RANDOMISED EVALUATIONS IN SINGLE AGENT SEARCH

Ingo Althoefer*
Faculty of Mathematics and Computer Science
Friedrich–Schiller–University Jena
07737 Jena, Germany

Susanne Heuser†
ZAIK, University of Cologne
Weyertal 80
50931 Cologne, Germany

February 16, 2005

Abstract

Most game playing programs are based on heuristic game tree search:
the nodes at distance t from the root are evaluated by a heuristic evaluation function, and these values are backed up to the root by minimaxing.
Following an old investigation by Beal and Smith [1], we discuss depth-t search with *randomised* evaluation functions: the heuristic leaf evaluations are artificially randomised before backing them up. Our experiments were done for single-agent search, namely in the sliding tile puzzle *SlideThree* [7]. The main finding is that applying a random disturbance to the heuristic leaf values may considerably improve the playing strength.

1 Introduction

A standard method in game playing programs is heuristic game tree search. This method has been extremely successful and largely investigated for two player games like chess (see e.g. [2]). Beal and Smith conducted experiments with randomised evaluation functions for a full minimax search in chess [1]. They found that randomising the heuristic evaluation function led to better play. Their conjecture was that the randomised version of the evaluation function tends to prefer (and select) positions with higher mobility. Other experiments conducted for several simple two player games (like Blobs, Domineering, and others) by Rolle showed mixed results of success when randomising the heuristic evaluation function [6].
We conducted first experiments with randomised evaluations for single player games and describe our findings in this report. Our subject of investigation was Troyka's puzzle *SlideThree* [7] with its about 11.6 million different positions. One advantage of this relatively small search space was that results of perfect retrograde analysis were available for comparison.
The note is organized as follows: In Section 2 we present the puzzle *SlideThree* and state some of its properties. Section 3 explains the method that was used for the tree search and discusses why randomisation might be helpful when searching the *SlideThree* game tree. The design of our experiments is explained in Section 4. Results are shown and discussed in Section 5. Section 6 consists of conclusions, conjectures, and open problems. This article is based on [4].

*email:althofer@minet.uni-jena.de
†email:heuser@zpr.uni-koeln.de

2 The Puzzle *SlideThree*

The puzzle *SlideThree* was invented and programmed for *Zillions of Games* [8] by W.D. Troyka [7]. There are nine tiles that are numbered from 1 to 9 on a 4×4 board. The task is to reverse the order of the tiles by sliding blocks of *three* tiles in horizontal or vertical direction. The blocks must be selected from a row or a column. If a block from a row is chosen, it may only be slid within this row. A block from a column may only slide within this column, respectively. Figure 1 depicts the Win and Start positions and a sample sequence of two legal moves from the starting position. The

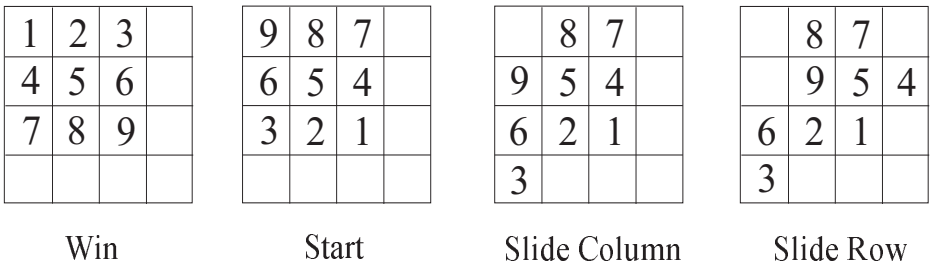

Figure 1: Win and Start positions and a sequence of two legal moves

'game' tree of *SlideThree* is rather uniform. Each position has either 4 or 5 or 6 successors. There are no cycles of odd length in the graph. With DTW (Distance To Win), we denote the minimal number of moves that are needed to transfer a *SlideThree* position into the Win position. We did retrograde analysis on all 11,612,160 legal positions and found that the Start position is in DTW 24, and the maximum DTW is 26 (cf. [4], Chapter 3).

