
Known for Outstanding Performance
Sticks & Stones May Break My Bones
Rock'n'Roll
Hurt Me
Can never
Redefining the Limits since 2004
Hurt Me

Friedrich Schiller University Jena
Faculty of Mathematics and Computer Science
07743 Jena, Germany

Winter Term 2005/2006

Student Research Project

Rock'n'Roll

A Cross-Platform Engine for the Board Game "EinStein würfelt nicht"™

Andreas Schäfer*

January 2, 2006

Prof. Dr. Ingo Althöfer (Supervisor)
Prof. Dr. Wilhelm Rossak (Supervisor)

*gentryx@gmx.de

Contents

1 Road Map

In 2004, Ingo Althöfer invented the highly addictive board game "EinStein würfelt nicht"™ (called EinStein for short in the following). Its key features are relatively short runs and a unique mixture of chance and strategy. Particularly, the optimal strategy is not obvious and since he is the chair of mathematical optimization, there was soon a lively debate at the faculty, how to place the stones, how to play the opening, where to defend and when to capture. Despite its straightforward rules, the exact analysis of this game is rather complex (read: compute intensive). To cast light on these problems, he needed some means of analysis: a game engine or – more euphonious – artificial intelligence which could play EinStein well. Since I study computer science and have always been interested in maths and game theory, I took on the offer to code the engine as a student project. This task included the mathematical/theoretical analysis of the game, the realization of the therefrom distilled algorithms in a computer program, and a sufficient tuning of parameters.

The acceptance term was that it should win at least 10 out of 20 games against Dr. Stefan Schwarz – the strongest player at that time[1] – at the end of the term. The surprising result was that Rock'n'Roll won 17-3 and proved that sometimes even computers have the luck of the devil[2].

The chapters of this report closely relate to my workflow while designing the program. In **Introduction to the Game** on page 9 the rules of the game are explained. **Engine Design** on page 11 presents the blueprints of the algorithms built into the engine and **Infrastructure** includes a description of the tools used during the implementation. The rubber meets the road in **AI Implementation** on page 23, where structural and technical details of the source code are given. To fine-tune Rock'n'Roll's camshafts, the chapter **Benchmarks** on page 31 deals with the right setting of the algorithm parameters. Finally, in **GUI Implementation** on page 43 a short description of the additional, graphical user interface (GUI) is given.

If you take fancy to the game while reading this document: A real world version of EinStein can be purchased at the 3-Hirn-Verlag `http://www.3-hirn-verlag.de`.

[1] Stefan has impressively defended this title at the EinStein tournament during the "Sternstunden Science Night" in Jena on 18th November 2005. He won the finals against Rico Walter by a great margin (see `http://www.3-hirn-verlag.de/einstein-turnier-jena.html`).

[2] Incidentally the strength of Rock'n'Roll did not rub off on me. During the "Sternstunden EinStein Tournament" Stefan defeated me (not Rock'n'Roll) by 10-0.

Remarks

In the following, a number of different computer players will be investigated. For each there are some parameters to be set. To prevent confusion a certain notation is used for each. An example: `FooBar(param1, param2)`. In this case, `FooBar` would be the heuristic (and class name) of the player. `param1` etc. are the necessary parameters for the object creation. The class can be looked up in Section 5.4 on page 26. For each class there is a subsection that will explain which parameters are needed and what they mean.

Although the original EinStein is available in a number of different colors, I will assume that the only two colors used will be red and blue. Sometimes I will refer to the players by their color (e.g. the red player or Red) and – for symmetry in name length – shorten the blue player just to Blu.

Acknowledgments

This work would never have been become what it is if there hadn't been so many people to guide me. Thank you:

Ingo Althöfer for the game itself, the opportunity to work on this project, his patient support and his suggestions.

Stefan Schwarz for his clever heuristic and brave sparring matches.

Christian Kauhaus for the never-ending computing time on the JETI-CL (Jenaer Cluster Technische Informatik [Kau05]) and his innumerable tips on cluster computing practice.

Jörg Sameith for repeatedly explaining the details of XOR Hashing.

Georg Schnatzke for pointing me to the Monte Carlo approach and its application in game theory.

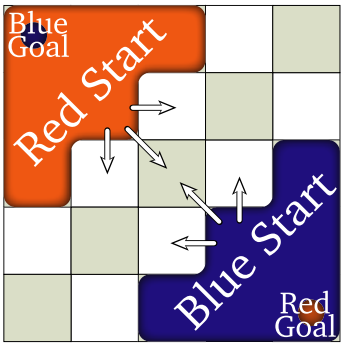
Erich Novak for his introduction into Monte Carlo Methods and pseudo random number generators.

Stefan Meyer-Kahlen for explaining his use of the hybrid hash map approach in his chess program Shredder.

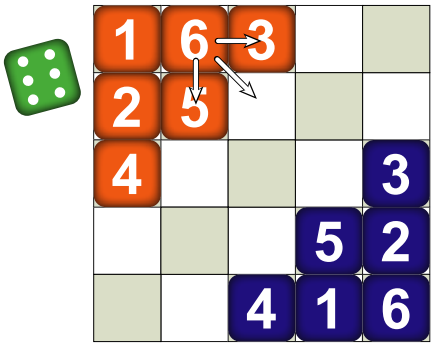
2 Introduction to the Game

EinStein's rules are straightforward: It is played on a 5×5 board. Each player has six stones numbered from one to six. At the beginning the players place their stones in opposite corners. Red starts in the upper left, Blue in the lower right corner (Fig. 2.1(a)). Their goal is to either reach the opposite corner with one stone or to capture all the enemy's stones. They move alternatingly and determine which stone to move by dicing. It is not allowed to pass a move. Red's stones may only move right, down and down-right. Blue moves to the opposite directions. It is allowed to capture both the enemy's and own pieces. Surprisingly, this ability to self-capture is important to increase mobility. If the number of a stone which is not on the board anymore is rolled, the player can choose the stone with the next higher or lower number on the board. Figure 2.1(c) on the following page shows such a situation. Blue has rolled a three and, since three and four are already missing on the board, she can now either move her five or her two. A player has won when he has either reached his goal or captured all the enemy's pieces.

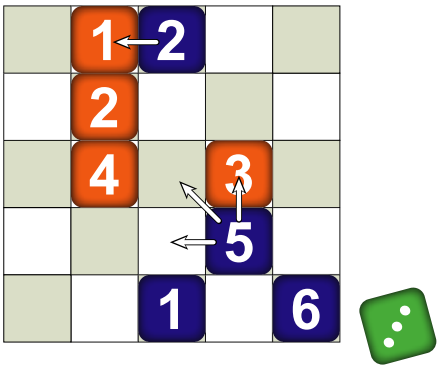
For an in-depth introduction (in German) refer to `http://www.3-hirn-verlag.de/MasterGame/regel.html`



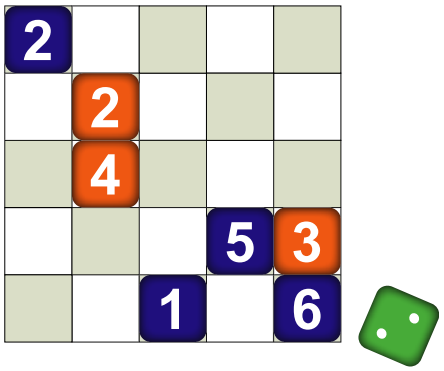
(a) Red places his stones in the six upper left fields, Blue uses the lower right fields. Their goals lie in the opposite corners. Each player has to move in one of the three directions towards his goal.



(b) A typical start setting. Red rolled a six and may now either capture his three, capture his five or move to the free field diagonally below.



(c) Here is an advanced game. The dice shows a three which Blue has already lost. Her adjacent numbers on the board are the two and five. Her legal moves are indicated by the arrows.



(d) It payed off for Blue to move the two. After Red moved his three directly before his goal, she rolled a two and won the game.

Figure 2.1: Key points during an EinStein game.

3 Engine Design

The best way to play the game would be to build a database containing every possible situation on the board and compute for each the optimal move. For games like Nine Men's Morris this has already be done. The problem with EinStein is that the number of different situations is far too large.

Besides the database, there are three major approaches for building a game engine. The probably best known is Minimax which would ideally investigate every possibility in the game tree and therefore could yield the exact solution. The Monte Carlo Method tries to get round of the immense calculation costs by using cheap random experiments for an estimation. Eventually the third, much more colorful family is that of the heuristics. Often they embody specialized rules of thumb and are generally required by the former two.

3.1 Minimax

Minimax, also known as Game Tree Search [Wik05d], is a brute force method for solving this decision problem. The Game Tree [Wik05c] is a directed graph whose vertices are made of the positions in a game and whose edges correspond to the legal moves which lead from position to position. The leaves of this tree are the terminal positions. When large numbers are assigned to those where Red wins and low values to those favoring Blue, Red will always seek high-rated positions and vice versa. Red becomes the maximizing player, Blu the minimizing player.

Since the players take turns, one can divide the tree into levels which correspond to the players on the move. Because of the players goals, each level is either a minimizing or a maximizing level which gives the method's name. A minimizing node's value is the least value of its descendants. Figure 3.1 on the next page illustrates the backward propagation of values from the leaves to the top. For the sake of simplicity I removed the need for minimizing and maximizing nodes by using only maximizing nodes. Figure 3.2 on the following page shows how their values are inverted when they are forwarded upwards. This variation is also known as Negamax Search.

The tree usually grows exponentially in depth and therefore seldom the whole tree can be searched. Rather it is rather cut off at a certain depth. Those artificial leaves have no children to gather the rating from, so they are rated by a heuristic. The level at

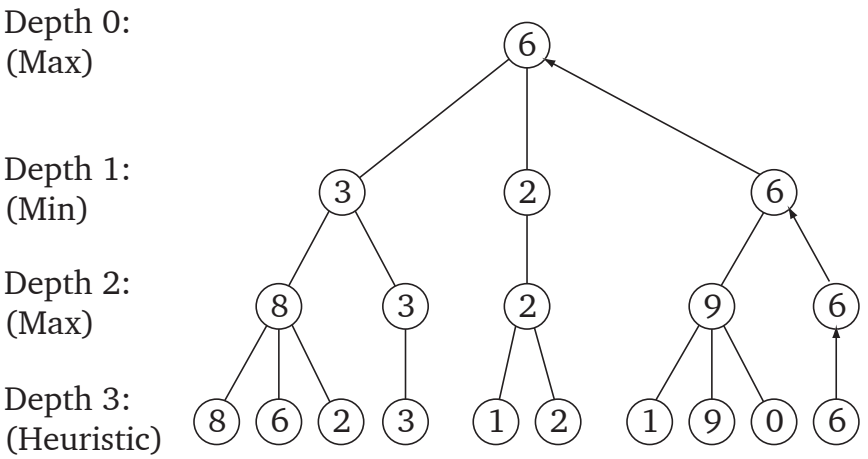


Figure 3.1: The final value of a board is determined by the leaves of the game tree. Depending on which player is on the move, either the minima or maxima are propagated upwards. The arrows mark the path on which the final result – the six – is moved upwards.

