

# Der perfekte Mühle-Computer

Autor: Thomas Weber

Datum: 20. Juli 2009

## 1 Einführung

Das Prinzip des perfekt spielenden Mühle-Computers basiert auf einer Datenbank, die Informationen über jede mögliche Spielsituation enthält. Jede Spielsituation ist dabei aus der Perspektive des Spielers, der gerade am Zug ist, zu betrachten. Die gespeicherte Information pro Spielsituation besteht aus „Situationswert“ und der Mindestanzahl an Zügen zum Gewinn bzw. der Maximalanzahl an Zügen bis das Spiel verloren ist, im Folgenden „Zuganzahlwert“ genannt. Der Situationswert wird in der Datei „database.dat“ gespeichert und kann folgende vier Werte annehmen:

- +, wenn es unabhängig von den Aktionen des Gegners möglich ist zu gewinnen,
- 0, wenn der Spielausgang unentschieden ist,
- , wenn der Gegner durch perfektes Spielen gewinnen kann.
- x, wenn der Situationswert noch nicht berechnet worden ist oder die Situation ungültig ist.

Der Zuganzahlwert ist eine natürliche Zahl und wird in der Datei „plyInfo.dat“ gespeichert. Wie kann nun der perfekte Zug anhand dieser Informationen abgeleitet werden? Nehmen wir eine beliebige Spielsituation an, die den Situationswert '+' hat. Abbildung 1 zeigt für diese Situation vier mögliche Züge sowie den Situationswert nach jedem dieser vier Züge. Hier wird ersichtlich, dass die Durchführung des zweiten oder des vierten Zuges angestrebt werden sollte, da diese zu Spielsituationen führen die aus der Sicht des Gegners verloren zu sein scheinen.

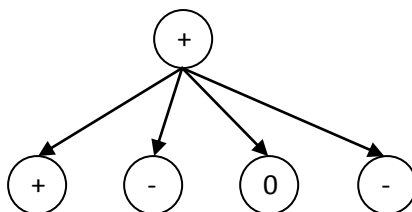


Abbildung 1: Situation mit Wert '+' und vier Zugmöglichkeiten

Nach dem gleichen Prinzip ist es auch für Situationen mit den Wert '0' möglich den perfekten Spielzug zu ermitteln, während es für Situationen mit dem Wert '-' irrelevant ist, welcher Spielstein bewegt wird, da der Gegner bei perfektem Spiel so oder so gewinnen wird.

## 2 Die Spielregeln

Das Mühlespiel wird von zwei Spielern gespielt und umfasst neun weiße als auch neun schwarze Spielsteine sowie das Spielbrett (siehe Abbildung 2) mit 24 Feldern. Der Spielablauf besteht aus drei Phasen und wird auch in der genannten Reihenfolge durchspielt: Setzphase, Zugphase und Springphase. Während der Setzphase positioniert jeder Spieler abwechselnd einen seiner neun Steine auf ein freies Feld. Die Zugphase zeichnet sich, wie der Name schon sagt, durch abwechselndes Ziehen der Steine entlang der eingezeichneten Linien, die jeweils zwei Felder verbinden. Die abschließende Springphase wird durch den Spieler eingeleitet, der nur noch im Besitz dreier Steine ist und somit mit seinen Spielsteinen auf jedes freie

Feld springen darf. Eine sogenannte Mühle wird geschlossen durch das Anordnen von drei Steinen eines Spielers in einer Reihe. Dies kann in jeder der drei Spielphasen der Fall sein und hat das sofortige Entfernen eines beliebigen gegnerischen Spielsteines zur Folge, sofern dieser nicht Teil einer geschlossenen Mühle ist.

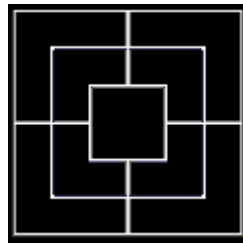


Abbildung 2: Spielbrett

### 3 Umsetzung und Implementierung

#### 3.1 Zuordnung der Spielsituationen zu natürlichen Zahlen

Zunächst einmal ist Entwicklung eines Algorithmus, welcher die Zuordnung jeder möglichen Spielsituation in eine natürliche Zahl, im Folgenden Zustandsnummer genannt, beherrscht, unerlässlich. Ebenso wichtig ist die Umkehrfunktion, welche bei gegebener Zustandsnummer die entsprechende Spielsituation wiedergibt. Der Idealfall nummeriert jede Situation bei eins beginnend bis zur Anzahl aller möglichen Zustände durch. Erforderlich sind diese Zuordnungen um den Situationswert abspeichern bzw. abfragen zu können. Ein besonders einfacher, allerdings zu speichergefräbiger, Zuordnungsalgorithmus würde 24 Stellen eines Trinärsystems nutzen, wobei jede Stelle einem Feld entspricht und jede Stelle den Wert „kein Stein“, „weißer Stein“ oder „schwarzer Stein“ annehmen kann. Hier wäre die größte erzeugte Zustandsnummer

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

Bedenkt man aber, dass sich maximal 9 "unterscheidbare" Steine von jeder Farbe auf dem Spielbrett befinden können, so lässt sich die Anzahl auf

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

reduzieren. Um einen weiteren Faktor 16 lässt sich die Anzahl beim Berücksichtigen von Symmetrieeoperationen verringern. Die 16 Symmetrieeoperationen sind jede Kombination aus Rotation um 0°, 90°, 180° und 270°, Spiegelung an der waagrechten, senkrechten und den beiden diagonalen Symmetrieachsen als auch die Inversion des Spielfeldes von innen nach außen, siehe Abbildung 3.