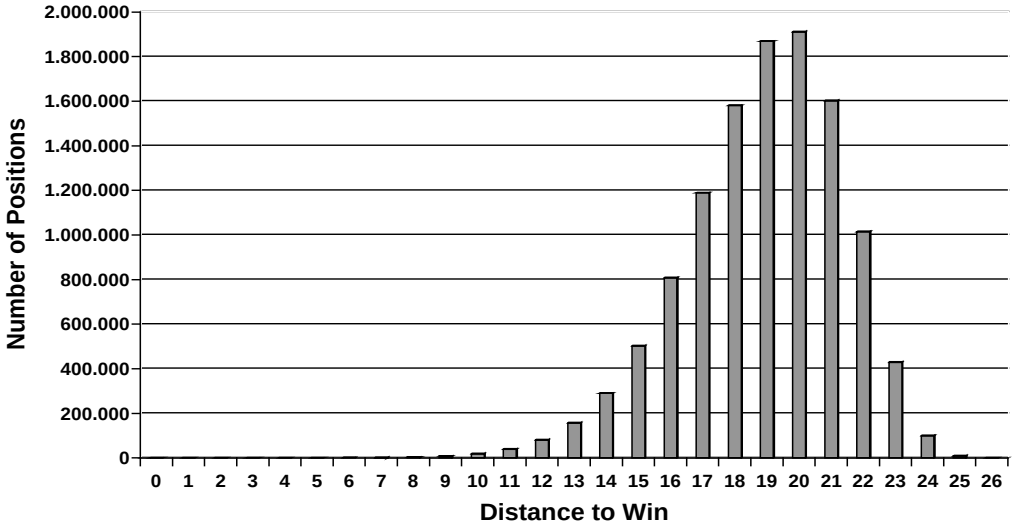

Figure 2: Distribution of *SlideThree* Positions

Figure 2 shows the distribution of *SlideThree* positions according to their DTW.

There is a straightforward generalisation of *SlideThree* to *SlideK*, the corresponding puzzle played with K^2 tiles on a board of size $(K+1) \times (K+1)$. *SlideFour* was programmed for Zillions by S. Heuser [3]. The quickest *SlideFour* solution discovered so far takes 92 moves, but the DTW of the *SlideFour* Start position remains an open problem.

3 Game Tree Search

3.1 Algorithm

We search the tree up to a fixed depth t. Basically, the following steps are performed:

1. Create the first t plies of the game tree.
2. Evaluate the leaves with a heuristic evaluation function.
3. Up-propagate the values in the tree by repeated minimising.
4. Execute the move that leads from the root to the successor with the minimal value. Ties are broken by random choice.

In the multi-game program *Zillions of Games* [8] single agent search is executed just in this way, in an iterative-deepening procedure (first a search of depth 1, then depth 2, ...).

3.2 Why Randomisation?

Applying random disturbance to deterministic heuristic leaf values might have the following benefits for a game playing program.

1. **Stepping out of Cycles**
   Deterministic evaluation functions behave always identically in the same situation. If the evaluation function chooses a certain 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. Randomisation offers a possibility to leave such cycles, thus solving this problem.
2. **Nextbest Candidate Moves**
   If randomisation is intense enough, it is able to switch the ranks of the moves evaluated best. Thus not only the best, but also the second best, third best, or k-th-best moves become candidates for execution.
3. **Tiebreak-Descend in Promising Subtrees**
   In case of *coarse-grained* heuristics 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, randomisation increases the chance to step into a subtree with many good leafs. Consider the situation shown in Figure 3. In the left subtree there are three "optimal" leaves, in the right subtree there is only a single one. After a modest randomisation, the leaf with the minimal value will be in the left subtree with probability (3/4 =) 75 percent and in the right subtree with only (1/4 =) 25 percent.

In the multi-game program *Zillions of Games* [8] the user can adjust several parameters for the computer engine. One of these parameters is called *Variety*. There are 11 different levels, ranging from *small* over *moderate* to *large*. *Variety* determines to what extent leaf values in the depth-t search are randomised: on the lowest level there is no randomisation at all, and on the other ten levels the average degree of randomisation grows linearly with the level number. There are a number of games and puzzles where large *Variety* improves the playing strength of Zillions.

