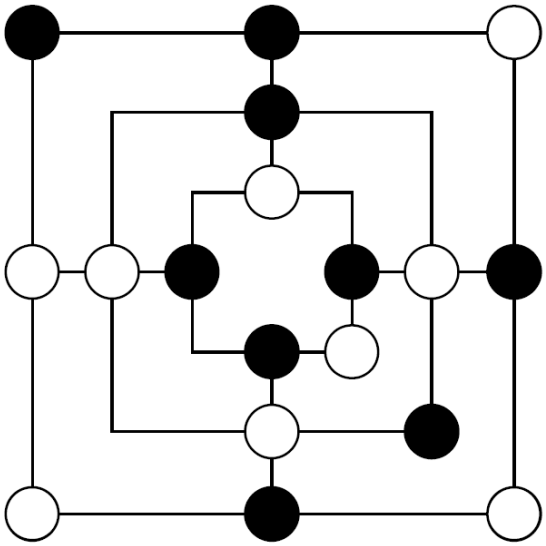
Suchoptimierung und Evaluation bei Zwei-Personen-Nullsummen-Spielen am Beispiel von Mühle



Studienarbeit

am Lehrstuhl für Informatik VI
Prof. Dr. Frank Puppe
Bayerische Julius-Maximilians-Universität Würzburg

19. April 2005

von

Florian Lemmerich Bastian Späth

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Das Mühlespiel

Das Spiel **Mühle** ist neben Spielen wie Go, Pachisi, Backgammon, Dame und Schach eines der ältesten und beliebtesten Brettspiele für zwei Personen. So wurden Mühlebretter in vielen historischen Gebäuden gefunden. Das älteste stammt aus der Zeit um 1400 vor Christus und wurde in einem Tempel in Ägypten gefunden. Bis heute erfreut sich das Mühlespiel großer Beliebtheit. Im Englischen ist es unter dem Namen Nine-Mens-Morris bekannt.

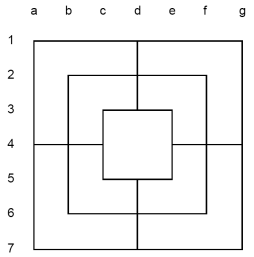


Abbildung 1.1: Mühle Spielfeld

Das Standard-Spielbrett besteht aus drei konzentrisch liegenden Quadraten, die in den Seitenmitten verbunden sind. Die Eckpunkte und die Mittelpunkte bilden 12+12=24 Felder, auf die Spielsteine gesetzt werden können. Ein Mühlespielfeld ist in Abbildung 1.1 dargestellt.
Zu Beginn des Spiels ist das Brett leer. Die beiden Spieler bekommen jeweils neun Spielsteine in Schwarz beziehungsweise Weiß. Der Spieler mit den weißen Steinen beginnt die Partie. Zwölf Felder besitzen je zwei Nachbarfelder, acht je drei Nachbarfelder und vier je vier Nachbarfelder. Jedes Feld ist zweimal durch jeweils eine gerade Verbindung mit je zwei anderen Feldern verbunden. Stehen auf drei so verbundenen Feldern Steine einer Farbe, so bilden sie eine "Mühle".
Ziel des Spiels ist es, Mühlen zu schließen und dem Gegner somit einen Stein nach dem anderen zu nehmen. Sobald ein Spieler nur noch zwei Steine besitzt oder keinen gültigen Zug mehr machen kann, hat dieser die Partie verloren.
Im Spiel unterscheidet man drei Phasen. Die Anfangsphase ist die Setzphase. Hier setzen die Spieler abwechselnd einen ihrer Steine auf ein freies Feld. Schließt einer damit eine Mühle, kann er einen beliebigen, nicht in einer Mühle befindlichen Stein des Gegners aus dem Spiel nehmen.
Sind alle Steine gesetzt, beginnt das Mittelspiel (Schiebephase). Nun können eigene Steine entlang einer Linie um jeweils ein Feld verschoben werden. Auch hier können Mühlen gebil-

det und dem Gegner somit Steine geschmissen werden. Kann ein Spieler nicht mehr ziehen, da seine Steine von den gegnerischen Steinen eingesperrt sind, hat er verloren.
Die letzte Phase beginnt, sobald ein Spieler nur noch drei Steine auf dem Spielfeld hat. Man nennt sie Endspiel. Nun ist es dem Spieler mit den drei Steinen erlaubt, mit einem seiner Steine auf ein beliebiges freies Feld auf dem Spielbrett zu springen. Der andere Spieler spielt, sofern sich noch mehr als drei seiner Steine auf dem Spielfeld befinden, weiter nach den Regeln der Schiebephase. Das Schließen von Mühlen und das Schmeißen von gegnerischen Steinen folgt ebenso den oben beschriebenen Regeln.
Für das Mühlespiel sind heute zahlreiche Varianten verbreitet. Insbesondere wird die Frage, ob in der Setzphase beim Schliessen zweier Mühlen durch das Setzen nur eines Steins ein oder zwei Steine des Gegners geschmissen werden dürfen, unterschiedlich beantwortet. Darüber hinaus gibt es auch Varianten, in denen in bestimmten Fällen das Schmeißen von Steinen innerhalb einer Mühle erlaubt ist. In unserer Arbeit haben wir uns auf die am weitesten verbreiteten Grundregeln beschränkt, bei denen stets nur ein einziger Stein entfernt werden kann und laut denen das Schmeißen von Steinen in Mühlen generell nicht erlaubt ist.

1.2 Ziel dieser Arbeit

Aus spieletheoretischer Sicht ist Mühle ein Zug-basiertes, deterministisches Null-Summen-Spiel. Das heißt, alle Gewinne eines Spielers sind automatisch Verluste des Gegners. Kooperation beider Spieler existiert in einem solchen Spiel natürlich nicht. Anfang der 90er Jahre wurde das Mühlespiel vom Schweizer Ralph Gasser durch eine „Retrogerade Analyse“ untersucht und schließlich als unentschieden gelöst (siehe [7]). Er generierte eine Datenbank, in der alle legalen Spielfeld-Kombinationen gespeichert waren. Zu diesen speicherte er zusätzlich das Spielergebnis. So konnte man feststellen, dass das Spiel Remis endet, sofern beide Spieler perfekt spielen. Nach dieser Erkenntnis geriet das Mühlespiel als informatisches Problem in den Hintergrund. Dennoch gibt es trotz der Existenz einer perfekten Datenbank Gründe, die für die Programmierung eines auf heuristischen Spieleinschätzungen beruhenden Mühleprogramms sprechen. Zum einen spielt eine solche Datenbank zwar theoretisch fehlerfrei, aber es ist ihr nicht möglich, Spielsituationen zu erkennen, die nicht perfekte Spieler zu Fehlern verleiten. Zum anderen wird durch die Größe der Datenbank (etwa 8 GB) die Anwendung für einen normalen Nutzer unbrauchbar.
Hauptziel dieser Arbeit ist es nun deswegen, ein Heuristik-basiertes Mühleprogramm zu schreiben, das nicht nur gegen eine solche perfekte Mühledatenbank möglichst wenige Niederlagen erleidet, sondern auch gegen Menschen und andere, fehlerbehaftete Programme möglichst gut abschneidet. Dabei stellen sich im Wesentlichen zwei große Aufgabenbereiche: Zum einen die Programmierung eines effizienten Suchalgorithmus, zum anderen die Gewinnung einer möglichst genauen und schnell berechenbaren Heuristik. Als Suchalgorithmus soll dabei der Minmax-Algorithmus mit verschiedenen Verbesserungen verwendet werden. Den Nutzen dieser Verbesserungen zu testen, soll daher auch ein Ergebniss dieser Arbeit werden. Eine gute Heuristik soll durch maschinelles Lernen in einem Neuronalen Netzwerk erreicht werden. Neben diesen beiden Hauptaspekten soll die Geschwindigkeit und Spielstärke unseres Programms durch die Verwendung einer selbst erstellten Endspieldatenbank und durch eine möglichst effiziente Implementierung des Mühlespiels verbessert werden.
Als Programmiersprache wurde wegen der Portabilität auf verschiedene Betriebsysteme JAVA verwendet, die möglichen Performance-Verluste gegenüber anderen Programmiersprachen wurden dafür in Kauf genommen.

1.3 Ein Mühlespiel Agent im PEAS-Schema

1. Performance.
Das Hauptmaß für die Performance eines Mühlespieler-Agenten ist sicher seine Spielstärke, also bei welchem Anteil seiner Mühlespiele er einen Sieg oder zumindest ein Unentschieden erreicht. Da das Mühlespiel bereits als unentschieden gelöst ist, wäre es auch denkbar, die Quote der Niederlagen gegen einen perfekten Spieler zur Bewertung des Mühleagenten heranzuziehen. Ein sekundäres Maß für einen Mühlespieler-Agenten ist die - für einen Zug oder das gesamte Spiel - verbrauchte Zeit.

2. Environment.
Die Umgebung des Mühle spielenden Agenten ist vollständig beobachtbar. Weiterhin handelt es sich um eine strategische Umgebung, das heißt, die Umgebung ist bis auf die Aktionen des Gegners deterministisch bestimmt. Es handelt sich ferner um eine sequentielle Umgebung, da aufeinanderfolgende Aktionen im starken Maße voneinander abhängen. Sieht man von einer zeitlichen Beschränkung auf ein "vernünftiges"Maß ab, so kann man von einer statischen Umgebung sprechen, bezieht man die Denkzeit als Performancemaß mit ein, so handelt es sich beim Mühlespiel um eine semidynamische Umgebung. Schließlich spielen jeweils zwei Agenten ein Mühlespiel, es handelt sich also formal um eine Multiagentenumgebung.

3. Aktuatoren.
Einziger Aktuator beim Mühlspieler-Agenten ist das Ausführen eines legalen Zuges auf einem (virtuellen) Mühlefeld.

4. Sensoren.
An Sensoren benötigt ein Mühlespieler-Agent lediglich eine Möglichkeit, die Aktionen seines Gegenspielers zu erfahren, oder eine Möglichkeit, die Position auf dem vorliegenden Spielfeld zu erkennen.

Kapitel 2

Minmax Suche

2.1 Minmax

Der in dieser Arbeit verwendete Algorithmus zum Finden eines richtigen Zuges ist der Minmax-Algorithmus. Seine grundlegende Funktionsweise ist folgendermaßen:
Von der aktuellen Spielposition wird der komplette Spielbaum mittels Tiefensuche durchsucht. Der am Zug befindliche Spieler sei der „Max“-Spieler, sein Gegner der „Min“-Spieler. An den Blättern dieses Baumes ist der Spielwert mit dem Ausgang des Spiels identisch, also +1 bei einem Sieg für den ursprünglich aktiven Spieler, -1 bei seiner Niederlage und 0 im Falle eines Unentschiedens. Mit dieser Information kann nun von den Blättern bis zur Wurzel des Spielbaums rekursiv der Spielwert eines Knotens ermittelt werden. Dies geschieht, indem ein Knoten das Maximum der Werte seiner Nachfolger annimmt, falls der Max-Spieler am Zug war, das Minimum, falls der Min-Spieler am Zug war. Die dahinterstehende Idee ist es, dass der aktive Spieler in der Spielsituation dieses Knotens den jeweils für sich günstigsten Nachfolger wählen wird. Sind nun alle Knoten des Spielbaums mit einem solchen Wert belegt, so entspricht der Wert der Wurzel dem aktuellen Spielwert, also dem Spielausgang der Partie bei perfektem Spiel.

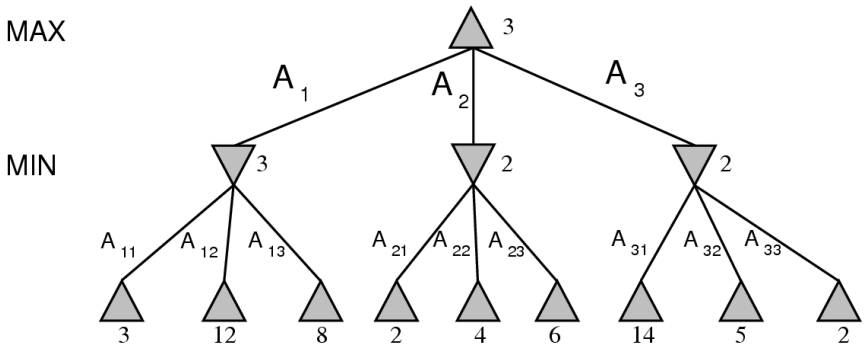


Abbildung 2.1: Minmax

Ein richtiger Zug ist bei diesem Vorgehen ein Zug, der den Spielwert für den aktiven Spieler nicht vermindert, also alle Züge, die zu Nachfolgern der Wurzel des Spielbaums führen, deren Spielwert identisch mit dem der Wurzel ist.
Die Laufzeit-Komplexität des Minmax-Algorithmus liegt bei einer maximalen Baumtiefe m und einem maximalen Verzweigungsfaktor b des Spielbaums in $O(b^m)$. Wegen dieses exponentiellen Wachstums ist es in der Praxis nicht möglich, den Spielbaum bis zu seinen

Blättern zu durchsuchen. Stattdessen wird die Suche maximal bis zu einer bestimmten, festen Tiefe durchsucht und die Knoten dieser Ebene durch eine Heuristische Funktion bewertet. Offensichtlich hängt die Güte des Ergebnisses des Minmax-Algorithmus in der Praxis in entscheidendem Maße von der Güte der Heuristik ab. Beim Mühlespiel konnten wir einen durschnittlichen Verzweigungsfaktor von $b = 13.5$ ermitteln, wobei dieser in der Setzphase im Bereich um $b = 16.5$ liegt und in der Schiebephase ungefähr bei $b = 11.1$.

```
int Minmax(int depth) {
 return Max(depth);
}

int Max(int depth) {
    int best = -INFINITY;
    if (depth <= 0) return Evaluate();

    GenerateLegalMoves();

    while (MovesLeft()) {

        MakeNextMove();
        val = Min(depth - 1);
        UnmakeMove();

        if (val > best) best = val;
    }

    return best;
}

int Min(int depth) {
    int best = INFINITY;
    if (depth <= 0) return Evaluate();

    GenerateLegalMoves();

    while (MovesLeft()) {

        MakeNextMove();
        val = Max(depth - 1);
        UnmakeMove();

        if (val < best) best = val;
    }

    return best;
}
```

2.2 Negamax

Wie man sieht, unterscheiden sich die „Min“-Methode und die „Max“-Methode des Minmax-Algorithmus nur unwesentlich voneinander. Daher werden beide gerne in einer Methode zusammengefasst. Der Rückgabewert dieser Methode, der im Spielbaum nach oben propagiert wird, ist dann der negierte Wert des besten Nachfolge-Zuges für den gerade aktiven

Spieler. Der dadurch enstehende Algorithmus Negamax unterscheidet sich zwar von der Vorgehensweise nicht vom Minmax-Algorithmus, ist jedoch für den Programmierer einfacher zu handhaben, da Änderungen und Verbesserungen im Algorithmus nicht an zwei Stellen im Programmcode (in der „Min“- und in der „Max“-Methode) geändert werden müssen.

```
int Negamax(int depth) {
    int best = -INFINITY;
    if (depth <= 0) return Evaluate();

    GenerateLegalMoves();

    while (MovesLeft()) {

        MakeNextMove();
        val = -Negamax(depth - 1);
        UnmakeMove();

        if (val > best) best = val;
    }

    return best;
}
```

2.3 Alpha-Beta

Beim Minmax-Algorithmus werden ausnahmslos alle Baumblätter ausgewertet, sei es explizit oder mit Hilfe einer Heuristik. Dabei ist bei vielen Blättern diese zeitaufwändige Auswertung unnötig, da sie das Ergbnis ohnehin nicht beeinflussen können.

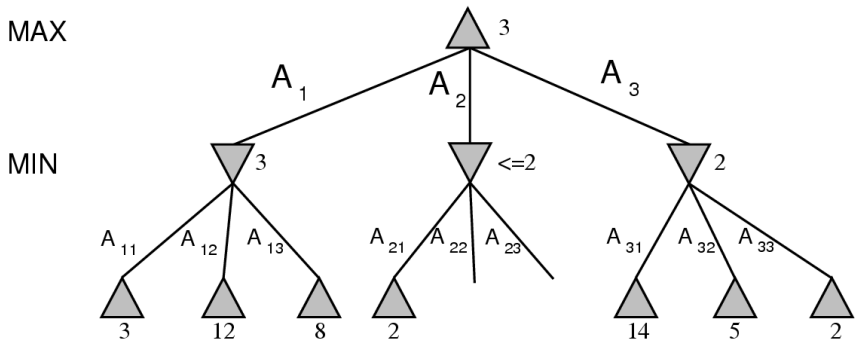


Abbildung 2.2: AlphaBeta

Beim Alpha-Beta-Algorithmus wird nun versucht, solche unnötigen Auswertungen zu vermeiden. Dazu wird der Minmax-Aufruf in jedem Knoten mit einem zusätzlichen Fenster mit zwei Werten *alpha* und *beta* aufgerufen, zwischen denen nach einem exakten Ergebnis gesucht wird. Wird ein Ergebnis außerhalb des Fensters erzielt, so wird die obere (alpha-) bzw. untere (beta-)Schranke zurückgegeben. Seinen Name erhält der Algorithmus von diesen beiden Parametern. Wie auch theoretische leicht nachgewiesen werden kann, unterscheidet sich das Endergebnis dabei nicht von dem des Minmax-Algorithmus, sofern die Schranken folgendermaßen gewählt werden:
Alpha wird auf den höchsten Wert gesetzt, den der Max-Spieler in einem beliebigen Knoten,

in dem er die Entscheidung trifft, auf einem Pfad vom aktuellen Knoten zur Spielbaumwurzel erreicht hat. Beta wird analog dazu auf den kleinsten Wert, den der Min-Spieler in dem Baumknoten, in denen er über den Fortgang entscheidet, erreicht, gesetzt. Fasst man die Min-Methode nach Muster des Negamax wieder zu einer Methode zusammen, so müssen die Schranken beim Aufruf der nächsten Ebene vertauscht und negiert werden. Es ergibt sich folgende Form des Alpha-Beta-Algorithmus:

```
int AlphaBeta(int depth, int alpha, int beta) {
    if (depth == 0) return Evaluate();
    GenerateLegalMoves();

    while (MovesLeft()) {

        MakeNextMove();
        val = -AlphaBeta(depth - 1, -beta, -alpha);
        UnmakeMove();

        if (val >= beta) return beta;
        if (val > alpha) alpha = val;
    }

    return alpha;
}
```

Offensichtlich hängt die Anzahl der durch den Alpha-Beta-Algorithmus abgeschnittenen Blätter und damit auch die Laufzeit des Algorithmus in entscheidender Weise von der Reihenfolge ab, in der die Nachfolger eines Knotens ausgewertet werden. Das Finden einer idealen Sortierung dieser Nachfolger ist ein offenes Problem. Für den Fall einer optimalen Sortierung konnte jedoch nachgewiesen werden, dass die Laufzeit des Minmax-Algorithmus von b^d bei einem Verzweigungsfaktor b und einer durchsuchten Baumtiefe d durch Alpha-Beta-Abschneiden auf $b^{d/2}$ verbessert werden. Für den Fall einer zufälligen Sortierung der Nachfolger reduziert sich die Laufzeit auf $b^{3d/4}$. Eben dieses Ergebniss zeigt auch der Zeittest (Abbildung 2.3) bei unserem Mühlespiel. Der auf $\frac{3}{4}$ reduzierte Exponent des alpha-beta-Algorithmus wird deutlich bei einer logarithmischen Skalierung (Abbildung 2.4).
Für den Alpha-Beta-Algorithmus bieten sich nun vielfältige Möglichkeiten der Optimierung. Prinzipiell bieten sich dabei 3 verschiedene Möglichkeiten an (vgl. [4]):

- Eine Verbesserung der Zugsortierung, die die Zahl der durch den Alpha-Beta-Algorithmus abgeschnittenen Blätter vergrößert
- Eine Minimierung des vom Alpha-Beta-Algorithmus durchsuchten Fensters
- Eine wiederholte Nutzung von gewonnener Information

Auf verschiedene solche Verbesserungsmöglichkeiten soll nun im Folgenden eingegangen werden.