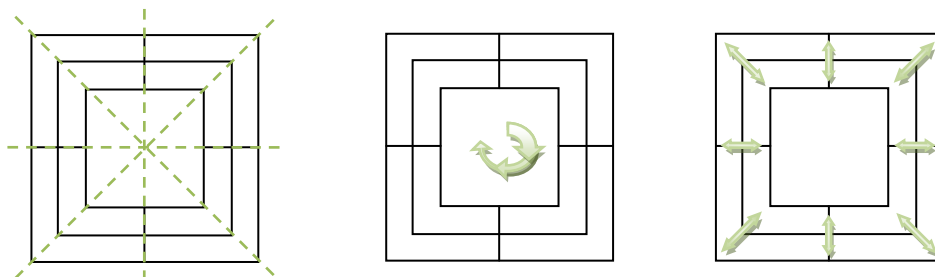


Abbildung 3: Symmetrieeoperationen

Somit resultiert eine maximale Zustandsnummer von

$$N_c = 7.673.759.269.$$

Die hier tatsächlich umgesetzte Implementierung sieht allerdings eine maximale Zustandsnummer von

17.980.008.240 für die Setzphase

17.169.514.230 für die Zug – und Springphase

vor, was aus einer nicht optimalen Symmetrienausnutzung herrührt.

### 3.2 Die Berechnung der Datenbank

Die Situationswerte als auch die Zuganzahlwerte müssen vom Spielende beginnend, rückwärts und aufeinander aufbauend berechnet werden. Dabei finden zwei verschiedene Verfahren Verwendung: Die Retro-Analyse [1] für die Zug- und Springphase und der Alpha-Beta-Algorithmus für die Setzphase, welcher sich aufgrund von zyklischen Bäumen für die beiden anderen Spielphasen nicht anwenden lässt.

#### 3.2.1 Retro-Analyse

Zunächst werden die Situationswerte für sämtliche Spielsituationen in der Datenbank als ungültig, unentschieden bzw. als gewonnen oder verloren markiert. Als verloren gelten die Situationen, wo der Spieler, der am Zug ist, zugunfähig oder weniger als drei Steine hat. Umgekehrt wird der Situationswert als gewonnen gewertet, wenn einer der beiden Fälle für den Gegenspieler zutrifft. Ungültig sind Situationen bei denen ein Spieler z.B. nur noch einen Stein besitzt. Unentschieden sind somit alle restlichen Spielsituationen. Ebenso wird der Zuganzahlwert bei gewonnen oder verlorenen Situationen auf null gesetzt, da dies das Ende das Spiel bedeutet und somit keine weiteren Züge notwendig sind.

Das Prinzip der Retro-Analyse ist nun für jeden möglichen Zuganzahlwert jeweils eine Liste mit den Spielsituationen, dessen Situationswerte berechnet worden sind, zu führen. Dabei verarbeitet eine Schleife sukzessive alle Elemente jeder Liste, beginnend bei der Liste, die zum Zuganzahlwert null gehört. Wurde zum Beispiel berechnet, dass für eine Situation der Situationswert '-' ist, so kann gefolgert werden, dass alle Vorgänger-Situationen, die durch einen Spielzug zu dieser Spielsituation führen können, als gewonnen gelten (siehe Abbildung 4). Umgekehrt liefert der Situationswert '+' lediglich die Information, dass ein Zweig vom Gegenspieler nicht in Erwägung gezogen werden kann. Erst wenn alle Zweige blockiert sind, kann eine Situation als verloren gewertet werden.

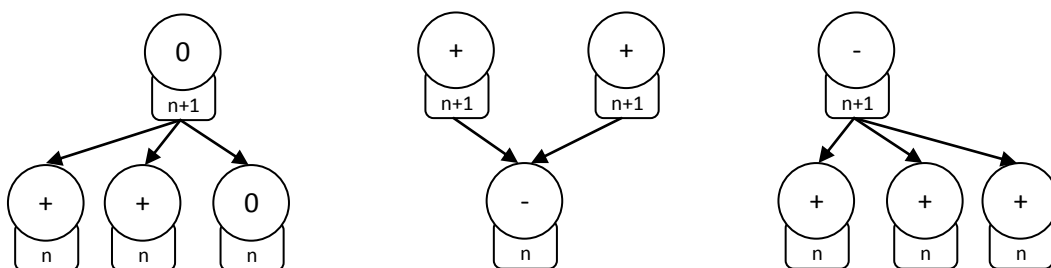


Abbildung 4: Logik der Situationswerte

Dies erfordert somit einen Zähler (siehe Array 'Count[]' im Pseudocode) für jede Spielsituation, der mit der Anzahl an möglichen Zügen initialisiert wird und bei jeden blockierten Zweig um eins reduziert wird. Wird der Wert Null erreicht so kann gefolgert, dass es sich um eine verlorene Spielsituation handelt.

