# The perfect playing Nine Men's Morris computer

Author: Thomas Weber

Datum: 20th June 2009

## 1 Introduction

The principle of the perfect playing Nine Men's Morris game computer is based on a database that contains information about any game situation. Each game situation is from the perspective of the player who is currently in turn. The stored information for each game situation consists of 'state value' and the minimum number of moves to win or the maximum number of moves until the game is lost, called 'ply value'. The ply value, stored in the file 'plyInfo.dat', is a natural number and the state value, stored in the file 'database.dat', may be in one of the following four values:

+, if winning is possible regardless of the actions of the enemy,
0, if the outcome of the game is a draw,
-, if the opponent can win with perfect play,
x, if the situation has not yet been calculated or is invalid.

How can we derive the perfect move according to this information? Suppose an arbitrary game situation with a state value of '+'. Figure 1 shows four possible moves for this situation and the situation value after each of these four features. Here it is apparent that the second or the fourth move should be aspired, since they lead to game situations which seem to be lost from the perspective of the opponent.

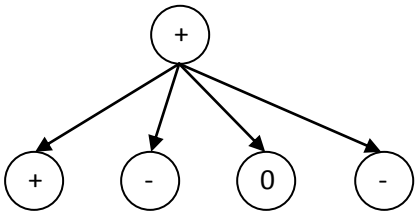



Figure 1: situation with value '+' and four possible moves

The same principle is also applicable to situations with the value '0' in order to identify the perfect turn, while it is irrelevant for situations with the value '-', which stone is moved, because the opponents is going to win anyway with perfect playing.

## 2 Rules of the game

The game is played by two players and includes nine white stones, nine black stones and the game board (see Figure 2) with 24 fields. The gameplay consists of three phases and will be played through in the following order: setting phase, moving phase and jumping phase. During the setting phase each player places in turn one of his nine stones on a free field. The moving phase is characterized, as the name suggests, by alternately pulling the stones along the marked lines, which each connects two fields. The final jumping phase is initiated by the player who is in possession of only three stones, who are allowed to therefore jump with his stones on each free field. A so-called mill will be closed by the arrangement of three stones of a player in a row. This can be the case in any of the three phases of the game and has the immediate removal of any opponent's stone as consequence, provided it is not part of a closed mill.

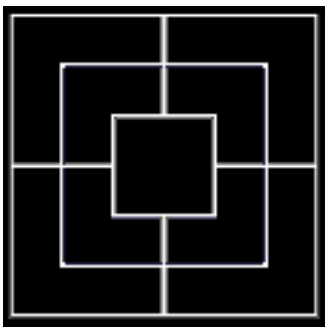


Figure 2: board of the Nine Men's Morris game

# 3 Implementation

## 3.1 Mapping of the game situations to native numbers

First of all, developing an algorithm which masters the assignment of every possible game situation in a natural number, now called the 'state number', is essential. Equally important is the inverse function, which returns for a given state number the corresponding game situation. The ideal case would be, starting the numbering at one until the total number of all possible states. These assignments are required in order to retrieve and to save the state value. A very simple, but a very memory expensive, mapping algorithm would use 24 numbers of a ternary system, where each number corresponds to a field adopting the value "no stone", "white stone" or "black stone". Here the largest produced state number would be

$$N_a = 3^{24} = 282.429.536.481$$

However, considering that at maximum 9 "distinguishable" stones of each of both colors can be placed on the board, the number of states reduces to

$$N_b = \sum_{i=0}^{9}\sum_{j=0}^{9} \frac{24!}{(24-i)!} \cdot \frac{(24-i)!}{(24-i-j)!} = \cdots$$

When considering symmetry operations the amount of states reduces nearly another factor of 16. I say nearly, since some symmetry operation lead to the same state for some special states, e.g. the situation with one stone on each corner and the rotation of the board. The 16 symmetry operations are any combination of rotation around 0°, 90°, 180° and 270°, mirroring in the horizontal, vertical and two diagonal axes of symmetry and the inversion of the board from the inside to outside, see Figure 3.

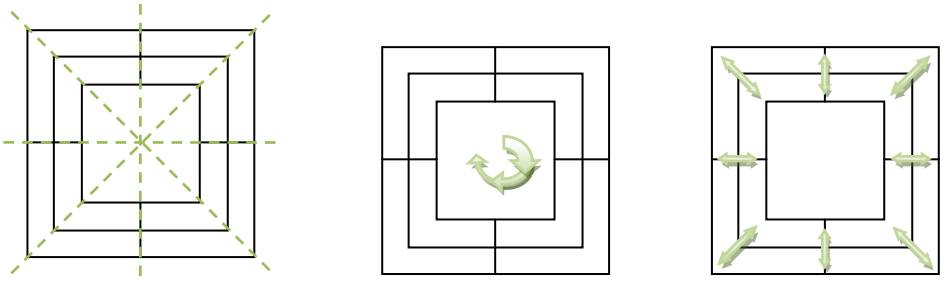


Figure 3: the basic symmetry operations

Thus a maximum state number of

$$N_c = 7.673.759.269.$$

results. The realized implementation here uses a maximum state number of

$$17.980.008.240 \text{ for the setting phase}$$

$$\text{and } 17.169.514.230 \text{ for the moving and jumping phase}$$

This circumstance yields from a not optimally use of symmetry.