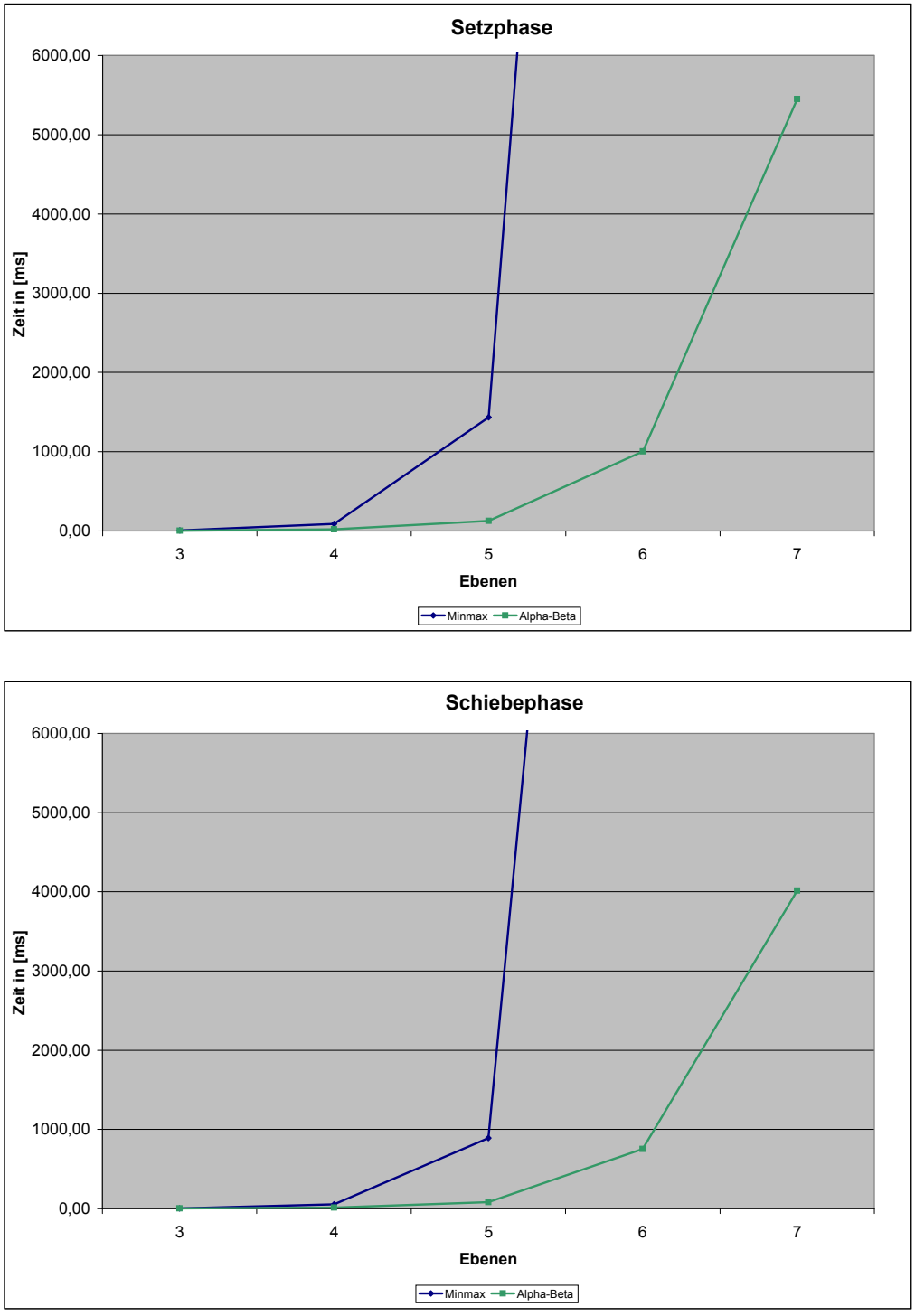


Abbildung 2.3: Minmax vs. Alpha-Beta: Denkzeit pro Zug in ms

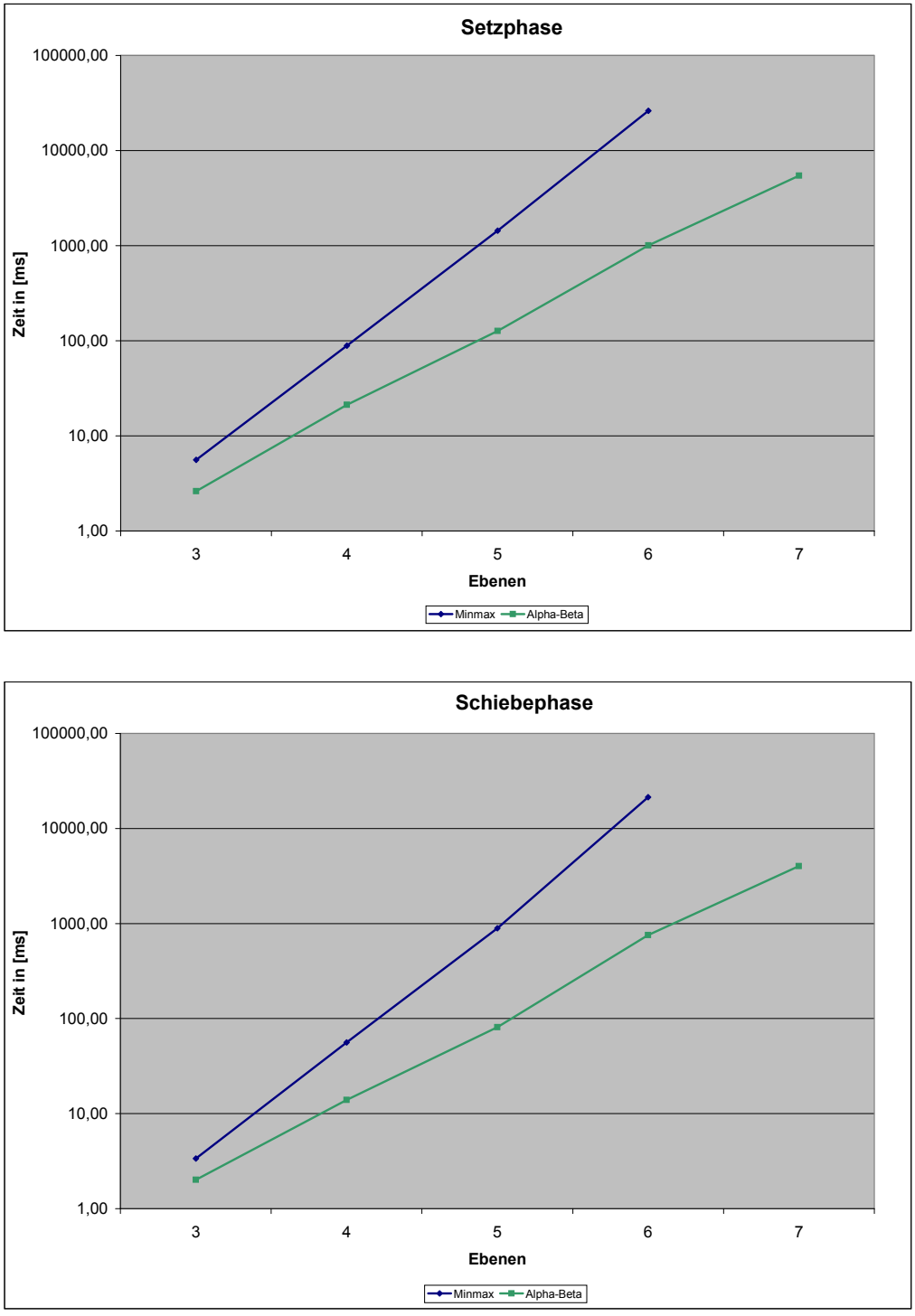


Abbildung 2.4: Minmax vs. Alpha-Beta: Denkzeit pro Zug in ms

2.4 Iterative Deepening

In der Spielpraxis steht einem Mühleagenten für einen Zug nur ein bestimmter Zeitraum zur Verfügung. Auf der anderen Seite schwankt aber die vom Alpha-Beta-Algorithmus bei einer festen Suchtiefe verbrauchte Zeit abhängig von der Spielsituation in hohem Maße. Um dennoch eine möglichst große Suchtiefe zu erreichen, wird das Konzept der iterierten Tiefensuche verwendet, das heißt, es wird wiederholt der Spielbaum mit dem Alpha-Beta-Algorithmus mit einer aufsteigenden Ebenen-Begrenzung durchsucht, also bis zur Tiefe $d = 1, 2, \ldots, n$ bis schließlich eine festgelegte Zeitschranke überschritten wird. Dann wird der nach der letzten vollständigen Suche als am besten eingeschätzte Zug zurückgegeben. Dies mag auf den ersten Blick nach einem immensen, überflüssigen Aufwand klingen, doch wird durch den exponentiell wachsenden Aufwand für die Iterationen $1, 2, \ldots, d-1$ tatsächlich nur ein Bruchteil der Iteration d benötigt. Beim Mühlespiel etwa kann man davon ausgehen, dass für die Iteration mit Tiefe $d-1$ maximal ein Fünftel der Zeit der Iteration mit Tiefe d benötigt wird. Darüber hinaus liefert die Suche mit einer Tiefe von $d-1$ meist eine ausgezeichnete Zugsortierung der Startzüge für die darauffolgende Suche bis Tiefe d. Durch diese Zugsortierung können im Alpha-Beta-Algorithmus tatsächlich oft soviele zusätzliche Knoten abgeschnitten werden, dass dadurch sogar deutlich mehr Zeit eingespart wird als durch die ersten $d-1$ Iterationen verbraucht wurde. Dieses Resultat ergab sich auch beim Zeittest in unserem Mühleprogramm (Abbildung 2.5).
Der einfache Alpha-Beta-Aufruf

```
val = alphaBeta(fixed_depth, -INFINITY, INFINITY);
```

wird beim Iterative Deepening ersetzt durch:

```
for (depth = 1;; depth++) {
    val = alphaBeta(depth, -INFINITY, INFINITY);
    if (TimedOut()) break;
}
```

2.5 Aspiration Window

Eine Möglichkeit, die Laufzeit des Alpha-Beta-Algorithmus zu verbessern, bietet das Konzept des Aspiration Window. Hier wird versucht, durch eine Verkleinerung des Suchfensters des Alpha-Beta-Algorithmus Zeit zu sparen. Diese Methode beruht auf der Idee, dass sich beim iterierten Anwenden des Minmax-Algorithmus der Spielwert von einem zum nächsten Iterationsschritt nur noch geringfügig ändern sollte. Unter dieser Annahme wird im darauf folgenden Iterationsschritt die Alpha-Beta-Schranke auf ein symmetrisches Intervall (Aspiration Window) um die heuristische Einschätzung des Spielwerts (Aspiration Value) in der letzten Iteration gelegt. Somit können beim nächsten Minmax-Durchlauf mehr Pfade abgeschnitten werden. Nun gibt es nach dem Ausgang der nächsten Iteration zwei Möglichkeiten: Die erste Möglichkeit ist, dass man einen Spielwert im Aspiration Window - also zwischen alpha und beta - gefunden hat. Der Alpha-Beta-Algorithmus liefert dann das selbe Ergebnis wie in der Grundversion, allerdings konnte durch die Verkleinerung des Suchfensters Zeit eingespart werden. Die zweite Möglichkeit ist, dass das Ergebnis des Alpha-Beta-Algorithmus gleich einer Intervallgrenze - also gleich alpha oder beta - ist. Dies bedeutet, der eigentliche Spielwert liegt möglicherweise außerhalb des Aspiration Window und wurde auf die Schranke angehoben beziehungsweise abgesenkt. In diesem Fall ist es nötig, den Iterationsschritt mit einem angepassten Wertebereich noch einmal neu durchzuführen. Wurde die obere Schranke zurückgegeben, so liegt der Spielwert oberhalb von beta, es muss also im Intervall [beta; $+\infty$] gesucht werden. Analog muss bei Rückgabe der alpha-Schranke das Intervall [$-\infty$; alpha] durchsucht werden.

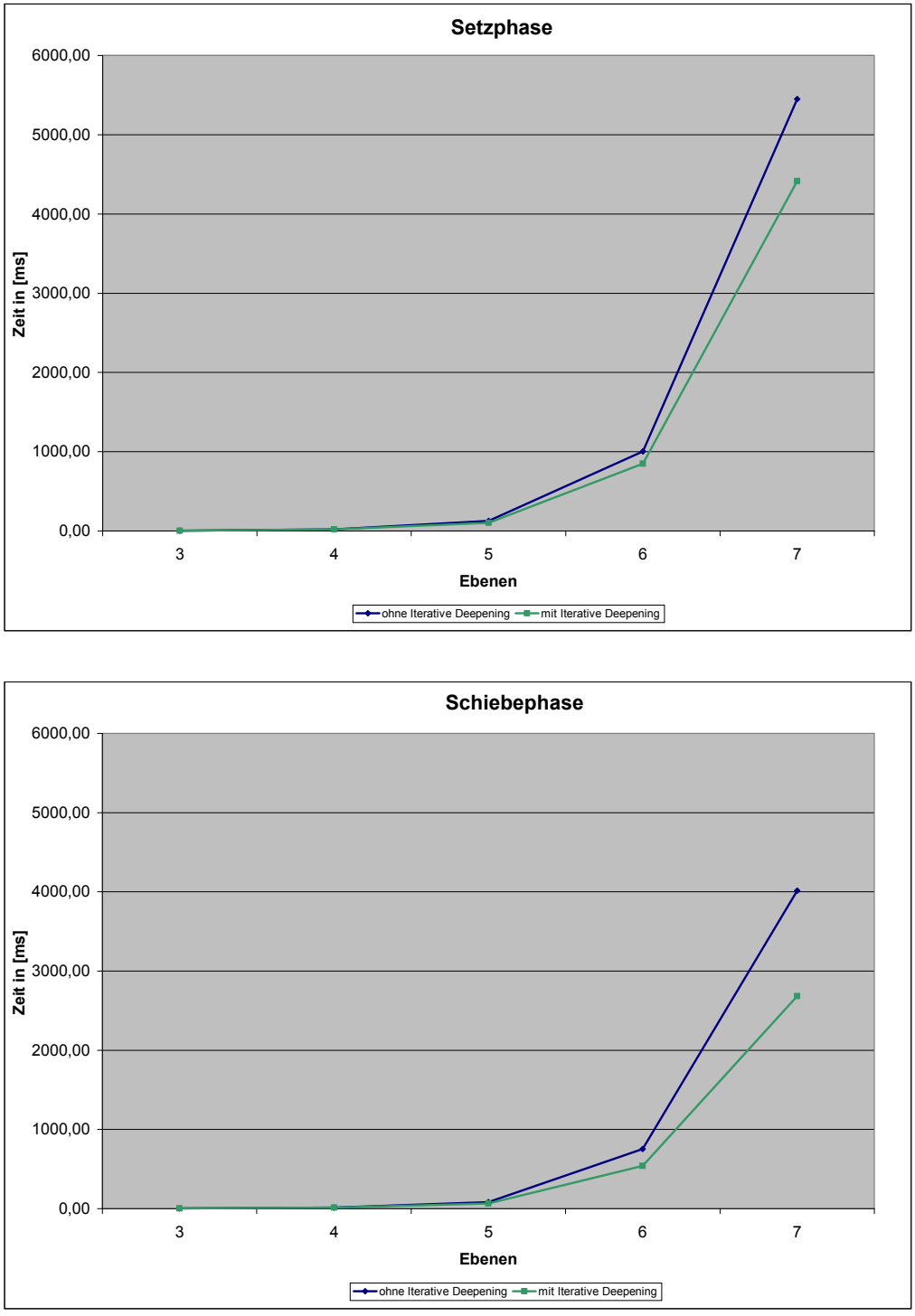


Abbildung 2.5: Iterative Deepening: Denkzeit pro Zug in ms

```
for (depth = 1;; depth++) {
    alpha = getLastBestValue() - ASPIRATION_INTERVAL;
    beta = getLastBestValue() + ASPIRATION_INTERVAL;

    val = AlphaBeta(depth, alpha, beta);
    if (val == alpha) val = AlphaBeta(depth, -INFINITY, alpha);
    if (val == beta) val = AlphaBeta(depth, beta, INFINITY);
    if (TimedOut()) break;
}
```

Bei unseren Tests mussten wir jedoch feststellen, dass Aspiration Window in unserem Fall keine Verbesserung gebracht hat. Wir haben die durchschnittliche Denkzeit pro Zug in der Setz- und in der Schiebephase von drei bis sieben Ebenen einmal ohne und einmal mit Aspiration Window ermittelt. Dabei haben wir drei Intervalle mit Radius 0.05, 0.1 und 0.2 getestet. Wie man in Tabelle 2.2 sieht, sind die Werte der Aspiration Windows in der Schiebephase fast identisch, aber alle dennoch knapp schlechter als ohne Aspiration Window. Auch in der Setzphase (Tabelle 2.1) ist kein Zeitgewinn feststellbar. Bei einem Aspiration Window mit Radius 0.05 ist sogar das Gegenteil der Fall, denn hier ist das Ergebnis erheblich schlechter. Das liegt vor allem daran, dass sich in der Setzphase die Spieleinschätzung bei einer nicht perfekten Heuristik von einer zur nächsten Iteration noch erheblich ändern kann. Zusammenfassend lässt sich feststellen, dass gerade die Zeit, die bei manchen erfolgreichen Aspiration Values gewonnen wird, wieder verbraucht wird, um bei einer erfolglosen Suche eine Iteration neu zu berechnen.

Ebenen	3	4	5	6	7
ohne Aspiration Window	2,23	19,18	103,37	847,66	4415,71
mit Aspiration W. (0.05)	2,66	25,03	122,05	1035,93	5245,60
mit Aspiration W. (0.1)	2,24	18,89	100,73	840,54	4399,55
mit Aspiration W. (0.2)	2,25	18,94	101,02	848,78	4418,31

Tabelle 2.1: Aspiration Window: Denkzeit pro Zug in ms (Setzphase)

Ebenen	3	4	5	6	7
ohne Aspiration Window	1,84	12,26	63,94	538,50	2685,06
mit Aspiration W. (0.05)	1,88	12,40	66,28	548,62	2874,62
mit Aspiration W. (0.1)	1,84	12,32	66,23	557,37	2901,05
mit Aspiration W. (0.2)	1,83	12,34	66,18	557,75	2908,66

Tabelle 2.2: Aspiration Window: Denkzeit pro Zug in ms (Schiebephase)

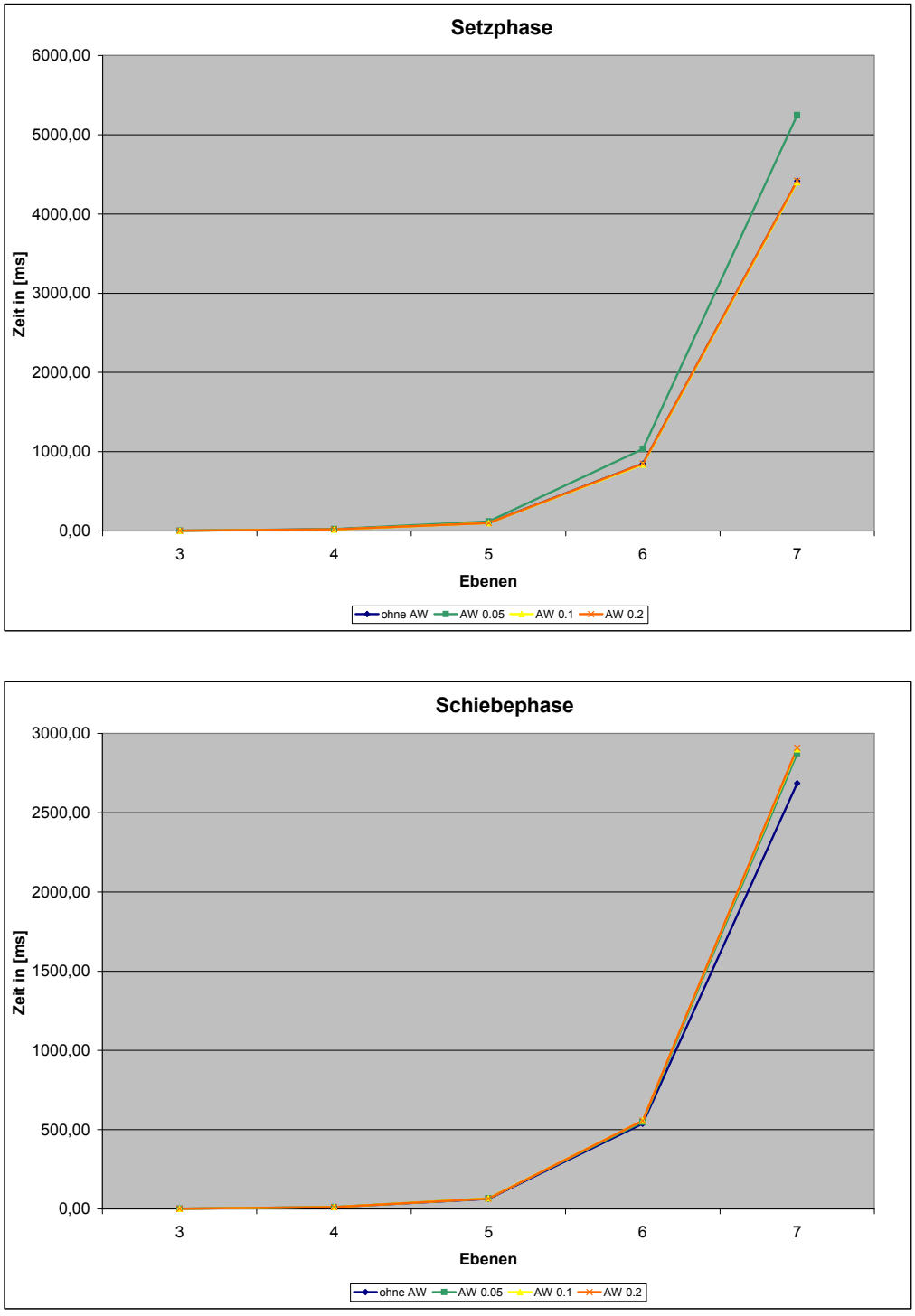


Abbildung 2.6: Aspiration Window: Denkzeit pro Zug in ms

2.6 Minimal Window

Eine weitere Idee zur Beschleunigung der Denkzeit im Minmax-Algorithmus ist die Verwendung des sogenannten Minimal Window. Dieses ist eine Verschärfung des Konzeptes des Aspiration Window. Hier wird nicht nach jeder Iterationsstufe die Alpha- und Betaschranke angepasst, sondern direkt auf oberster Ebene im Minmax-Baum. Beim Minimal Window wird darauf spekuliert, dass der erste Knoten des Baumes auch der beste Nachfolgeknoten für den aktiven Spieler ist. Bei einer Zugsortierung mit Hilfe der iterierten Tiefensuche besteht zu dieser Annahme durchaus Hoffnung. Setzt man nun voraus, dass der unter den bisher durchsuchten Nachfolgern beste auch der insgesamt beste Nachfolger ist, so kann man die Alpha- und Betaschranke für den nächsten Knoten auf ein minimales Intervall um diesen angenommenen besten Wert legen.
Für den berechneten Wert des nächsten Knotens bieten sich nun drei Möglichkeiten: Die erste Möglichkeit besteht darin, dass die Alpha-Beta-Suche ein Ergebnis unterhalb des bisherigen besten Werts liefert, insbesondere die alpha-Schranke. In dieser Situation hat man durch die Begrenzung des Suchbereiches Zeit gespart. Dennoch ist sichergestellt, dass der Wert des durchsuchten Knotens kleiner ist als der bisher beste und daher nicht als insgesamt beste Zugmöglichkeit in Frage kommt. An der Wahl des Minimal Windows für den nächsten Knoten ändert sich in diesem Fall nichts. (Abbildung 2.7 a)) Eine zweite Möglichkeit ist es, dass der Spielwert überhalb des angenommenen besten Wertes, aber noch innerhalb des Suchfenster liegt. In diesem Fall wurde der exakte Wert eines neuen besten Knotens bestimmt. Für die weitere Suche dient er als Basis für das neue Suchfenster. (Abbildung 2.7 b)) Die dritte Möglichkeit besteht darin, dass der Alpha-Beta-Algorithmus als Ergebnis die beta-Schranke zurückgibt. Dies bedeutet, dass der eigentliche Spielwert besser ist als der angenommene beste Spielwert. In diesem Fall muss der Knoten noch einmal mit angepasstem Suchfenster neu berechnet werden, um seinen exakten Wert zu bestimmen. Hier genügt es dann aber, im Intervall [beta; ∞] zu suchen, da ja bereits festgestellt wurde, dass der Wert des Knotens größer oder gleich beta ist. Mit diesem Wert des neuen besten Knotens kann nun für die darauf folgenden Knoten ein neues minimales Suchfenster angelegt werden. (Abbildung 2.7 c))

```
int alphaBeta_MinWindow (int depth, int alpha, int beta) {
    if (depth == 0) return Evaluate();
    GenerateLegalMoves();

    while (MovesLeft()) {
        MakeNextMove();
        val = -alphaBeta_MinWindow(depth - 2, -beta, -alpha);

        // MinimalWindow fehlgeschlagen
        if (val == beta) val = -alphaBeta_MinWindow (depth - 2, -WIN_VALUE, -alpha);

        //alpha neusetzen
        alpha = Math.max(alpha, val - POSITIVE_ZERO);

        //Minimal Window setzen
        beta = alpha + 2 * POSITIVE_ZERO;

        UnmakeMove();

        if (val >= beta) return beta;
        if (val > alpha) alpha = val;
    }
    return alpha;
}
```