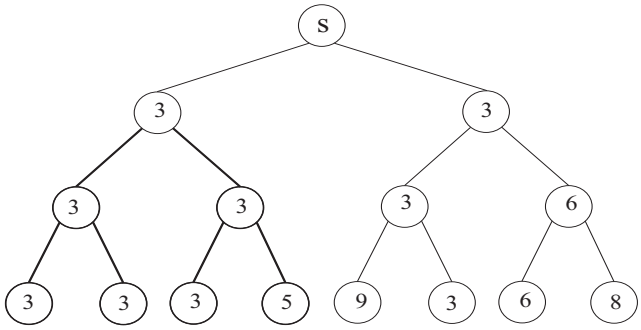

Figure 3: Example for a tie situation in a min-tree of depth 3

4 Experimental Design

We conducted autoplay experiments for samples of *SlideThree* positions. This means that we let our algorithm play *SlideThree* for up to 100 moves starting from a sample position. As a measure for the playing strength we used the winning quota WQ,

$$\mathrm{WQ} = \frac{\text{number of games won within 100 moves}}{\text{number of games played}}.$$

For each sample, 1068 starting positions were randomly chosen from several fixed DTWs. This sample size gives us a maximum standard deviation of 3% on a significance level of 95%. This means that, if two winning quotas Q_1 and Q_2 for two different evaluation functions E_1 and E_2 differ by more than 6% ($Q_1 - Q_2 > 6\%$), we can conclude that E_1 plays better than E_2. The probability that this conclusion is correct is 95% or better.

Heuristic Evaluation Functions in our Test
We tested six different heuristic evaluation functions which estimate the distance to the Win position by different criteria. The first three are natural ones whereas the others look rather artificial. This was fully intended as we wanted to analyse the randomisation approach also for strange evaluations. Here, the multi-game program *Zillions of Games* with its automatically generated and sometimes strange evaluation functions was our role model.

	8	7	
	9	5	4
6	2	1	
3			

Position P

Figure 4: Position for the Example Computations of the Evaluation Functions

Definition 1 *Let Pos be any SlideThree position. In the examples we compute the respective heuristic value for the position depicted in Figure 4, denoted by P in the following.*
(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

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

Example:

$$\begin{aligned} Eu(P) &= \sqrt{r(1)^2 + c(1)^2 + r(2)^2 + c(2)^2 + \ldots + r(9)^2 + c(9)^2} \\ &= \sqrt{2^2 + 2^2 + 2^2 + 0^2 + \ldots + 1^2 + 1^2} \\ &= \sqrt{54} \approx 7.348 \end{aligned}$$

(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

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

Example:

$$\begin{aligned} Man(P) &= d(1) + d(2) + \ldots + d(9) \\ &= 4 + 2 + \ldots + 2 \\ &= 26 \end{aligned}$$

(3) ***Neighbour Distance***
The value of Pos according to the Neighbour Distance is the sum of various mali. 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 4-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

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

Example:
Consider tile 3 in position P. Its current position is in the fourth row, first column; its goal square is first row, third column. So we already have a malus of +1 because the tile is not on its goal square. Next, we take a look at the neighbours. In P, tile 3 has no left neighbour, tile 6 as an upper neighbour, a blank as a right neighbour, and no lower neighbour. Compared to its neighbours in the Win position, it misses tile 2 as its neighbour on the left. This gives a malus of +1 for a missing neighbour. In the Win position, tile 3 is located in the first row, thus having no upper neighbour. Therefore we get another malus of +1, because of an additional upper neighbour in the current position. The right neighbour of tile 3 in P is a blank, which is consistent with its right neighbour in the Win position. So we have no malus for the right neighbour. The lower neighbour of tile 3 in the Win position is tile 6. But in P, a lower neighbour is missing, which gives us another malus of +1. Altogether, the malus for tile 3 in P is $m(3) = 1 + 1 + 1 + 0 + 1 = 4$. *Treating the other tiles analogously, we obtain*

$$\begin{aligned} Nb(P) &= m(1) + m(2) + m(3) + m(4) + m(5) + m(6) + m(7) + m(8) + m(9) \\ &= 5 + 5 + 4 + 5 + 5 + 5 + 5 + 5 + 5 \\ &= 44 \end{aligned}$$

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

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

The additive term 0.5 in the root guarantees that Perm(Pos) becomes 0 only for the Win position.