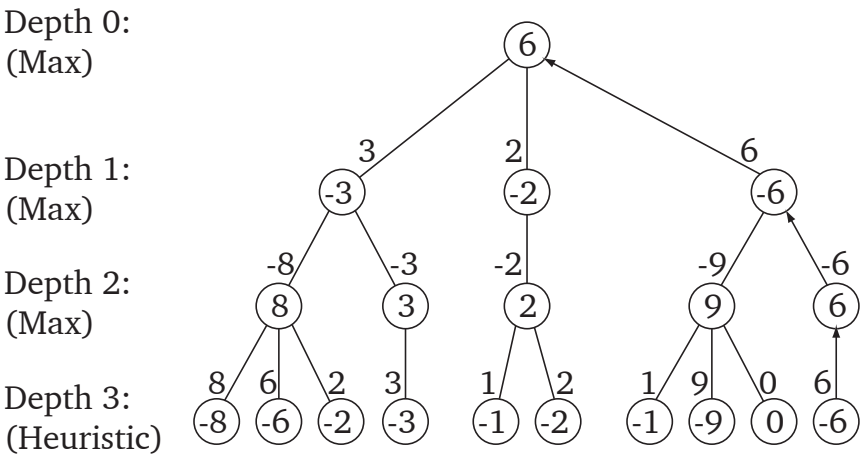


Figure 3.2: In contrast to Figure 3.1, only maximizing nodes are used here. The numbers outside the nodes are the inverted values moved upwards. Notice that, depending on the depth, the heuristic leaves have to be inverted. In Rock'n'Roll this happens automatically because the board is being flipped after each move.

which this cutoff is performed is called *search depth*[1]. For *good* models it can be proven that the higher the search depth is, the more accurate the rating of the top level nodes will become. In games like chess the exponential growth can often be mitigated by using alpha-beta pruning. This means essentially to leave out unattractive branches by proving that they wouldn't be played anyway. This allows much higher search depths and is surely one of the reasons for Deep Blue's success against Kasparov [Wik05a]. Unlike chess however, EinStein is highly randomized. This effectively prevents most (but not all) pruning since the dice might still enforce certain moves.

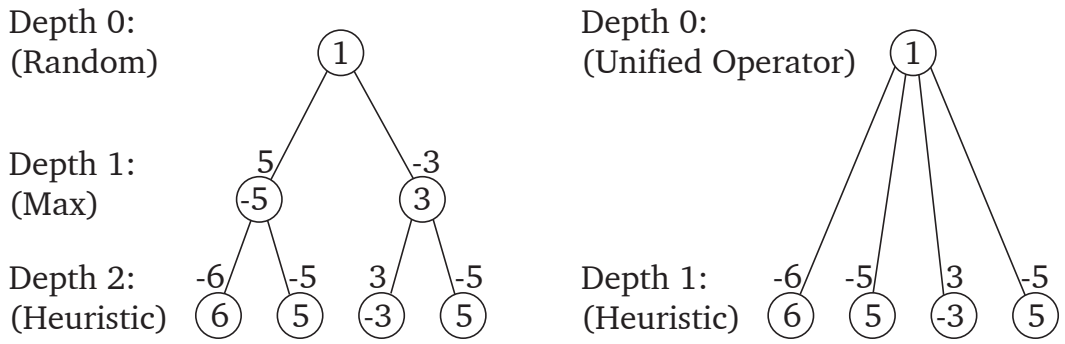
Expect Minimax

The dice does not only introduce problems with pruning but also disturbs the simple layering of the game tree. According to the Expect Minimax [Wik05b], which is the standard method to cope with randomization, one would have to introduce dice layers as can be seen in Figure 3.3(a) on the following page. These chance layers simulate the changes opposed by the random component. In practice a dice is seldom fair. Sometimes the differences of its probabilities are astonishing and a player could very well draw advantage from them. Therefore Rock'n'Roll does not assume the dice to be fair but allows any artificial setting of the dice probabilities $p_1 \dots p_6$. The value of dice nodes is defined as the dot product of their child node vectors with the dice probabilities.

Dicing Layer Reduction

Imagine a position situation like the one in figure 2.1(c) on page 10 where a player has only few pieces left. Most of the six branches of the random node would lead to the same possible moves (in this case the three and four). Obviously this leads to an immense overhead because equal positions are repeatedly assessed. I have therefore

[1] The definition of depth used throughout this document might be confusing: When a player rates his boards using a search depth of 0, this means that every direct successor of the root node is evaluated via a heuristic. Generally this is known as "depth 1" search. In my opinion this doesn't make much sense because of two reasons: Firstly this would leave depth 0 undefined (which would be unsatisfying – clearly 0 is a natural number). Secondly this confuses the process of *board evaluation* with the process of *move choice*. Both are similar, but still not the same. *move choice* means the enumeration of each legal move and the choice of the best one among them. *board evaluation* is the process undertaken to get to rate a certain move. An example: According to the classic definition, the random player `MrRandom` performs a "depth 1" search by simply enumerating every move and assigning random values to the moves. According to my definition this would be a "depth 0" search which sounds a bit more sensible in my humble opinion. If you don't trust this, refer to the implementation Chapter 5 on page 23 (source code is just a very formal way of defining algorithms) and see how they differ and where they belong. If you still don't trust me, please just increase all depths given in this report by one.



(a) Expect Minimax (b) Unified Operator

Figure 3.3: Removal of chance layers from the Expect Minimax algorithm by using my Unified Operator.

thought of a method to eliminate the dice layers and alongside with them the redundant computation. The central idea is to replace the dice node's averaging- and the player's maximizing-operator by a new, unified one (see Figure 3.3). Its main steps are:

1. For each of the current player's stones on the board every possible move is rated. Since they are all unique, no redundant calculation is done.
2. Each stone has at most three different possibilities to move. From them the best one according to the ratings is selected.
3. The dice layer is now simulated bottom up. For each number one could dice, the stones one could play are determined. From those the best one is chosen using the values from step two. This yields the ratings $r_1 \dots r_6$.
4. Now best moves for every dice result are known and the final rating R is derived by multiplying the ratings $r_1 \dots r_6$ with the dice probabilities $p_1 \dots p_6$:

$$R = \sum_{i=1}^{6} r_i p_i$$

Hash Maps

Still, there is redundancy the unified operator alone can not avoid: Different sequences of moves often lead to the same positions – which would be evaluated twice or more. The common countermeasure to this is to use a cache to store the recent evaluations. The cache is an associative map of keys (the positions on the board) and values (their ratings). It has to provide only the two operations insert and lookup. Hash maps

(or hash tables) can perform both in $O(1)$, which is optimal. Hash maps outperform binary trees ($O(\log n)$ access cost) in this application by allowing more operations for the keys. These are used to generate a function $f(key) \mapsto \mathbb{N}$ which is called a hash function. By storing the data in a simple array the $O(1)$ access costs are realized. The index is provided by the hash function.

For hash maps there are alway two problems to be solved:

- Which hash function should be chosen?
- How are collisions handled?

In the literature many generic hash functions can be found and I have tried out many of then, including $\mod$ a prime number, some homebrewn, and the golden ratio. However, none seemed satisfying. Some caused too many collisions – bad distribution – while others spread well but took so long to compute that the performance gain was absorbed. The solution was to introduce XOR-Hashing[2]. The hash value $\mathrm{hash}(p)$ of a given position p in a game (i.e. positions of all pieces on the board) is calculated by XORing values assigned to the positions the pieces occupy:

- The hash being used can store up to n elements.
- Let $\mathcal{S}$ be the set of stones on the board.
- $\mathcal{B}$ is the set of positions the board consists of.
- $\mathrm{pos} : \mathcal{S} \mapsto \mathcal{B}$ yields the position of a given stone.
- $\mathrm{hashWord} : \mathcal{S} \times \mathcal{B} \mapsto \mathbb{N}$ is a table filled at program startup with random integers.
- From this we obtain:

$$\mathrm{hash}(p) \quad = \quad \operatorname*{XOR}_{s \in \mathcal{S}} \left(\mathrm{hashWord}\left(s, \mathrm{pos}\left(s\right)\right)\right) \quad \mod n$$

Thanks to the straightforward nature of the XOR hash function, a hash value for a node in the game tree can be derived from its predecessor (note that one needs the value before applying the $\mod$ operator) by XORing out the old position of the moving stone(s) and XORing in the new one. The probability for two different position to collide in a hash of size n is – when using decent random numbers – is equal to $1/n$.

Usually the number of positions to be stored in the hash is much larger than the size of the hash map, which is limited by the physical RAM of the computer. When an index collision occurs one has to decide which entry is more valuable. The most prominent heuristics for this decision are:

[2]XOR stands for eXclusive OR. It is also named Zobrist Hashing (after its developer) and the standard technique in chess programs since the 1970s.

Recently Used prefers the youngest boards. It assumes that the occurrence of equal positions happens in a short interval. This is true for the lower regions of the search tree but requires frequent reevaluation of positions in the upper branches since their value will most likely be kicked out of the cache.

Remaining Search Depth stores the element which represents the larger subtree. Larger (i.e. higher) subtrees are favored because they are more expensive to compute. However, in lower branches the cache is rendered ineffective because it is filled up with valuable, but currently useless, entries from the upper levels.

From modern chess programs like Shredder Rock'n'Roll has adopted a hybrid strategy. Those make the best out of both heuristics by using two tables: the first acts according to Remaining Search Depth and, if it rejects a new entry, it is forwarded to the second Recently Used map. Due to their nature I will refer to the first map as the *upper* and to the second as the *lower* map.

As a special case, a stepping counter has to be stored in each hash entry. The counter is updated for each tree search. This enables the safe deallocation of old entries while still being able to profit from them. Table flushes are also unnecessary.

Looking at the hash maps it might seem that the effort laid in optimizing the unified operator above was superfluous. It was not. The operator significantly reduces the load on the hash maps and their required size and that is important since the a computer is always short of memory bandwidth and size.

Figure 3.4 on the next page summarizes the workflow of Rock'n'Roll's Minimax variation.

3.2 The Monte Carlo Method

Monte Carlo Methods (MCMs) are a class of algorithms which employ random numbers during the computation. They are distinguished from the Las Vegas algorithms. While Las Vegas algorithms always yield an deterministic result of a randomized way of computation (to avoid worst case situations during the computation), MCMs perform their calculation is a (structurally) deterministic way and yield non-deterministic results (read: estimates).

The probably best known method is Direct Simulation where a random variable X is constructed such that the expected value $E(X)$ equals the value to be calculated. By repeatedly creating instances of the variable its expectancy value is estimated. Behind this is the hope that creating instances of X can be done very fast which leads to both, a good and cheap approximation of $E(X)$.

