output files. The input file must be of the same structure as for the CoM experiments (cf. G.2). If the input file is in the same directory as autoplay.exe, the filename suffices. Otherwise, the path has to be specified by the user.

Random Evaluation The program can be set to run experiments with Random Evaluation by the following parameters:

- All weights for evaluation functions 0
- Randomization 1 (Add)
- Random numbers 3 ((0, 1))
- Weight for random numbers 1

The choice of the open interval guarantees that the Win is identified in the tree search.

Output The program produces an output file containing the input parameters as a header. Thus it is possible to detect typing mistakes and to reproduce an experiment. The program keeps track of the number of won and played games, from which the winning quota is calculated. Winning quota and fat quota are written at the end of the file. The fat quota is called *break rate* in the output file. Furthermore, some other, probably interesting properties of the experiments are written to the file:

1. Maximum number of moves needed to win a game
2. Minimum number of moves needed to win a game
3. Mean number of moves needed to win a game
4. Number of ties (sum over all games)

In some of the autoplay result files included on the CD-ROM, you will also find a number called "Collisions". This is supposed to give the number of Type1 collisions that occurred in the transposition table during the experiment. It is mere additional information and in the early experiments, it was even computed wrong. Therefore, it was finally erased from the output of the program. Also included in the older result files is a list of all played games. One entry consists of the starting position of the game, the number of moves played and an indication whether the Win was found.

G.4 autoplayIA.exe

This program also plays games of *SlideThree* automatically, but lists the single moves of the game. When starting the program, the following parameters have to be entered in the start menu by the user:

1. `float` Weights for the different evaluation functions:
   `Man, L2, Eu, Perm, Nb, Tile, Inv`
2. `int` Type of randomization
3. `int` Type of random numbers
4. `float` Weight for random numbers
5. `int` Search depth
6. `int` Seed for random numbers generator
7. `char[30]` Name of output file

These parameters are the same as for *autoplay.exe*. The maximum number of moves per game is 100 and cannot be set by the user.
For each game that is played, the user is asked to specify the starting position. This is done by entering the codes for pattern and order of the position (cf. 3.3.2). Then playing starts.
While playing, the following information appears on the screen and is written to the output file:
First, the current position is put out by its codes for pattern and order and its DTW. Beneath the current position, the possible successors are listed. From left to right each entry contains

1. pattern code of the position
2. order code of the position
3. heuristic value of the position after depth-t-search with randomized evaluation function
4. heuristic value of the position after depth-t-search with deterministic evaluation function
5. number of minimal nodes in the subtree of the position

In case that no randomization was chosen for playing, Points 3 and 4 will both show the heuristic value of the position. The number of minimal nodes as shown in Point 5 counts the minimal nodes for this successor. I.e. if the successor obtains the heuristic value k, all nodes with value k in its subtree are counted. Note that for this, the values with respect to a depth-t-search with the deterministic evaluation function are considered. Beneath this list, the number of collisions in the transposition table is reported (Type1 collision, cf. 4.1.2). Also, a tie with respect to the deterministic depth-t-search is announced and if it is broken in the direction of the successor with the most minimal nodes, the message "Tie broken to fuller side!" and the number of minimal nodes are printed on the screen. Finally, the move that was executed is given by the new current position. On pressing "Return", the next move is executed.
After each game, the user can choose whether he wants to quit the program or see another game.