Der folgende Pseudocode soll die Initialisierung der Situations- und Zuganzahlwerte sowie deren Berechnung verdeutlichen:

```
// initialization
for (stateNumber=0; stateNumber<numStates; stateNumber++) {
    if (situationIsValid(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 Alpha-Beta-Algorithmus

Zur Erläuterung des Algorithmus ein Zitat aus der Wikipedia [2]:

*Die Alpha-Beta-Suche ist eine optimierte Variante des Minimax-Suchverfahrens, also eines Algorithmus zur Bestimmung eines optimalen Zuges bei Spielen mit zwei gegnerischen Parteien. Während der Suche werden zwei Werte Alpha und Beta aktualisiert, die angeben, welches Ergebnis die Spieler bei optimaler Spielweise erzielen können. Mit Hilfe dieser Werte kann entschieden werden, welche Teile des Suchbaumes nicht untersucht werden müssen, weil sie das Ergebnis der Problemlösung nicht beeinflussen können. Der Minimax-Algorithmus analysiert den vollständigen Suchbaum. Dabei werden aber auch Knoten betrachtet,*

die in das Ergebnis nicht einfließen. Die Alpha-Beta-Suche versucht, möglichst viele dieser Knoten zu ignorieren.

Für eine detaillierte Erklärung der Funktionsweise des Alpha-Beta-Algorithmus sei der Leser auf den Wikipedia-Artikel im Internet verwiesen. Entscheidend für die Berechnung der Mühle-Datenbank sind jedoch folgende beiden Punkte.

Der Knoten- bzw. Situationswert nach folgenden Regeln berechnet:

'+' wenn mindestens ein Unterknoten den Wert '-' hat

'-' wenn alle Unterknoten den Wert '+' haben

'0' wenn kein Unterknoten den Wert '-' aufweist und mindestens einer den Wert '0'

Der Zuganzahl wird mittels folgenden Regeln ermittelt:

0 wenn die Situation gleichzeitig Spielende ist

(Maximum der Zuganzahlwerte der Unterknoten) + 1 wenn der Situationswert '-' ist

(Minimum der Zuganzahlwerte der Unterknoten) + 1 wenn der Situationswert '+' ist

### 3.2.3 Überlegungen zum Speicherverbrauch

Ganz entscheidend für die Erstellung der Datenbank ist die Ausnutzung des Computerspeichers. Für die gegenwärtige Implementierung werden 16GB RAM benötigt. Gleich vorweggenommen sei der Hinweis, dass zwei historisch bedingte ungünstige Konstellationen vorliegen. Zum einen handelt es sich dabei um die Tatsache, dass jede Spielsituation doppelt abgespeichert wird: Einmal für den ganz normalen Fall und einmal für den Fall, dass ein Stein entfernt werden muss. Zum anderen wirkt sich das Zählen in Halbzügen beim Zuganzahlwert negativ aus, was einen maximalen Wert über 255 hinaus bedeutet, so dass ein Byte zur Abspeicherung nicht mehr ausreicht. Zwei Halbzüge (von Spieler und Gegenspieler) ergeben einen Vollzug.

Die vier möglichen Situationswerte benötigen jeweils 2 Bit, wohingegen für den Zuganzahlwert jeweils 16 Bit zum Abspeichern verwendet werden. Tatsächlich notwendig wären nur 9 Bit bzw. 8 Bit bei der Benutzung von Vollzügen. Bei einer angenommen maximalen Zustandsnummer von achteinhalb Milliarden wären somit