In this special case I want to estimate how good a certain move on the board is. Its quality is equal to the win probability. This probability is hard to calculate, so a quick

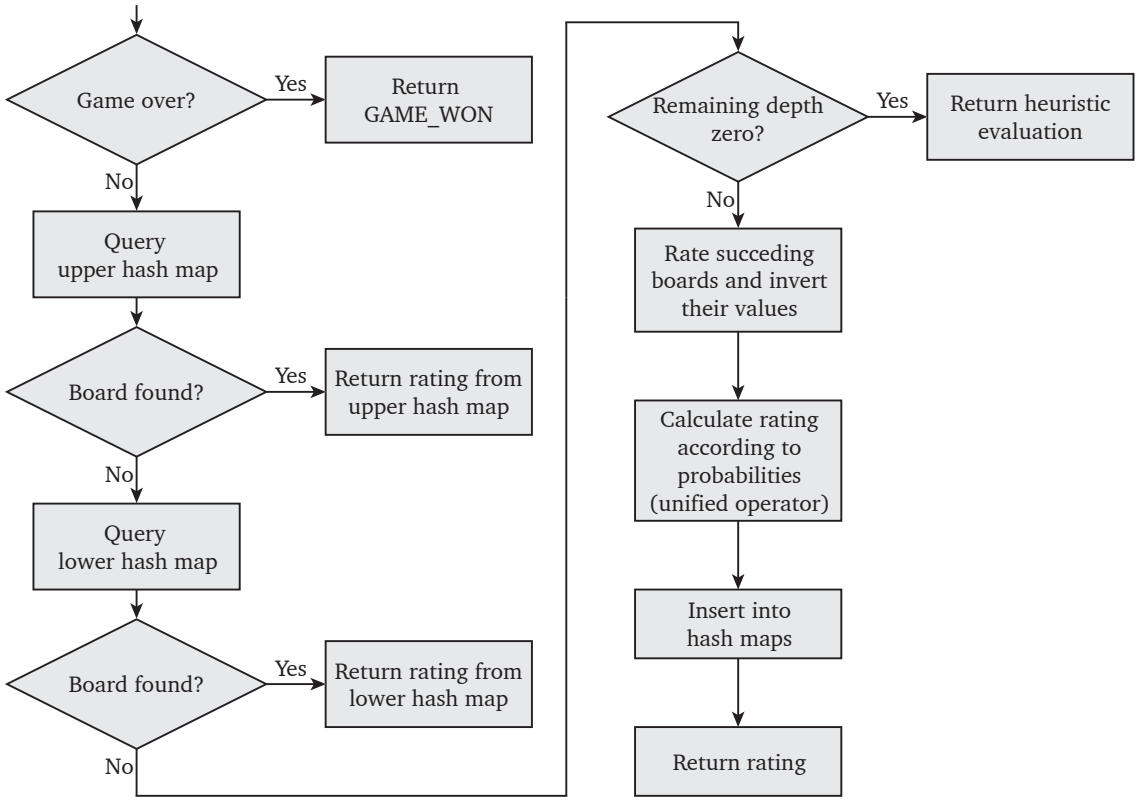


Figure 3.4: Rock'n'Roll's Minimax flowchart shows the main tasks that have to be accomplished per node. Much effort is made to prevent the execution of the right block which contains the recursive branch. GAME_WON is a constant with a large value (currently 1000). This causes the algorithm to choose sure wins over probable ones.

experiment is conducted: Two primitive players – e.g. acting according to a simple heuristic – finish the game repeatedly. The intention is that the percentage of games won by the primitive players converge to the probability in a real game.

The advantage of this approach is that much knowledge about the long term development of the game is gathered. Its drawback is that it might miss some tactical considerations due to the simple nature of the simulated players.

3.3 Heuristics

Heuristics are a very widespread field, many functions ranging from rules of thumb to neural networks pass for that. Whichever function is chosen, the idea behind is always the same: Trade accuracy for speed. That's clearly a tough tradeoff and thus good heuristics are hard to find.

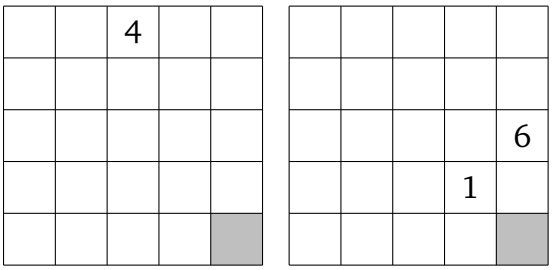
Random Moves

On the other, hand *bad* heuristics are easily found. Rating positions by random values seems ridiculous. It is. But nevertheless this heuristic is important for comparing the benchmark results and having a starting point. That can be seen in the Chapter **Benchmarks** on page 31.

Schwarz Tables

Stefan Schwarz came up with the first heuristic to evaluate positions. It abstracts from the concrete position by only considering for each stone its distance to the goal. For each player the expected number of remaining moves – while leaving piece capturing and the enemy's stones aside – is calculated. The difference of the two values (for both players) gives the heuristic rating of the position.

The naive approach to calculate the remaining moves would use a (time wasting) recursive algorithm. This can be improved. Ignoring win positions, each stone can only take on one of five distances: 1, 2, 3, 4 and ∞ (i.e. off board). Having six stones, there are at most $5^6 = 15625$ different parameter sets – perfectly suitable for a lookup table. Figure 3.5 on the next page shows some examples for the "remaining moves" calculation.



(a) $r = 4$ (b) $r = \frac{5}{6} \cdot 1 + \frac{1}{6} \cdot 2 = \frac{7}{6}$

Figure 3.5: Here are some examples for the Schwarz Tables. Only Red's stones are displayed. His goal is to reach the lower right corner. r denotes the remaining moves for him according to the heuristic.

FarmerBoy

While Stefan's heuristic is pretty accurate, it still requires a number of unpredictable branches for the distance calculation. Modern CPUs depend on accurate branch prediction to fill their long pipelines they require due to their high performance clocks. Random branches often result in pipeline flushes and spoil the performance. Thus I conceived another, more classical heuristic. It rates positions according to the average value of the occupied fields. The name FarmerBoy comes from this work on fields. Fields nearer to the goal get a higher value. More formally:

1. Let $\mathcal{S}_{red}$ and $\mathcal{S}_{blu}$ be the set of stones on the board for each of the players.
2. $\mathcal{B}$ is the set of positions the board consists of.
3. $\mathrm{pos} : \mathcal{S} \mapsto \mathcal{B}$ yields the position of a given stone.
4. $\mathrm{fieldRg} : \mathcal{B} \mapsto \mathbb{R}$ returns the (artificial) values assigned to fields.
5. A board's rating is defined as:

$$\mathrm{rating} \quad = \quad \left(\frac{1}{|\mathcal{S}_{red}|} \sum_{r \in \mathcal{S}_{red}} \mathrm{fieldRg}(\mathrm{pos}(r)) \right) - \left(\frac{1}{|\mathcal{S}_{blu}|} \sum_{r \in \mathcal{S}_{blu}} \mathrm{fieldRg}(\mathrm{pos}(r)) \right)$$

This heuristic requires just some multiply-add operations – for which, thanks to MMX, 3DNow!, and SSE, todays ALUs are well suited – and two divisions. The intent is to give up a bit of quality to gain improved performance.

4 Infrastructure

To prepare the implementation undertaken in Chapter 5 on page 23, this chapter will explain which tools have been used and why.

One of the first decisions to make is the choice of the implementation language. Sadly there is no supreme language for every use case – and there will probably never be one – so I had to make my choice depending on the requirements. Here is a list of the most important ones:

- Execution performance to increase the prediction quality (see Section **Minimax** on page 11).
- Rich expressiveness of the language to reduce complexity.
- Cross-platform support to run on an end user's Windows desktop as well as on my Linux development machine or the testbed on the JETI-CL cluster.
- Memory access performance to make efficient use of the hash map advantages.

As the base language I have chosen C++. It has roughly the same efficience as C and is both, slightly faster and more expressive, than Java. Higher level languages such as Lisp or Ruby have been discarded on account of being too slow and Assembler was rejected for being a hell to code in.

The graphical user interface has been written using Qt from Trolltech; further details are given in Chapter **GUI Implementation** on page 43. Besides, Qt provides all the classes for the engine which are platform dependent and not part of the standard library.

To automate the testing during the development cycle, unit tests were written using CxxTest [Vol05]. This framework allows the coder to write very slick and precise test cases and generates the auxiliary code using a Perl script. A comparison of many similar frameworks is given in [Llo05]. Automatic tests are only useful when the rest of the build cycle is also automated. For the engine Make has been used because of its well adaptation to C++ and Qt's QMake.

This report is written using LaTeX from the TeTeX distribution [Ess05]. The data taken from the various measurements is parsed by some Ruby [Mat05] scripts and plotted for the diagrams using Gnuplot [TW05]. Every figure and table derived from measurements is created completely automatically from the data files by the build tool. This

requires a considerable amount of scripting and therefore Make was discarded for the report and replaced by Rake [Wei05]

A unified storage for the whole source code was provided by Subversion [Bla05]. It is a version control system that allowed to keep the code consistent, no matter if the development took place on the cluster, on my desktop or on a mobile Windows notebook.

5 AI Implementation

So far the algorithms were presented from a theoretical point of view. In this chapter the transformation from mathematical concepts into live code is explained. The main objective is to create both, comprehensible and efficient source code.

I have approached this by using an evolutionary prototype. At first it implemented just the most basic use cases (i.e. algorithms) and got equipped with the more advanced ones during the term. This agile development allowed clean code and easy refactoring to integrate new features (of which I came to know most just during the term in which the implementation took place).

A good heuristic for breaking down the requirements is to divide them according to real world metaphors. A game consists of the board, the stones, and two players. In more sophisticated games or when dealing with rude players also a referee handling details as setting up the board, watching who's on the move etc. becomes necessary.

A bad thing to do would be to create a class for every tiny piece of information. There could be classes like `RedStoneNo2`, `FieldB3` or `CapturingRule32`. This *sounds* like modular, single minded code. But it would also be bloated and – because of the sheer mass of classes – hard to comprehend. The programmer would be tacked in his own fen of classes. Another bad thing would be to glue everything together into one monolithic class. The best solution must be somewhere in between.

Figure 5.1 on the next page shows the class diagram. First I will have a glance at the most important classes before taking the in-depth tour in the following sections. The `Board` class encapsulates the playing field, the stones, and the rules according to which the stones are moved across the fields. A `Player` is an object which can decide upon a move to take. Subclasses are `StdInPlayer` – a human interface – and `AIPlayer`. `AIPlayer` (Artificial Intelligence Player) is the parent class of all computer players and implements the Expect Minimax from Section 3.1 on page 11. At a first glance it might be striking that all the other approaches for player AIs are subclasses of `AIPlayer`, but at a second glance it becomes obvious that they all have one thing in common: They map boards to numbers describing their value. Therefore each of them can be used with the Minimaxing algorithm as a kind of heuristic. The result is a building set of players where the behavior and strength of a player depends on the building blocks employed during its construction.