Example:
First we determine the permutation of the tiles within P, starting in the upper left corner and proceeding rowwise to the lower right corner. Blanks are skipped:

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

Now we can compute the heuristic value according to Perm:

$$\begin{aligned} Perm(P) &= \sqrt{0.5 + (8-1)^2 + (7-2)^2 + (9-3)^2 + \ldots + (3-9)^2} \\ &= \sqrt{0.5 + 49 + 25 + 36 + \ldots + 36} \\ &= \sqrt{222.5} \approx 14.916 \end{aligned}$$

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

$$Inv(Pos) = \begin{cases} 0.5 + number\ of\ inversions\ of\ \pi & \text{if } Pos \neq Win \\ 0 & \text{if } Pos = Win \end{cases}$$

Example:
An inversion is a switch of the lexicographical order of two numbers, e.g. (8,7) is an inversion in π, (8,9) is not. The number of all such switches in $\pi(P)$ is 30.

$$\begin{aligned} Inv(P) &= 0.5 + inv(\pi) \\ &= 0.5 + 30 = 30.5 \end{aligned}$$

(6) ***L2 Metric***
To define this evaluation function, we convert Pos into a vector v(Pos) in the 16-dimensional Euclidian space $\mathbf{R}^{16}$. This is done rowwise, starting in the upper left corner. The first four entries of the vector correspond to the first row of Pos, the second four entries to the second row of Pos, and so on. 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):

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

Example:
The vector corresponding to P is $v(P) = (0,\ 8,\ 7,\ 0,\ 0,\ 9,\ 5,\ 4,\ 6,\ 2,\ 1,\ 0,\ 3,\ 0,\ 0,\ 0)$.

$$\begin{aligned} \text{L2(P)} &= \sqrt{(0-1)^2 + (8-2)^2 + (7-3)^2 + (0-0)^2 + \ldots + (0-0)^2} \\ &= \sqrt{1 + 36 + 16 + 0 + \ldots + 0} \\ &= \sqrt{192} \approx 13.856 \end{aligned}$$

Randomisations in our Test
In our experiments each of the six evaluation functions was randomised in five different ways:

1. **Add:** Add a uniformly distributed random value (rv) to the deterministic evaluation function (evf). So, evf becomes evf + rv , with $-a \leq \text{rv} \leq a$.
   This is the easiest way of randomising a deterministic evaluation function. The intensity of the randomisation is scalable by changing the range of the random value, but depends on the absolute values of the heuristic evaluation function.

2. **Mult:** Multiply the evaluation function by (1+rv). So evf becomes $\text{evf} \cdot (1 + \text{rv})$, with $-a \leq \text{rv} \leq a$.
   This is a relative randomisation of the evaluation function. Its intensity no longer depends on the range of the deterministic evaluation function.

3. **Norm+:** Substitute evf by $\frac{\text{evf+rv}}{\text{evf}} = 1 + \frac{\text{rv}}{\text{evf}}$, where $0 \leq \text{rv} \leq a$.
   When using this randomisation, the ranking of the moves as determined by the deterministic evaluation function is reversed on average. A high deterministic evaluation is likely to give a low heuristic value when using Norm+. Thus, the moves considered as being bad by the deterministic evaluation function are preferred for execution when using this randomisation. One could say that the "knowledge" of the deterministic evaluation function is rejected. This very strange randomisation is not meant as a serious proposal for application! Instead its investigation shall only demonstrate that it is possible to decrease performance by randomisation.

   Example: Consider two *SlideThree* positions P and R with deterministic values evf(P)= 3 and evf(R)= 2. Drawing uniformly distributed random numbers rv from the interval $[0, 5]$ gives Norm+(P) in the interval $[1, 1 + \frac{5}{3}]$ and Norm+(R) in the interval $[1, 1 + \frac{5}{2}]$. So, on average, P, which has a worse heuristic value before randomisation, gets a better randomised value.