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

Count-Array      Situationswert      Zuganzahlwert      Setz- und Springphase

für die Datenbank erforderlich, welche nach bisher beschriebenen Vorgehensweise, im Arbeitsspeicher vollständig Platz finden müsste. Jedoch kann dies umgangen werden durch die Ausnutzung der Tatsache, dass während der Setzphase nur Steine hinzugefügt und während der Zugphase nur entfernt werden. Wie dies geschieht wird im Folgenden Kapitel beschrieben.

#### 3.2.3.1 Einteilung der Spielsituationen in Schichten

Um nicht die Situations- und Zuganzahlwerte sämtlicher Spielsituationen im Arbeitsspeicher halten zu müssen bietet es sich beim Mühlespiel an die Spielsituationen in Schichten abhängig von der Anzahl der Steine eines jeden Spielers auf dem Spielbrett einzuteilen. Alle Spielsituationen mit m schwarzen Steinen (Spieler der am Zug ist) und n weißen Steinen werden stets zusammenfasst zu einer Schicht  $S_2(m,n)$  für die

Zugphase und  $S_s(m,n)$  für die Setzphase. Insgesamt gibt es 200 Schichten, 100 für die Setzphase und 100 für die Zug- und Springphase, wobei einige lediglich ungültige Spielsituationen beinhalten (siehe Abbildung 5).

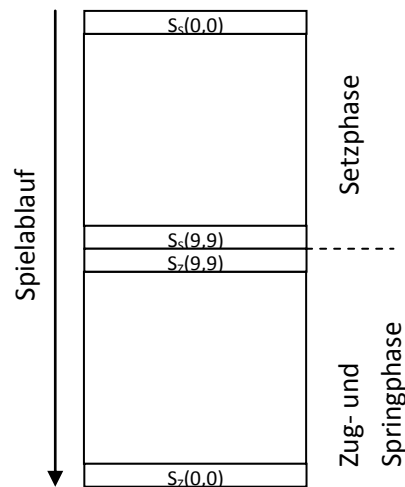


Abbildung 5: Veranschaulichung der Schichten

Beim sukzessiven Berechnen der Situations- und Zuganzahlwerte der Schichten  $S_z(m,n)$  und  $S_z(n,m)$  mittels Retro-Analyse ist es nun ausreichend lediglich diese beiden Schichten sowie die bereits berechneten direkten Nachfolgerschichten im Arbeitsspeicher zu halten. Jede Schicht  $S_z(m,n)$  hat eine Partnerschicht  $S_z(n,m)$ , deren Spielsituationen durch bloßes Ziehen eines Spielsteines erreicht werden kann. Direkte Nachfolgerschichten sind

- $S_z(n-1,m)$  - Ein Stein wurde entfernt und der Gegner ist nun am Zug.
- $S_z(m-1,n)$  - Gleiche Sachverhalt wie  $S_z(m-1,n)$  für die Partnerschicht.
- $S_z(m,n-1)$  - ???
- $S_z(n,m-1)$  - ???

Die Berechnung der Setzphasenschicht  $S_s(m,n)$  mittels Alpha-Beta-Algorithmus kann von folgenden Schichten abhängen:

- $S_s(m+1,n)$  - Ein eigener Stein wurde gesetzt und eine Mühle geschlossen.
- $S_s(n-1,m)$  - Ein fremder Stein wurde entfernt und der Gegner ist nun am Zug.
- $S_s(n, m+1)$  - Ein eigener Stein wurde gesetzt und der Gegner ist nun am Zug.
- $S_z(n-1,m)$  - Ein fremder Stein wurde entfernt und die Setzphase beendet. Der Gegner ist am Zug.
- $S_z(n,m+1)$  - Der letzte Stein wurde gesetzt und die Setzphase beendet. Der Gegner ist nun am Zug.
- $S_z(m,n)$  - ???
- $S_z(n,m)$  - ???

Begonnen wird die Datenbankberechnung bei den Schichten  $S_z(3,2)$  und  $S_z(2,3)$ , die nur Endsituationen enthalten.

### 3.2.3.2 Implementierung der Symmetrienausnutzung

Prinzipiell ist eine surjektive Zuordnung eines jeden der  $N_a$  möglichen Zustände des Spielfeldes in eine natürliche Zahl zwischen 1 und  $N_c$  erforderlich, wobei symmetrische Zustände die selbe Nummer zugeordnet bekommen. Leider ist es mir nicht gelungen eine mathematische Beschreibung dieser