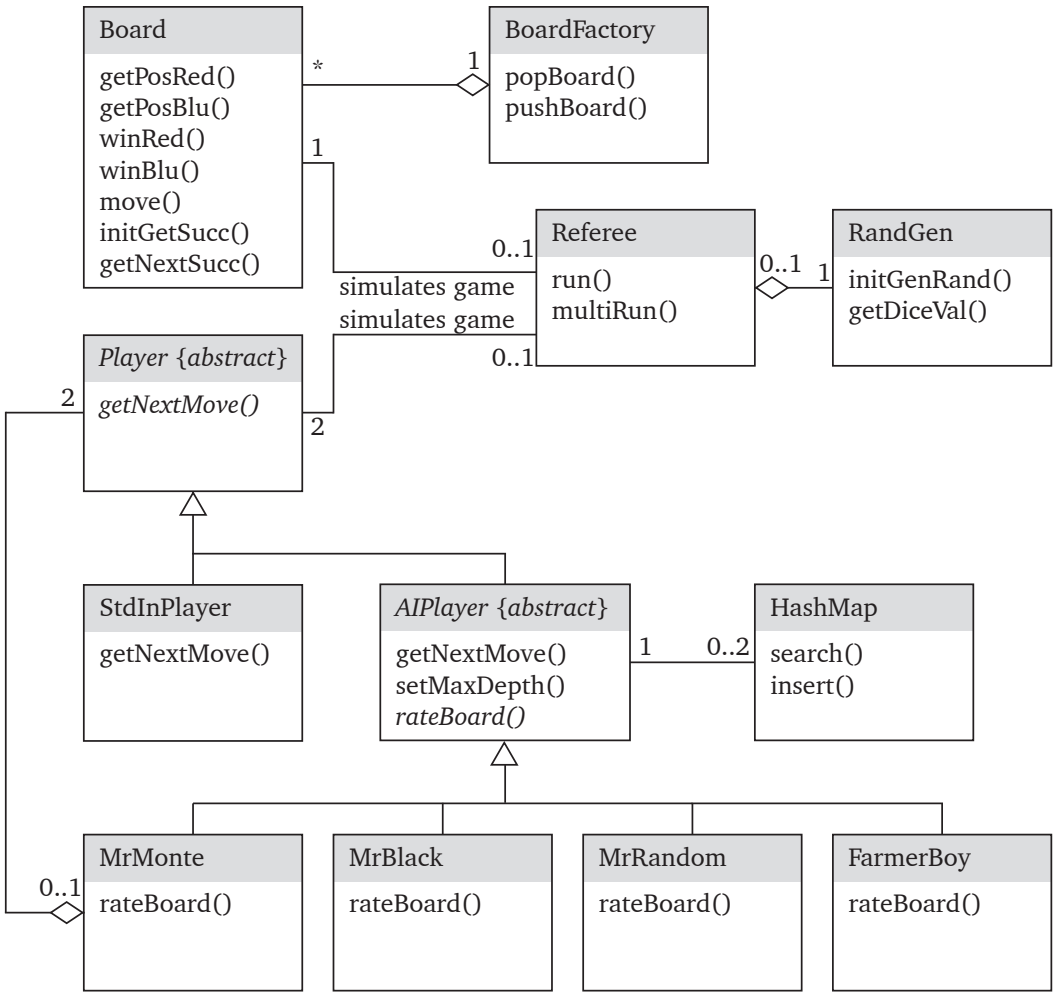


Figure 5.1: Rock'n'Roll's main classes and methods. Abstract items are set in *italics*. Utility classes and helper methods have been left out.

5.1 Board

`Board` entirely encapsulates the field, the stones, and their movement. In its move generator the rules of the game are embedded. It is perhaps the central class since most others operate on it.

There are two possibilities to store the stones' positions on the field: An array of stones where each element contains a field or an array of fields where each element denotes which stone – if any – occupies it. The former has the drawback that it is expensive to detect piece capturing during a move (one would have to iterate through all of the stones). The later suffers from the same problem when deciding which pieces are allowed to move: Every field has to be searched for the stone it contains. During tests both have proven to be way too slow, so I have implemented a dualistic approach by storing both. This has the drawback that for every move both data structures are updated, but the saved iterations still make this a bargain. The process of enumerating each directly succeeding board is set up by calling `initGetSucc()` and propelled by `getNextSucc()`. The main steps are:

1. `initGetSucc()` looks up the table entry containing the sequence of legal moves.

2. For each successor `getNextSucc()` calls `move()`:

    a) Allocate a new board from the `BoardFactory`.

    b) Clone the board's data to the new one. Mind that it is always assumed that player Red is on the move so one has to flip the data (exchange players) while copying it.

    c) Move the stone according to the lookup table.

    d) Detect if the goal corner has been reached.

    e) Update `movesBlu` if one of the edges was encountered.

    f) Remove any stricken pieces and detect if the opponents last piece has been removed.

    g) Correct the board's hash key.

    h) Return the new board.

The methods `winRed()` and `winBlue()` detect terminal positions while `getPosRed()` and `getPosBlu()` provide access to the positions of the stones on the board.

5.2 Player

The second abstraction from the real game is the player. It incorporates a way of playing the game. `Player` itself is an abstract class. Subclasses have to implement the abstract method `getNextMove()`.

5.3 StdInPlayer

`StdInPlayer` was first introduced for testing purposes. It prints out the current board to the standard output and reads a move to return from the standard input (stdin).

5.4 AIPlayer

`AIPlayer` implements the Minimax algorithm in its helper method `_rateBoard()`. The class itself remains abstract. It is up to the subclasses to implement the rating heuristic `rateBoard()` for the leaves. The behavior of the game tree search can be modified by setting parameters in the constructor.

- `depth` sets the maximum search depth. If one wants to perform iterative deepening, he can override this by calling `setMaxDepth()` with an increasing parameter first and `getNextMove()` second. This results in constantly improving results of `getNextMove()`.
- The number of entries stored by the upper and lower hash maps is controlled by `hashSizeHi` and `hashSizeLo`.
- Finally `_probs` sets the assumed probabilities of the dice.

```
MrRandom(depth)
```

`MrRandom` implements the Random Moves heuristic from Section 3.3 on page 18 by assigning random values to the cutoff leaves. It depends on the random number generator from 5.7 on page 29.

0	1	1	1	1
1	2	2	2	2.5
1	2	4	4	5
1	2	4	8	10
1	2.5	5	10	16

(a) $r = 4$

Figure 5.2: FarmerBoy field ratings for player Red.

`MrBlack(depth)`

From its name it is easy to guess that `MrBlack` makes use of the Schwarz Tables (see 3.3 on page 18). Its implementation of `rateBoard()` calculates the distance of each stone from its goal. The resulting six dimensional vectors for each player are used to lookup the expected remaining distance in the `MrBlack::remMoves` table. The difference of both values is scaled up to better match averaging effects when going up the game tree and being mixed with win or lose positions. `MrBlack::remMoves` is a static table, so it is shared by every `MrBlack` instance. This has the benefit that just little space in the CPU's caches is consumed and it has to be initialized only once when the engine is being loaded.

`FarmerBoy(depth)`

Rivaling `MrBlack`, this class depends on my much simpler heuristic from page 19. Figure 5.2 shows the ratings being applied to the fields.

`MrMonte(depth, iterations, red, blu)`

`MrMonte` provides an interface to the Monte Carlo Method. What might seem strange is that it is a subclass of `AIPlayer` which implements game tree search. The reason for this is simple: The result of the MCM is a float approximating the win probability. This can very well be fed into the game tree search as a kind of super heuristic. Beside the parameters for its superclass, `MrMonte`'s constructor takes the following parameters:

iterations sets the number of simulation runs. The higher this value is, the lesser random artifacts will become and the longer it will take.

red is a pointer to the object which will be treated as player Red and

blu is the corresponding object for the blue player. The smarter those players are, the closer the simulation will match reality. Note that when one knows his opponent, he can exploit this using an asymmetric setup where the opponent's simulation player is matched to his real playing style.

If this parameter is left out, it is assumed to be equal to `red`.

In chapter 6 on page 31 I will investigate if it is more desirable to increase the `iterations` or the `depth` or even use smarter simulation players.

5.5 Referee

So far the `Board` class, which encapsulates the move rules, and the `Player` classes containing strategies were explained. `Referee` arranges a smooth interaction between the `Player` objects and provides basic batch testing capabilities. While `run()` conducts a single game between the given `Players` starting with the specified `Board`, `multiRun()` runs a given number of games. It comes in two flavors: Either each game starts with the same board (useful for `MrMonte`) or with a random starting board (good for comparing players). `multiRun()` returns the number of games Red has won.

5.6 BoardFactory

One of the most error prone tasks a coder has to handle is memory management. Modern languages such as Java, Python etc. come with a garbage collector (GC) build in and even for C++ Hans Boehm has released a GC. This relieves the programmer from a great burden but also has the drawback that when a great number of object with a short life time are created, the load generated by the GC becomes inacceptable. Even worse, the number of cache misses in the CPU increases because at first free memory is allocated – it is hardly possible for the GC to detect unused objects instantly. Conversely, data structures like the call stack achieve very good hit rates while being easy to use. Not all of the `Boards` used in the engine can live on the stack; some are used as return values and others are stored longer than the creating procedure is executed.

The `BoardFactory` takes care of the memory management for the boards. It mimics the built-in stack by providing two methods:

- `popBoard()` returns a pointer to a new board and via

- `pushBoard()` an old one can be given back.

Old boards are stored on a stack and are not freed. `popBoard()` will return the top element of this stack and only allocate a new board if the stack is empty. The result is a high cache hit rate in the CPU's 1st and 2nd level caches and – since the amount of necessary boards usually reaches a saturation value quickly – a low load on the standard library's `malloc()` facility.

5.7 RandGen

The quality and performance of every MCM depends upon the employed random number generator. Defining the exact qualities a decent generator should have would fill several pages, therefore only the most important features are given:

- long period and
- high-dimensional equidistribution to avoid simulation artifacts, and
- efficient computability for performance.

Internally `RandGen` relies on the Mersenne Twister by Takuji Nishimura and Makoto Matsumoto. This pseudo-random number generator has a period of $2^{19937} - 1$ and outperforms e.g. Linux' standard generator by the factor two (on my desktop ≈ 72899400 numbers$/s$ vs. ≈ 36283000 numbers$/s$).

Externally a `RandGen` object behaves like a virtual dice with configurable "fairness": the probabilities of the numbers generated in `getDiceVal()` can be tweaked in the constructor. This is realized by inverting the distribution function and applying the equidistribution. The generator can be re-seeded by `initGenRand()`.

5.8 HashMap

In Hash Maps on page 14 the hashing algorithm itself was explained. Its implementation is straightforward. A large array of structs is allocated. Each struct has the following fields:

id is a unique board identifier derived from the board positions,

rating is the stored rating,

stepping specifies the entries "generation", and

remDepth gives the remaining search depth used when evaluating the board.

5.9 ThreadWrapper

Rock'n'Roll is designed as a cross-platform program. Most parts of the engine are not platform dependent. However, for a clean iterative deepening implementation one would have to use threads. The Windows thread model differs from Linux' model. To hide these differences, `ThreadWrapper` provides a unified interface behind which different bindings can be used according to the current platform.

The external method `concurrentExecute()` allows to start a given function in the background. The internal implementation can be automatically switched on demand during the build process.

6 Benchmarks

Every good engine needs a drop of oil on its bearings and Rock'n'Roll is no exception. In this chapter a brief overview is given, how to tweak the parameters and make Rock'n'Roll play really strong. Parameter sweeps consume a lot of computing power – much more than my PC could offer in reasonable time – therefore most of the tests were run on the JETI-CL Cluster.

6.1 Optimal Setup of Stones on the EinStein Board

So far a lot has been dealt with deciding which move to take. But a pressing topic has been left out: How to set up the board? Does it matter how the stones are placed or can some strength be drawn from special setups? To answer these questions I ran a little tournament, testing each of the initial setups against each other.