## 3.2 The calculation of the database

The state values and the ply values must be calculated backwards based on each other starting from the end of the game. Two different methods were used: the retro-analysis [1] for the moving and jumping phase and the minimax algorithm for the setting phase, which cannot be applied for the other two phases of the game due to cyclic trees.

### 3.2.1 Retro-Analysis

First, the state values for all game situations in the database are marked as invalid, as undecided, as won or as lost. The state of loss is set when a player turn begins and none of his tokens can be moved or he has fewer than three left. Conversely, the situation is seen as won when one of the two mentioned cases applies for the opponent. Invalid for example are states where a player possesses only one stone. Drawn states are all the resting game situations. Similarly, the ply value will be zero in won or lost situations, as this means the end of the game and thus no more moves are needed.

The principle of retro-analysis is now to manage for each single ply value a list of game situations, which state value has already been calculated. A loop processes successively all elements of each list, starting with the list that belongs to ply value zero. For example if for a situation the state value '-' was calculated, it can be concluded that all previous situations are won (see Figure 4). Conversely, the situation value '+' merely provides the information that this branch may not be considered for the opponent player. Only when all branches are blocked, a situation will be valued as lost.

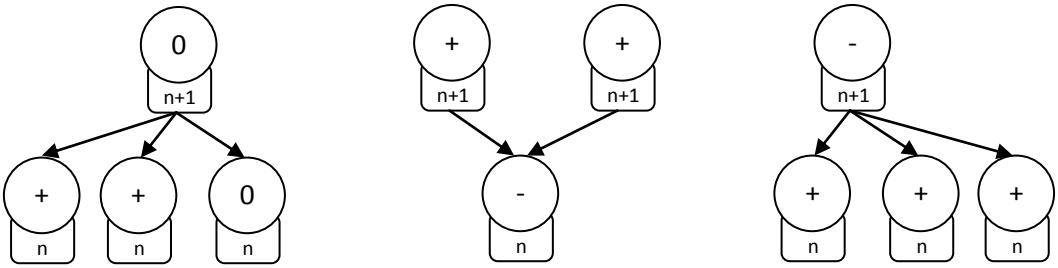



Figure 4: logic of the state values

This requires a counter (see Array 'Count[]' in the pseudo code) for each game situation, which is initialized with the number of possible moves and is reduced by one for each blocked branch. If the value reaches zero one may conclude that it is a lost game situation.

The following pseudo code is intended to illustrate the initialization of the state and ply values and their calculation:

```
// initialization
for (stateNumber=0; stateNumber<numStates; stateNumber++) {
        if (situationIsInvalid(stateNumber)) {
                situationValue[stateNumber]   = INVALID;
                plyValue[stateNumber]         = INVALID;
        } else if (situationIsWon(stateNumber)){
                situationValue[stateNumber]   = WON;
                plyValue[stateNumber]         = 0;
                list[0].push_back(stateNumber);
        } else if (situationIsLost(stateNumber)) {
                situationValue[stateNumber]   = LOST;
                plyValue[stateNumber]         = 0;
                list[0].push_back(stateNumber);
        } else {
                situationValue[stateNumber]   = DRAWN;
                Count[stateNumber]            = numPredecessors(stateNumber);
                plyValue[stateNumber]         = INFINITE;
        }
}

// iteration
for (curNumPlies=0; curNumPlies<maxNumPlies; curNumPlies++) {

  while (list.size() > 0) {
    curStateNumber = *(list.begin());

    for (curPred=0; curPred < numPredecessors(stateNumber); curPred++) {
      predStateNumber = getStateNumberOfPredecessor(curStateNumber, curPred);

      // only drawn states are relevant here,
      // since the other are already calculated
      if (situationValue[predStateNumber] == DRAWN) {
        // if a state is lost then all predecessors are won
        if (curStateValue == LOST) {
          situationValue[predStateNumber]    = WON;
          plyValue[predStateNumber]          = plyValue[curStateNumber] + 1;
          list.push_back(predStateNumber);
        } else {
          //
          if (Count[predStateNumber] > 0) {
            Count[predStateNumber ]--;
            if (plyValue[predStateNumber] < plyValue[curStateNumber] + 1) {
              plyValue[predStateNumber]      = plyValue[curStateNumber] + 1;
            }
          }

          // when all successor are won states then this is a lost state
          if (Count[predStateNumber] == 0) {
            situationValue[predStateNumber] = LOST;
            list.push_back(predStateNumber);
          }
        }
      }
      // remove first element from list
      list.erase(list.begin());
    }
}
```

### 3.2.2 Minimax Algorithm

As introduction a quotation from Wikipedia [2] may serve:

> *Minimax is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss while maximizing the potential gain. [...] Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves. It has also been extended to more complex games and to general decision making in the presence of uncertainty.*
>
> *[...] Suppose the game being played only has a maximum of two possible moves per player each turn. The algorithm generates the tree on the right, where the circles represent the moves of the player running the algorithm (maximizing player), and squares represent the moves of the opponent (minimizing player). Because of the limitation of computation resources, as explained above, the tree is limited to a look-ahead of 4 moves.*
>
> *The algorithm evaluates each leaf node using a heuristic evaluation function, obtaining the values shown. The moves where the maximizing player wins are assigned with positive infinity, while the moves that lead to a win of the minimizing player are assigned with negative infinity. At level 3, the algorithm will choose, for each node, the smallest of the child node values, and assign it to that same node (e.g. the node on the left will choose the minimum between "10" and "+∞", therefore assigning the value "10" to himself). The next step, in level 2, consists of choosing for each node the largest of the child node values. Once again, the values are assigned to each parent node. The algorithm continues evaluating the maximum and minimum values of the child nodes alternatively until it reaches the root node, where it chooses the move with the largest value (represented in the figure with a blue arrow). This is the move that the player should make in order to minimize the maximum possible*

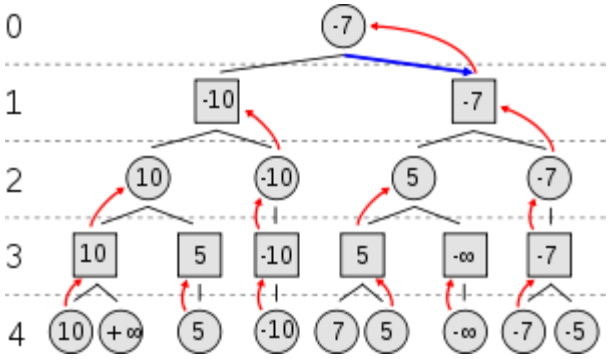



**Figure 5: illustration of a tree used in the minimax algorithm**

For a further detailed explanation of the functioning of the minimax algorithm the reader is referred to the Wikipedia or other articles in the Internet. Important for the calculation of the database are the following two points.

The state value is calculated by the following rules:

| | |
|---|---|
| '+' | if at least one subnode has the value '-' |
| '-' | if all subnodes have the value '+' |
| '0' | if no subnode has the value '-' and at least one the value '0' |

The ply value is determined by the following rules:

| | |
|---|---|
| 0 | if the situation is the end of the game |
| (Maximum ply value of the subnodes) + 1 | if the state value is '-' |
| (Minimum ply value of the subnodes) + 1 | if the state value is '+' |

### 3.2.3 Thoughts about memory consumption

Absolutely crucial for the creation of the database is a delicate utilization of the computer memory. For the current implementation 16GB of RAM are required. Already here should be mentioned that there are two historically-related disadvantageous circumstances. Firstly, the fact that each game situation is saved twice: Once for the normal case, and once in the case that a stone must be removed. Second, counting in half-moves[1] leads to bigger ply values, which means a maximum value over 255, so that one byte of storage is no longer sufficient.

The four possible values of each state value will need 2 bits, while 16 bits are used to store each ply value. In fact, it would require only 9 bits or 8 bits when counting in whole-moves. Assumed at a maximum state number of eight and a half billion therefore would be required

$$(8 + 2 + 8)\text{Bit} \cdot 8{,}5 \cdot 10^9 \cdot 2 = 38{,}25 \cdot 10^9\text{Bytes}$$

Count-Array — state value — ply value — setting and moving phase

for the database, which should be store completely in the random accessible memory, when doing so as described above. However, this can be avoided by taking advantage of the fact that during the setting phase stones are set only and during the moving phase removed only. How it's donewill be described in the following chapter.

[1] Two half-moves or two plies (of player and opponent) are equal to one move, see [3].

### 3.2.3.1 Division of the game situations in layers

To not hold the state and ply values of all game situations in memory at the same time it is one option in the Nine Men's Morris game to divide the game situations into layers depending on the number of stones of each player on the board. All game situations with m black stones (player to move) and n white stones are always summarized to a layer $S_z(m,n)$ for the moving phase and $S_s(m,n)$ for the setting phase. In total there are 200 layers, 100 for the setting phase and 100 for the moving and jumping phase, whereas some contain invalid game situations only (see Figure 6).

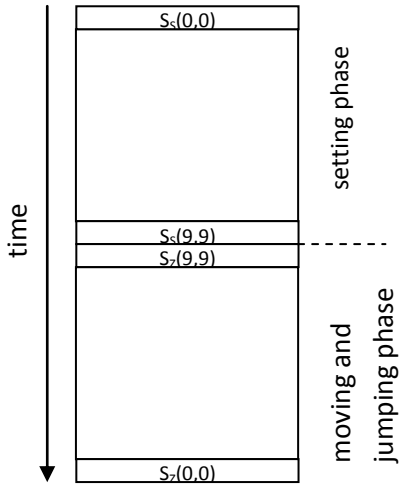



Figure 6: illustration of the layers

In the successive calculation of the state and ply values of the layers $S_z(m,n)$ and $S_z(n,m)$ by means of retro-analysis, it is now sufficient to keep in memory only these two layers and the layers with the direct successor states, which have already been calculated. Each layer $S_z(m,n)$ has a partner layer $S_z(n,m)$, whose game situations can be achieved simply by moving a stone.