Zuordnung zu finden, so dass die Zuordnung als Datenfeld im Arbeitsspeicher bereitgelegt werden muss. Bei Einteilung der Spielsituationen in Schichten genügt es 32-bit-Variablen des Typs 'positive Ganzzahl' zu benutzen. Allerdings wären dafür

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

Speicher notwendig, weswegen die Zuordnung folgendermaßen realisiert wurde. Zunächst einmal werden die 24 Felder in zwei Gruppen A und B eingeteilt wie in Abbildung 6 gezeigt. Zwar kann nun die Symmetrie der Spielzustände nicht vollständig ausgenutzt werden, aber es ist möglich für beide Gruppen getrennt die Datenfelder für die Zuordnungen im Speicher zu halten, da Gruppe A lediglich  $3^8 = 6561$  und Gruppe  $3^{16} = 43.046.721$  Zustände umfasst.

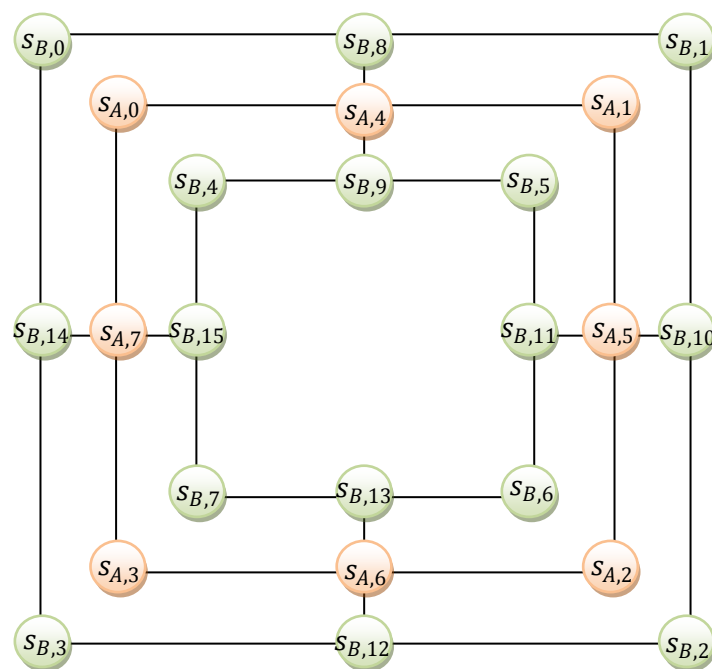


Abbildung 6: Gruppe A in Orange und Gruppe B in Grün

Die nun folgende detaillierte Beschreibung der Symmetrienausnutzung wird dem Leser nun ein wenig Mühe abverlangen, so dass er sich ruhig Zeit nehmen möge zum Lesen und Verstehen. Beginnen wir mit der mathematischen Darstellung der Spielsteine auf dem Spielbrett sowie die Anwendung von Symmetrieoperationen. Jedes Feld kann einen von drei Zuständen annehmen, je nach dem ob es leer, mit einem weißen oder schwarzen Stein belegt ist. Die Werte der Felder zusammen ergeben den Zustandsvektor, dem je eine Zustandsnummer zugeordnet ist.

i-tes Feld vom gesamten Spielbrett:	$s_{G,i} = \begin{cases} 0 & \text{leeres Feld} \\ 1 & \text{weißer Stein} \\ 2 & \text{schwarzer Stein} \end{cases}$
i-tes Feld der Steingruppe A :	$s_{A,i} = \begin{cases} 0 & \text{leeres Feld} \\ 1 & \text{weißer Stein} \\ 2 & \text{schwarzer Stein} \end{cases}$
i-tes Feld der Steingruppe B :	$s_{B,i} = \begin{cases} 0 & \text{leeres Feld} \\ 1 & \text{weißer Stein} \\ 2 & \text{schwarzer Stein} \end{cases}$

Zustandsvektor der Steingruppe A:  $\vec{Z}_A = (s_{A,7}, \dots, s_{A,1}, s_{A,0})$   
 Zustandsvektor der Steingruppe B:  $\vec{Z}_B = (s_{B,15}, \dots, s_{B,1}, s_{B,0})$   
 Zustandsvektor des gesamten Spielfeldes:  $\vec{Z}_G = (s_{G,23}, \dots, s_{G,1}, s_{G,0})$

Anzahl möglicher Zustände für Steingruppe A:  $N_A = 3^8$

Anzahl möglicher Zustände für Steingruppe B:  $N_B = 3^{16}$

Anzahl aller möglichen Zustände:  $N_G = 3^{24}$

Zustandsnummer des Zustandes  $\vec{Z}^A$ :  $z_A = \phi(\vec{Z}_A) = \sum_{i=0}^7 s_{A,i} \cdot 3^i$

Zustandsnummer des Zustandes  $\vec{Z}^B$ :  $z_B = \phi(\vec{Z}_B) = \sum_{i=0}^{15} s_{B,i} \cdot 3^i$

Zustandsnummer des Zustandes  $\vec{Z}^G$ :  $z_G = \phi(\vec{Z}_G) = \sum_{i=0}^{23} s_{G,i} \cdot 3^i$

Zustand zur Zustandsnummer  $z_A$ :  $\vec{Z}_A = \vec{\phi}^{-1}(z_A)$

Zustand zur Zustandsnummer  $z_B$ :  $\vec{Z}_B = \vec{\phi}^{-1}(z_B)$

Zustand zur Zustandsnummer  $z_G$ :  $\vec{Z}_G = \vec{\phi}^{-1}(z_G)$