Ignoring symmetries, there are obviously $6! = 720$ starting positions for each player and thus $720^2 = 518400$ pairings when considering both. With each of these pairings 20 games were simulated, leading to a total number of 10368000 simulated games. As simulation players two copies of `MrMonte(0, 200, MrBlack(0))` have been used. They provide acceptable playing strength while keeping time consumption within reasonable limits.

A quasi-lexicographic ordering of the 720 setups can be obtained by enumerating every permutation of the six starting fields.

The number of games in which a specific setup won divided by the total number of games it took part in yields a good estimate for the real win probability. A plot of every setup against its win probability can be seen in Figure 6.1 on the next page. The advantage of beginning the game is eliminated in the plot because each setup belonged to the starting player the same number of games.

Two observations can be made:

- The start setup greatly affects the chances (ranging from 42% up to 56%).
- Despite the drop between 500 and 600, the quasi-lexicographic ordering doesn't tell much about the quality of a given setup.

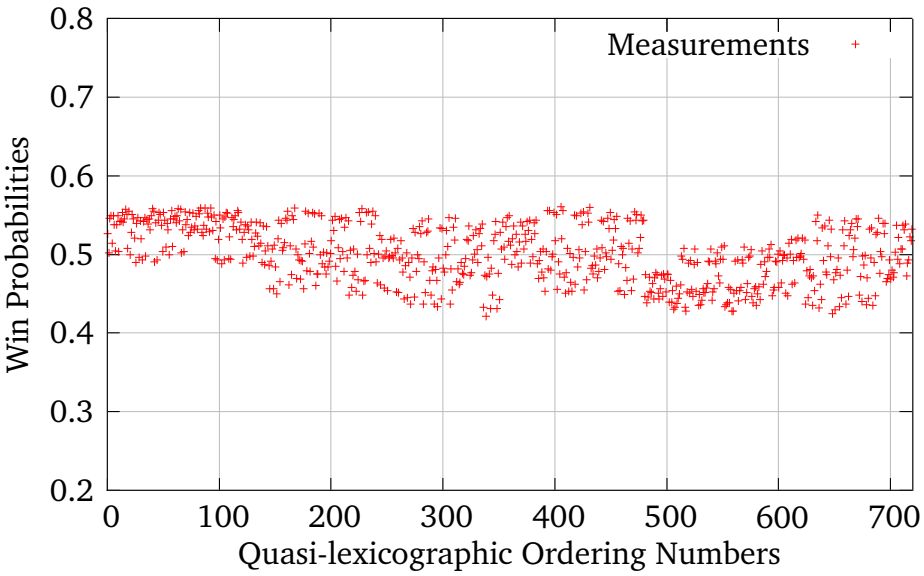


Figure 6.1: Each plus stands for the measured probability to win with the setup belonging to the given ordering number against another random board. The diffuse spreading of the plusses shows that the choice of the right setup is vital for the further success (see 6.1 on the previous page).

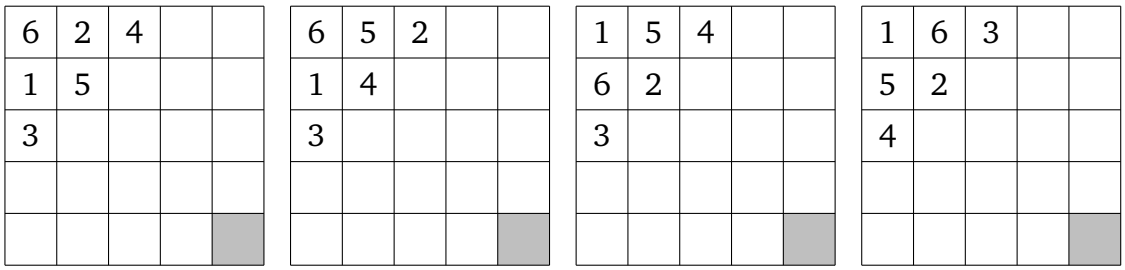
To get to know the characteristics of *good* and *bad* setups, Figures 6.2 on the facing page and 6.3 on page 34 show the sixteen best and worst positions according to the previous plot 6.1. The observable range is surprising: With a clever setup, the odds against an equally strong playing opponent who's choosing a random board can be risen by 6% while a bad setup can cost even a bit more.

Now some patterns can be observed; the most obvious is that it seems to be a good idea to keep the 1 and the 6 in the background (preferably to capture the 2 or 5). Placing them in the 1st line (especially side by side) doesn't look like a survival trait.

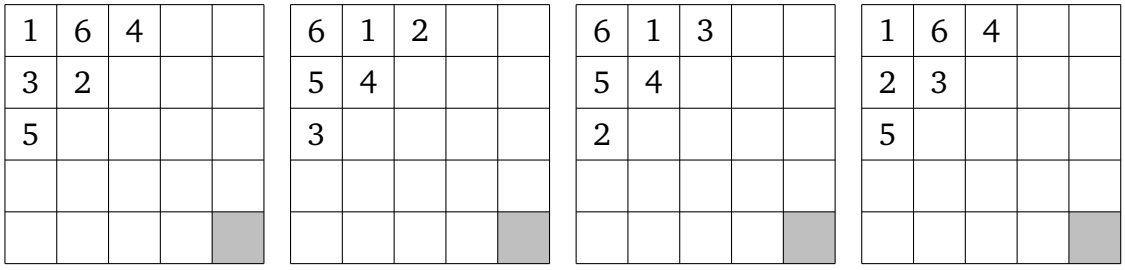
6.2 Tournament

> *Premature optimization is the root of all evil...*
> **Donald Knuth, Computer Programming as an Art**

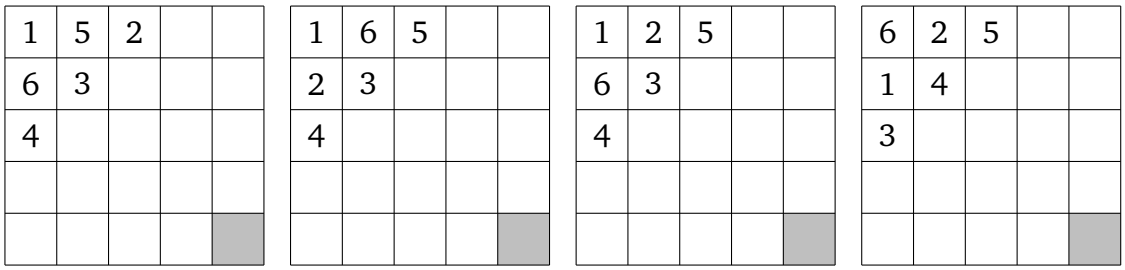
It is hard to guess the optimal parameters to create an optimal computer player. To bring light to this speculative topic, I have conducted a large tournament. The participants were 24 different parameter sets. I have tried to represent the most important variations of search depth and heuristics. The tables 6.1 on page 38 and 6.2 on page 39



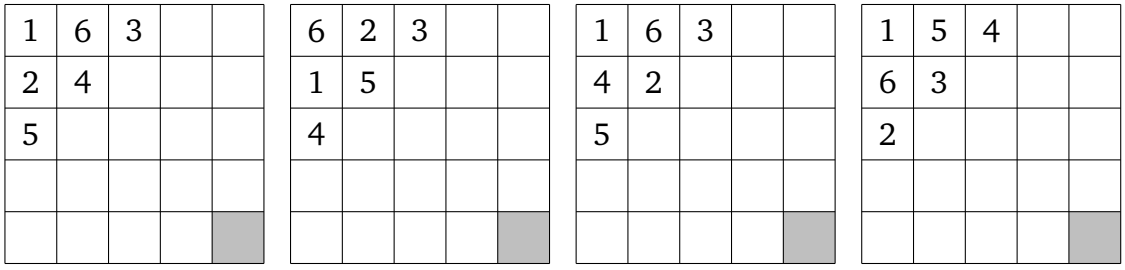
(a) $p(405) = 56.056\%$ (b) $p(431) = 56.021\%$ (c) $p(92) = 55.913\%$ (d) $p(83) = 55.903\%$



(e) $p(87) = 55.865\%$ (f) $p(167) = 55.861\%$ (g) $p(227) = 55.837\%$ (h) $p(63) = 55.795\%$

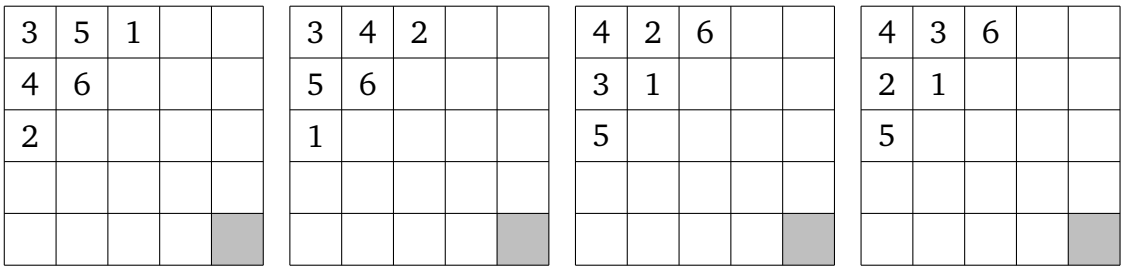


(i) $p(40) = 55.785\%$ (j) $p(65) = 55.750\%$ (k) $p(16) = 55.736\%$ (l) $p(407) = 55.611\%$

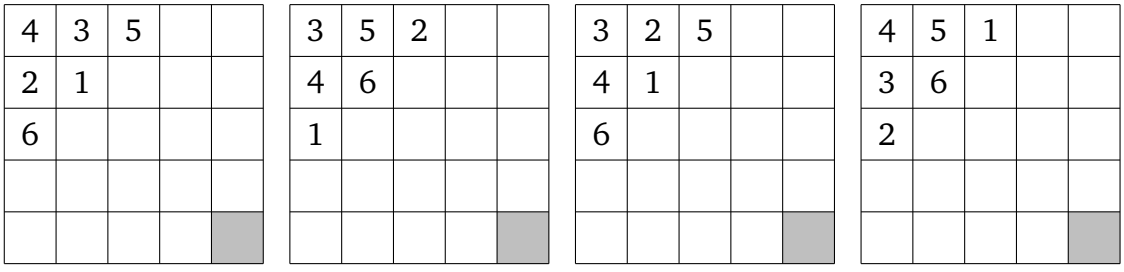


(m) $p(57) = 55.604\%$ (n) $p(395) = 55.583\%$ (o) $p(81) = 55.562\%$ (p) $p(116) = 55.555\%$

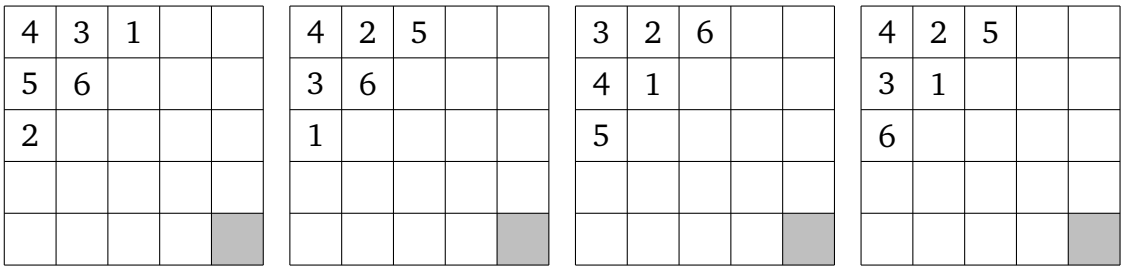
Figure 6.2: Best starting positions; the number in brackets is the position's quasi-lexicographic index, the percentage denotes the probability to win against another randomly chosen position (see Section 6.1 on page 31).