4. **Norm–:** $\frac{\text{evf-rv}}{\text{evf}} = 1 - \frac{\text{rv}}{\text{evf}}$, $0 \leq \text{rv} \leq a$.
   Norm– is the mirrored version of Norm+. If Norm+ orders two positions in one way, Norm– is likely to reverse the order. In other words, the ranking of the moves as obtained by the deterministic evaluation function is reproduced on average.

5. **Comb:** Add (rv $\cdot Perm$) to the deterministic evaluation function.
   So evf becomes $\text{evf} + \text{rv} \cdot Perm$, with $-a \leq \text{rv} \leq a$.
   $Perm$ is the Permutation Distance as defined in Definition 1, (4). So, here a linear combination of two deterministic evaluation functions is attempted with a random weight for the Permutation Distance. The deterministic evaluation function consults a second "expert" (which is $Perm$) and integrates this "knowledge" randomly into its own evaluation.

   Example: Consider a position P with deterministic value E_P, where E is one of the evaluation functions defined in Definition 1. The randomised value of P is Comb(P)= E_P + rv $\cdot$ Perm(P). If the absolute value of rv is small, only little information from the Permutation Distance is included in the final value Comb(P). If it is large, a considerable amount of the Permutation Distance evaluation is contributed to Comb(P).

In our experiments all random numbers are generated by the pseudo random number generator "Mersenne Twister" by Matsumoto and Nishimura [5] and are uniformly distributed in an interval $[-a,\ a]$ or $[0,\ a]$, $a \in \mathbf{R}^+$. We tried different values for a in lots of different runs.

5 Results of the Experiments

All results in detail may be found in [4]. Here we give an overview. We conducted autoplay experiments for each of the heuristic evaluation functions and each kind of randomisation for a sample of 1068 *SlideThree* positions in DTW 14. The search depths were 4, 5, and 6. Tables 1 to 3 list our findings from the experiments with randomised heuristic evaluation functions on this sample. Here, "+" means a statistically significant increase, "–" a decrease, and "≈" no significant change of the winning quota.

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

Table 1: Effects of Randomisation for a Depth-4 Search on Sample in DTW 14

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

Table 2: Effects of Randomisation for a Depth-5 Search on Sample in DTW 14

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

Table 3: Effects of Randomisation for a Depth-6 Search on Sample in DTW 14

In many cases, we observed a quite considerable increase of the winning quota when randomising the deterministic heuristic evaluation function. Also, the positive impact of randomisation on the winning quota increased with the search depth, as may be seen by the more technical tables in the appendix.

When taking a closer look at the results, we can give a refined distinction of the behaviour of the winning quota in dependence on the randomisation intensity. Five different patterns were observed.

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

2. The winning quota increases only after a certain intensity of randomisation has been reached. When the randomisation becomes too intense it decreases again.
3. The winning quota does not change when randomisation is used. Once the randomisation reaches a certain intensity, the winning quota decreases.
4. The winning quota drops to 0 as soon as randomisation 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 4: Behaviour of Winning Quota

Table 4 lists the evaluation functions and their type of behaviour. It represents a summary of all the experiments that were conducted and of which only a part is shown in detail in this note. The entries with a 5 are additionally marked with a +, −, or ≈ sign, to indicate the change in the winning quota when using randomisation. The classification of the behaviour of the L2 Metric fails because of too low winning quotas.

In Figure 5 we show the winning quotas of the deterministic evaluation function Eu and its randomised version Eu-Add (using the Add randomisation with random values taken from [–0.1, 0.1]). Here we took samples from several DTWs (DTW = 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18). Each data point shown in the figure is the winning quota achieved in the experiment on the respective sample set of 1068 *SlideThree* positions. The DTW of the positions in the sample set is indicated on the x-axis.
In the figure, "Det" refers to the winning quota of the deterministic heuristic evaluation function Eu and "Rand" to its randomised version. Both for deterministic and randomised evaluations the winning quotas decrease with increasing DTW, and for larger DTW the positive effect of randomisation is more explicit.

6 Conclusions

We investigated randomised evaluation functions in single agent search. For this purpose, six different deterministic heuristic evaluation functions for the puzzle *SlideThree* were constructed, which show different playing strengths. These evaluation functions were randomised in five different ways. The experiments were set up as autoplay experiments and applied a depth-t search to select a move for execution. Playing strength was measured by the winning quota.