i-ter Symmetrieoperator:  $\hat{S}_i$

i-ter Umkehrsymmetrieoperator:  $\hat{S}_i^{-1}$

Nun ein Beispiel zur Erläuterung der Zustandsnummern und der Wirkung des Symmetrieoperators. Betrachten wir den Zustand in der Abbildung 7, welcher die Zustandsvektoren

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

$$\vec{Z}_B = (0, 0, 0, 1, 0, 2, 0, 2, 0, 0, 0, 0, 0, 0, 1, 1) \text{ und}$$

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

sowie die Zustandsnummern

$$z_A = 2 \cdot 3^6 + 3^4 = 1539,$$

$$z_B = 3^{12} + 2 \cdot 3^{10} + 2 \cdot 3^8 + 3^1 + 3^0 = 649.543 \text{ und}$$

$$z_G = 2 \cdot 3^{22} + 3^{20} + 3^{12} + 2 \cdot 3^{10} + 2 \cdot 3^8 + 3^1 + 3^0 = 66.249.553.162 \text{ hat.}$$

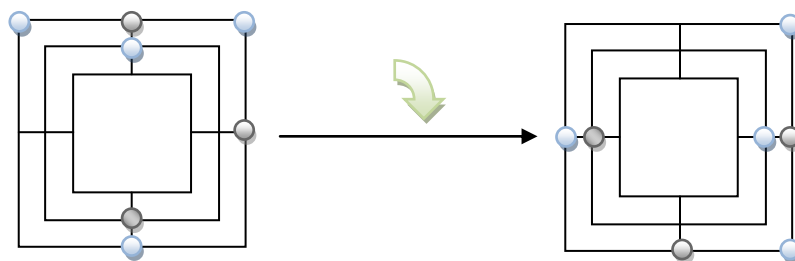


Abbildung 7: Drehung nach rechts



Wird nun die eine Symmetrieoperation angewendet, z.B. die Drehung nach rechts, so ergibt sich der Zustandsvektor  $\hat{S}_r \vec{Z}_G = (2,0,1,0,0,0,0,0,1,0,2,0,2,0,0,0,0,0,0,0,1,1,0)$  sowie die Zustandsnummer  $\phi(\hat{S}_r \vec{Z}_G) = 198.752.674.818$ . Zwei symmetrische Zustände besitzen also zwei verschiedene Zustandsnummern. Eine Zuordnungstabelle dient nun dazu beiden Zuständen dieselbe Zahl  $\tilde{z}_G$  zuzuordnen.

Zuordnung der Zustandsnummer  $z_B$  in  $\tilde{z}_B$  und umgekehrt:  $\tilde{z}_B = t_B[z_B]$   $z_B = t_B^{-1}[\tilde{z}_B]$

Zuordnung der Zustandsnummer  $z_G$  in  $\tilde{z}_G$  und umgekehrt:  $\tilde{z}_G = t_G[z_G]$   $z_G = t_G^{-1}[\tilde{z}_G]$

Wie bereits erwähnt ist es jedoch nicht möglich die Zuordnungstabelle  $t_G$  im Arbeitsspeicher unterzubringen, sondern nur die erheblich kleinere Zuordnungstabelle  $t_B$ . Zwar kann mit dieser nicht  $\tilde{z}_G$  berechnet werden, aber immerhin eine Zustandsnummer  $\tilde{z}'_G$ , die dieser mithilfe folgender Formel sehr nahe kommt:

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

mit  $SO(z_B)$  als Index derjenigen Symmetrieoperation für die gilt

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

und  $\tilde{N}^B$  als die Anzahl der Zustände für Steingruppe B nach Entfernung der symmetrischen Zustände. Dabei gilt:

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

Der Leser möge sich nun fragen weswegen der Symmetrieoperator  $\hat{S}_{SO(z_B)}$  auf den Zustand  $\vec{Z}_A$  angewandt wird. Dies ist für die Umkehrberechnung notwendig. Anhand der Zustandsnummer  $\tilde{z}'_G$  kann somit auch der Zustand  $\vec{Z}_G = (\vec{Z}_A, \vec{Z}_B)$ , welcher sich aus den Zuständen

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

zusammensetzt berechnet werden. Die beiden Zustandsnummern ergeben sich aus  $\tilde{z}'_G$  wie folgt:

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

Die Funktion  $\lfloor x \rfloor$  ist dabei als abrundende Gaußklammer zu verstehen.

### 3.3 Programmdetails

Entwicklungsumgebung:	MS Visual Studio 2008