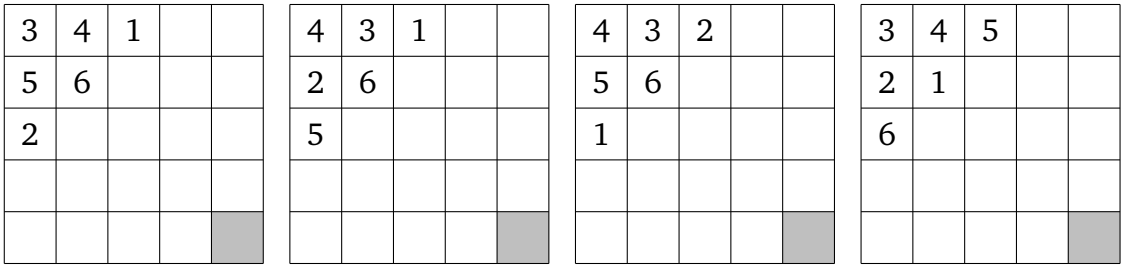
(a) $p(338) = 42.175\%$ (b) $p(648) = 42.446\%$ (c) $p(517) = 42.776\%$ (d) $p(559) = 42.783\%$



(e) $p(558) = 42.797\%$ (f) $p(650) = 42.977\%$ (g) $p(506) = 43.057\%$ (h) $p(348) = 43.082\%$



(i) $p(342) = 43.151\%$ (j) $p(636) = 43.227\%$ (k) $p(507) = 43.273\%$ (l) $p(516) = 43.314\%$



(m) $p(336) = 43.321\%$ (n) $p(295) = 43.332\%$ (o) $p(654) = 43.373\%$ (p) $p(552) = 43.391\%$

Figure 6.3: Worst starting positions; the number in brackets is the position's quasi-lexicographic index, the percentage denotes the probability to win against another randomly chosen position (see Section 6.1 on page 31).

show the whole mass of results. However, they are merely meant for reference. For a quick overview, please have a look at Table 6.3 on page 40 which summarizes them.

Measurement Details

As seen in **Optimal Setup of Stones** on page 31 it is important to choose a good starting position carefully as the odds may be moved by 6% up or down. On the one hand choosing a purely random starting positions seems to be bad because it might upset the tournaments results. On the other hand choosing just one (e.g. the best) or a symmetric setup might bear unknown inbreeding problems. And of course it would be unrealistic to assume that players would play with boards which are known to be worse than others.Therefore each pair in the tournament plays with 16^2 different setups. They are formed by the combinations of the 16 best starting positions from Figure 6.2 on page 33.

To even out the advantage of being the first player each game was played twice: one time Red began, another time Blue.

Finally, to increase the quality of the samples even more, the whole tournament has been repeated 20 times. From this we derive the distribution properties of the table samples p_{ij}:

$$
\begin{aligned}
n &= \underbrace{2}_{\text{start flip}} \cdot \underbrace{16 \cdot 16}_{\text{setups}} \cdot \underbrace{20}_{\text{repeats}} = 10240 \\
p_{ijk} &= \text{Result of } k\text{th game between players } i \text{ and } j \\
p_{ij} &= \frac{1}{n} \sum_{k=1}^{n} p_{ijk} \\
V(p_{ijk}) &= p_{ijk} - p_{ijk}^2 \\
V(p_{ij}) &= \frac{n}{n^2} V(p_{ijk}) = \frac{p_{ijk} - p_{ijk}^2}{n} \\
\sigma_{ij} &= \sqrt{V(p_{ij})} = \frac{\sqrt{p_{ijk} - p_{ijk}^2}}{\sqrt{n}}
\end{aligned}
$$

According to the 95% confidence interval rule the exact values should lie within a $4\sigma_{ij}$ broad interval around the samples. For instance the players 6 and 18 are roughly equally strong. We yield $p_{618} \approx 49.93\% = 0.4993$ and $\sigma_{618} = 0.00494$ and therefore the real value lies with a likelihood of 95% in the interval $[48.94, 50.92]$.

Parameter Choices and Interpretations

The choice of the right parameter sets for the players was a hard tradeoff. I tried to cover the most interesting combinations, generate as strong players as possible, and still keep all of them simple enough to raise the sample quality by increasing the repeats.

The exact analysis is left to the reader, as this is not the scope of this report – I just had to find out which is the best one. However, I will outline, why which parameter sets have been chosen and what I have concluded from them.

In the afore mentioned tables the use of the `MrRandom` player becomes obvious: The better another player competes against `MrRandom`, the better it will probably play against others.

With the `MrMonte` players 2 to 5 I have tried to figure out how many repeats would be necessary to get reliable results. As one can see, more than 100 repeats result in only marginal improvements.

The `MrMonte` players 6 and 7 differ in the place where they perform the tree search. I was interested if one should rather rely on smarter, more realistic simulations, like player 7 or try to think further ahead like player 6. They are nearly up to par, but not quite: There is a slight advantage for player 6.

The players 8 to 12 investigate, if and if so, how much, an increased search depth improves the quality of the `MrBlack` heuristic. The players 13 to 17 do the same for the `FarmerBoy` heuristic. It is a bit disappointing for me, although not unexpected, to see that `MrBlack` proves to be the stronger player. Really disappointing was to see that there was no measurable speedup by using the simpler `FarmerBoy` heuristic. At least the results show that deep searches significantly improve the player's strength, although there is a saturation at higher depths.

No. 18 and 19 went in as mavericks, but their results surprised me. They trade accuracy of the MCM for increased search depth. Before I had seen the results, I expected them to perform rather bad because their MCMs rely on only few repeats. Obviously those artifacts are mitigated by the Expect Minimax algorithms. Even better: The increased depth allows the players to discover at least simple tactical combinations. The result is that player 19 is the strongest player of the whole tournament, although he consumes about as little CPU power as No. 6 or 7.

Player 20 was another (failed) attempt to improve `MrMonte` by using not a simple heuristic for the simulated players but using another `MrMonte` himself. Although this spoils a lot of performance, the results are not really convincing. This is most likely caused by the low repetition factors one can use in such a setup.

The last pack of players (21 to 24) doesn't differ so much from the 2nd pack (2 to 5), except that they use the `FarmerBoy` heuristic for their simulations. From the results

above one should expect that they should be slightly worse than those relying on Schwarz Tables. Funnily, the opposite is the case: In essence they tend to perform a little bit better.

Table 6.1: Tournament Part 1; the entries denote the rounded win percentages of the row player versus the column player. The main diagonal is indicated by dashes.

	Player vs. Player	1	2	3	4	5	6	7	8	9	10	11	12
1	`MrRandom`	–	11.09	9.73	9.52	8.60	7.62	8.57	10.02	9.33	7.57	7.22	7.07
2	`MrMonte(0, 50, MrBlack(0))`	88.91	–	46.29	46.27	45.07	42.05	41.73	55.93	51.22	49.31	47.56	44.78
3	`MrMonte(0, 100, MrBlack(0))`	90.27	53.71	–	48.12	47.51	45.14	45.35	58.66	53.53	52.84	51.28	47.63
4	`MrMonte(0, 150, MrBlack(0))`	90.47	53.73	51.88	–	49.36	45.84	45.82	60.65	55.85	53.90	51.17	49.37
5	`MrMonte(0, 200, MrBlack(0))`	91.40	54.93	52.49	50.64	–	47.75	47.36	61.89	56.37	54.51	52.71	49.24
6	`MrMonte(1, 200, MrBlack(0))`	92.38	57.95	54.86	54.16	52.25	–	49.93	60.21	58.32	56.85	55.62	52.90
7	`MrMonte(0, 200, MrBlack(1))`	91.43	58.27	54.65	54.18	52.64	50.07	–	60.52	59.74	58.35	54.62	52.53
8	`MrBlack(0)`	89.98	44.07	41.34	39.35	38.11	39.79	39.47	–	47.10	43.60	41.23	41.24
9	`MrBlack(1)`	90.67	48.78	46.47	44.15	43.63	41.68	40.26	52.90	–	47.14	45.58	43.79
10	`MrBlack(2)`	92.43	50.69	47.16	46.10	45.49	43.15	41.65	56.40	52.86	–	45.51	42.92
11	`MrBlack(3)`	92.78	52.44	48.72	48.83	47.29	44.38	45.38	58.77	54.42	54.49	–	46.92
12	`MrBlack(4)`	92.93	55.22	52.37	50.63	50.76	47.10	47.47	58.76	56.21	57.08	53.08	–
13	`FarmerBoy(0)`	84.23	38.42	36.04	35.53	33.24	31.71	32.61	43.09	39.71	35.81	35.36	33.11
14	`FarmerBoy(1)`	84.49	39.47	35.54	34.82	33.35	30.81	31.27	40.98	38.69	38.13	36.85	34.86
15	`FarmerBoy(2)`	88.59	46.97	44.61	41.55	41.67	38.38	38.90	49.20	46.51	46.94	43.68	41.63
16	`FarmerBoy(3)`	89.28	46.57	43.09	40.99	40.32	39.09	40.35	49.58	46.94	47.49	43.59	41.40
17	`FarmerBoy(4)`	91.56	51.79	47.84	46.58	45.58	43.50	44.62	54.17	51.88	52.14	49.51	47.09
18	`MrMonte(2, 20, FarmerBoy(0))`	93.14	58.76	56.23	54.58	53.80	49.96	51.47	59.61	58.16	57.67	56.20	52.65
19	`MrMonte(2, 40, FarmerBoy(0))`	93.07	59.63	56.07	55.60	54.96	50.02	51.33	60.87	58.79	58.54	57.11	54.47
20	`MrMonte(0, 20, MrMonte(0))`[1]	85.52	45.67	42.01	39.33	40.11	36.45	37.78	45.24	44.02	44.45	42.22	39.96
21	`MrMonte(0, 50, FarmerBoy(0))`	89.31	50.52	46.20	46.68	44.73	41.55	43.28	52.98	49.89	49.53	48.48	45.47
22	`MrMonte(0, 100, FarmerBoy(0))`	90.63	53.41	51.08	48.86	48.56	44.77	45.82	56.66	53.72	52.56	51.88	48.31
23	`MrMonte(0, 150, FarmerBoy(0))`	90.89	55.54	52.18	51.17	49.95	46.53	47.61	57.96	55.49	54.04	52.95	50.56
24	`MrMonte(0, 200, FarmerBoy(0))`	91.64	56.27	52.78	51.34	50.36	46.84	48.58	59.81	56.10	53.85	53.42	50.57

[1] Relies on two `MrMonte(0, 10, FarmerBoy(0))`

Table 6.2: Tournament Part 2; the entries denote the rounded win percentages of the row player versus the column player. The main diagonal is indicated by dashes.