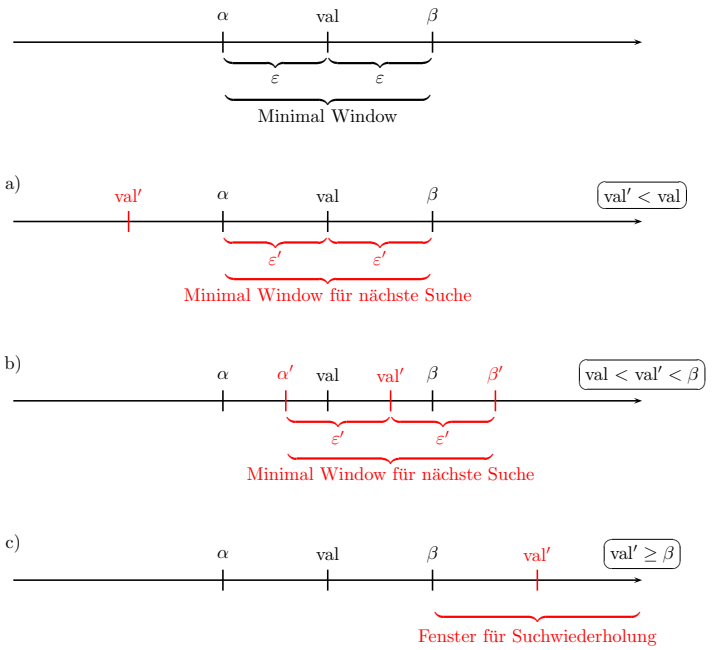


Abbildung 2.7: Minimal Window Skizze

Bei der Verwendung von Minmal Window haben wir bei den Tests einen erfreulichen Zeitgewinn verzeichnen können. In der Setzphase (Tabelle 2.3) liegt dieser Gewinn bei 41,5%. In der Schiebephase (Tabelle 2.4) ist immernoch ein Zeitgewinn von 33,8% zu sehen.

Ebenen	3	4	5	6	7
ohne Minimal Window	2,23	19,18	103,37	847,66	4415,71
mit Minimal Window	2,08	14,02	74,63	513,56	2583,55

Tabelle 2.3: Minimal Window: Denkzeit pro Zug in ms (Setzphase)

Ebenen	3	4	5	6	7
ohne Minimal Window	1,84	12,26	63,94	538,50	2685,06
mit Minimal Window	1,70	9,48	52,44	347,40	1777,79

Tabelle 2.4: Minimal Window: Denkzeit pro Zug in ms (Schiebephase)

2.7 Transposition Table

Beim Durchlaufen des Minmax-Baumes kommt es häufig vor, dass eine Spielfeldkonstellation in mehreren Knoten identisch ist. Das ist in folgendem kleinen Beispiel der Fall: Angenommen Weiß setzt zuerst auf das freie Feld A, dann setzt Schwarz auf Feld B und hierauf Weiß auf Feld C. Das sich ergebende Spielfeld ist identisch mit der Setzfolge $C \rightarrow B \rightarrow A$. Jedoch würde dieses Spielfeld in jedem auftretenden Knoten neu evaluiert bzw. dessen Teilbaum weiter verfolgt werden. Es wäre also praktisch, wenn man bereits evaluierte Spielfelder speichert und dann beim Betreten einer bekannten Spielfeldkonstellation den gespeicherten Wert direkt verwendet. Diese Art des Speicherns von Spielfeldern mit den dazugehörigen

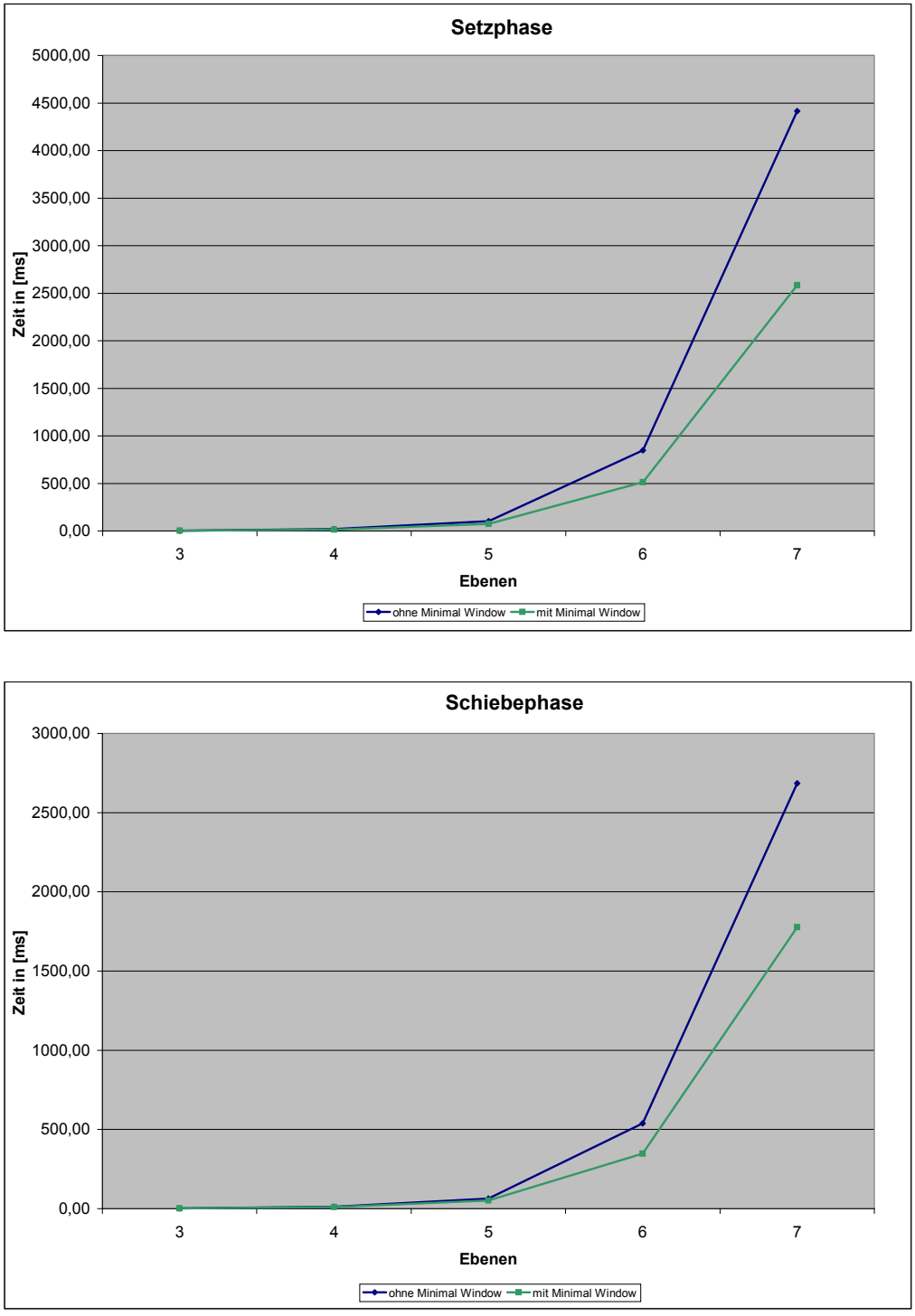


Abbildung 2.8: Minimal Window: Denkzeit pro Zug in ms

Spielwerten ist in der Litartur als Transposition Table bekannt.
Sie wird also während eines Minmax-Aufrufs verwendet, um bereits evaluierte Spielfelder und deren Spieleinschätzung zu speichern. Jedoch ist beim Alpha-Beta-Algorithmus von einem Spielfeld nicht immer die genaue Spieleinschätzung bekannt. So kann es sein, dass nur die Alpha- oder Betaschranke bekannt ist, also der eigentliche Spielwert nur mindestens oder maximal so gut ist wie alpha beziehungsweise beta. Aber auch in diesem Fall ist es ein Gewinn, wenn man für einen Knoten schon weiß, dass er mindestens oder maximal so gut ist wie die gegebene Schranke. In der praktischen Umsetzung muss demnach aber unterschieden werden, ob der Wert für diesen Knoten exakt, kleiner einer Alphaschranke oder größer einer Betaschranke ist. So speichern wir zusätzlich zum Spielwert ein Flag, das diese Information beinhaltet.
Beim Betreten eines neuen Knotens wird erst in der Transposition Table überprüft, ob die Spielsitation bereits gespeichert ist. Ist die Spielfeldkonstellation gespeichert und ist der zugehörige Wert als exakt gekennzeichnet, so kann dieser sofort zurückgegeben werden. Ist nur der Alpha- oder Beta-Wert bekannt, kann immerhin die Alpha- bzw. Beta-Schranke neu gesetzt werden. Wenn die Spielsitation dagegen noch nicht in der Transposition Table vorhanden ist, dann wird der Knoten evaluiert beziehungsweise dessen Teilbaum weiter verfolgt und anschließend darin gespeichert.
Da ein Minmax-Baum trotz des Alpha-Beta-Abschneidens mehr als eine Million Knoten haben kann, ist es wichtig die Spielfelder in einer Struktur zu speichern, die einen schnellen Datenzugriff ermöglicht. Für diese Zwecke bietet sich ein Hash an, der als Keys eindeutige Kennnummern der Spielfelder besitzt und als Wert eine Struktur aus dem Spielwert, der aktuellen Denktiefe und dem Flag, das die Art des Spielwerts angibt.

2.7.1 Zobrist Keys

Wie gerade erklärt, verwenden wir für das Speichern der Transposition Table eine Hash-Datenstruktur. Hierfür wird eigentlich ein eindeutiger Hash-Key benötigt, zu dem eine bijektive Abbildung auf eine Spielsituation möglich ist. Das Mühle-Spielfeld hat 24 Felder, von denen jedes Feld entweder leer, von weiß oder von schwarz besetzt sein kann. Es gibt also maximal 3^{24} mögliche Spielfeldkonstellationen. Das entspricht ungefähr $2,8 \cdot 10^{11}$ Möglichkeiten. In Java sind Hashkeys jedoch nur 32 Bit lang. So könnten nur $2^{32} \approx 4,2 \cdot 10^9$ verschieden Felder gespeichert werden. Eine bijektive Abbildung von den Spielfeldern zu den Hash-Keys ist demnach nicht möglich.

Es muss also zwangsweise mit einer Doppelbelegung eines Keys gerechnet werden. Bei einer geigneten Wahl der Keys lässt sich dieses Risiko jedoch stark reduzieren. Bei naiver Vorgehensweise würde man Keys erzeugen, die sich bei einer kleinen Änderung des Spielfeldes nur geringfügig verändern. Dies führt sofort zu Doppelbelegung. Eine gute Methode ist die Verwendung von Zobrist Keys:
Jedes der 24 Felder wird mit zwei 32-Bit Zufallszahlen initialisiert. Eine Zufallszahl für einen möglichen schwarzen und eine für einen mögliche weißen Stein, der auf dem Feld sitzt. Der Hash-Key eines Spielfeldes wird nun daraus erzeugt, dass die Zufallszahlen aller gesetzen weißen und schwarzen Steine logisch geodert werden. So werden Keys mit großer Distanz selbst bei kleiner Spielfeldänderung erzeugt, die in der Praxis kaum Doppelbelegungen verursachen.

2.7.2 Implementierung

```
int alphaBeta_TransTable(int depth, int alpha, int beta) {

    int hashf = hash_ALPHA;

    if (hashEntry = searchHash (playground.getZobristKey) != null) {
        if (hashEntry.type == hash_EXACT) return hashEntry.value;
        if (hashEntry.type == hash_ALPHA) {
            if (hashEntry.value <= alpha) return alpha;
            if (hashEntry.value < beta) beta = hashEntry.value;
        }
        if (hashEntry.type == hash_BETA) {
            if (hashEntry.value >= beta) return beta;
            if (hashEntry.value > alpha) alpha = hashEntry.value;
        }
    }

    if (depth == 0) {
        val = Evaluate();
        recordHash(depth, val, hash_EXACT);
        return val;
    }

    GenerateLegalMoves();

    while (MovesLeft()) {

        MakeNextMove();
        val = -alphaBeta_TransTable(depth - 1, -beta, -alpha);
        UnmakeMove();

        if (val >= beta) {
            recordHash(depth, beta, hash_BETA);
            return beta;
        }
        if (val > alpha) {
            hashf = hash_EXACT;
            alpha = val;
        }
    }
    recordHash(depth, alpha, hashf);
    return alpha;
}
```

2.7.3 Test und Auswertung

Wie nicht anders erwartet bringt die Verwendung der Transposition Table einen enormen Zeitgewinn. In der Setzphase beträgt dieser 59,6%. In der Schiebephase ist sogar ein Zeitgewinn von 62,5% im Vergleich zur Version ohne Transposition Table zu messen.

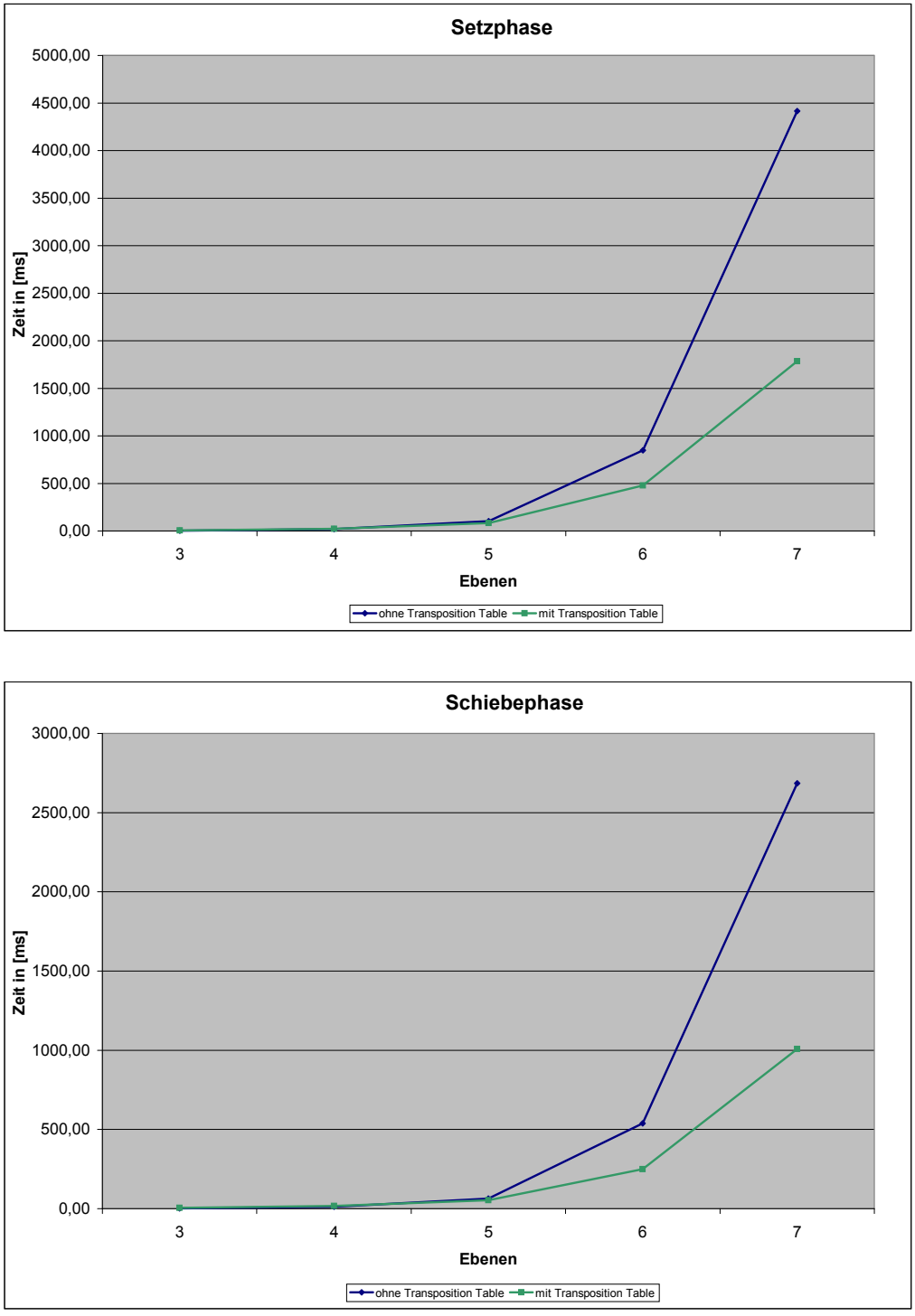


Abbildung 2.9: Transposition Table: Denkzeit pro Zug in ms

Ebenen	3	4	5	6	7
ohne Transposition Table	2,23	19,18	103,37	847,66	4415,71
mit Transposition Table	6,29	24,27	83,24	477,51	1784,93

Tabelle 2.5: Transposition Table: Denkzeit pro Zug in ms (Setzphase)

Ebenen	3	4	5	6	7
ohne Transposition Table	1,84	12,26	63,94	538,50	2685,06
mit Transposition Table	5,77	17,97	52,49	250,09	1007,51

Tabelle 2.6: Transposition Table: Denkzeit pro Zug in ms (Schiebephase)

2.8 Mühlespezifische Zugsortierung

Wie bereits bei der Beschreibung des Alpha-Beta-Algorithmus erwähnt wurde, wird für einen effizienten Einsatz des Algortihmus eine möglichst präzise Vorsortierung aller Nachfolger eines Knotens benötigt. Eine sehr gute Zugsortierung für die Züge auf der obersten Ebene des Spielbaums wird dabei von der bereits beschriebenen iterierten Tiefensuche geliefert. Für die Sortierung der Knoten auf tieferen Ebenen mittels der während der iterativen Tiefensuche gewonnen Spieleinschätzungen wäre eine sehr aufwendige Speicherung aller Knoten nötig. Dieser Aufwand stellte sich jedoch in der Praxis als viel zu groß heraus.
Ein zweiter Ansatz war es, bis zu einer bestimmten Tiefe im Spielbaum auf alle Nachfolger einer Situation direkt die benutzte Heuristik anzuwenden und die möglichen Züge anhand dieser Einschätzung zu sortieren. Auch bei dieser Vorgehensweise wurde jedoch mehr Zeit für die Sortierung benötigt, als im Alpha-Beta-Algorithmus eingespart werden konnte.
Erfolgreich war dagegen der folgende Ansatz. Im Allgemeinen kann man davon ausgehen, dass das Schließen einer Mühle oft zu einer besseren Situation führt, als eine Mühle nicht zu schließen, wenngleich dies auch bei weitem nicht immer zutrifft. Unter dieser Annahme kann man eine Sortierung konstruieren, die zuerst alle mühleschließenden Züge durchgeht und hierauf alle, die keine Mühle schließen. Da das Erkennen einer Mühle ohnehin für den Aufbau des weiteren Spielbaums benötigt wird und nicht jeden Folgeknoten heuristisch bewerten muss, ist eine solche Sortierung in jedem inneren Knoten des Spielbaums sehr schnell zu berechnen. Dass diese Sortierung auch tatsächlich eine sinnvolle Zugsortierung ergibt, wird in den untenstehenden Testergebnissen (Tabellen 2.7, 2.8 und in Abbildung 2.10) deutlich. Somit führte diese Art der Zugsortierung, die in der Version 2.43 unseres Mühleprogrammes erstmals verwendet wurde, nochmals zu einer erheblichen Performance-Steigerung des Programmes.