Programmiersprache:	C++
verwendete Bibliotheken:	Standard Template Library (STL) MS Win32 API MS DirectX 9 SDK (April 2007)
Bibliotheksdateien:	Msmg32.lib, d3d9.lib, d3dx9.lib, ddraw.lib, comctl32.lib, shlwapi.lib

### 3.4 Programmhierarchie

Die hier implementierte Klasse „MuehleWin“, siehe Abbildung 8, greift auf Routinen der Windows-API zu. Denkbar wäre es jedoch eine andere GUI-Umgebung zu wählen, um z.B. Linux-Kompatibilität einzubauen. Wobei allerdings einige Anpassungen bei den restlichen Klassen umgesetzt werden müssten, da diese aus Bequemlichkeit teilweise auch die Windows-Bibliothek nutzen. Die Verwaltung des Mühle-Spiels, zu der das Halten des aktuellen Spielstandes und des Zugprotokolls gehört, geschieht durch die Klasse „muehle“, welche entweder auf die Eingabe eines menschlichen Spielers wartet oder automatisch die zuvor eingestellte KI-Funktion aufruft. Neben der perfekt spielenden KI sind noch (in Abbildung 8 nicht dargestellt) eine Zufalls-KI und eine KI, welche den Alpha-Beta-Algorithmus nutzt, implementiert. Der Minimax-Algorithmus als der Algorithmus für die Retro-Analyse sind in der Klasse „miniMax“, deren Namen zugegebenermaßen ein wenig unpassend ist, untergebracht.

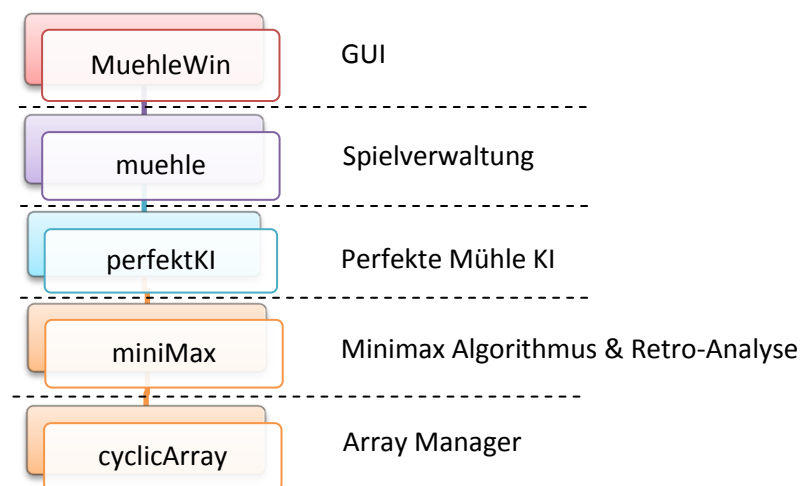


Abbildung 8: Skizze der Programmhierarchie

Die Klasse „cyclicArray“ dient lediglich zur Verwaltung eines sehr großen Arrays auf der Festplatte mit kontinuierlichem Schreib- und Lesezugriff in eine Richtung. Sobald der Schreib- bzw. Lesezeiger am Ende des Arrays angekommen ist wird er auf die Position null zurückgesetzt. Dabei ist das Array in kleinere Arrays

aufgeteilt, die stets im Ganzen von bzw. zur Festplatte transferiert werden um Skaleneffekte auszunutzen. In Abbildung 9 sind zwar nur 9 Unterteilungen dargestellt, jedoch können es beliebig viele sein.

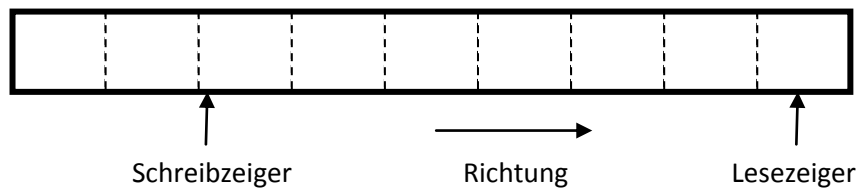


Abbildung 9: Unterteilung des Arrays in kleinere

### 3.5 Systemvoraussetzungen und Rechendauer

6 GB für Situationswerte

16 GB für Zuganzahlwerte

4 Wochen Rechenzeit auf Intel Dualcore E6750

⇒ 40 GB große Datenbank

## 4 Optimierungspotential

- 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 Statistiken

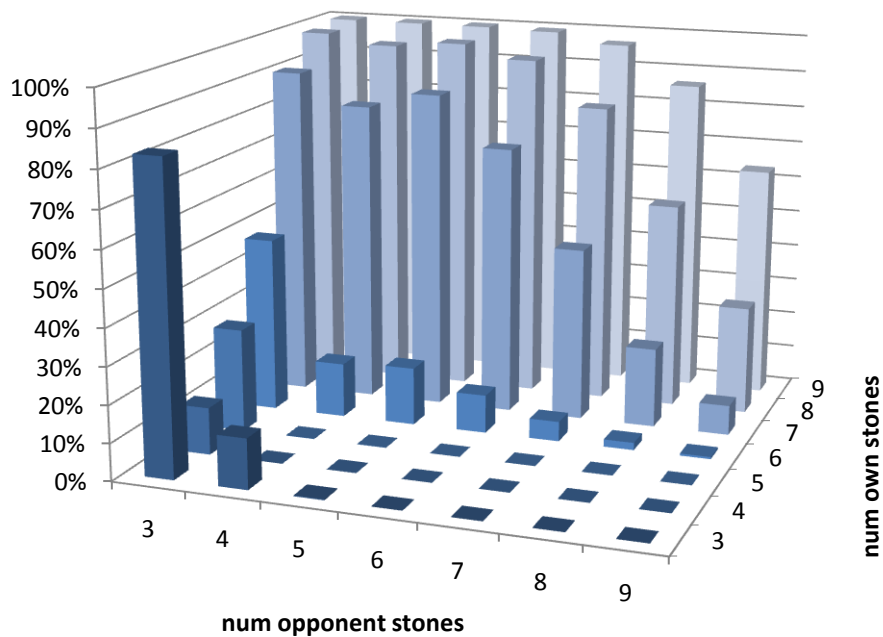


Abbildung 10: Anteil der Spielsituationen, welche zum Sieg führen

## 6 Die Benutzeroberfläche

Direkt nach dem Programmstart der Datei „MuehleWin.exe“ präsentiert sich folgender Fensterdialog:

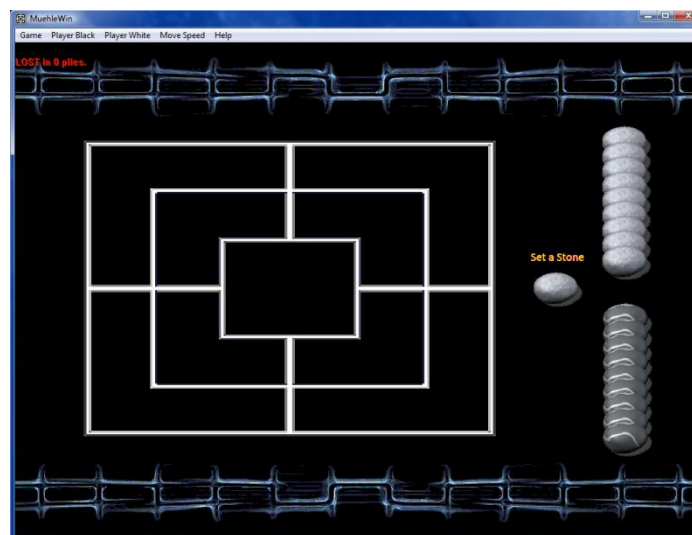


Abbildung 11: Benutzeroberfläche nach dem Programmstart

Standardmäßig ist ein Spiel zwischen zwei menschliche Spieler eingestellt, so dass der Spieler mit den schwarzen Steinen sofort durch klicken auf die Spielfelder loslegen kann. Um einen Computergegner auszuwählen, klicken Sie in der Menüleiste am oberen Fensterrand auf den Eintrag „Player Black“ bzw. „Player White“ um entweder einen Computer, der nach Zufall spielt, einen, der mithilfe des Alpha-Beta-Algorithmus seinen Spielzug berechnet, oder einen, der perfekt spielt, auszuwählen. Für den zweitgenannten Computer kann ggf. eine feste oder eine automatisch Suchtiefe eingestellt werden, siehe Abbildung 12.