Direct successor layers are

$S_z(n-1,m)$ - A stone was removed and now it's the opponents turn.
$S_z(m-1,n)$ - Same facts as $S_z(m-1,n)$ for the partner layer.
$S_z(m,n-1)$ - ???
$S_z(n,m-1)$ - ???

The calculation of setting phase $S_s(m,n)$ using the minimax algorithm may depend on the following layers:

| | | |
|---|---|---|
| $S_S(m+1,n)$ | - | An own stone was placed and closed a mill. |
| $S_S(n-1,m)$ | - | An opposing stone was removed. Now it's the turn of the opponent. |
| $S_S(n, m+1)$ | - | An own stone was placed. Now it's the turn of the opponent. |
| $S_z(n-1,m)$ | - | An opposing stone was removed and setting phase terminated. Now it's the turn of the opponent. |
| $S_z(n,m+1)$ | - | The last stone was placed and finished the setting phase. Now it's the turn of the opponent. |
| $S_z(m,n)$ | - | ??? |
| $S_z(n,m)$ | - | ??? |

The database calculation starts at the layers $S_z(3,2)$ und $S_z(2,3)$, which only contain end game situations.

#### 3.2.3.2 Implementation and symmetry

In principle a surjective mapping of each of the possible $N_a$ states of the game to a natural number between 1 and $N_c$ is needed, with the same number assigned to symmetric states. Unfortunately, I failed to find a mathematical description of this assignment, so that the assignment must be stored as a data field in the memory. For classification of game situations in layers, it is sufficient to use 32 bit variables of type unsigned integer. However, there would be

$$32\text{Bit} \cdot N_b = \cdots \text{Bytes}$$

needed in memory, so the mapping was realized as follows. First, the 24 fields are divided into two groups A and B as shown in Figure 7. It is true that now the symmetry of the game states are not fully exploited, but it is possible to hold separately for both groups, the data fields for the assignments in the memory, as group A only includes $3^8 = 6561$ and group B $3^{16} = 43.046.721$ states.

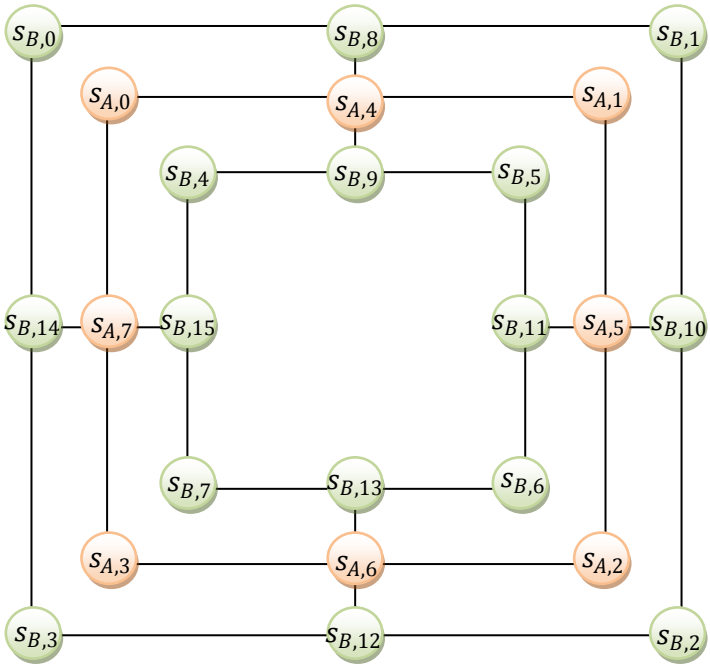



Figure 7: group A in Orange and group B in green

The ensuing detailed description of the symmetry exploitation will require the reader a little effort, so he might take time for reading and understanding. Let's start with the mathematical representation of the stones on the board and the application of symmetry operations. Each field can be in one of three states, depending on whether it is empty, covered with a white or black stone. The values of the fields together form the state vector, which is assigned to each a state number.

i[th] field of the whole board: $s_{G,i} = \begin{cases} 0 & \text{free field} \\ 1 & \text{white stone} \\ 2 & \text{black stone} \end{cases}$

i[th] field of group A: $s_{A,i} = \begin{cases} 0 & \text{free field} \\ 1 & \text{white stone} \\ 2 & \text{black stone} \end{cases}$

i[th] field of group B: $s_{B,i} = \begin{cases} 0 & \text{free field} \\ 1 & \text{white stone} \\ 2 & \text{black stone} \end{cases}$

state vector of group A: $\vec{Z}_A = \left(s_{A,7},...,s_{A,1},s_{A,0}\right)$

state vector of group B: $\vec{Z}_B = \left(s_{B,15},...,s_{B,1},s_{B,0}\right)$

state vector of the whole board: $\vec{Z}_G = \left(s_{G,23},...,s_{G,1},s_{G,0}\right)$