Ebenen	3	4	5	6	7
ohne Zugsortierung	2,23	19,18	103,37	847,66	4415,71
mit Zugsortierung	2,22	13,84	60,79	389,85	1653,81

Tabelle 2.7: Zugsortierung nach Mühlen: Denkzeit pro Zug in ms (Setzphase)

Ebenen	3	4	5	6	7
ohne Zugsortierung	1,84	12,26	63,94	538,50	2685,06
mit Zugsortierung	1,84	8,95	34,50	215,91	882,76

Tabelle 2.8: Zugsortierung nach Mühlen: Denkzeit pro Zug in ms (Schiebephase)

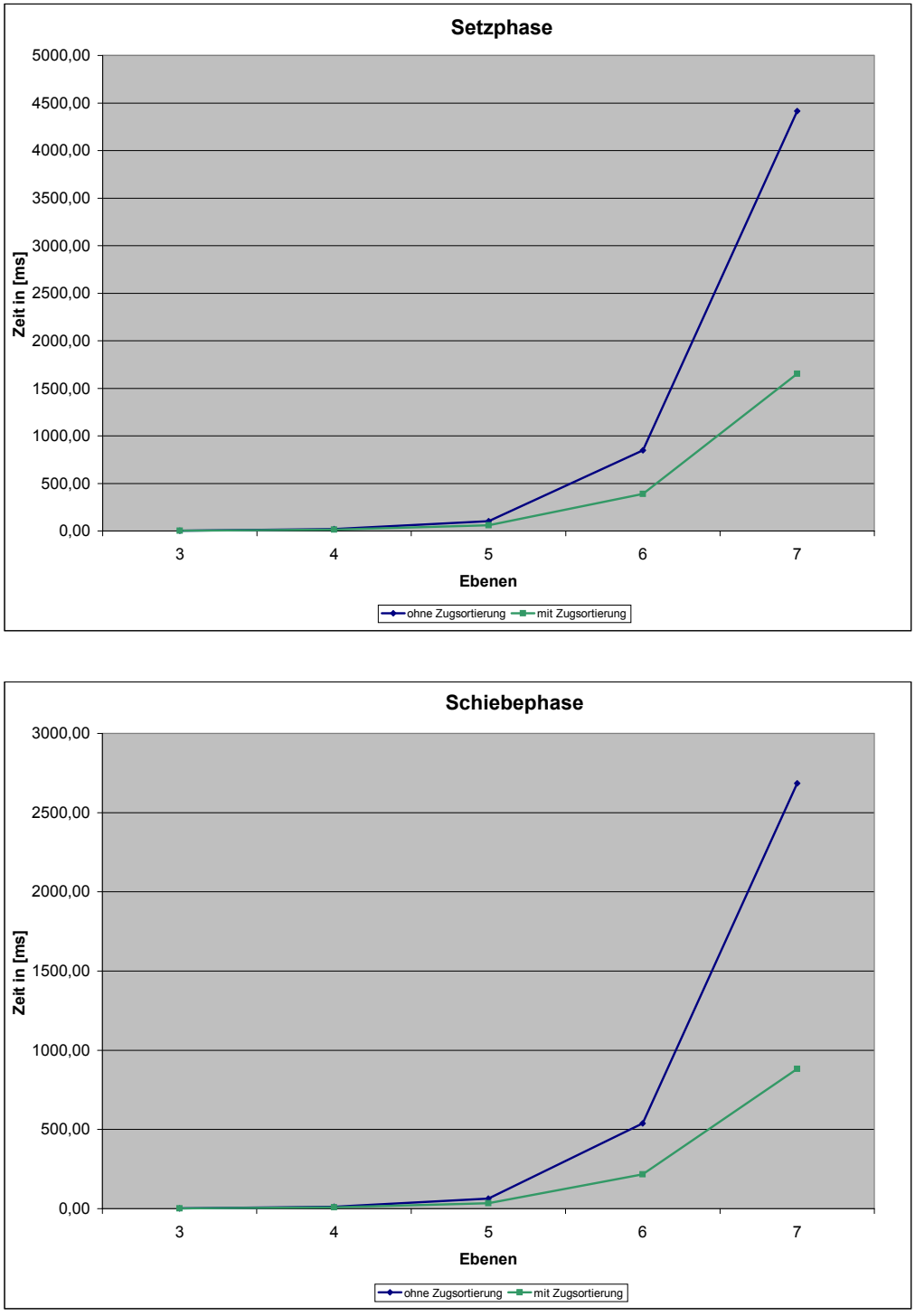


Abbildung 2.10: Zugsortierung nach Mühlen: Denkzeit pro Zug in ms

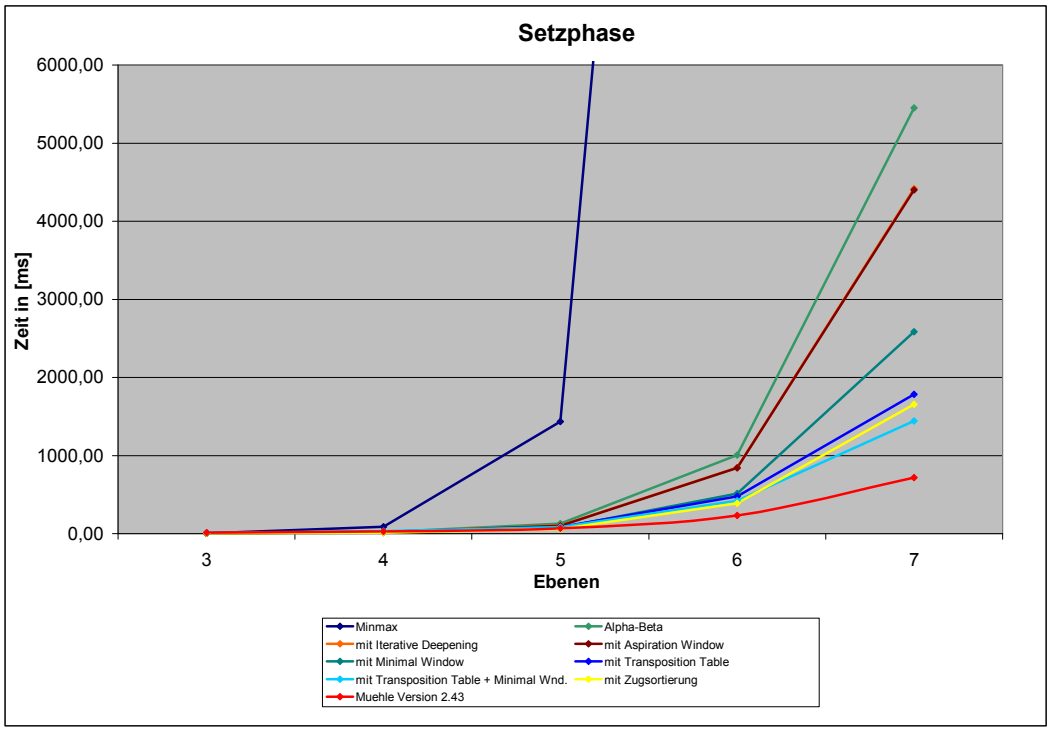
2.9 Auswertung

Zeittests

Die verschiedenenen oben vorgestellten Verbesserungen des Minmax-Algorithmus, wurden von uns zunächst einem reinen Zeittest unterzogen. Dazu generierten wir für das Setzen und das Schieben jeweils 2000 zufällige Spielsituationen. Für jede dieser Stellungen sollte dann der Minmax-Algorithmus mit den entsprechenden Verbesserungen den heuristisch besten Folgezug ermitteln. Dabei wurde jeweils eine feste Denktiefe vorgegeben und die im Algorithmus benötigte Zeit gemessen. Als Testkonfiguration wurde die weiter unten beschriebene Heuristik „Standard“ auf dem Referenzrechner verwendet. Teilergebnisse aus diesen Tests wurden bereits in den Beschreibungen der jeweilgen Verbesserung vorweggenommen. In den Tabellen 2.9 und 2.10 und in Abbildung 2.11 sind die Ergebnisse zusammengefasst und illustriert. Bei den in den Tabellen unterhalb der Trennlinie aufgeführten Tests wurde jeweils der alpha-beta-Algorithmus mit iterative Deepening und den angegebenen zusätzlichen Modifikationen verwendet. In der Mühleversion 2.43 sind als zusätzliche Verbesserungen Minimal-Window, Transposition-Table und die Zugsortierung nach Mühlen enthalten. Insgesamt lassen sich dabei folgende Ergebnisse festhalten:
Die Tests in Setz- und Schiebephase ergaben im Wesentlichen die gleichen Resultate. Der Einsatz von Zugsortierung, Minimal-Window und Transposition-Table allein brachte jeweils einen enormen Zeitgewinn. Lediglich das Aspiration-Window brachte keinen Nutzen. Durch eine Kombination von mehreren gewinnbringenden Modifikationen konnten in der Summe weiterhin deutliche Fortschritte erzielt werden, allerdings sank der relative Zeitgewinn der einzelnen Verbesserungen leicht. Beispielsweise konnte in der Setzphase durch den Einsatz der Zugsortierung bei sieben Ebenen gegenüber dem Alpha-Beta-Algorithmus mit iterative Deepening ein Zeitgewinn von etwa $62,5\%$ erzielt werden, bei zusätzlichem Einsatz von Minimal-Window und Transpostion-Table, wie in Mühle 2.43 der Fall, sank der Zeitvorteil durch die Zugsortierung auf ungefähr 50%. Dennoch stellte sich Mühle 2.43, die alle bisher beschriebenen Verbesserungen bis auf das Aspiration-Window verwendet, als die bei weitem schnellste Programmversion heraus.

Ebenen	3	4	5	6	7
Minmax	5,59	88,89	1434,87	26233,81	
mit Alpha-Beta	2,62	21,21	126,80	1004,47	5450,12
mit Iterative Deepening	2,23	19,18	103,37	847,66	4415,71
mit Aspiration Window	2,24	18,89	100,73	840,54	4399,55
mit Minimal Window	2,08	14,02	74,63	513,56	2583,55
mit Transposition Table	6,29	24,27	83,24	477,51	1784,93
mit Transposition Table + Minimal Wnd.	6,92	30,17	83,28	421,03	1441,72
mit Zugsortierung nach Müehlen	2,22	13,84	60,79	389,85	1653,81
Muehle Version 2.43	12,14	27,94	68,87	233,12	715,82

Tabelle 2.9: Denkzeit pro Zug in ms (Setzphase)



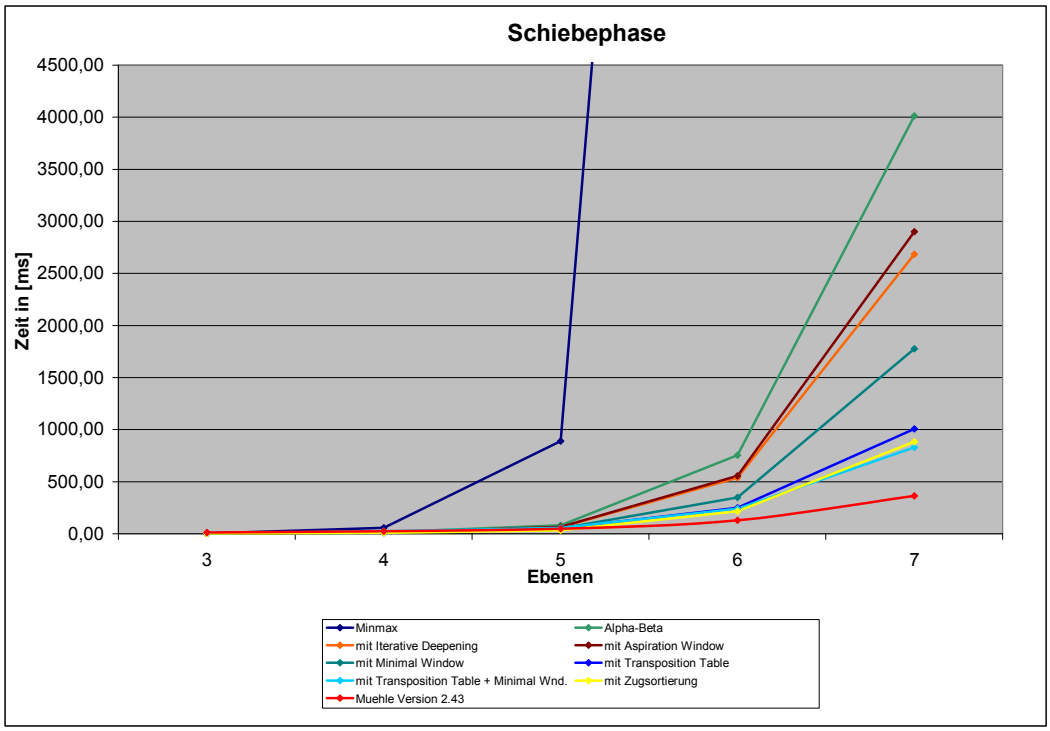


Abbildung 2.11: Überblick aller Minmax-Verbesserungen: Denkzeit pro Zug in ms

Ebenen	3	4	5	6	7
Minmax	3,36	56,23	889,63	21311,72	
mit Alpha-Beta	2,01	13,96	81,33	753,63	4013,98
mit Iterative Deepening	1,84	12,26	63,94	538,50	2685,06
mit Aspiration Window	1,84	12,32	66,23	557,37	2901,05
mit Minimal Window	1,70	9,48	52,44	347,40	1777,79
mit Transposition Table	5,77	17,97	52,49	250,09	1007,51
mit Transposition Table + Minimal Wnd.	5,83	22,63	56,98	239,20	829,27
mit Zugsortierung nach Müehlen	1,84	8,95	34,50	215,91	882,76
Muehle Version 2.43	11,28	22,63	47,94	130,15	363,58

Tabelle 2.10: Denkzeit pro Zug in ms (Schiebephase)

Praxistests

Nachdem diese verschiedenen Modifikationen auf ihre Verbesserung der Denkzeit hin getestet wurden, sollten nun in einem weiteren Test die Auswirkungen auf die Spielstärke in der Praxis getestet werden. Dazu wurden fünf Programmversionen ausgewählt, die in einem Turnier gegeneinander antraten. Darin spielte jede Version 1000 Partien mit t=5000 ms Bedenkzeit pro Zug auf dem Referenzrechner mit der „Standard“-Heuristik gegen jeden anderen Turnierteilnehmer, insgesamt also 4000 Partien. Die Ergebnisse dieser Partien sind in der Tabelle 2.11 dargestellt und in Abbildung 2.12 illustriert. In der Tabelle sind in der jeweiligen Zeile die Gewinne einer Programmversion gegen die in der Spalte obenstehende Version eingetragen. Umgekehrt stehen in jeder Spalte die Niederlagen gegen den entsprechenden Gegner, bei den auf 1000 fehlenden Spielen handelt es sich um Remis.
Beim Blick auf die Ergebnisse fällt zunächst auf, dass die Spielstärke der einzelnen Versionen stark variiert und sich tatsächlich proportional zur Geschwindigkeit, die in den vorigen Zeittests ermittelt wurde, verhält. Je schneller eine Suche also ist, desto besser spielt das entsprechende Programm. Dabei macht sich offenbar auch eine geringe Vergrößerung der durchschnittlichen Denktiefe - oft um weniger als eine Ebene pro Zug - bei selber Heuristik durchaus deutlich in der Spielstärke bemerkbar.

	Minmax	Alpha-Beta	Minimal-Window	Transposition-Table	Mühle Version 2.43	Gewinne
Minmax	-	83	72	58	38	251
Alpha-Beta	481	-	226	260	77	1044
Minimal-Window	527	353	-	318	115	1313
Transposition-Table	602	408	338	-	85	1433
Mühle Version 2.43	710	669	522	582	-	2483
Verluste	2320	1513	1158	1218	315	6524

Tabelle 2.11: Minmax-Verbesserungen in der Praxis

2.10 Quiescence Search

Wie wir bereits gesehen haben, ist es im Allgemeinen nicht möglich, den Spielbaum komplett auszuwerten. Deshalb beschränkt man die Suche auf eine festgelegte Suchtiefe, auf der dann eine heuristische Bewertungsfunktion der Spielsituation angewendet wird. Dabei muss aber stets in Betracht gezogen werden, dass im nächsten, nicht mehr betrachteten Spielzug eine gravierende Änderung der Spielsituation und damit der heuristischen Bewertung eintreten

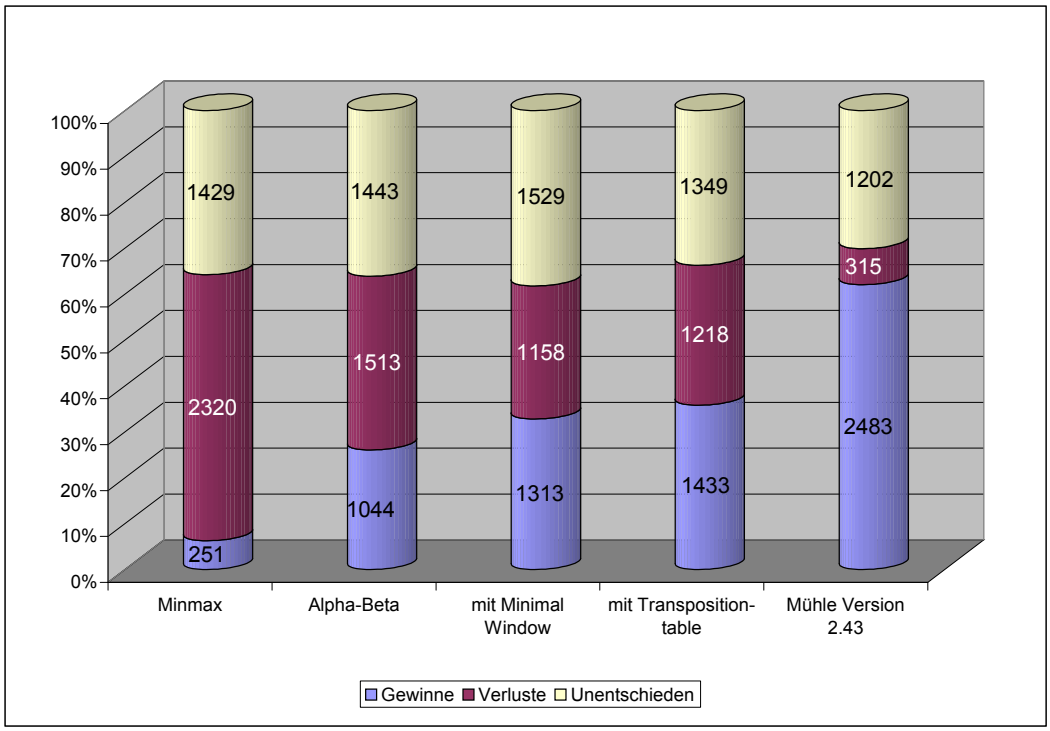


Abbildung 2.12: Minmax-Verbesserungen in der Praxis

kann. Dieses Problem ist als *Horizont-Problem (horizon problem)* bekannt und ist generell nicht vollständig zu vermeiden. Ein Ansatz, dieser Problematik entgegenzuwirken, ist die sogenannte *Ruhesuche (quiescence search)*. Der Grundgedanke der Ruhesuche ist es, vor der heuristischen Auswertung von Blättern des Suchbaumes „unruhige“ Spielsituationen zu erkennen und diese weiter zu verfolgen. Als unruhig werden dabei Situationen bezeichnet, wenn in den nächsten Zügen eine deutliche Schwankung der Spieleinschätzung zu erwarten ist. Ein naheliegender Ansatz beim Mühlspiel ist, Spielsituationen mit einer offenen Mühle als unruhig zu bezeichnen. Für die Weiterverfolgung von unruhigen Spielsituationen gibt es verschiedene Möglichkeiten:

- Die einfachste Möglichkeit, die Spieleinschätzung eines unruhigen Knotens zu präzisieren, ist eine vollständige Expansion des Knotens. Es werden also von einer Spielsituation alle Zugmöglichkeiten betrachtet, auf dem Spielfeld ausgeführt und die entstandenen Situationen erneut zur Bewertung herangezogen. Eine solche Vorgehensweise ist sehr zeitaufwendig, da viele neue Knoten untersucht werden müssen. Unter diesen befinden sich wiederum zahlreiche unruhige Stellungen, da die Unruhe bei vielen durchgeführten Aktionen selbst nicht beseitigt wird. Daher ist diese Möglichkeit nur dann praktikabel, wenn jede Zugmöglichkeit innerhalb kurzer Zeit zu einer ruhigen Spielsituation führt, etwa beim Schachspiel ein Schachgebot. Beim Mühlespiel, mit Betrachten offener Mühlen als unruhige Situation, kann in der Regel keine vollständig ruhige Situation erreicht werden, da eine offene Mühle ja auch nicht unbedingt geschlossen werden muss.