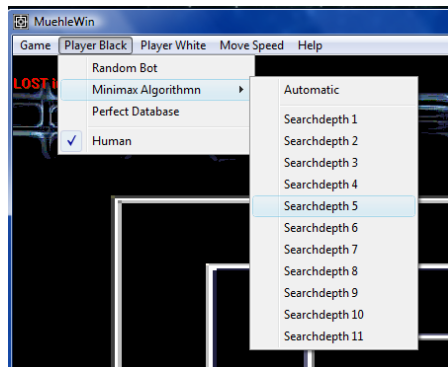


Abbildung 12: Einstellen eines Computergegners

Über den Menüpunkt „Move Speed“ in der Menüleiste lässt sich einer von vier festeingestellten Wartezeiten (100ms, 500ms, 1s, 3s) des Computergegners einstellen. Die tatsächliche Dauer für einen Zug liegt jedoch darüber, da noch die Rechenzeit desjenigen sowie die Animationsdauer hinzugerechnet werden müssen.

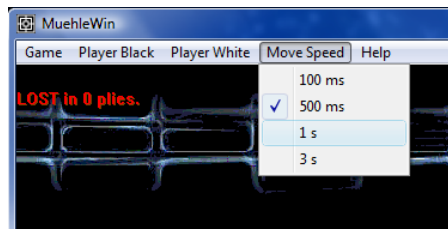


Abbildung 13: Einstellen der Wartezeit des Computergegners

Nun kann noch über den Menüpunkt „Game“ ein neues Spiel gestartet, ein beliebiger Spielzustand eingegeben, die Anzeige der Zustandsnummer als auch des perfekten Spielzuges eingestellt, der letzte Zug rückgängig gemacht und das Programm beendet werden.

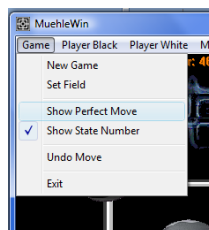












Abbildung 14: Einstellen der Anzeige der Zustandsnummer und des perfekten Zuges

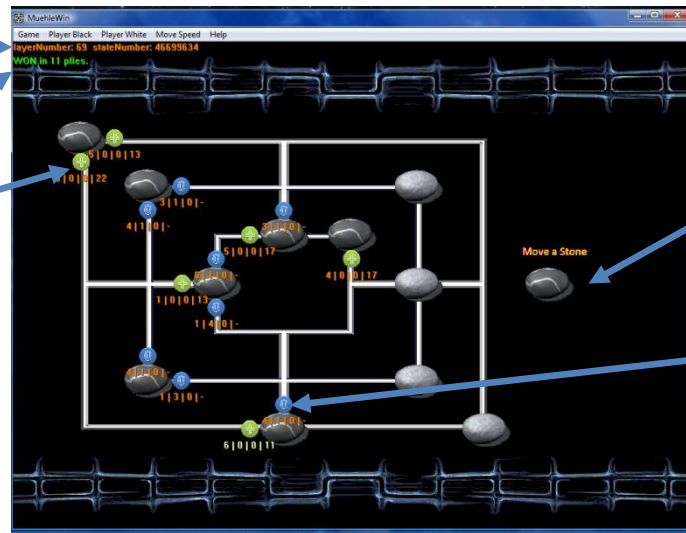
Die Anzeige der Schicht- und Zustandsnummer befindet sich oben links und wird in Orange dargestellt. Direkt eine Zeile darunter wird der Situationswert angezeigt. Der jeweilige Ausgang eines jeden Zuges wird mit einem der folgenden drei Symbole sowie 4 Zahlen gekennzeichnet:

-  Gewonnenes Spiel in 22 Zügen - Der nächste Zug beinhaltet 5  und keine  bzw.  .  
 5 | 0 | 0 | 22 Der nächste Zug wird vom Gegenspieler ausgeführt, sofern keine Mühle geschlossen wird.
-  Unentschieden - Der nächste Zug beinhaltet 4  und ein  .  
 4 | 1 | 0 | - Hierbei ist natürlich keine Anzahl Züge geboten, da ein Unentschieden unendlich lange ist.
-  Verlorenes Spiel in 7 Zügen -  
 0 | 11 | 2 | 7 Der nächste Zug beinhaltet 11  und 2  .

Schicht- und  
Zustandsnummer

Situationswert

Das grüne Plus  
bedeutet, dass  
dieser Zug zum  
Sieg führen kann.



Steinfarbe des zu  
ziehenden Spielers

Die blaue Null  
bedeutet, dass  
dieser Zug zum  
Unentschieden  
führt.

Abbildung 15: Anzeige des perfekten Zuges

Im folgenden Bild wurde der perfekte Zug aus Abbildung 15, Schieben des unteren schwarzen Steines nach links, durchgeführt. Zu erkennen ist nun die ausweglose Situation des Gegenspielers, d.h. jede seiner 6 Zugmöglichkeiten wird mit einem roten Minus markiert. Um dennoch den Gegner zu möglichst vielen Zügen zu zwingen sollte er den weißen Stein rechts unten bewegen.

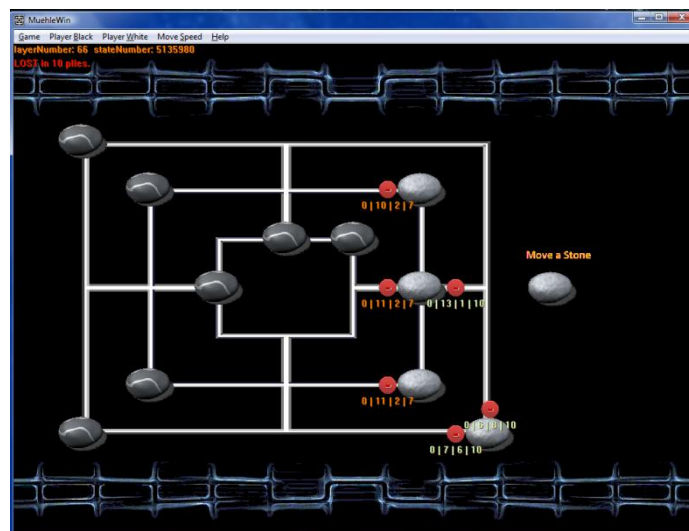


Abbildung 16: Anzeige des perfekten Zuges bei aussichtsloser Situation

**Ich wünsche Euch viel Spaß beim Ausprobieren!**

## Referenzen

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, Alpha-Beta-Suche, Bearbeitungsstand: 14. Oktober 2009, 11:46 UTC <http://de.wikipedia.org/w/index.php?title=Alpha-Beta-Suche&oldid=65572753>