number of states of group A: $N_A = 3^8$

number of states of group B: $N_B = 3^{16}$

total number of states: $N_G = 3^{24}$

state number of state $\vec{Z}^A$: $z_A = \phi\left(\vec{Z}_A\right) = \sum_{i=0}^{7} s_{A,i} \cdot 3^i$

state number of state $\vec{Z}^B$: $z_B = \phi\left(\vec{Z}_B\right) = \sum_{i=0}^{15} s_{B,i} \cdot 3^i$

state number of state $\vec{Z}^G$: $z_G = \phi\left(\vec{Z}_G\right) = \sum_{i=0}^{23} s_{G,i} \cdot 3^i$

state corresponding to state number $z_A$: $\vec{Z}_A = \vec{\phi}^{-1}\left(z_A\right)$

state corresponding to state number $z_B$: $\vec{Z}_B = \vec{\phi}^{-1}\left(z_B\right)$

state corresponding to state number $z_G$: $\vec{Z}_G = \vec{\phi}^{-1}\left(z_G\right)$

i[th] symmetry operator: $\hat{S}_i$

i[th] reverse symmetry operator: $\hat{S}_i^{-1}$

The following example shall explain the state numbers and the impact of the symmetry operators. Let's look at the state in Figure 8 whose state vectors are

$\vec{Z}_A = (0,2,0,1,0,0,0,0)$,
$\vec{Z}_B = (0,0,0,1,0,2,0,2,0,0,0,0,0,0,1,1)$ and
$\vec{Z}_G = (0,2,0,1,0,0,0,0,0,0,0,0,1,0,2,0,2,0,0,0,0,0,0,1,1)$

and state numbers are

$z_A = 2 \cdot 3^6 + 3^4 = 1539$,
$z_B = 3^{12} + 2 \cdot 3^{10} + 2 \cdot 3^8 + 3^1 + 3^0 = 649.543$ and
$z_G = 2 \cdot 3^{22} + 3^{20} + 3^{12} + 2 \cdot 3^{10} + 2 \cdot 3^8 + 3^1 + 3^0 = 66.249.553.162$.

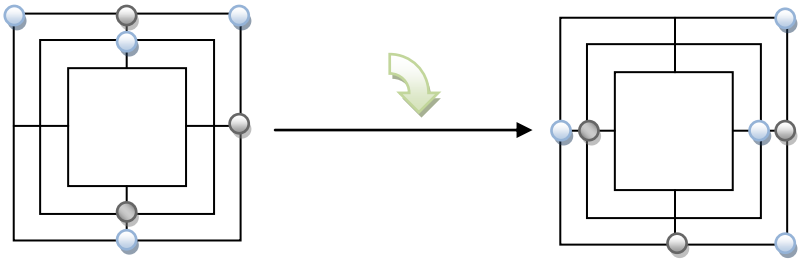


Figure 8: rotation to the right

Application of a symmetry operation, for example the rotation to the right, yields the state vector $\hat{S}_r\vec{Z}_G = (2,0,1,0,0,0,0,0,0,0,1,0,2,0,2,0,0,0,0,0,0,0,0,1,1,0)$ and state number $\phi(\hat{S}_r Z_G) = 198.752.674.818$. So two symmetric states possess two different state numbers. An allocation table serves to assign both states the same number $\tilde{z}_G$.

Allocation of the state number $z_B$ to $\tilde{z}_B$ and vice versa: $\tilde{z}_B = t_B[z_B] \quad z_B = t_B^{-1}[\tilde{z}_B]$

Allocation of the state number $z_G$ to $\tilde{z}_G$ and vice versa: $\tilde{z}_G = t_G[z_G] \quad z_G = t_G^{-1}[\tilde{z}_G]$

As mentioned above, it is not possible to accommodate the mapping table $t_G$ in memory, but at least the much smaller allocation table $t_B$. While it cannot be calculated $\tilde{z}_G$ with this table, but nevertheless a state number $\tilde{z}'_G$ that is close-by to this using the following formula:

$$\tilde{z}'_G = \phi(\hat{S}_{SO(z_B)}\vec{Z}_A)\cdot\tilde{N}_B + t_B[\phi(\vec{Z}_B)]$$

with $SO(z_B)$ as index of the symmetry operation, which satisfies

$$\hat{S}_i^{-1} = \hat{S}_{SO(\phi(\hat{S}_i\vec{Z}_B))} \text{ for } \forall i, \vec{Z}_B$$

and $\tilde{N}^B$ as the number of states for group B after removal of the symmetric states. Where:

$$\tilde{N}_B = \max(\tilde{z}_B) < N_B$$

The reader may now ask why the symmetry operator $\hat{S}_{SO(z_B)}$ is applied to the state $\vec{Z}_A$. This is necessary for the reverse calculation. Based on the state number $\tilde{z}'_G$ it is also possible to calculate the state $\vec{Z}_G = (\vec{Z}_A, \vec{Z}_B)$, which is composed by the states

$$\vec{Z}_A = \hat{S}_{SO(z_B)}^{-1}\vec{\phi}^{-1}(z_A) \quad \text{and} \quad \vec{Z}_B = \vec{\phi}^{-1}(t_B^{-1}[\tilde{z}_B]).$$

The two numbers result from state $\tilde{z}'_G$ as follows:

$$z_A = \left\lfloor \frac{\tilde{z}'_G}{\tilde{N}_B} \right\rfloor \quad \text{and} \quad \tilde{z}_B = \tilde{z}'_G \bmod \tilde{N}_B$$

The function $\lfloor x \rfloor$ is to be understood as a floor function.