	Player vs. Player	13	14	15	16	17	18	19	20	21	22	23	24
1	MrRandom	15.77	15.51	11.41	10.72	8.44	6.86	6.93	14.48	10.69	9.38	9.11	8.36
2	MrMonte(0, 50, MrBlack(0))	61.58	60.52	53.03	53.43	48.21	41.24	40.36	54.33	49.48	46.59	44.46	43.73
3	MrMonte(0, 100, MrBlack(0))	63.95	64.45	55.39	56.91	52.16	43.77	43.93	57.99	53.80	48.92	47.82	47.22
4	MrMonte(0, 150, MrBlack(0))	64.47	65.18	58.45	59.01	53.42	45.42	44.40	60.67	53.32	51.14	48.83	48.66
5	MrMonte(0, 200, MrBlack(0))	66.76	66.65	58.33	59.68	54.42	46.20	45.04	59.89	55.27	51.44	50.05	49.64
6	MrMonte(1, 200, MrBlack(0))	68.29	69.19	61.62	60.91	56.50	50.04	49.98	63.55	58.45	55.23	53.47	53.16
7	MrMonte(0, 200, MrBlack(1))	67.38	68.73	61.10	59.65	55.38	48.53	48.67	62.22	56.72	54.18	52.39	51.42
8	MrBlack(0)	56.91	59.02	50.80	50.42	45.83	40.39	39.13	54.76	47.02	43.34	42.04	40.19
9	MrBlack(1)	60.29	61.31	53.49	53.06	48.12	41.84	41.21	55.98	50.11	46.28	44.51	43.90
10	MrBlack(2)	64.19	61.88	53.06	52.51	47.86	42.33	41.46	55.55	50.47	47.44	45.96	46.15
11	MrBlack(3)	64.63	63.15	56.32	56.41	50.49	43.80	42.89	57.78	51.52	48.12	47.05	46.58
12	MrBlack(4)	66.88	65.14	58.37	58.60	52.91	47.35	45.53	60.04	54.53	51.69	49.44	49.43
13	FarmerBoy(0)	–	46.97	41.40	40.27	35.83	31.75	30.62	45.64	36.82	33.13	32.96	32.15
14	FarmerBoy(1)	53.03	–	42.47	43.12	36.93	30.43	29.19	45.82	38.75	34.17	32.47	31.10
15	FarmerBoy(2)	58.60	57.53	–	48.87	44.16	38.97	37.06	53.05	46.14	43.31	41.32	40.82
16	FarmerBoy(3)	59.73	56.88	51.13	–	44.85	37.67	36.68	53.28	46.77	42.31	39.97	40.19
17	FarmerBoy(4)	64.17	63.07	55.84	55.15	–	43.05	42.29	57.41	52.34	47.73	45.33	46.28
18	MrMonte(2, 20, FarmerBoy(0))	68.25	69.56	61.02	62.33	56.95	–	49.93	63.77	57.75	54.16	54.39	53.27
19	MrMonte(2, 40, FarmerBoy(0))	69.38	70.81	62.94	63.32	57.71	50.07	–	63.13	59.44	55.58	53.86	53.06
20	MrMonte(0, 20, MrMonte(0))[1]	54.36	54.18	46.95	46.72	42.59	36.22	36.86	–	45.27	42.02	40.46	40.03
21	MrMonte(0, 50, FarmerBoy(0))	63.18	61.25	53.86	53.23	47.66	42.25	40.56	54.73	–	46.73	45.78	44.04
22	MrMonte(0, 100, FarmerBoy(0))	66.88	65.83	56.69	57.69	52.27	45.84	44.42	57.98	53.27	–	49.75	47.14
23	MrMonte(0, 150, FarmerBoy(0))	67.04	67.53	58.68	60.03	54.67	45.61	46.14	59.54	54.22	50.25	–	49.91
24	MrMonte(0, 200, FarmerBoy(0))	67.84	68.90	59.18	59.81	53.72	46.73	46.94	59.97	55.96	52.86	50.09	–

[1] Relies on two MrMonte(0, 10, FarmerBoy(0))

Table 6.3: Tournament Summary; the symbols stand for the win percentages p of the row players against the column player: $\ominus$ if $p \leq 45\%$, $\oplus$ if $p \geq 55\%$ and $\bullet$ else. The main diagonal is hinted by dashes.

Player vs. Player	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1 `MrRandom`	–	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖
2 `MrMonte(0, 50, MrBlack(0))`	⊕	–	•	•	•	⊖	⊖	⊕	•	•	•	⊖	⊕	⊕	•	•	•	⊖	⊖	•	•	•	⊖	⊖
3 `MrMonte(0, 100, MrBlack(0))`	⊕	•	–	•	•	•	•	⊕	•	•	•	•	⊕	⊕	⊕	⊕	•	⊖	⊖	⊕	•	•	•	•
4 `MrMonte(0, 150, MrBlack(0))`	⊕	•	•	–	•	•	•	⊕	⊕	•	•	•	⊕	⊕	⊕	⊕	•	•	⊖	⊕	•	•	•	•
5 `MrMonte(0, 200, MrBlack(0))`	⊕	•	•	•	–	•	•	⊕	⊕	•	•	•	⊕	⊕	⊕	⊕	•	•	•	⊕	⊕	•	•	•
6 `MrMonte(1, 200, MrBlack(0))`	⊕	⊕	•	•	•	–	•	⊕	⊕	⊕	⊕	•	⊕	⊕	⊕	⊕	⊕	•	•	⊕	⊕	⊕	•	•
7 `MrMonte(0, 200, MrBlack(1))`	⊕	⊕	•	•	•	•	–	⊕	⊕	⊕	•	•	⊕	⊕	⊕	⊕	⊕	•	•	⊕	⊕	•	•	•
8 `MrBlack(0)`	⊕	⊖	⊖	⊖	⊖	⊖	⊖	–	•	⊖	⊖	⊖	⊕	⊕	•	•	•	⊖	⊖	•	•	⊖	⊖	⊖
9 `MrBlack(1)`	⊕	•	•	⊖	⊖	⊖	⊖	•	–	•	•	⊖	⊕	⊕	•	•	•	⊖	⊖	⊕	•	•	⊖	⊖
10 `MrBlack(2)`	⊕	•	•	•	•	⊖	⊖	⊕	•	–	•	⊖	⊕	⊕	•	•	•	⊖	⊖	⊕	•	•	•	•
11 `MrBlack(3)`	⊕	•	•	•	•	⊖	•	⊕	•	•	–	•	⊕	⊕	⊕	⊕	•	⊖	⊖	⊕	•	•	•	•
12 `MrBlack(4)`	⊕	⊕	•	•	•	•	•	⊕	⊕	⊕	•	–	⊕	⊕	⊕	⊕	•	•	•	⊕	•	•	•	•
13 `FarmerBoy(0)`	⊕	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	–	•	⊖	⊖	⊖	⊖	⊖	•	⊖	⊖	⊖	⊖
14 `FarmerBoy(1)`	⊕	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	•	–	⊖	⊖	⊖	⊖	⊖	•	⊖	⊖	⊖	⊖
15 `FarmerBoy(2)`	⊕	•	⊖	⊖	⊖	⊖	⊖	•	•	•	⊖	⊖	⊕	⊕	–	•	⊖	⊖	⊖	•	•	⊖	⊖	⊖
16 `FarmerBoy(3)`	⊕	•	⊖	⊖	⊖	⊖	⊖	•	•	•	⊖	⊖	⊕	⊕	•	–	⊖	⊖	⊖	•	•	⊖	⊖	⊖
17 `FarmerBoy(4)`	⊕	•	•	•	•	⊖	⊖	•	•	•	•	•	⊕	⊕	⊕	⊕	–	⊖	⊖	⊕	•	•	•	•
18 `MrMonte(2, 20, FarmerBoy(0))`	⊕	⊕	⊕	•	•	•	•	⊕	⊕	⊕	⊕	•	⊕	⊕	⊕	⊕	⊕	–	•	⊕	⊕	•	•	•
19 `MrMonte(2, 40, FarmerBoy(0))`	⊕	⊕	⊕	⊕	•	•	•	⊕	⊕	⊕	⊕	•	⊕	⊕	⊕	⊕	⊕	•	–	⊕	⊕	⊕	•	•
20 `MrMonte(0, 20, MrMonte(0))`[1]	⊕	•	⊖	⊖	⊖	⊖	⊖	•	⊖	⊖	⊖	⊖	•	•	•	•	⊖	⊖	⊖	–	•	⊖	⊖	⊖
21 `MrMonte(0, 50, FarmerBoy(0))`	⊕	•	•	•	⊖	⊖	⊖	•	•	•	•	•	⊕	⊕	•	•	•	⊖	⊖	•	–	•	•	⊖
22 `MrMonte(0, 100, FarmerBoy(0))`	⊕	•	•	•	•	⊖	•	⊕	•	•	•	•	⊕	⊕	⊕	⊕	•	•	⊖	⊕	•	–	•	•
23 `MrMonte(0, 150, FarmerBoy(0))`	⊕	⊕	•	•	•	•	•	⊕	⊕	•	•	•	⊕	⊕	⊕	⊕	•	•	•	⊕	•	•	–	•
24 `MrMonte(0, 200, FarmerBoy(0))`	⊕	⊕	•	•	•	•	•	⊕	⊕	•	•	•	⊕	⊕	⊕	⊕	•	•	•	⊕	⊕	•	•	–

[1] Relies on two `MrMonte(0, 10, FarmerBoy(0))`

6.3 Tweaking the Hash Map Size

In 3.1 hash maps were discussed to speed up the Minimax algorithm. Their success relies on the fact that memory access is usually much cheaper than reevaluating a whole subtree. Modern computers are equiped with a number of layered caches. Their purpose is to mitigate the effects of the interconnection crisis according to which there is a constantly growing gap between the CPUs internal performance and external bandwidth. On the one hand, when the hash map's size is increased, fewer boards need to be reevaluated. But on the other, the efficience of the cache memories is degraded. To evaluate these contrary effects, I have run a short benchmark. It consisted of a `MrBlack(6)` performing a tree search beginning with a fixed starting position. It used an upper hash map of variable size and no lower hash map. Figure 6.4 on the next page allows four interesting observations:

- An increase of the hash map can dramatically improve the performance.
- A map with less than 1024 entries does hardly improve performance.
- Beginning with 2^{23} entries, the search requires more time. At this size, the simulation machine began swapping out memory pages because its main memory was filled up. Hard disks are some magnitudes slower than RAM, which caused the whole simulation to run slower.
- Contrary to all expectations, there are nearly no observable CPU-cache effects. There is just a slight bend upwards between 2^{14} and 2^{16} entries, which drops again afterwards. This is about the size when the hash map didn't fit anymore into the second level cache of the simulation machine and was moved to the RAM.

Judging from the measurement results, the hash map should be as large as possible, but not larger than the computer's RAM can host.