- Eine zweite Möglichkeit ist es, nur die Pfade weiter zu verfolgen, die nach einem vorgegebenem Muster als beste Pfade vermutet werden und bei denen mit einer Beruhigung der Situation zu rechnen ist. Hier kommt zum Beispiel das Schließen einer offenen Mühle in Frage. Als Bewertung des Knotens wird anschließend das Maximum der betrachteten Pfade verwendet. Diese Methode ist offensichtlich einerseits weit we-

niger aufwändig als die erste Methode. Auf der anderen Seite wird nur ein Teil der möglichen Züge betrachtet, bessere Züge werden unter Umständen ignoriert. Beim Mühlespiel gibt es beispielsweise viele Situationen, bei denen das Schließen der Mühle eben nicht die beste Spielmöglichkeit ist. Dies wird im folgenden Beispiel (Abbildung 2.13) illustriert.

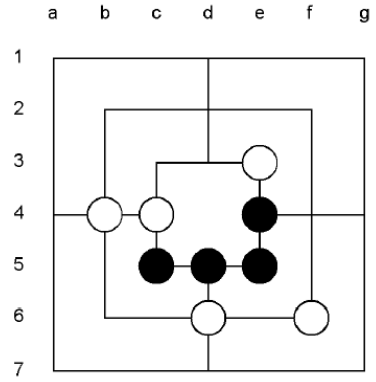


Abbildung 2.13: Mühle-Schließen als nicht-optimales Spiel

Das Schließen der Mühle (b4-b6) und das zwangsläufige Schmeißen des Steins e4 führt zu einem Unentschieden. Der Zug f6-f4 dagegen sperrt Schwarz ein und führt zu einem sofortigen Sieg für Weiß. Die Spielbewertung für den aktiven Spieler ist also entweder richtig oder wie in diesem Beispiel schlechter als bei einer vollständigen Expansion.

- Auch bei der dritten Möglichkeit werden wiederum nur die Pfade weiter verfolgt, die nach einem vorgegebenem Muster als beste Pfade vermutet werden und bei denen mit einer Beruhigung der Situation zu rechnen ist. Als Bewertung des expandierten Knotens aber wird das Maximum der betrachteten Pfadergebnisse und der Einschätzung des expandierten Knotens selbst verwendet. Die Grundannahme dabei ist, dass der am Zug befindliche Spieler die Möglichkeit besitzt, sich bei optimalem Spiel nicht in der heuristischen Einschätzung zu verschlechtern. Alternativ dazu kann man auch eine geringfügige Verschlechterung der Spieleinschätzung zulassen.

Beim Mühlespiel ist es naheliegend, Spielsituationen mit offenen Mühlen als unruhig anzusehen. Allerdings hängt die „Ruhe“ eines Spielfeldes ganz entscheidend von der verwendeten Heuristik ab. Wenn in der Heuristik bereits offene Mühlen als Feature berücksichtigt werden, und wenn der Nutzen einer offenen Mühle über alle Spiele gesehen relativ konstant ist, dann kann eine solche Situation eigentlich nicht als unruhig bezeichnet werden, denn das Schließen der Mühle wird wahrscheinlich nicht zu einer überdurchschnittlich großen Veränderung der Spieleinschätzung führen, wenn der Nutzen der offenen Mühle in der Heuristik richtig beurteilt wird. Genau auf eine solche Konstellation scheinen unsere Testergebnisse hinzudeuten.

Bei 1000 Testspielen einer Heuristik, die offene Mühlen nicht mit in die Einschätzung einbezieht, mit einer solchen Ruhesuche gegen die selbe Heuristik ohne Ruhesuche ergab sich folgendes Testergebnis (2.12):
Wie man sieht, bringt die Verwendung der Ruhesuche in jedem Fall einen deutlichen Vorteil. Allerdings war die erreichte Spielstärke immer noch deutlich unter der eines Spielers, der eine Heuristik mit Bewertung offener Mühlen verwendet.

Bei 1000 Testspielen einer Heuristik, die offene Mühlen mit in die Einschätzung einbezieht, mit einer solchen Ruhesuche gegen die selbe Heuristik ohne Ruhesuche ergab sich folgendes Testergebnis (Tabelle 2.13):

Spieler mit Ruhesuche	Gewonnen	Verloren	Unentschieden
Bei fester Suchtiefe: 6 Ebenen	521	178	301
Bei fester Denkzeit: 700 ms	347	219	434

Tabelle 2.12: Spielausgang mit Heuristik, der keine offenen Mühlen bewertet

Spieler mit Ruhesuche	Gewonnen	Verloren	Unentschieden
Bei fester Suchtiefe: 6 Ebenen	294	361	345
Bei fester Denkzeit: 700 ms	137	539	324

Tabelle 2.13: Spielausgang mit Heuristik, der offene Mühlen explizit bewertet

Wie man deutlich erkennen kann, bringt die Ruhesuche selbst bei fester Grundsuchtiefe hier keinen Nutzen. Obwohl die für die Ruhesuche benötigte Denkzeit keine Rolle spielt, ergibt sich letztlich eine schlechtere Spieleinschätzung als ohne Ruhsuche. Wie bereits erwähnt, ist dies wohl darauf zurückzuführen, dass das Schließen einer Mühle nicht immer den besten Spielzug darstellt. Wird wie in der Praxis eine feste Denkzeit vorausgesetzt und damit die Denkzeit für den gewöhnlichen Alpha-Beta-Algorithmus durch den zusätzlichen Zeitaufwand für die Ruhesuche vermindert, verwundert es nicht, dass dies zu einer deutlichen Niederlage des Spielers mit Ruhesuche führt.
Um zu testen, ob durch eine geeignete Ruhesuche auch Spieler mit Bewertung von offenen Mühlen noch verbessert werden können, wurde von uns folgender Ansatz gewählt. Anstatt Spielsituationen mit offenen Mühlen als unruhig zu bezeichnen, wurden nun solche Spielsituationen als unruhig definiert, bei denen in der Setzphase beim Setzen eines Steins gleich zwei geöffnete Mühlen entstehen (vgl. Abbildung 2.14). In der Schiebephase wurde das Spiel als ruhig angesehen.

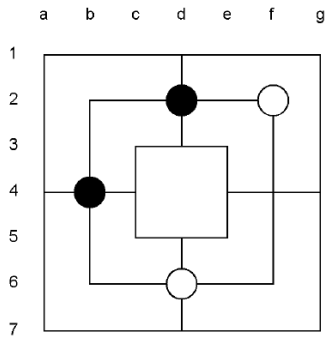


Abbildung 2.14: Weiß kann mit Setzen auf f6 zwei Mühlen öffnen

Auch hier haben wir 1000 Testspiele zwischen einem Spieler mit dieser Ruhesuche und einem Spieler ohne sie durchgeführt, bei gleicher zu Grunde liegender Heuristik. Hier ergibt sich

Spieler mit Ruhesuche	Gewonnen	Verloren	Unentschieden
Bei fester Suchtiefe: 6 Ebenen	372	250	378
Bei fester Denkzeit: 700 ms	212	285	503

Tabelle 2.14: Spielausgang bei alternativer Ruhesuche

nun folgendes interessantes Bild. Wie im Versuch mit fester Grunddenktiefe zu erkennen ist, kann die Verwendung dieser alternativen Ruhesuche durchaus zu einer Verbesserung der Spieleinschätzung führen. Dies ist allerdings nur dann der Fall, wenn die zur Ruhesuche

benötigte Zeit nicht von der allgemeinen Denkzeit abgeht. Wird dagegen wie in der Praxis eine feste Denkzeit vorgegeben, dann hat der Aufwand für die Ruhesuche eine Verminderung der Denktiefe zur Folge. Dieser Verlust wirkt sich wie in Tabelle 2.14 ersichtlich leicht stärker aus als der Gewinn, den die Ruhesuche mit sich bringt.
Zusammenfassend lässt sich sagen, dass beim Mühlespiel bei Verwendung einer geeigneten Heuristik durch die Ruhesuche keine signifikante Verbesserung erreicht werden konnte. Dies überrascht, da bei einigen anderen Spielen, allen voran das Schachspiel, mit der Ruhesuche deutlich Fortschritte erzielt werden konnten. Zu erklären ist dies dadurch, dass das Mühlespiel vegleichsweise „gleichmäßig“ ruhig ist. Auch das Schließen einer offenen Mühle führt im Normalfall nicht zu einer so dramatisch veränderten Spielsituation wie etwa beim Schachspiel das Schlagen einer Dame oder auch ein Schachgebot. Daher ist das Auflösen solcher Situationen zum einen weniger nötig und zum anderen auch schwieriger, da das Schließen einer Mühle - wiederum im Gegensatz zum Schlagen einer Dame im Schach - oftmals keine gute Fortsetzung des Spiels findet. Abschließend zu diesem Kapitel ist festzustellen, dass der Nutzen der Ruhesuche natürlich in starkem Maße von der verwendeten Heuristik abhängt.

```
int quiescentSearch (int depth, int alpha, int beta) {

    if (depth < quiescentSearchLimit) return Evaluate();
    if (depth <= 0)
       if (isQuiet) return Evaluate();

       GenerateLegalMoves();

       while (MovesLeft()) {

       if (depth <= 0) & (not nextMove.closesMuehle) continue;
           MakeNextMove();
           val = -AlphaBeta(depth - 1, -beta, -alpha);
           UnmakeMove();

           if (val >= beta) return beta;
           if (val > alpha) alpha = val;
        }

    return alpha;
}
```

Kapitel 3

Endspieldatenbank

3.1 Einführung

Bei modernen Programmen für Brettspiele ist die Verwendung einer Endspieldatenbank üblich. Eine solche Datenbank ermöglicht es, am Ende des Spieles perfekte Spielzüge sehr schnell zu finden. Gerade in der Schlussphase des Mühlespiels, in der es zwar nur verhältnismäßig wenige zu speichernde Spielsituationen gibt, in denen die Spieler aber durch die Möglichkeit des Springens enorm viele Zugmöglichkeiten besitzen und damit der Verzweigungsfaktor des Spielbaums sehr groß wird, lohnt der Einsatz einer Datenbank besonders. Eine Ausdehnung der Endspieldatenbank auf das komplette Spiel ist beim Mühlespiel möglich und wurde erstmals von R. Gasser Anfang der 90er Jahre durchgeführt. Allerdings benötigt eine solche vollständige Mühledatenbank mehrere Gigabyte Speicher, so dass die Anwendung für den privaten Nutzer nicht geeignet ist. Darüber hinaus ist es gerade Ziel dieser Studienarbeit, ein Mühlespiel auf der Basis heuristischer Einschätzungen zu erstellen und eben nicht auf solche speicheraufwendigen kompletten Datenbanken zurückzugreifen. Für unser Mühleprogramm wurde daher eine Endspieldatenbank erstellt, die wirklich nur die Endphase des Spiels abdeckt. In der Datenbank sind alle Spielsituationen mit bis zu 8 Spielsteinen auf dem Spielbrett gespeichert.

3.2 Nutzung von Spiegelungen

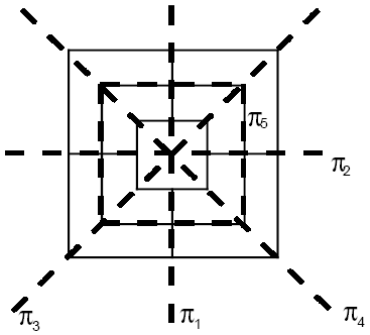


Abbildung 3.1: Spiegelachsen des Mühlefeldes

Auf dem Mühlefeld existieren fünf mögliche Spiegelachsen, von denen sich eine durch eine Kombination der anderen Spiegelachsen ergibt und daher redundant ist. Da der Spielwert aller durch solche Spiegelungen entstehenden Spielsituationen identisch ist, genügt es, in einer

Endspieldatenbank nur den Spielwert eines Vertreters festzuhalten. In unserer Implementierung werden dazu auf Bitebene alle durch Spiegelungen entstehenden Spielfelder berechnet und das Spielfeld, das den niedrigsten perfekten Hashcode aufweist, eingespeichert. Die Berechnung des verwendeten perfekten Hashcodes wird weiter unten beschrieben.

3.3 Retrograde Analysis

Zum Erstellen der Endspieldatenbank wurde die auch von R. Gasser genutzte Methode der „Retrograde Analysis“ verwendet. Dabei wird folgendermaßen vorgegangen: Grundsätzlich werden bei uns ausschließlich Spielfelder betrachtet, bei denen der Spieler mit den weißen Steinen am Zug ist. Das Verwenden der Datenbank durch den schwarzen Spieler erfolgt durch Invertierung des Spielfeldes. Die verschiedenen zu betrachtenden Spielsituation werden nach Steinanzahl der beiden Spieler gegliedert gespeichert. Mögliche Kombinationen mit bis zu 8 Steinen sind also: 5-3 Steine, 4-4 Steine, 4-3 Steine und 3-3 Steine. Da Lösungen von Situationen mit höherer Steinanzahl auf die Lösungen von Situationen mit kleinerer Steinanzahl zurückgreifen, werden die Spielsituationen von hinten nach vorne gelöst. Beispielsweise wird das Endergebnis für alle Situationen mit sieben Steinen auf dem Feld berechnet, bevor mit der Berechnung einer Datenbank mit acht Steinen begonnen wird. Für jede einzelne Kombination der Steinanzahl werden nun die folgende Schritte durchgeführt:

- Erstellung aller möglicher Spielsituationen mit dieser Steinanzahl.
  Als erstes wurden von uns alle Spielsituationen mit der vorgebenen Steinanzahl generiert. Dazu wurden zunächst die schwarzen und weißen Spielsteine ohne Unterscheidung der Farbe in allen möglichen Kombinationen auf die 24 Felder des Spielfeldes gesetzt. In einem zweiten Schritt wurden nun für jede dieser Kombinationen alle möglichen Permutationen von schwarzen und weißen Steinen generiert. Dieses Vorgehen wird von der von uns gewählten, weiter unten beschriebenen Codierung unterstützt. In einem letzten Schritt werden zu jeder so konstruierten Spielsituation alle Spiegelungen erstellt und der oben beschriebene Repräsentant der Gruppe gespeichert, falls er noch nicht eingespeichert ist.

- Feststellen aller direkten Gewinnsituationen.
  Als nächstes werden alle im vorherigen Schritt erzeugten Spielsituationen danach überprüft, ob mit ihnen ein direkter Sieg möglich ist. Als direkter Sieg wird eine Spielsituation bezeichnet, wenn der Gegner durch den eigenen Zug aller Zugmöglichkeiten beraubt wird, wenn der Gegner durch Schließen einer Mühle auf zwei Steine reduziert wird oder auch, wenn durch das Schmeißen eines gegnerischen Steins eine Spielsituation entsteht, die bereits in der zuvor erstellten Datenbank für einen Stein weniger als Verlustsituation des Gegners gespeichert wurde. Zu jeder dieser Situationen wird die Anzahl der Halbzüge n bis zum Gewinn bei optimalem Spiel gespeichert. Dabei gilt:

  $$n = \begin{cases} 1, & \text{falls Gegner eingesperrt oder auf zwei Steine reduziert wurde.} \\ k+1, & \text{falls Verlustsituation des Gegners in } k \text{ Zügen} \\ & \text{in der Datenbank für einen Stein weniger erreicht wurde.} \end{cases}$$

  Analog dazu müssten theoretisch auch die direkten Verlustsituationen erkannt werden, also alle Situationen, in denen der Spieler am Zug bereits verloren hat, obwohl er durch Schließen einer Mühle in ein Endspiel mit weniger Steinen für den Gegner kommen kann. Basierend auf den Ergebnissen von Gasser konnten wir diesen Fall jedoch für die betrachteten Endspiele mit bis zu acht Steinen ausschließen, das heißt, bei Mühleendspielen mit bis zu acht Steinen existieren für den aktiven Spieler bei einer offenen Mühle keine direkten Verlustsituationen.
  Damit wurden bereits alle Situationen als Siege oder Niederlagen kategorisiert, die wegen Schmeißen eines Steins direkt auf den Teil der Datenbank mit niedrigerer Steinzahl

zurückgreifen müssen. Daher muss im folgenden Schritt für das Finden weiterer Gewinn und Verlustsituationen nur noch die Datenbank für die aktuelle Steinanzahl in Betracht gezogen werden.

- Iteration von Loss-Backup und Win-Backup
  Schließlich werden wiederholt Loss-Backups und Win-Backups durchgeführt:
  Beim Loss-Backup werden alle Spielsituationen nach neuen Verlustsituationen durchsucht. Dazu wird überprüft, ob bei **allen** eigenen Zügen die enstehende Spielsituation bereits als Gewinn für den Gegner gespeichert ist. Neu gefundene Verlustsituationen werden mit der Verlustlänge $l + 1$ bei maximaler Gewinnlänge l einer Nachfolgesituation gespeichert.
  Beim Win-Backup werden alle Spielsituationen nach neuen Gewinnsituationen durchsucht. Hierzu genügt es, wenn **eine** Nachfolgesituation schon als Verlust für den Gegner gespeichert ist. Neu entdeckte Gewinnsituationen werden mit der Gewinnlänge $w + 1$ gespeichert, wobei w das Minimum der Verlustlängen des Gegners in den Nachfolgesituationen ist.
  Wie gerade gezeigt genügt es beim Loss- wie beim Win-Backup auf die Datenbank für die aktuelle Steinzahl zurückzugreifen. Die Suche bricht ab, sobald in einem aufeinanderfolgenden Loss-Backup und Win-Backup keine neuen Verlust- beziehungsweise Gewinnsituationen entdeckt wurden.

3.4 Speicherform und Verwendung im Spiel

Speicherform

Eine naheliegende Möglichkeit und auch unser erster Versuch, die aktuelle Spielsituation und die Gewinn- bzw Verlust-Distanz zu speichern, war es, einzeln die Position der weißen Steine, die Position der schwarzen Steine und die Gewinn- bzw Verlust-Distanz zu codieren und zusammenzufügen. Beim Mühlespielfeld, welches 24 Felder besitzt, belegt eine solche Codierung für die weißen sowie die schwarzen Steine demnach jeweils 24 Bit. Für die Distanz bis zum Spielende langt eine 8 Bit-Zahl, da es keine Gewinn- bzw Verlustsituationen gibt, die mehr als 256 Halbzüge bis zu ihrem Ergebnis benötigen.
Insgesamt braucht man also für diese Art der Codierung $24 + 24 + 8 = 56$ Bit, was 7 Byte entspricht. Abbildung 3.2 verdeutlicht diese Codierung.

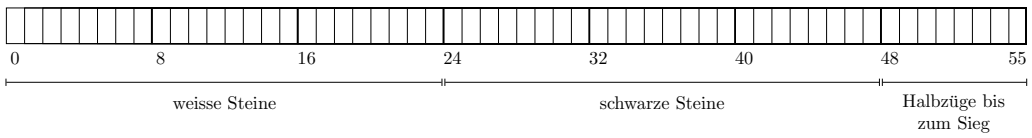


Abbildung 3.2: 7 Byte Codierung

Eine elegantere Lösung besteht darin, die Tatsache zu nutzen, dass sich nur maximal acht Steine auf dem Spielfeld befinden. Somit genügt es, in 24 Bit die gesetzen weißen und schwarzen Steine - also ohne Farbunterscheidung - zu speichern. In lediglich 8 Bit kann man dann die aktuelle Permutation speichern, in der die weißen und schwarzen Steine angeordnet sind. Dabei steht eine 0 für einen weißen Stein und eine 1 für einen schwarzen Stein. Die 8 Bit zum Speichern der Gewinn- bzw Verlust-Distanz wurde beibehalten. Abbildung 3.3 zeigt diese Codierung. Sie benötigt nur noch $24 + 8 + 8 = 40$ Bit, also nur noch 5 Byte.

Wie man sieht, lässt sich mit den ersten 32 Bit dieser Codierung eine bijektive Abbildung zu den Spielfeldern mit bis zu acht Steinen auf dem Spielbrett beschreiben. Wegen dieser

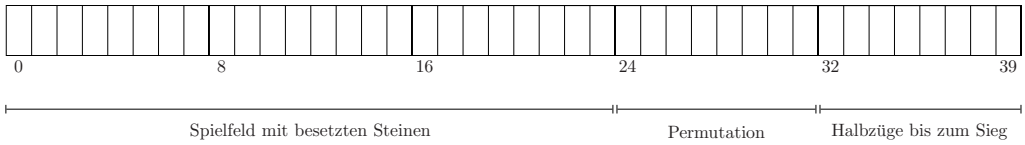


Abbildung 3.3: 5 Byte Codierung

Eigenschaft können wir diese 32-Bit-Zahl als perfekten Hashcode zum Speichern der Spielfelder in einer Hashtabelle nutzen.
In Abbildung 3.4 wird eine sich in der Endspieldatenbank befindliche Spielsitation gezeigt, bei welcher der weiße Spieler, der auch am Zug ist, mit folgender Zugfolge in fünf Halbzügen gewonnen hat.
Zugfolge: $c3 \rightarrow a4, e3 \rightarrow b4, c4 \rightarrow a1, c5 \rightarrow a7, a4 \rightarrow g1$.

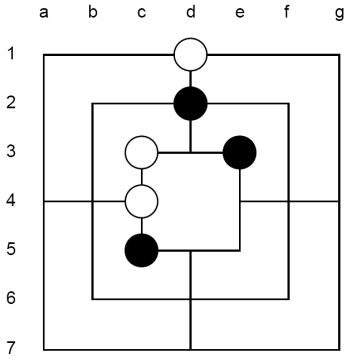


Abbildung 3.4: Sieg für Weiß in 5 Zügen

In Abbildung 3.5 ist die entsprechende Codierung zu sehen. Das nullte Bit entspricht dem Mühlefeld $a1$ links oben. Dann werden alle Felder zeilenweise durchnummeriert bis Bit 23, welches dem Feld $g7$ entspricht. Die Bits 24 bis 31 geben die Permutation der Steine an, in diesem Beispiel also abwechselnd ein weißer und ein schwarzer Stein. Die Bits 30 und 31 sind hier allerdings ohne Bedeutung, da sich nur 6 Steine auf dem Spielfeld befinden. In den Bits 32 bis 39 ist die Gewinn- bzw. Verlustdistanz gespeichert, hier also 5.