## 3.3 Programming environment

| | |
|---|---|
| development environment: | MS Visual Studio 2008 |
| programming language: | C++ |
| used libraries: | Standard Template Library (STL) |
| | MS Win32 API |
| | MS DirectX 9 SDK (April 2007) |
| library files: | Msimg32.lib, d3d9.lib, d3dx9.lib, ddraw.lib, comctl32.lib, shlwapi.lib |

## 3.4 Program hierarchy

The here implemented class "MuehleWin", see Figure 9, accesses the routines of the Windows API. Conceivable, would be to choose a different GUI environment for example for Linux compatibility. Although some adjustments would be necessary in the remaining classes, because they also use the Windows library partly out of convenience. The management of the Nine Men's Morris game, which includes keeping the current game state and movement protocols in memory, is done by the class "muehle", which either waits for the entry of a human player or automatically calls the previously set AI function. In addition to the playing perfectly AI (not shown in Figure 9) there is implemented a random AI and an AI, which uses the alpha-beta algorithm. The minimax algorithm and the algorithm for the retro-analysis are implemented in the class "miniMax" whose name is admittedly a bit inappropriate.

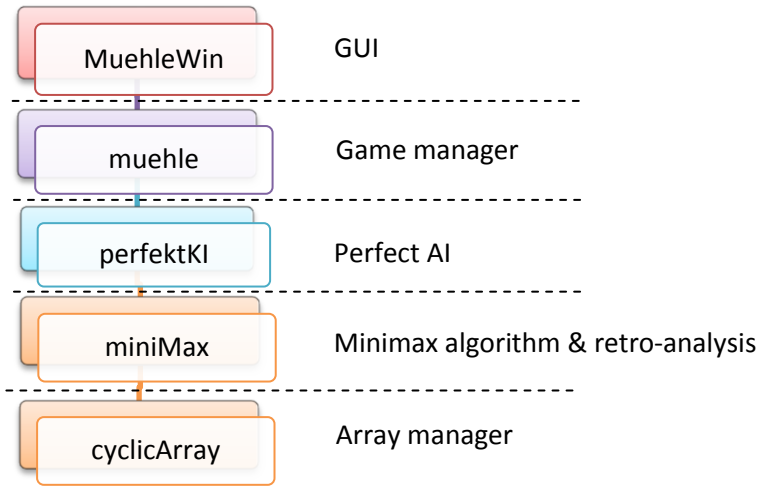



Figure 9: sketch of the program hierarchy

The class "cyclicArray" only serves to manage a very large array on the hard disk drive with continuous read/write access in one direction. Once the writing or reading pointer reaches the end of the array it is reset to position zero. The array is divided into smaller arrays, which are always transferred in the whole from or to the hard disk drive to exploit economies of scale. Although in Figure 10 are shown only 9 divisions, there can be any number.

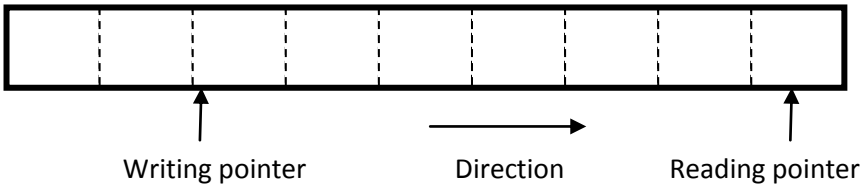



Figure 10: division into smaller arrays

## 3.5 System requirements and calculation time

6 GB für Situationswerte

16 GB für Zuganzahlwerte

4 Wochen Rechenzeit auf Intel Dualcore E6750

⇨ 40 GB große Datenbank

# 4 Optimization potential

- Halbierung: Vollzüge für Zuganzahlwert benutzen, da char, nicht short
- Halbierung: Situationen, bei denen ein Stein entfernt werden muss nicht extra betrachten
- Erfolgsaussicht Run-Time-Encoding schlecht