Appendix H

SlideThree

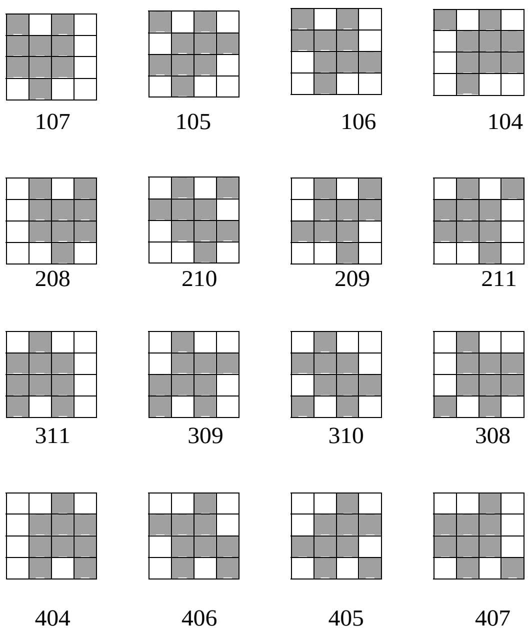


Figure H.1: Patterns that are Solvable for Odd Permutations

7	8	9	
5	4	6	
1	2	3	

6	7	8	
9	5	1	
2	3	4	

Figure H.2: Nice Starting Positions in DTW 26

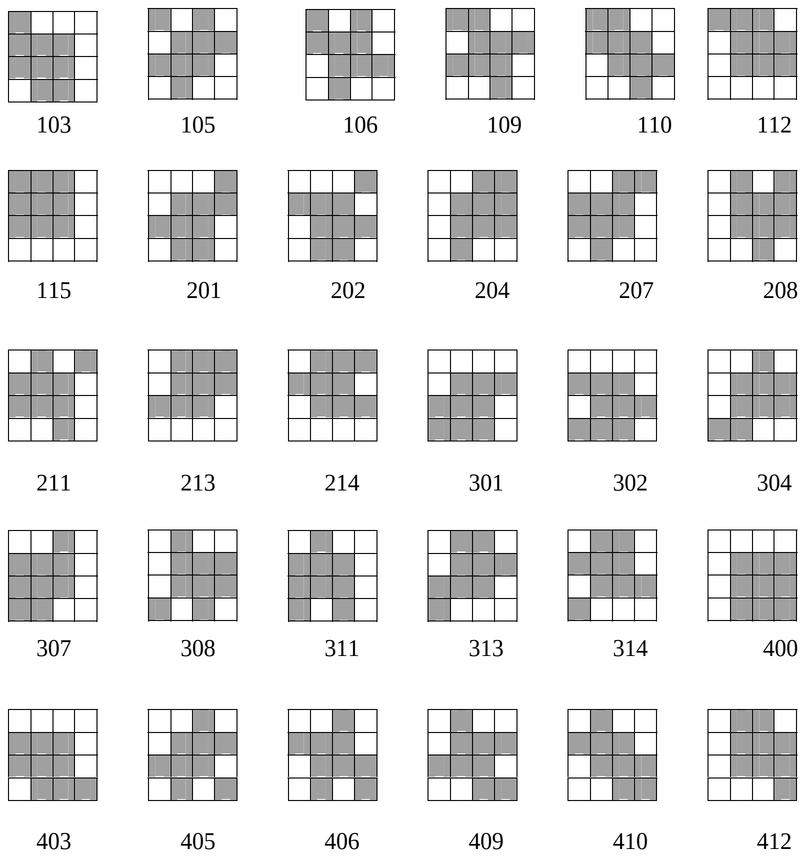


Figure H.3: Patterns of Positions in DTW=26 with Code

Appendix I

Zillions playing *SlideThree*

Zillions of Games (short: *Zillions*) is a software that can be easily programmed to play various games. The search algorithm and evaluation function are provided by Zillions as black box, only the rules of the game and the graphics used for generating the playing board have to be programmed. Before starting a game, the user can specify some parameters that vary the playing strength of *Zillions*.

- Playing Strength
  Choice between Beginner, Advanced, or Expert
- Variability
  Intensity of randomness
- Time Limit
  The time *Zillions* may use to determine the next move

Zillions was set up to play *SlideThree* and its variants *SlideTwo* and *SlideFour* (cf. 3.2.3). The following tables give the average number of moves *Zillions* needed to solve the puzzle. Playing always started at the Start position. In all experiments, the Playing Strength was set to Expert. The Variability (var) and Time Limit (time) were varied.

var	1 (low)	5 (medium)	11 (high)
av. moves	10	10	10.3

Table I.1: SlideTwo, Average Over 6 Games Each

	var=1 (low)	**var=5 (medium)**	**var=11 (high)**	*Sum*
time=1s	30	29.7	26.7	*96.4*
time=2s	36.7	39.3	35.3	*111.3*
time=5s	32	37.3	34.7	*104*
time=10s	35.3	34	30.7	*100*
Sum	*134*	*140.3*	*137.4*	

Table I.2: SlideThree, Average Moves Over 3 Games Each

The shortest solution for *SlideTwo* is 10 moves. For *SlideFour* the shortest solution is unknown. The shortest solution found by *Zillions* in the experiments was 124 moves.

	var=1 (low)	**var=5 (medium)**	**var=11 (high)**	*Sum*
time=1s	340.7	330.7	227.3	*898.7*
time=2s	146	220.7	228.7	*595.4*
time=5s	368.7	256.3	234.7	*868.7*
time=10s	214	138.7	131.7	*484.4*
Sum	*1069.4*	*946.4*	*822.4*	

Table I.3: SlideFour, Average Moves Over 3 Games Each

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel angefertigt habe.

Jena, den 13. 08. 2003