Our experiments show that a randomisation often is able to improve the playing strength. The positive effect of randomisation increases with the search depth. Moreover, there should be an upper bound for the playing strength of the deterministic evaluation function beyond which randomisation cannot lead to further improvement. This implies that randomisation is most profitable for deterministic evaluation functions of moderate playing strength.

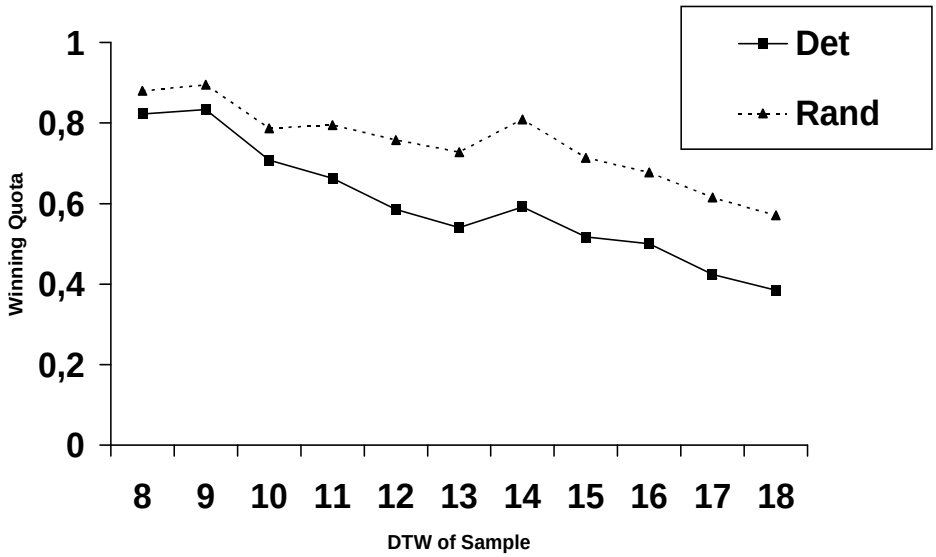

Figure 5: Effect of Randomisation

We finish with two open problems: (i) Given a single agent game with heuristic search: is there always some medium level of quality for the heuristic evaluation such that randomisation is most helpful for heuristics at this level? (ii) Does randomisation also help for other procedures instead of depth-t search, for instance for A*?

7 Acknowledgements

Thanks are due to W.D. Troyka for the publication of his nice puzzle *SlideThree*. Susanne Heuser wants to thank especially Prof. van den Herik and his research group at the University of Maastricht, where she had a very fruitful 3-month's stay during her diploma project. The FRZ (Fakultaets-Rechen-Zentrum) at Jena University with its director Dr. Schorr was so kind to provide extra computing time for very intense experimental sessions.

References

[1] D.F. Beal and M.C. Smith. *Random Evaluations in Chess.* ICCA Journal, Vol. 17, No. 1, pp.91-94, 1994.

[2] E.A. Heinz. *Scalable Search in Computer Chess.* Vieweg, 2000.

[3] S. Heuser. *Puzzle SlideFour.* 2003. An electronic version is available at http://www.zillionsofgames.com/cgi-bin/zilligames/submissions.cgi/99611?do=show;id=828.

[4] S. Heuser. *Randomised Evaluation Functions in Single Agent Search.* Diploma Thesis, Friedrich-Schiller-University Jena, Faculty of Mathematics and Computer Science, 2003. An electronic version is available on request from heuser@zpr.uni-koeln.de .

[5] M. Matsumoto and T. Nishimura. *Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator.* ACMTMCS: ACM Transactions on Modeling and Computer Simulation, Vol.8, pp. 3–30, 1998.

[6] T. Rolle. *Development of a multi-game engine.* Diploma Thesis, Friedrich-Schiller-University Jena, Faculty of Mathematics and Computer Science, 2003. An electronic version is available on request from althofer@minet.uni-jena.de .

[7] W. D. Troyka. *Puzzle SlideThree.* 2001. An electronic version is available at http://www.zillionsofgames.com/cgi-bin/zilligames/submissions.cgi/99744?do=show;id=829.