# 5 Statistics

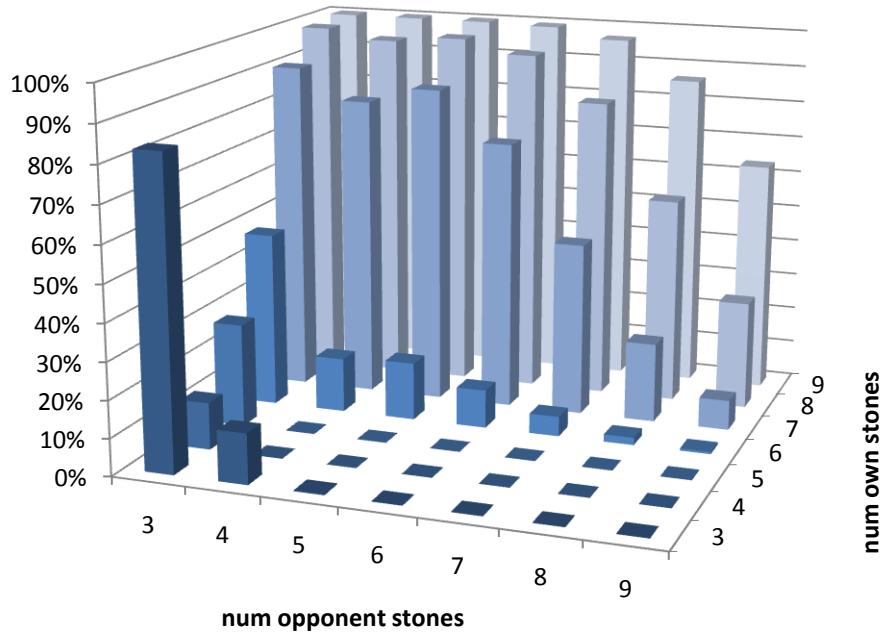
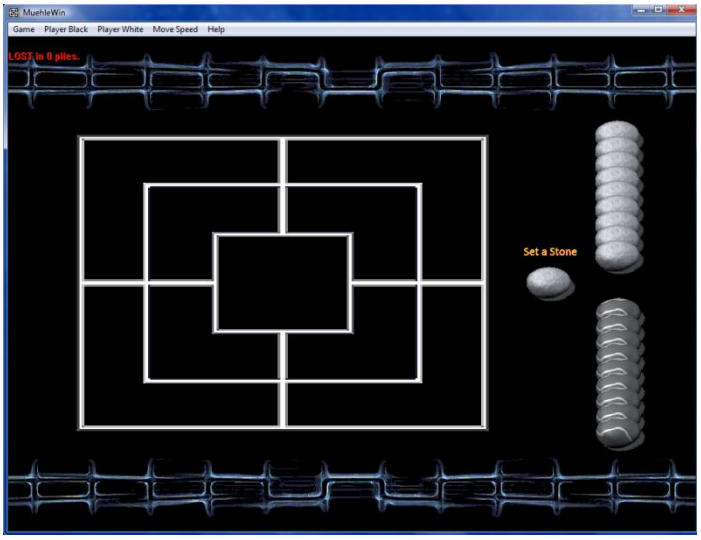


Figure 11: fraction of win situations

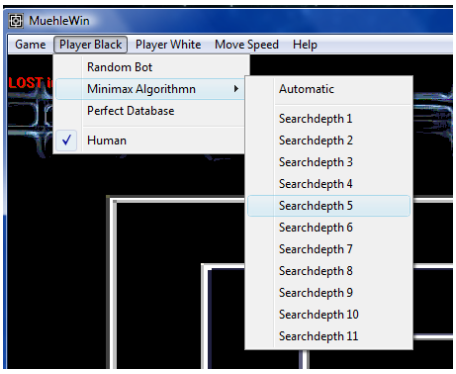
# 6 The graphical user interface

Directly after starting the program "MuehleWin.exe" the following dialog will be presented:



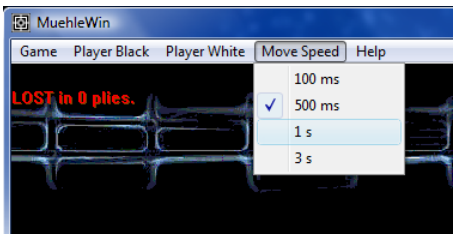

Picture 12: initial window present after program start

By default, a game between two human players is set so that the player with the black stones can start immediately by clicking on one of the playing fields. To select a computer opponent, click on "Player Black" or "White Player" in the menu bar at the top of the window to either select a computer, which plays randomly, one that calculates his moves by using the alpha-beta algorithm or one that plays perfectly. For the second mentioned computer a fixed or an automatic search depth can be chosen, see Picture 13.



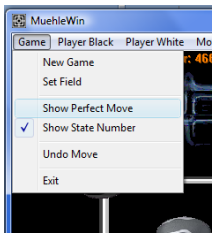

Picture 13: setting up a computer opponent

Via the menu item "Move Speed" in the menu bar one can selected out of a fixed set of four waiting times (100ms, 500ms, 1s, 3s) the desired one for the computer opponent. The actual time for a move is higher because the computation time and duration of the animation must be added.




Picture 14: adjustment of the animation speed of the computer opponent

Now you can still do the following actions with the menu item "Game": Start a new game, enter any game state, display the state numbers and perfect moves, undo the last move and or simply exit the program.