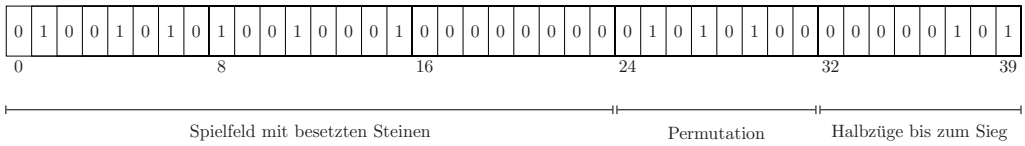


Abbildung 3.5: Codierung des Datenbankeintrags für das Spielfeld in Abbildung 3.4

Verwendung der Datenbank

Die Datenbank wird im Spiel anstelle der heuristischen Einschätzung verwendet, sobald sich nur noch 8 Steine auf dem vom Minmax-Algorithmus zu bewertenden Spielfeld befinden. Sie wird zum Zeitpunkt der ersten Verwendung in den Speicher geladen und bleibt dort bis zum Spielende. Organisiert ist sie im Programm durch mehrere Hashtables, die als Keys

den oben beschriebenen perfekten Hashcode besitzen und deren Werte die Gewinn- bzw. Verlustdistanz darstellen. Für jede Steinanzahl-Kombination (3-3, 4-3, 3-4, 5-3, 3-5, 4-4) verwenden wir einen jeweils eigenen win-Hash und loss-Hash. So müssen wir bei einer aktuell vorhandenen Steinanzahl nicht einen sehr großen Hash durchsuchen, sondern eben nur den wesentlich kleineren, der die Konstellationen mit entsprechender Steinanzahl gespeichert hat.

3.5 Auswertung

3.5.1 Statistik

Der Tabelle 3.1 kann man die Anzahl der Stellungen und die Größe unserer Endspieldatenbank entnehmen. Sie ist insgesamt 4,7 MB groß und enthält 943.317 Stellungen.

Datenbank	Stellungen	Größe in Byte
win3-3	140.621	703.105
win3-4	102.281	511.405
win3-5	6.301	31.505
win4-3	74.649	373.245
win4-4	152	760
win5-3	577.309	2.886.545
loss3-3	28.736	143.680
loss3-4	3.095	15.475
loss4-4	28	140
loss5-3	10.145	50.725

Tabelle 3.1: Endspieldatenbank Statistik

3.5.2 Spielstärke

Durch die Verwendung der Endspieldatenbank konnte die Gesamtspielstärke geringfügig gesteigert werden. Das Resultat einer Serie von Testspielen des Mühleprogramms mit Endspieldatenbank gegen das entsprechende ohne Endspieldatenbank bei einer Bedenkzeit von t=5000 ms pro Zug auf dem Referenzrechner ist in der folgenden Tabelle 3.2 festgehalten:

Programm	Gewinne	Verluste	Unentschieden
Programm mit Endspieldatenbank	2145	1773	1637
Programm ohne Endspieldatenbank	1773	2145	1637

Tabelle 3.2: Endspieldatenbank: Testspiele

Neben dieser Verbesserung der Spielstärke ist vor allem der Zeitvorteil entscheidend.Während ohne Verwendung der Datenbank wieder eine aufwendige Suche bis zu einer vorgegebenen Zeitschranke notwendig ist, genügen bei Verwendung einer Datenbank einige wenige Hashzugriffe. Ein korrekter Spielzug kann damit in wenigen Millisekunden gefunden werden.

Kapitel 4

Lernen von Heuristiken mit Neuronalen Netzwerken

4.1 Heuristiken im Spielbaum

Zum Bestimmen eines „guten“ Zuges ist es entscheidend, den Spielwert, also den Spielausgang bei perfektem Spiel beider Spieler, zu kennen. Leider ist ein solcher Spielwert im Allgemeinen nicht oder nur sehr aufwendig zu berechnen. Daher greift man auf eine Heuristische Funktion zurück. Diese Heuristische Funktion ist eine Abbildung aller möglichen Spielsituationen auf das Interval [-1;1], die den Spielwert der Situation möglichst gut approximieren soll, aber auch relativ schnell berechenbar sein soll.
Zum Erstellen einer Heuristischen Funktion muss man sich zunächst überlegen, welche Kennzeichen (Features) man bei einer gegebenen Spielsituation feststellen will und somit bewerten kann. Für das Mühlespiel scheinen intuitiv folgende Features geeignet:
Zum einen ist für eine Spieleinschätzung natürlich der Materialvorteil, also die Anzahl der eigenen und der gegnerischen Steine, sehr wichtig. Ein zweiter wichtiger Punkt ist beim Mühlespiel auch die Bewegungsmöglichkeit der eigenen und der gegnerischen Steine, da bei Zugunfähigkeit eine Niederlage droht. Weitere Features bilden offene und geschlossene Mühlen beider Spieler auf dem Spielfeld, sowie der Unterschied der potentiellen Bewegungsmöglichkeit beider Spieler. Diese richtet sich danach, wieviele Zugmöglichkeiten ein Spieler hätte, wenn kein anderer Spielstein sich auf dem Feld befinden würde. So werden Felder besonders hervorgehoben, die prinzipiell eine große Bewegungsfreiheit bieten, auch wenn dies in der aktuellen Spielsituation nicht der Fall ist.
Diese fünf Feature-Typen bilden die Grundlage unserer Standard-Heuristik. Da für das spätere Lernen einer geeigneten Gewichtung dieser Features aber binäre Eingabewerte benötigt werden, speichern wir sie in 39 binären Features. Die Heuristik mit von uns selbst bewerteten Gewichten ist in Tabelle 4.1 dargestellt.

Nr	Feature	Gewichtung
1	4 Steine: akt. Spieler	0.10
2	5 Steine: akt. Spieler	0.15
3	6 Steine: akt. Spieler	0.20
4	7 Steine: akt. Spieler	0.25
5	8 Steine: akt. Spieler	0.30
6	9 Steine: akt. Spieler	0.35
7	4 Steine: Gegener	-0.10
8	5 Steine: Gegener	-0.15
9	6 Steine: Gegener	-0.20
10	7 Steine: Gegener	-0.25
11	8 Steine: Gegener	-0.30
12	9 Steine: Gegener	-0.35
13	2 oder 3 Zugmöglichkeiten: akt. Spieler	0.10
14	4 oder 5 Zugmöglichkeiten: akt. Spieler	0.15
15	6 oder 7 Zugmöglichkeiten: akt. Spieler	0.20
16	8 oder 9 Zugmöglichkeiten: akt. Spieler	0.25
17	mehr als 10 Zugmöglichkeiten: akt. Spieler	0.30
18	2 oder 3 Zugmöglichkeiten: Gegner	-0.10
19	4 oder 5 Zugmöglichkeiten: Gegner	-0.15
20	6 oder 7 Zugmöglichkeiten: Gegner	-0.20
21	8 oder 9 Zugmöglichkeiten: Gegner	-0.25
22	mehr als 10 Zugmöglichkeiten: Gegner	-0.30
23	1 geschlossene Mühle: akt. Spieler	0.01
24	mehr als 1 geschlossene Mühle: akt. Spieler	0.02
25	1 geschlossene Mühle: Gegner	-0.01
26	mehr als 1 geschlossene Mühle: Gegner	-0.02
27	1 offene Mühle: akt. Spieler	0.02
28	mehr als 1 offene Mühle: akt. Spieler	0.04
29	1 offene Mühle: Gegner	-0.02
30	mehr als 1 offene Mühle: Gegner	-0.04
31	mehr als 7 potentielle Bewegungen weniger als der Gegner	-0.16
32	4 bis 6 potentielle Bewegungen weniger als der Gegner	-0.08
33	2 bis 3 potentielle Bewegungen weniger als der Gegner	-0.04
34	1 potentielle Bewegung weniger als der Gegner	-0.02
35	Genausoviele potentielle Bewegungen wie der Gegner	0.00
36	1 potentielle Bewegungen meher als der Gegner	0.02
37	2 bis 3 potentielle Bewegungen meher als der Gegner	0.04
38	4 bis 6 potentielle Bewegungen meher als der Gegner	0.08
39	mehr als 7 potentielle Bewegungen meher als der Gegner	0.16

Tabelle 4.1: Gewichtungen unserer Standard-Heuristik

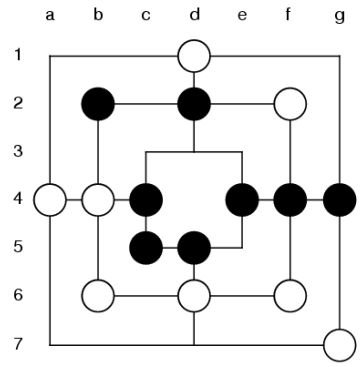


Abbildung 4.1: Beispiel: Spieleinschätzung mit Standard-Heuristik

In Tabelle 4.2 sieht man eine Beispielrechnung zur Bestimmung der Spieleinschätzung des in Abbildung 4.1 gezeigten Spielfeldes aus Sicht des weißen Spielers. Man erhält -0,11. Somit liegt der weiße Spieler knapp hinten.

Feature	Gewichtung
eigene Steine	0,30
gegnerische Steine	-0,30
eigene Zugmöglichkeiten	0,15
gegnerische Zugmöglichkeiten	-0,20
eigene offene Mühlen	0,00
gegnerische offene Mühlen	-0,02
eigene geschlossene Mühlen	0,01
gegnerische geschlossene Mühlen	-0,01
Differenz der potentiellen Zugmöglichkeiten	-0,04
Summe	-0,11

Tabelle 4.2: Beispielrechnung zur Bestimmung der Spieleinschätzung

Neben dieser Standard-Heuristik haben wir noch eine ganze Reihe weiterer Heuristiken getestet. Zum Beispiel versuchten wir eine Minimal-Heuristik, die nur den Steinvorteil mit einbezieht, und eine Klein-Heuristik, die sich nur auf den Steinvorteil und die Anzahl der Zugmöglichkeiten stützt. Diese Heuristiken haben, wie man später im Test sieht jedoch nicht gut gegen die Standard-Heuristik abgeschnitten, obwohl sie weniger Rechenzeit benötigen und demenstsprechend bei vorgegebener Zeitbeschränkung mehr Zeit für die Suche im Minmax zur Verfügung haben. Eine weiterer Versuch war eine Kombinations-Heuristik. Hier haben wir als Inputs alle Feature-Paare, also 39^2 Einzelinputs, angelegt. Eine solche Heuristik ermöglicht die Bewertung von Kombinationen mit einem eigenen Gewicht, etwa ergibt beispielsweise in einer Heuristik die Konstellation aus acht eigenen Steinen und einer gegnerischen offenen Mühle 15 Punkte. Eine solche Heuristik weist allerdings schon bei Weitem zu viele Features auf, um die einzelnen Gewichtungen manuell festzulegen. Aus diesem Grund haben wir die Gewichtung dieser, wie auch später der anderen Heuristiken maschinell lernen lassen.

In unserer Implementierung werden alle Heuristiken im XML-Format gespeichert. Dadurch bleibt einerseits auch bei komplexen Heuristiken eine gewisse Übersichtlichkeit gewährleistet, andererseits können Java-eigene Input- und Output-Funktionen genutzt werden.

4.2 Einführung in Neuronale Netzwerke

Eine manuelle Gewichtung der Heuristik-Features eines Spielfeldes ist auch für menschliche Experten sehr schwierig und zeitaufwendig. Daher versucht man günstige Gewichtungen maschinell zu erreichen. Der Mühlespiel-Agent soll also die Gewichtungen durch wiederholtes spielen eigenständig „lernen“. Eine weit verbreiteter Lösungsansatz dazu ist die Verwendung Neuronaler Netzwerke.
Kernidee ist die Nachahmung eines biologischen Gehirns, das aus großen „Netzen“ von miteinander durch unterschiedlich starke Synapsen verknüpften Neuronen besteht. Ein einzelnes Neuron empfängt dabei eine Reihe von verschieden starken Eingangssignalen, verarbeitet sie und sendet in Abhängigkeit davon ein Signal weiter.
In der maschinellen Umsetzung werden die unterschiedlichen Stärken der Synapsen durch verschieden starke Gewichtungen $W_{j,i}$ der Eingangssignale a_j in eine Neuron-Unit simuliert. Diese Eingangssignale werden aufsummiert und als Argument einer Aktivierungsfunktion g übergeben, die dann das Outputsignal a_i bestimmt. Als Output eines Neurons ergibt sich also:

$$a_i = g(\sum_{j=0}^{n} W_{j,i} a_j)$$

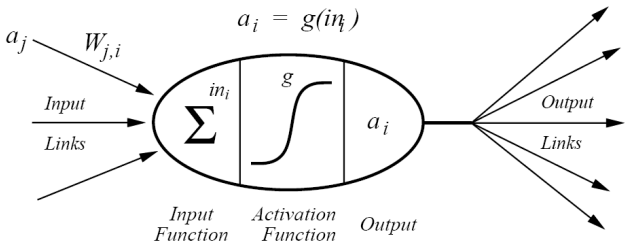


Abbildung 4.2: Neuron

4.2.1 Netzstrukturen

Wie bereits erwähnt bestehen Neuronale Netzwerke aus vielen einzelnen Neuronen, die prinzipiell beliebig miteinander verknüpft werden können. Im Rahmen dieser Arbeit haben wir uns allerdings auf einige wenige wichtige Strukturen beschränkt. Wir betrachten hier lediglich in Ebenen angeordnete Netze ohne Rückkopplung, bei denen zwei aufeinanderfolgende Ebenen vollvermascht sind. Weitere Beziehungen zwischen jeweils zwei Neuronen existieren nicht.
In der Literatur wird grundsätzlich zwischen Singlelayer-Netzwerken (auch Perceptron Netzwerke, Abb. 4.3) und Multilayer-Netzwerken (Abb. 4.4) unterschieden. Bei Singlelayer-Netzen handelt es sich lediglich um eine einzige Schicht Neuronen (Outputknoten). An jedem einzelnen Neuron liegen dabei alle Eingänge an. Vorteil dieser einfachen Struktur ist es, dass auf ihr schnelle und effiziente Lernalgorithmen durchführbar sind. Die Ausdrucksstärke einer solchen Anordnung ist jedoch stark begrenzt.

Multilayer-Netzwerken dagegen bestehen aus mehreren Ebenen mit unterschiedlicher Anzahl an Neuronen. Verknüpfungen zwischen Neuronen bestehen dabei lediglich zwischen Neuronen zweier aufeinanderfolgender Ebenen. Die inneren Ebenen des Netzwerkes, die sich nur indirekt auf den Output auswirken, werden dabei als „Hidden Layer“ bezeichnet. Lernvorgänge in Multilayer-Netzwerken sind komplexer und laufen in der Regel langsamer ab.

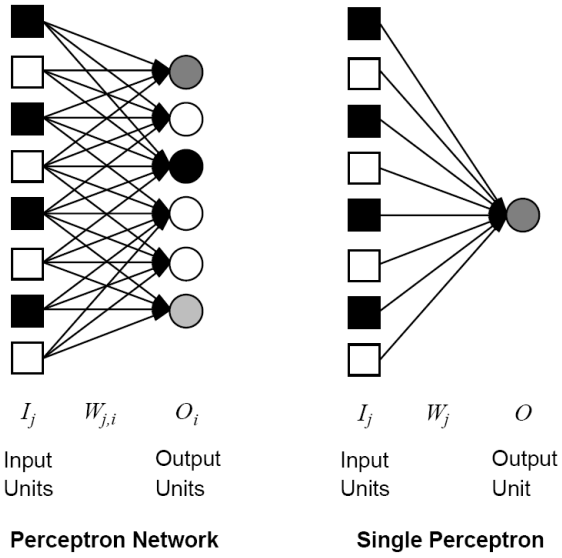


Abbildung 4.3: Singlelayer-Netzwerk

Sie können dafür aber bei einer ausreichenden Zahl von Neuronen in den „Hidden Layern“ und entsprechend langen Lernphasen beliebig schwierige Zusammenhänge lernen.

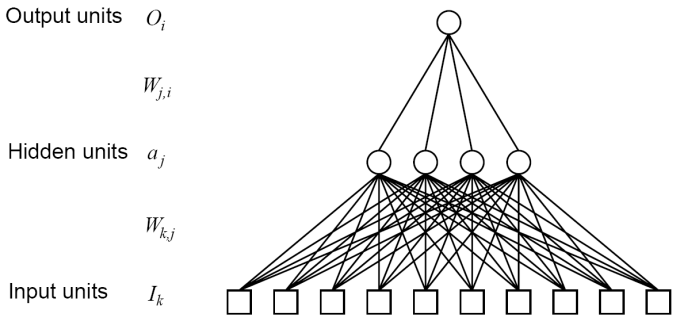


Abbildung 4.4: Multilayer-Netzwerk

4.2.2 Aktivierungsfunktion

Eine Aktivierungsfunktion ist eine Funktion, die auf die gewichtete Summe der Inputs *in* eines Neurons angewendet wird, und daraus das Outputsignal des Neurons berechnet. Die Verwendung einer solchen Aktivierungsfunktion hat mehrere Vorteile. Zum einen kann damit der Wertebereich des Outputs eines Neurons festgelegt werden, zum anderen ist die Verwendung einer Aktivierungsfunktion nötig, da ansonsten ausschließlich lineare Funktionen durch ein Neuronales Netzwerk beschrieben werden könnten. übliche Aktivierungsfunktionen sind:

- Die Stufenfunktion (*threshold function*), die unterhalb eines Schwellwertes 0 zurückliefert, oberhalb des Schwellwertes 1:

$$g(in) = \begin{cases} 1, & \text{falls } in \text{ größer als Schwellwert } t \\ 0, & \text{sonst} \end{cases}$$

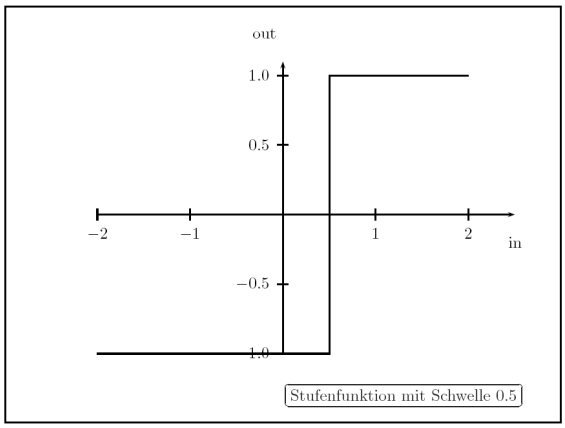


Abbildung 4.5: Stufenfunktion

Nachteile der Stufenfunktion sind, dass sie nicht differenzierbar ist und das Unsicherheiten im Output nicht ausgedrückt werden können. Gerade aus letzterem Grund ist diese Funktion für Neuronale Netzwerke zum Einschätzen von Positionen eines Mühlespiels ungeeignet.

- Die logistische Funktion (wegen ihrer Form auch als sigmoide Funktion bezeichnet) ist streng monoton steigend, stetig und differenzierbar. Ihr Wertebereich ist das Intervall $]0;1[$. Sie ist folgendermaßen definiert:

$$g(in) = \frac{1}{1 + e^{-in}}$$

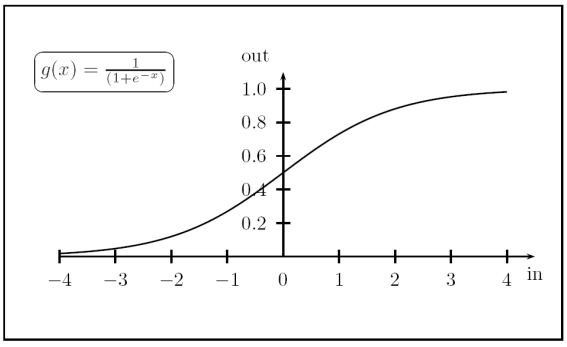


Abbildung 4.6: Logistische Funktion (sigmoid function)

Die Ableitung der logistischen Funktion ist gegeben durch:

$$g'(in) = g(in) \cdot (1 - g(in))$$

- Die Tangens Hyperbolicus Funktion. Auch sie ist streng monoton steigend, stetig und differenzierbar. Sie hat den Wertebereich $]-1;1[$ und ist wie folgt gegeben:

$$g(in) = \frac{e^{in} - e^{-in}}{e^{in} + e^{-in}}$$

Ihre Ableitung ist wie folgt definiert:

$$g'(in) = 1 - g(in)^2$$

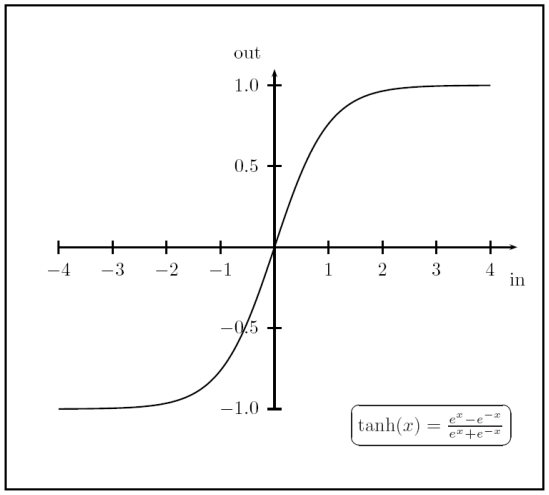


Abbildung 4.7: Tangens Hyperbolicus Funktion

Für die Einschätzung von Mühlepositionen verwenden wir eine Netzstruktur mit drei Outputknoten, je einen für Sieg, Unentschieden und Niederlage. Jeder dieser drei Outputknoten soll die Wahrscheinlichkeit angeben, mit der der entsprechende Spielausgang eintritt. Daher wird das Intervall $]0;1[$ als Wertebereich benötigt. Deshalb und der anderen oben erwähnten Eigenschaften wegen haben wir als Aktivierungsfunktion der Outputknoten die logistische Funktion verwendet. Für die *Hidden Units* haben wir auf Grund des benötigten Wertebereiches auf die Tangens Hyperbolicus Funktion als Aktivierungsfunktion zurückgegriffen.