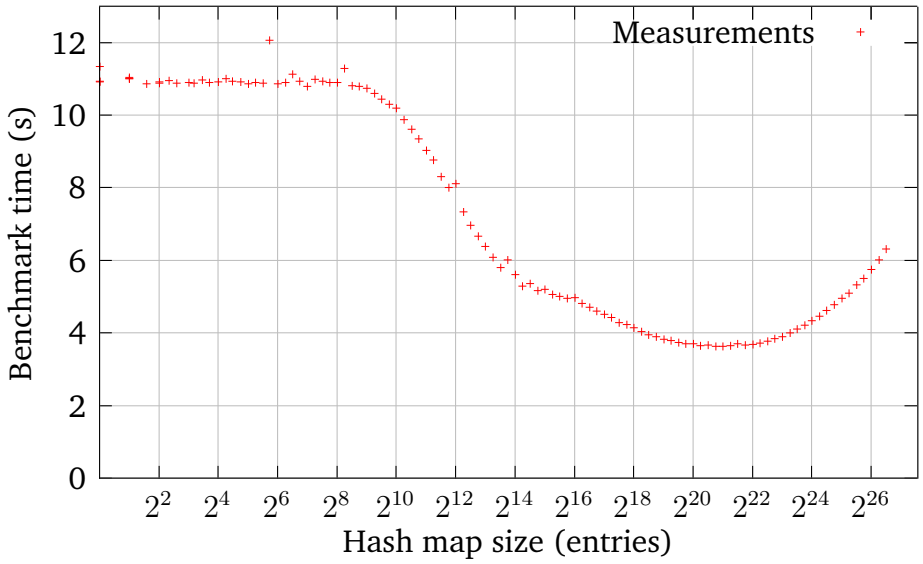


Figure 6.4: Benchmark runtimes depending on the hash map size. See Section 6.3 on the previous page.

7 GUI Implementation

The engine itself was designed from a performance and coding perspective. Soon it became obvious that the simple interface offered by `StdInPlayer` (see section 5.3 on page 26) was not convenient as a human interface. For a user (like me) it was to hard to continuously translate the machine's display. I knew that during the final match (Chapter 1 on page 7) we would play with a limited supply of time. I didn't want to waste the precious time for handling the computer input/output but use it for real computations.

Therefore I decided to code a simple graphical user interface (GUI). Figure 7.1 on the following page shows a screen shot. It is based on Trolltech's Qt Toolkit [Tro05] and written in C++, just like the engine. Qt has been recently re-released under a free license, too. It has the advantage of integrating seamlessly into the engine's build process, being reasonably fast and solving the cross-platform problem for the GUI. Its widgets integrate smoothly into a platform's standard widget set and so a Qt Application looks like a native Windows application under Windows and like a real Linux program when compiled for Linux.

Most of the GUI's classes are widgets. According to the state of the game they have to alter their behavior; internally they are implemented as finite state machines. Simple machines can be coded with switch/case statements in the methods to control the objects behavior. However, when the objects become more and more complex, maintaining the methods can become a real pain. Rock'n'Roll's widgets are implemented according to the State Pattern [EGV95]. Similar to the Delegate Pattern the State Pattern equips an object A with a background object B. All method calls are forwarded to this object. When A is required to alter its state, the background object B is replaced by another one representing the new state.

This makes it possible to maintain a states source in a dedicated class at one place (high cohesion) and to introduce new states without meddling in the others (low coupling). Sub-states are modeled as sub-classes of their super-states.

The internal communication infrastructure of the GUI is based on the Observer Pattern [EGV95]. The internal state of the game is encapsulated in a single object and the observing objects (players, dice-widgets, chat boxes etc.) are notified if anything interesting happens.

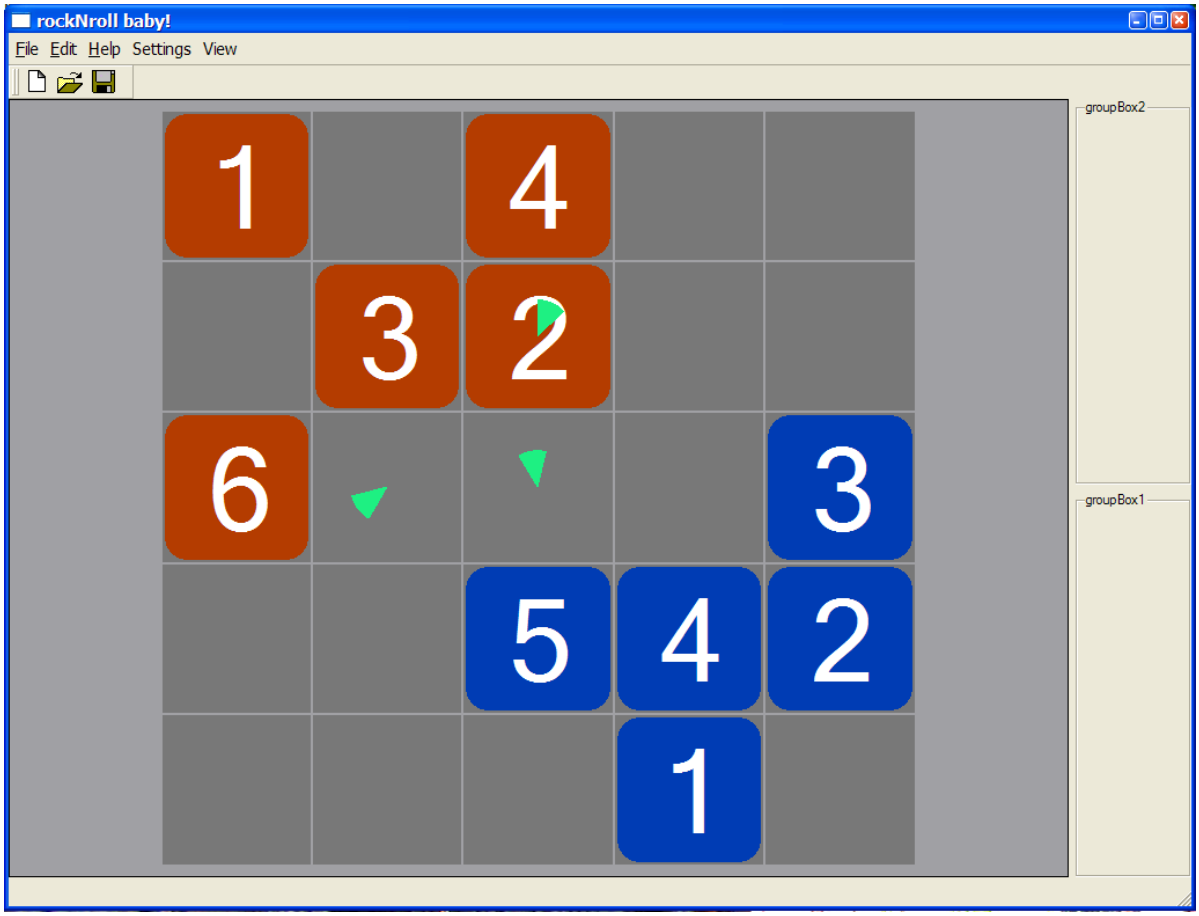


Figure 7.1: Rock'n'Roll GUI Screenshot in mid-game. The red player has thrown a 3 and may now decide on three different fields to move to.

8 Outlook

Although the project itself is finished as a universitary project, Rock'n'Roll is far from being dead. Here are some points I hope to cover in the near future:

- The GUI is not at all perfect. One reason for this is that, despite all the tricks used, static languages like C++ are not really suitable when it comes to implementing state charts. This is because they can not *really* change the class of an object but have to rely on error prone, hard-to-maintain multiway switch statements. Currently an advanced GUI is being developed using the Ruby language for the state machines, Qt bindings for the widgets and the proven C++ engine.
- Theo van der Storm has extended the Schwarz Tables by replacing the simple number of expected moves remaining with a probability distribution. This allows a higher precision of the heuristic. Rock'n'Roll is not yet equipped with this heuristic.
- The hash maps can still be further optimized. For one it could be tried to use two maps of different size or use just one map with a higher associativity and an advanced replacement strategy. This could be e.g. a combination of "least recently used" and "remaining search depth".
- For the end game situations, where typically a rather small number of pieces remains on the board, the optimal moves could be chosen from a database.
- Today, Rock'n'Roll is not the only EinStein program anymore. There are many competitors, numbered in temporal order of their occurrence:
  1. Hanfried by Jörg Sameith and Stefan Schwarz (Jena),
  2. MeinStein by Theo van der Storm (Amsterdam),
  3. Jonny by Johannes Zwanzger (Bayreuth),
  4. Jazzio by Munjong Kolss (Friborg, CH),
  5. Hubbie by Hubert Baumgarten (Hannover),
  6. Chess77 by Eiko Bleicher (Berlin),
  7. Fraggle by Ingo Schwab (Bonn),
  8. NoName by Lars Bremer (Hannover).

It would be interesting to find out, who's best and learn from each other by arranging a tournament.

A References

A.1 Bibliography

[Bla05] Jim Blandy. Subversion Version Control. Accessed on 29th November, 2005.
`http://subversion.tigris.org`.

[EGV95] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Ess05] Thomas Esser. TeTex Distribution. Accessed on 29th November, 2005.
`http://tug.org/teTeX`.

[Kau05] Christian Kauhaus. Jenaer Cluster Technische Informatik. Accessed on 29th November, 2005.
`http://www2.informatik.uni-jena.de/cmc/racluster`.

[Llo05] Noel Llopis. Exploring the C++ Unit Testing Framework Jungle. Accessed on 29th November, 2005.
`http://www.gamesfromwithin.com/articles/0412/000061.html`.

[Mat05] Yukihiro Matsumoto. The Object-Oriented Scripting Language Ruby. Accessed on 29th November, 2005.
`http://www.ruby-lang.org/en`.

[Tro05] Trolltech. Qt. Accessed on 28th November, 2005.
`http://www.trolltech.com/products/qt/index.html`.

[TW05] Colin Kelley Thomas Williams. Gnuplot. Accessed on 29th November, 2005.
`http://www.gnuplot.info`.

[Vol05] Erez Volk. CxxTest Framework. Accessed on 29th November, 2005.
`http://cxxtest.sourceforge.net`.

[Wei05] Jim Weirich. Rake – Ruby Make. Accessed on 29th November, 2005.
`http://rake.rubyforge.org`.

[Wik05a] Wikipedia. Deep Blue Article. Accessed on 24th November, 2005.
`http://en.wikipedia.org/wiki/Deep_Blue`.

[Wik05b] Wikipedia. Expect-Minimax Article. Accessed on 24th November, 2005. http://en.wikipedia.org/wiki/Expectiminimax_tree.

[Wik05c] Wikipedia. Game Tree Article. Accessed on 24th November, 2005. http://en.wikipedia.org/wiki/Game_tree.

[Wik05d] Wikipedia. Minimax Article. Accessed on 24th November, 2005. http://en.wikipedia.org/wiki/Minimax.

A.2 CD Contents

Here is an overview of the files on the included CD. Directories have a trailing slash "/".

File	Description
rockNrollPrint.pdf	This document
rockNrollOnline.pdf	The document version meant for online publishing
rockNroll/	A Windows binary release of the GUI and engine for instant testing. This version was also used during the final match against Stefan.
trunk/	The complete source code of the project
trunk/cxxtest/	Vendor sources of the CxxTest unit testing framework
trunk/database/	Preliminary sources for the end game database
trunk/engine/	The engine's C++ source files
trunk/gui/	GUI source code
trunk/misc/	Miscellaneous files of the build system
trunk/paper/	The LaTeX sources and measurement results for this document.
trunk/shellinterface/	The interface used for batch tests on both my development machine and the cluster
trunk/testsuites/	The engine's test suites for use with CxxTest
trunk/Makefile	In the Makefile the top level rules for the engine's build system are defined.