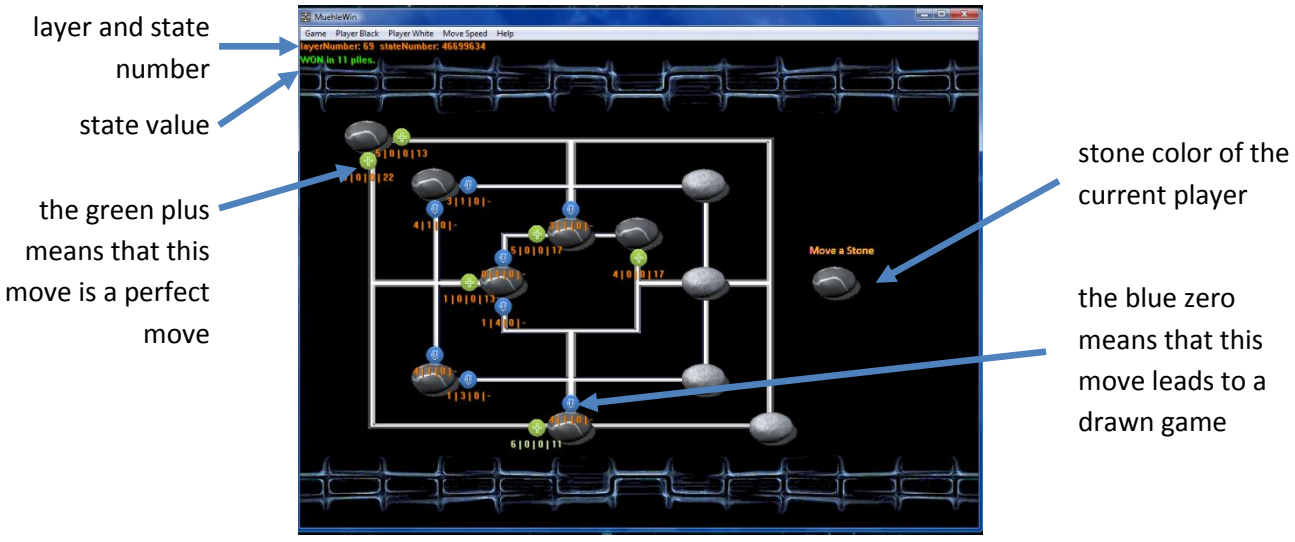

Picture 15: check boxes, saying if the state number and/or the perfect move shall be shown

The layer and state number are displayed in orange at the top left of the window. In the line below the situation value is shown. The outcome of each move is marked with one of three symbols accompanied by 4 numbers:

5 | 0 | 0 | 22 — The game is won in 22 moves - The next move will yield 5 and no / .
The next move will be done by the opponent player, as far as no mill will be closed.

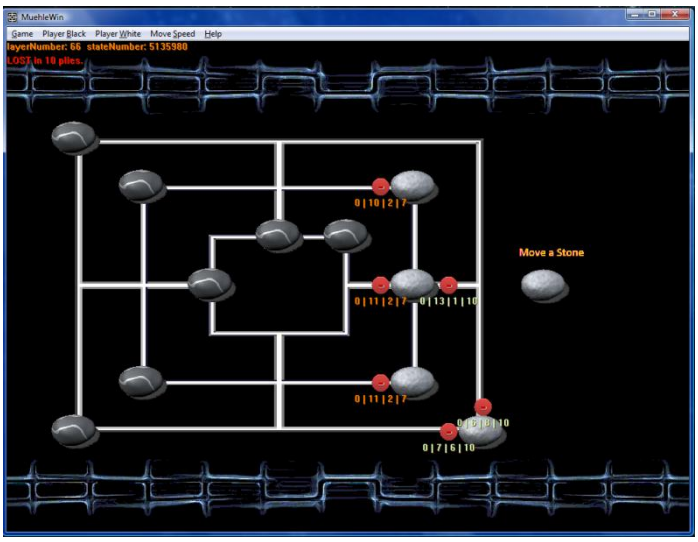
4 | 1 | 0 | - — Drawn - The next move contains 4 and one .
Hierbei ist natürlich keine Anzahl Züge geboten, da ein Unentschieden undendlich lange ist.

0 | 11 | 2 | 7 — Lost game in 7 moves -
The next move contains 11 and 2 .




Picture 16: display of the perfect moves

In the following image the perfect move, see Picture 16, pushing the black stone at the bottom to the left, was performed. Now the desperate situation of the opponent can be observed, each of his 6 possible moves is marked with a red minus. Nevertheless, in order to force the opponent to a maximum number of moves, he should move the white stone right below.




Picture 17: display when the game is unwinnable

**Have fun!**

## References


1. Ralph Gasser, Solving Nine Men's Morris,
   Games of No Chance MSRI Publications, Volume 29, 1996,
   http://www.msri.org/publications/books/Book29/files/gasser.pdf

2. Wikipedia, Minimax,
   http://en.wikipedia.org/wiki/Minimax (as of Feb. 14, 2010, 16:51 GMT)

3. Wikipedia, Ply (game theory),
   http://en.wikipedia.org/wiki/Ply_(game_theory) (as of Feb. 16, 2010, 13:28 GMT).