4.3 Lernen in Singlelayer Neural Networks (Perceptron Networks)

4.3.1 Gradientenabstiegsverfahren

Die verbreitetste Methode zur Gewichtsanpassung in Neuronalen Netzwerken besteht in einem Gradientenabstiegsverfahren. Hierbei wird der Fehler des Netzwerkoutputs als Funktion über den Gewichten des Netzwerks betrachtet. Die Funktionswerte dieser (differenzierbaren) Fehlerfunktion ergeben eine sogenannte "Fehlerfläche". Ziel des Gradientenabstiegsverfahrens ist es, ein Minimum der Funktion zu finden. Dies versucht man zu erreichen, indem man von einem aktuellen Punkt auf der Fehlerfläche auf dem steilst möglichen Weg bergab geht (*steepest-descent*). Die Richtung des steilsten Abstiegs wird mathematisch durch den Gradienten bestimmt, also den Vektor der partiellen Ableitungen nach den einzelnen Gewichten.
Als Fehler für ein Muster betrachtet man den quadratischen Abstand zwischen erwartetem und realem Ausgang.

$$E_p = \frac{1}{2} \sum_i (t_{pi} - a_{pi})^2.$$

Dabei ist t_{pi} der reale Ausgangswert für ein Muster p und a_{pi} die Ausgabe des Neurons i bei Muster p. Der Faktor $\frac{1}{2}$ wird aus rechnerischen Gründen verwendet. Einen Einfluss auf das Ergebnis hat er nicht, da es egal ist, ob man den Fehler oder den halben Fehler minimiert.

Werden mehrere Muster in einem Optimierungsschritt betrachtet, dann ergibt sich der Gesamtfehler E aus der Summe der Einzelfehler E_p:

$$E = \sum_p E_p.$$

Die Gewichtsänderung wird nun bestimmt aus dem Gradienten der Fehlerfunktion und einem Faktor η, der sogenannten *learning rate*, der die Schrittweite der Gewichtsänderung beeinflusst:

$$\Delta W = -\eta \nabla E(W),$$

wobei $E(W) = E(w_1, ..., w_n)$.

Für ein einzelnes Gewicht gilt dann:

$$\Delta w_{ji} = -\eta \frac{\partial}{\partial w_{ji}} E(W).$$

Diese Ableitung lässt sich leicht bestimmen durch die Delta-Regel.

4.3.2 Delta-Regel

Die Gewichtsänderung in einstufigen Netzen wird durch die Delta-Regel bestimmt. Sie soll im Folgenden kurz hergeleitet werden. Als Aktivierungsfunktion der Netzwerkknoten dient der Einfachheit halber zunächst die Identität. Es ergibt sich mit der Verwendung der Kettenregel:

$$\begin{aligned}
\Delta w_{ji} &= -\eta \frac{\partial}{\partial w_{ji}} E(W) \\
&= \sum_p -\eta \frac{\partial}{\partial w_{ji}} E_p \\
&= \sum_p -\eta \frac{\partial E_p}{\partial a_{pi}} \frac{\partial a_{pi}}{\partial w_{ji}} \\
&= \sum_p -\eta \frac{\partial}{\partial a_{pi}} \frac{1}{2} \sum_i (t_{pi} - a_{pi})^2 \frac{\partial}{\partial w_{ji}} \sum_j a_{pj} w_{ji} \\
&= \sum_p -\eta(-(t_{pi} - a_{pi}))a_{pj} \\
&= \sum_p \eta(t_{pi} - a_{pi})a_{pj} \\
&= \sum_p \eta \delta_{pi} a_{pj} \\
&= \eta \sum_p a_{pj} \delta_{pi}
\end{aligned}$$

mit dem Fehlersignal $\delta_{pi} = (t_{pi} - a_{pi})$.

Damit ergibt sich für ein einzelnes Muster p bei Annahme der Identität als Aktivierungsfunktion die Delta-Regel in der üblichen Form:

$$\Delta_p w_{ji} = \eta a_{pj} \delta_{pi} = \eta a_{pj} (t_{pi} - a_{pi})$$

4.3.3 Lernregel für Singlelayer-Netzwerke

Analog zur Herleitung der Deltaregel erhält man bei Verwendung einer monotonen, differenzierbaren Aktivierungsfunktion $g(x)$ für das Fehlersignal δ:

$$\delta_{pi} = g'(in_{pi})(t_{pi} - a_{pi})$$

wobei in_{pi} die gewichtete Summe der Inputs eines Knoten i bei Muster p ist.

$$in_{pi} = \sum_j a_{pj} w_{ji}$$

Damit folgt die allgemeine Lernregel für Singlelayer-Netzwerke:

$$\Delta_p w_{ji} = \eta a_{pj} \delta_{pi} = \eta a_{pj} g'(in_{pi})(t_{pi} - a_{pi})$$

4.4 Lernen in Multilayer Neural Networks

4.4.1 Backpropagation-Regel

Die Backpropagation-Regel ist eine Verallgemeinerung der Delta-Regel für:

- Netzte mit mehreren Ebenen und
- semilineare (monotone, differenzierbare, nicht unbedingt lineare) Aktivierungsfunktionen

Bei mehreren Ebenen ist die Eingabe eines Neurons i nun nicht mehr nur ein Wert, sondern die gewichtete Summe der Outputs der Ebene darunter. Der Input eines einzelnen Neurons bei einem Muster p lautet dann:

$$in_{pi} = \sum_j a_{pj} w_{ji}$$

Beim Anwenden einer Aktivierungsfunktion gilt dann für den Output des Neurons i:

$$a_{pi} = g(in_{pi})$$

Nun lässt sich analog zur Delta-Regel eine Verallgemeinerung der Delta-Regel, die sogenannte *Backpropagation-Regel* herleiten:

$$\begin{aligned}
\Delta w_{ji} &= \sum_p -\eta \frac{\partial E_p}{\partial w_{ji}} \\
&= \sum_p -\eta \frac{\partial E_p}{\partial in_{pi}} \frac{\partial in_{pi}}{\partial w_{ji}} \\
&= \sum_p -\eta \frac{\partial E_p}{\partial in_{pi}} \frac{\partial}{\partial w_{ji}} \sum_j a_{pj} w_{ji} \\
&= \sum_p -\eta \frac{\partial E_p}{\partial in_{pi}} a_{pj} \\
&= \sum_p \eta \delta_{pi} a_{pj} \\
&= \eta \sum_p a_{pj} \delta_{pi}
\end{aligned}$$

Das Fehlersignal δ_{pi} sieht dann folgendermaßen aus:

$$\begin{aligned}
\delta_{pi} &= -\frac{\partial E_p}{\partial in_{pi}} \\
&= -\frac{\partial E_p}{\partial a_{pi}} \frac{\partial a_{pi}}{\partial in_{pi}} \\
&= -\frac{\partial E_p}{\partial a_{pi}} \frac{\partial}{\partial in_{pi}} g(in_{pi}) \\
&= -\frac{\partial E_p}{\partial a_{pi}} g'(in_{pi})
\end{aligned}$$

Für den ersten Faktor muss nun unterschieden werden, ob es sich bei dem Neuron mit dem Index i um einen Outputknoten oder einen inneren Knoten handelt. Im ersten Fall gilt analog zum einstufigen Netz:

$$-(\frac{\partial E_p}{\partial a_{pi}}) = (t_{pi} - a_{pi})$$

Für innere Knoten gilt:

$$\begin{aligned}
-(\frac{\partial E_p}{\partial a_{pi}}) &= -\sum_h \frac{\partial E_p}{\partial in_{ph}} \frac{\partial in_{ph}}{\partial a_{pi}} \\
&= \sum_h (\delta_{ph} \frac{\partial}{\partial a_{pi}} \sum_j a_{pj} w_{jh}) \\
&= \sum_h \delta_{ph} w_{ih}
\end{aligned}$$

Für ein einziges Muster p lautet die Backpropagation-Regel dann:

$$\Delta_p w_{ji} = \eta a_{pj} \delta_{pi}$$

mit

$$\delta_{pi} = \begin{cases} g'(in_{pi})(t_{pi} - a_{pi}), & \text{falls } i \text{ Output Knoten ist} \\ g'(in_{pi}) \sum_h \delta_{ph} w_{ih}, & \text{falls } i \text{ verdeckter Knoten ist} \end{cases}$$

Wie man hier unmittelbar erkennen kann, reduziert sich die Backpropagation Regel bei Verwendung eines Singlelayer-Netzwerkes auf die einfache Lernregel für Singlelayer-Netzwerke. In unserer Implementierung stellt sie die Basis für das Lernen in Neuronalen Netzwerken dar.

4.4.2 Momentum-Term

Ein allgemeines Problem von Gradientenabstiegsverfahren besteht darin, dass auf flachen Plateaus in der Fehlerfläche der Gradient gegen Null geht. Dies bedeutet, dass eine Gewichtsveränderung sehr langsam vorangeht und bis zum nächsten Minimum viele Iterationsschritte benötigt werden. Die Schwierigkeit dabei ist auch, dass man gar nicht unterscheiden kann, ob man sich auf einem Plateau oder gar in einem Minimum befindet. Eine übliche Methode diesem Problem entgegenzuwirken ist die Einführung eines Trägheitsmomentes, des sogenannten Momentum-Term. Die Grundidee dieses Verfahrens besteht darin, dass die letzte Gewichtsänderung in die neue Gewichtsänderung mit einfließt. Dies äußert sich dann durch ein schnelleres Vorankommen in solchen Situationen. Abbildung 4.8 verdeutlicht dies. Ein

weiteres Problem, was sich mit dem Momentum-Term lösen lässt, ist die Oszillation um ein lokales Minimum. Denn ist der Gradient auf der Seite einer steilen Schlucht sehr groß, so kann ein sich wiederholender Sprung auf die jeweils andere Seite auftreten. Ein Vorankommen auf der Fehlerfläche wäre in diesem Fall gar nicht mehr möglich, ein Festhängen im lokalen Minimum die einzige Alternative. Mit dem Momentum-Term dagegen lassen sich solche lokale Minima überspringen. Abbildung 4.9 verdeutlicht diesen Sachverhalt.

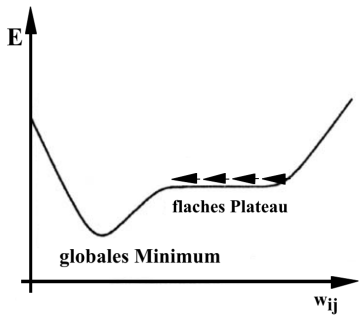


Abbildung 4.8: Momentum-Term: Beschleunigung bei flachen Plateaus

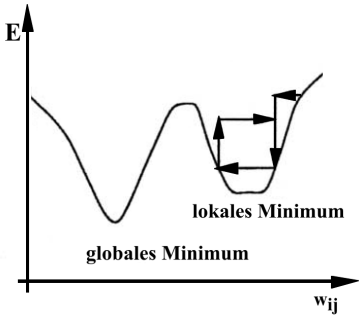


Abbildung 4.9: Momentum-Term: Abbremsen bei steilen Schluchten

Die Gewichtsänderung bei der Verwendung des Momentum-Terms ergibt sich aus einer durch den Parameter α geregelten Gewichtung von letzter Gewichtsänderung und der aktuellen Gewichtsänderung ohne Momentum-Term. Übliche Werte für α liegen dabei ungefähr im Bereich um 0.6 (vgl. [15]). Als Formel ergibt sich:

$$\Delta_p w_{ji}(t+1) = (1-\alpha)\eta a_{pj}\delta_{pi} + \alpha\Delta_p w_{ji}(t).$$

4.4.3 Adaptive Learning Rate

Um bei weiter Entfernung vom Idealwert eines bestimmten Gewichts schnelleres Annähern zu ermöglichen, aber gleichzeitig bei Gewichten, die sich schon sehr nahe am Optimum befinden, feine Änderungen zu erleichtern, wurde das Konzept der adaptiven Learning-Rate entwickelt. Dabei wird jedem Gewicht eine eigene Lernrate zugeordnet. Die Änderung dieser individuellen Lernrate wird dabei von der Folge der Änderungen dieses Gewichtes bestimmt. Ändert sich ein Gewicht in zwei aufeinanderfolgenden Lerndurchgängen in die gleiche Richtung, so wird diese Lernrate erhöht, andernfalls erniedrigt. In der Literatur (vgl. [15]) wird

dazu ein linearer Anstieg um einen kleinen Summanden κ und ein exponentieller Abfall um einen Faktor ϕ der Lernrate empfohlen. Dadurch werden kleine Änderungen nahe am Optimum besonders begünstigt.

Das Update eines Gewichts sieht dann unter Beibehaltung des Momentum-Terms folgendermaßen aus:

$$\Delta w_{ji}(t+1) = (1-\alpha)\eta_{ji}(t)a_j\delta_i + \alpha\Delta w_{ji}(N)$$

Hierbei ist $\eta_{ji}(N)$ die Lerning Rate, die wie folgt angepasst wird:

$$\eta_{ji}(t) = \begin{cases} \eta_{ji}(t-1) + \kappa, & \text{falls } (a_j\delta_i)(t) \cdot (a_j\delta_i)(t-1) > 0 \\ \eta_{ji}(t-1) \cdot \varphi, & \text{sonst} \end{cases}$$

Gebräuchliche Parametereinstellungen sind dabei $\kappa \approx 0.05$ und $\varphi \approx 0.5$.

4.5 Lernen beim Mühlespiel

4.5.1 Gewinnung eines Teaching-Wertes beim Mühlespiel

Für den Lernprozess wird, wie gerade eben beschrieben, für jedes Inputmuster ein möglichst exakter Spielwert benötigt. Dabei ergeben sich allerdings zwei Schwierigkeiten: Ein Problem ist, dass im Allgemeinen kein exakter Spielwert für jede einzelne Situation bekannt ist, sondern nur ein Gesamtausgang der Partie. Aus diesem Grund muss auf der Basis des Spielausgangs ein geeigneter Teaching-Wert für alle Spielsituation einer Partie ermittelt werden. Prinzipiell bieten sich hier zwei Möglichkeiten an. Auf der einen Seite ist es möglich, jede Spielsituation direkt mit dem Wert des Spielausgangs zu lernen. Dieses Extrem bedeutet allerdings, dass auch Spielsituationen gleich zu Beginn der Partie, die noch nicht entschieden sind, gegebenenfalls als Gewinn- beziehungsweise als Verluststellung gelernt werden. Auf der anderen Seite kann man lediglich die letzte Spielsituation mit dem Wert des Spielausgangs lernen, und alle vorherigen mit der heuristischen Einschätzung des Neuronalen Netzwerkes der jeweils nachfolgenden Spielsituation. Der Nachteil dieses Verfahrens ist es, dass bei schlechten Ausgangswerten der Heuristik für Stellungen am Beginn der Partie Werte gelernt werden, die dem letztlichen Ausgang des Spiels widersprechen. Bis solche Situationen richtig gelernt werden können, also der Spielausgang auch Einfluss auf den Start der Partie nimmt, sind zahlreiche Partien nötig. Aus diesem Grund wurden bei der Implementierung unseres Neuronalen Netzwerks ein Weg zwischen diesen beiden Extrema gewählt. Dabei wird der Teaching-Wert zusammengesetzt aus der heuristischen Spieleinschätzung der nachfolgenden Stellung und deren Teaching-Wert, in den auch durch die allerletzte Stellung der Spielausgang eingeflossen ist. Es gilt also:

$$t_{p-1} = \lambda t_p + (1-\lambda) \cdot out_p$$

mit Teaching-Wert t, dem Netzwerkoutput out_p bei Eingabe des Musters p und einem Parameter λ zwischen 0 und 1.
Das Verhalten des Teachingwertes bei unterschiedlich gewählten Parametern λ wird in den beiden Beispielen 4.10 und 4.11 illustriert. Die grüne Linie markiert dabei die Spieleinschätzung des Lernenden während eines Spiels, mit roter Farbe ist der exakte Spielwert zum jeweiligen Zeitpunkt eingetragen. Die übrigen Linien zeigen den Verlauf des Teachingwertes bei verschiedenen λ.

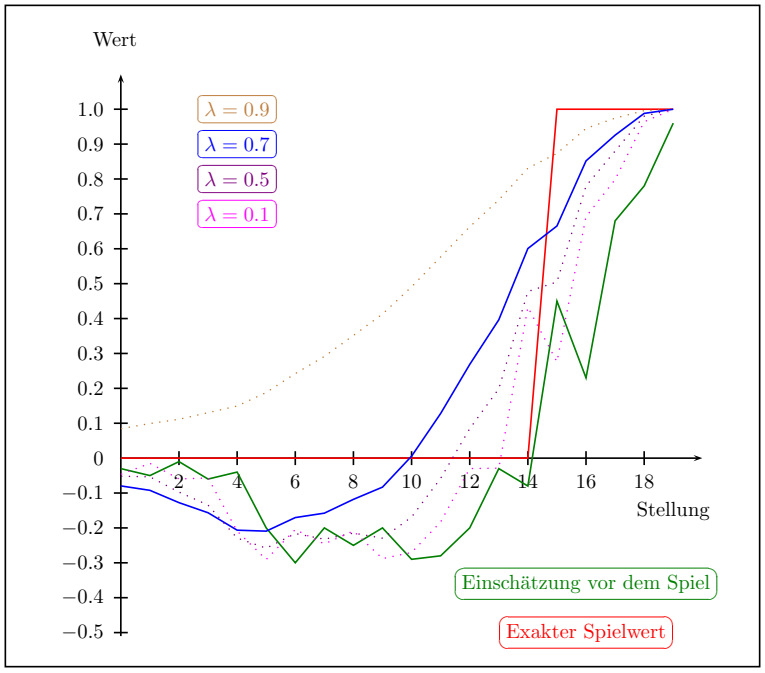


Abbildung 4.10: Bsp1: Verhalten des Teaching-Wertes bei unterschiedlichen λ

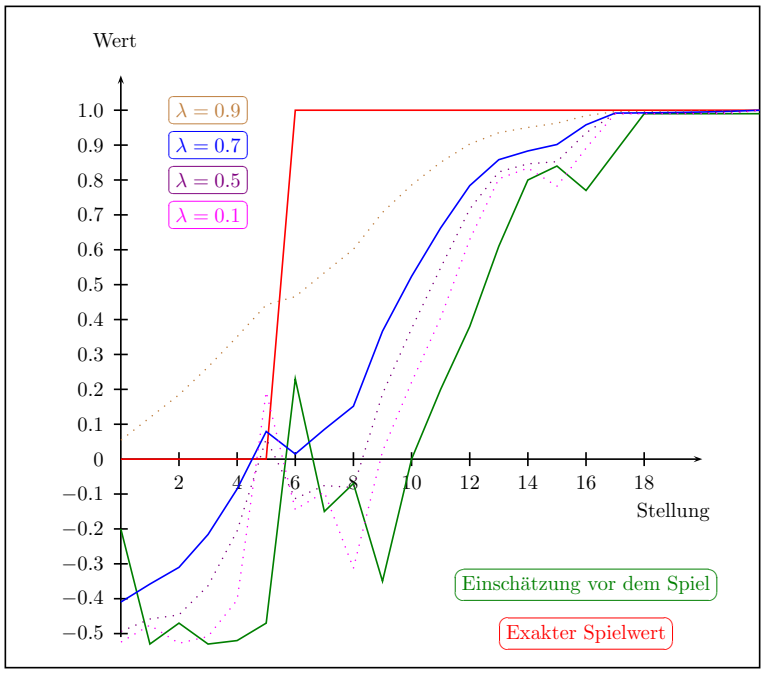


Abbildung 4.11: Bsp2: Verhalten des Teaching-Wertes bei unterschiedlichen λ

Ein zweites entscheidendes Problem besteht darin, dass die zur Einschätzung verwendeten Features nicht ausreichen, alle denkbaren Spielsituationen exakt in Gewinne, Verluste und Remis zu separieren (Generalisierungsproblem). Man muss also davon ausgehen, dass eine Belegung der Bits der Inputfeatures verschiedene Spielfelder repräsentiert, die tatsächlich auch unterschiedliche korrekte Spielwerte besitzen. Ein „richtiger“ Output des Neuronalen Netzwerks ist in einem solchen Fall also überhaupt nicht möglich. Ein Beispiel für einen solchen Fall ist in Abbildung 4.12 dargestellt.