[8] Zillions. *Multi-game program Zillions of Games.* 1998. http://www.zillionsofgames.com . Updated weekly.

Appendix

Tables A1 to A3 show the best winning quotas that were achieved in the experiments. For each pair of evaluation and randomisation, we tried several intervals for the random numbers. The tables show the highest winning quotas achieved and in [ ]-brackets the corresponding intervals from which the random numbers were taken. These intervals are meant for orientation. In fact, our experiments showed a significant increase of the winning quotas for a broad range of intensities of randomisation. Column Det refers to the deterministic evaluation function without randomisation.

	Det	**Add**	**Mult**	**Norm+**	**Norm–**	**Comb**
Eu	0.355	0.421 [-0.1, 0.1]	0.452 [-0.1, 0.1]	0.001 [0, 0.5]	0.380 [0, 1]	0.343 [0, 2]
Man	0.169	0.254 [-2.5, 2.5]	0.245 [-0.3, 0.3]	0.001 [0, 1]	0.271 [0, 1]	0.238 [0, 2]
Nb	0.046	0.099 [-5, 5]	0.110 [-0.3, 0.3]	0	0.117 [0, 1]	0.109 [0, 2]
Perm	0.045	0.052 [-5, 5]	0.040 [-0.3, 0.3]	0	0.045 [0, 0.2]	
Inv	0.050	0.065 [-0.5, 0.5]	0.047 [-0.5, 0.5]	0	0.054 [0, 0.5]	0.051 [0, 1]
L2	0.003	0.016 [-2.5, 2.5]	0.007 [-0.5, 0.5]	0.001 [0, 1]	0.017 [0, 0.5]	0.010 [0, 0.5]

Table A1: Winning Quotas for a Depth-4 Search on Sample in DTW 14

	Det	**Add**	**Mult**	**Norm+**	**Norm–**	**Comb**
Eu	0.463	0.638 [-0.005, 0.005]	0.645 [-0.1, 0.1]	0.001 [0, 1]	0.568 [0, 2]	0.531 [0, 0.5]
Man	0.271	0.460 [-2.5, 2.5]	0.443 [-0.2, 0.2]	0	0.481 [0, 1]	0.431 [0, 0.5]
Nb	0.086	0.181 [-0.5, 0.5]	0.137 [-2.5, 2.5]	0	0.219 [0, 0.3]	0.200 [0, 2]
Perm	0.064	0.107 [-1, 1]	0.078 [-0.1, 0.1]	0	0.104 [0, 0.2]	
Inv	0.063	0.132 [-0.5, 0.5]	0.074 [-0.2, 0.2]	0.001 [0, 2]	0.138 [0, 1]	0.083 [0, 0.5]
L2	0.007	0.038 [-5, 5]	0.013 [-0.2, 0.2]	0.001 [0, 0.5]	0.041 [0, 0.2]	0.022 [0, 1.5]

Table A2: Winning Quotas for a Depth-5 Search on Sample in DTW 14

	Det	**Add**	**Mult**	**Norm+**	**Norm–**	**Comb**
Eu	0.592	0.799 [-0.5, 0.5]	0.818 [-0.1, 0.1]	0	0.737 [0, 2]	0.682 [0, 2]
Man	0.353	0.666 [-2.5, 2.5]	0.623 [-0.2, 0.2]	0	0.684 [0, 2]	0.632 [0, 1.5]
Nb	0.148	0.326 [-5, 5]	0.301 [-0.5, 0.5]	0.001 [0, 0.5]	0.400 [0, 0.3]	0.344 [0, 3]
Perm	0.139	0.192 [-1, 1]	0.178 [-0.5, 0.5]	0.001 [0, 0.5]	0.203 [0, 2]	
Inv	0.134	0.220 [-1, 1]	0.197 [-0.5, 0.5]	0	0.228 [0, 2]	0.193 [0, 1]
L2	0.009	0.092 [-2.5, 2.5]	0.022 [-0.5, 0.5]	0.001 [0, 0.5]	0.088 [0, 0.5]	0.037 [0, 1]

Table A3: Winning Quotas for a Depth-6 Search on Sample in DTW 14