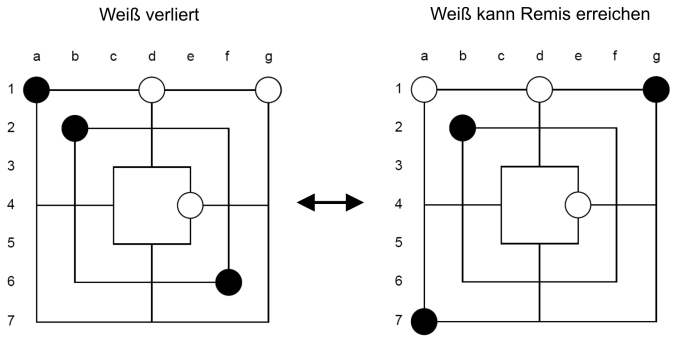


Abbildung 4.12: Feature Kollision

In beiden Spielsituationen, die aus der Setzphase stammen, besitzen beide Spieler drei Steine und haben exakt fünf Zugmöglichkeiten. Die Situation-Value-Differenz beträgt bei beiden Stellungen genau zwei, offene und geschlossenen Mühlen sind nicht vorhanden. Somit werden beide Spielsituationen im Neuronalen Netzwerk durch die selben Inputmuster abgebildet. Die Separierung der ersten Stellung, die ein sicheres Verlieren bedeutet, von der zweiten Stellung, mit der ein Remis möglich ist, lässt sich also nicht erreichen.

4.5.2 Lernvorgang

Lernvorgang ohne perfekte Datenbank

Für den Lernvorgang ohne perfekte Datenbank wird von uns die Methode des „Reinforcement Learnings“ verwendet. Dazu lassen wir stets zwei Computerspieler mit verschiedenen Heuristiken gegeneinander antreten. Diese spielen eine vollständige Partie Mühle, die Abfolge der Spielsituationen wird dabei gespeichert. Nach Spielende wird für beide Spieler eine Lernphase mit dem Wert des Spielausgangs aufgerufen. Dabei wird wie oben beschrieben für jede Situation ein Teaching-Wert berechnet und mit dem Backpropagation-Algorithmus im Neuronalen Netzwerk gelernt. Die Spiele wurden dabei aus rein praktischen Gründen mit einer festen Denktiefe pro Zug durchgeführt, um Unabhäbgigkeit von der Rechnerperformance zu gewährleisten.
Im Verlauf zahlreicher Lern- und Testphasen konnten wir hierbei einige Resultate festhalten. Zum einen stellten wir fest, dass die Lernvorgänge am erfolgreichsten sind, wenn beide am Lernspiel beteiligten Spieler ungefähr gleich stark sind und etwa ebensoviele Siege wie Niederlagen auftreten. Dies lässt sich dadurch erklären, dass zum Beispiel für den stärkeren Spieler auch ein schwaches Spielen wegen Fehlern des Gegners als Sieg endet und so als positiv gelernt werden würde. Diese Überlegung führt dazu, dass wir zum Lernen jeweils zwei Spieler mit gleichen Features, gleicher Netzstruktur und gleichen Lernparametern gegeneinander antreten lassen, da für diesen Fall ein gleiches Lerntempo für beide Spiele zu erwarten ist. Auch das Fortführung des Trainings einer erlernten Heuristik gegen eine andere starke Heuristik führte nur selten zu einer Verbesserung. Hier spielt möglicherweise die mangelnde Differenzierung der einzelnen Spielsituationen durch die verallgemeinernde Charakterisierung des Spielfeldes mittels der Features eine Rolle, die es nicht zulässt, einer gelernten Heuristik Stärken einer anderen Heuristik hinzuzufügen, ohne dass eigene Stärken

verloren gehen.
Zum anderen konnten wir die Beobachtung machen, dass Lernvorgänge mit großen Denktiefen und damit zwangsläufig weniger gespielten Partien in der selben Zeit zu besseren Ergebnissen führten als zahlreiche Spiele mit sehr niedrigen Denktiefen. Das lässt sich dadurch erklären, dass die Heuristik bereits eine gute direkte Spielbewertung benötigt, um bei niedriger Denktiefe nicht rein zufällig zu ziehen, wodurch zu viele Partien mit einem Remis enden. Dies wird bei hoher Suchtiefe zumindest gegen Ende des Spieles dadurch vermieden, dass ein direkter Sieg schon im Minmax-Algorithmus eher erkannt wird.
Aus dem selben Grund konnte der Beginn des Lernprozesses durch die Verwendung einer Endspieldatenbank deutlich beschleunigt werden, da zumindest sichere Siege im Endspiel nicht durch zufälliges Ziehen in Unentschieden verwandelt werden. Als geeigneter Lernparameterwert erwies sich $\lambda = 0.7$. Die Verwendung des Momentum-Terms brachte bei verschieden gewählten Einstellungen keine nennenswerten Verbesserung. Dies ist im Nachhinein nicht verwunderlich, da sich die Fehlerfläche bei unserer Vorgehensweise ohnehin ständig verändert und lokale Minima damit kein Problem darstellen. Auch für die adaptive Learning-Rate wurden verschiedene Einstellungen getestet. Die im Endeffekt spielstärksten Heuristiken wurden dann mit den Parametern $\kappa = 0.08$ und $\varphi = 0.5$ trainiert.

Lernvorgang mit Hilfe einer perfekten Datenbank

Neben dieser Methode des „Reinforcement Learnings“ wurde von uns auch versucht, eine Heuristik durch „Supervised Learning“ mit Hilfe einer vollständigen Mühledatenbank zu lernen. Um dabei vor allem mit in der Praxis wirklich vorkommenden Spielsituationen zu trainieren, wurden wie beim Lernen ohne Datenbank normale Mühlepartien zwischen zwei Heuristiken mit gleichen Features und gleicher Netzstruktur und gleichen Lernparametern gespielt. Nach jedem Spielzug wurde dann die aktuelle Spielsituation mit dem Neuronalen Netzwerk der Heuristik trainiert, wobei als Teaching-Wert der aus der Datenbank erfragte korrekte Spielwert verwendet wurde.
Insgesamt konnten wir feststellen, dass der Ansatz des „Supervised Learnings“ mit einer Datenbank weniger spielstarke Heuristiken erzeugte als der Ansatz des „Reinforcement Learnings“. Eine der stärksten mit diesem Verfahren gelernten Heuristiken war die Heuristik „SinglePerf“, die im nächsten Abschnitt gegen durch „Reinforcement Learning“ trainierte Heuristiken getestet werden wird.

4.6 Auswertung

4.6.1 Spielstärke verschiedener Heuristiken

Während unserer Testphase haben wir zahlreiche Heuristiken erstellt und einem maschinellen Lernprozess unterzogen. Dabei wurden von uns sowohl verschiedene Features zum Bewerten einer Spielsituation als auch unterschiedliche Strukturen des neuronalen Netzwerkes verwendet. In diesem Kapitel sollen nun einige erfolgreich gelernte Heuristiken hinsichtlich der aus ihnen resultierenden Spielstärke getestet werden. Für die Tests der Spielstärke wurden Spiele zwischen den einzelnen gelernten Heuristiken und einer manuell erstellten Heuristik, einer perfekten Datenbank und menschlichen Spielern durchgeführt.

Getestete Heuristiken

- *„Standard“* ist eine von uns manuell erstellte Heuristik. Sie basiert auf den oben bereits beschriebenen Standard-Features. Als Netzstruktur liegt ein Singlelayer-Netzwerk vor.
- *„Single“* besteht ebenfalls aus einem Singlelayer-Netzwerk mit dem Standard-Feature-Set. Die Gewichtungen wurden wie alle weiteren angeführten Heuristiken durch den Backpropagation-Algorithmus im Spiel gegen äquivalente Heuristiken gelernt.

- *„Combi“* ist eine gelernte Heuristik, die ebenfalls ein Singlelayer-Netzwerk nutzt. Als Eingangsfeatures für das Netzwerk wurden jedoch jeweils Feature-Paare der Standard-Features genutzt. So besitzt das Neuronale Netzwerk dieser Heuristik $39^2 = 1521$ Eingänge. Durch die Kombination der Features wird die Ausdrucksfähigkeit des Netzwerkes erhöht, doch sinkt wegen des größeren Zeitaufwandes die Suchtiefe.

- *„Small“* ist eine gelernte Singlelayer-Heuristik, deren Feature-Set nur aus einem ausgewählten Teil der Standard-Features besteht. Hier wurde auf die zeitaufwendige Bewertung offener und geschlossener Mühlen verzichtet. Dadurch verschlechtert sich zwar zwangsläufig die heuristische Einschätzung, doch vergrößert sich durch die gesparte Zeit die Suchtiefe.

- *„SinglePerf“* besitzt die selbe Struktur wie der „Single“, wurde jedoch ausschließlich mit Spielen gegen eine perfekte Datenbank gelernt.

- *„Multi“* ist die am besten gelernte Multilayer-Heuristik. Sie besteht wie „Single“ aus den 39 Eingängen der Standard-Features und drei Ausgängen, dazwischen aber befindet sich ein Hidden-Layer von 30 Knoten. Auch sie wurde gegen eine äquivalente Heuristik trainiert.

Spielstärke gegen Menschen

Zunächst wurden alle betrachteten Heuristiken von verschiedenen menschlichen Spielern getestet. Gegen menschliche Anfänger gelang dabei allen Heuristiken fast immer ein Sieg. Auch gegen erfahrene und starke Mühlespieler konnten die meisten Heuristiken gut abschneiden, in den ersten Spielen gegen einen solchen starken menschlichen Gegner erzielten die Computerspieler einige Siege und fast keine Niederlagen. Hier wurde von den Testspielern bestätigt, dass die Computerspieler sogar häufiger Partien gewinnen konnten als Spieler, die mit Hilfe einer Datenbank lediglich korrekte Züge auswählten. Allerdings konnten sich die menschlichen Spieler nach einigen Partien auf die Spielweise einer speziellen Heuristik einstellen und gezielte Gewinnstrategien dagegen entwickeln. Hier wurde deutlich, dass jede Heuristik während des Lernprozesses tatsächlich eine eigene charakteristische Spielweise entwickelt hatte, die in verschiedenen Spielsituationen zu stärkeren oder schwächeren Zügen führte. Bei geschickter Ausnutzung der Schwächen einer Heuristik konnten geübte Mühlespieler stets einen Weg zum Sieg finden.
Auf Grund der individuellen Stärken und Schwächen in verschiedenen Situationen war eine Differenzierung der Spielstärken der Heuristiken äußerst schwierig. Subjektiv bildeten sich zwei Gruppen von Spielstärken heraus. Zu den besseren Heuristiken zählten dabei „Standard“, „Multi“, „Single“, zur schwächeren Gruppe gehörten „Small“, „SinglePerf“ und „Combi“.

Spielstärke gegen die manuell erzeugte Standard-Heuristik

Zunächst wurde die Spielstärke im Spiel gegen die Standard-Heuristik betrachtet. Dazu wurden jeweils 3000 Testspiele bei einer Bedenkzeit von $t = 5000ms$ pro Zug auf dem Referenzrechner durchgeführt. Um wiederkehrende identische Spiele zu vermeiden, wurde als Zufallselement zu jeder heuristischen Spieleinschätzung ein Wert zwischen -0.05 und 0.05 hinzuaddiert. In der folgenden Tabelle (4.3) sind die jeweiligen Siege, Remis und Niederlagen der entsprechenden Heuristik dargestellt. In der letzten Spalte ist die relative Häufigkeit p der mehr gewonnenen als verlorenen Spiele gegeben

$$p = \frac{wins - losses}{Gesamtspielzahl}$$

Die Tests gegen den Standard-Spieler bestätigten im Wesentlichen den subjektiven Eindruck aus den Tests gegen Menschen. Deutlich am besten schnitten die beiden Heuristiken „Multi“

Heuristik	Wins	Losses	Remis	p
Multi	1043	350	1697	0.231
Single	1030	495	1476	0.178
Combi	837	730	1434	0.036
Standard	-	-	-	0.000
Small	708	1039	1254	-0.110
SinglePerf	306	1492	1202	-0.395

Tabelle 4.3: Ergebnisse verschiedener Heuristiken gegen die Standard-Heuristik

und „Single“ ab, klarer Verlierer des Tests war die Heuristik „SinglePerf“. Insgesamt gelang jedoch jeder Heuristik mindestens in jedem zehnten Spiel ein Sieg gegen den Standard-Spieler. Beachtlich, angesichts der Tatsache, dass Mühle als unentschieden gelöst ist, aber nicht überraschend war die hohe Zahl an unentschiedenen Spielen.

Spielstärke gegen eine perfekte Datenbank

Anschließend wurden die Heuristiken in je 500 Spielen gegen eine vollständige Mühledatenbank getestet. Den Heuristischen Mühleprogrammen stand dabei wiederum eine Bedenkzeit von t = 5000 ms pro Halbzug auf dem Referenzrechner 2 zur Verfügung, während von der seperat auf einem Server laufenden Datenbank stets ein beliebiger korrekter Zug ausgewählt wurde. Die Ergebnisse der Tests sind in der folgenden Tabelle 4.4 dargestellt. Es zeigt sich

Heuristik	Remis	Losses	p
Multi	424	76	0.847
Single	397	103	0.794
Standard	349	151	0.698
Combi	331	169	0.662
Small	291	209	0.582
SinglePerf	261	239	0.522

Tabelle 4.4: Ergebnisse verschiedener Heuristiken gegen die perfekte Datenbank

ein sehr ähnliches Bild wie im Test gegen den Standard-Spieler. Als bester Spieler zeigte sich hier die Heuristik „Multi“, die immerhin in fast 85% aller Spiele ein Unentschieden gelang. Bei perfektem Spiel der Datenbank und etwa 40 Zügen pro Spiel bedeutet dies, dass dieser Heuristik nur etwa alle 240 Züge ein Fehler beim Ziehen unterläuft.
Schwach zeigten sich die beiden Heuristiken „Small“ und „SinglePerf“.

Turnier verschiedener Heuristiken

Abschließend wurde ein „Turnier“ aller vorgestelleten Heuristiken veranstaltet. Dabei wurden 10000 Partien zwischen jeweils zwei zufällig ausgewählten Spielern durchgeführt, die wiederum pro Zug $t = 5000$ ms auf dem Referenzrechner zur Verfügung hatten.
In der Tabelle 4.5 und in Abbildung 4.13 sind die Ergebnisse dargestellt. Wie oben ist in der letzten Spalte die relative Häufigkeit p der mehr gewonnenen als verlorenen Spiele gegeben. Wiederum erweist sich „Multi“ als Testsieger vor der Heuristik „Single“. Abgeschlagen belegte „SinglePerf“ den letzten Platz.

Heuristik	Wins	Losses	Remis	p
Multi	686	205	722	0.299
Single	621	282	801	0.199
Combi	424	428	891	-0.003
Standard	424	467	763	-0.026
Small	426	588	585	-0.101
SinglePerf	186	796	705	-0.361

Tabelle 4.5: Ergebnisse verschiedener Heuristiken im Turnier

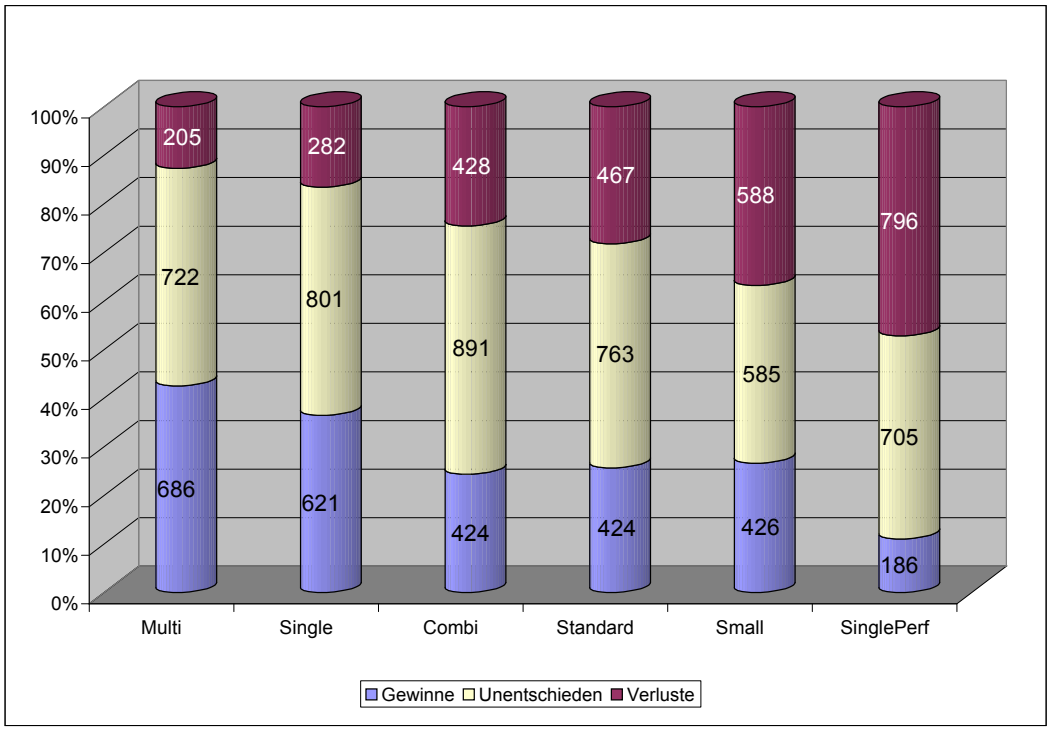


Abbildung 4.13: Ergebnisse verschiedener Heuristiken im Turnier

Zusammenfassung

Insgesamt erwiesen sich alle Test als untereinander stimmig. Sieger wurde „Multi“, gefolgt von „Single“, am schlechtesten schnitt die Heuristik „SinglePerf“ ab. Prinzipiell überrascht der Erfolg des „Multi“ nicht. Durch die komplexere Struktur seines Neuronalen Netzwerks ist es mit dieser Heuristik möglich, auch komplexe Sachverhalte zusammenzufassen und zu bewerten. Zwar ist das Lernen in einem solchen Netzwerk schwieriger und aufwendiger, offensichtlich zahlt sich dieser Aufwand aber letztlich aus. Die Heuristik „Single“ zeigte sich deutlich stärker als der von Hand eingestellte „Standard“. Hier erwies sich das maschinelle Lernen dem Austesten und Variieren des Menschen als überlegen. Den Versuch der Heuristik „Small“, mit einer schnell zu berechnenden, gröberen Heuristik größere Suchtiefen zu erreichen, kann als gescheitert betrachtet werden. Wie sich in den Tests herausstellte, führte die gewonnene Suchtiefe wegen der Genauigkeitsverluste der Heuristik eher zu einer Verschlechterung als zu einer Verbesserung.
Enttäuschend war auch das Abschneiden der Heuristik „Combi“. Die hier verfolgte Idee, durch Kombination je zweier Grundfeatures Ausdruckststärke zu gewinnen und gleichzeitig die einfache Netzwerkstruktur eines Singlelayer-Netzwerks beizubehalten, brachte nicht den gewünschten Erfolg. Möglicherweise wären hier noch zahlreiche weitere Testspiele für eine optimale Einstellung der Gewichte notwendig. Das besonders schlechte Abschneiden des „SinglePerf“ überrascht. Das Lernen exakter Spielwerte brachte offenbar keinen Vorteil. Im Gegenteil führte die mangelnde Differenzierung zwischen Situationen, die bei perfektem Spiel zwar gleichermaßen in Unentschieden münden, aber eben mehr oder weniger Spielraum für Fehler lassen, tatsächlich zu einem Verlust an Spielstärke.

4.6.2 Singlelayer-Heuristiken im Vergleich

Im Folgenden sollen nun einmal die Gewichtungen der einzelnen Features bei verschiedenen Heuristiken betrachtet werden. Eine ausführliche Beschreibung der Feature findet sich weiter oben in dieser Arbeit.

Die Heuristik „Standard“

Die Heuristik „Standard“ wurde von uns im Laufe eines langwierigen Prozesses kreiert. Die einzelnen Features wurden dabei per Hand mit Gewichten belegt und immer wieder nach der subjektiven Spielstärke nachgebessert. Am wichtigsten erschien uns dabei die Steinanzahl und die Anzahl der Zugmöglichkeiten beider Spieler. Nach unserer im Nachhinein naiven Vorgehensweise wurde dabei jeder eigene Stein mit einer positiven Punktzahl belohnt, jeder gegnerische Stein mit gleicher Punktzahl bestraft. Die Bedeutung von offenen und geschlossenen Mühlen wurde zunächst als in gleichem Maße wichtig eingeschätzt, allerdings ergab sich bei einer starken Reduzierung dieser Gewichte subjektiv eine deutliche Steigerung der Spielwerte. Daher sind die Gewichte für diese Features in der Heuristik „Standard“ sehr gering gewählt. Dies erschien uns auch zunächst plausibel, da geschlossene Mühlen bereits zu einer Änderung der Steinzahl geführt haben und offene Mühle bei Vergrößerung der Suchtiefe um nur einen Halbzug oft zu einem Steinvorteil führen und damit beide ohnehin schon maßgeblich in die heuristische Bewertung einfließen.
Die Gewichtungen der einzelnen Features dieser Heuristik sind in Diagramm 4.14 dargestellt.

Die Heuristik „Single“

Wie sich in den obenstehenden Tests gezeigt hat, handelt es sich bei der Heuristik „Single“ um unsere spielstärkste gelernte Heuristik auf Basis eines Singlelayer-Netzwerkes. Bei genauer Betrachtung der einzelnen Gewichte, dargestellt in Diagramm 4.15, fallen einige interessante Unterschiede zur „Standard“-Heuristik auf. Zunächst erkennt man, dass die Gewichtungen für die Steinanzahlen ungleichmäßiger verlaufen als die des „Standard“. Auffällig ist vor allem die Tatsache, dass vier oder fünf eigene Steine negativ und damit schlechter als

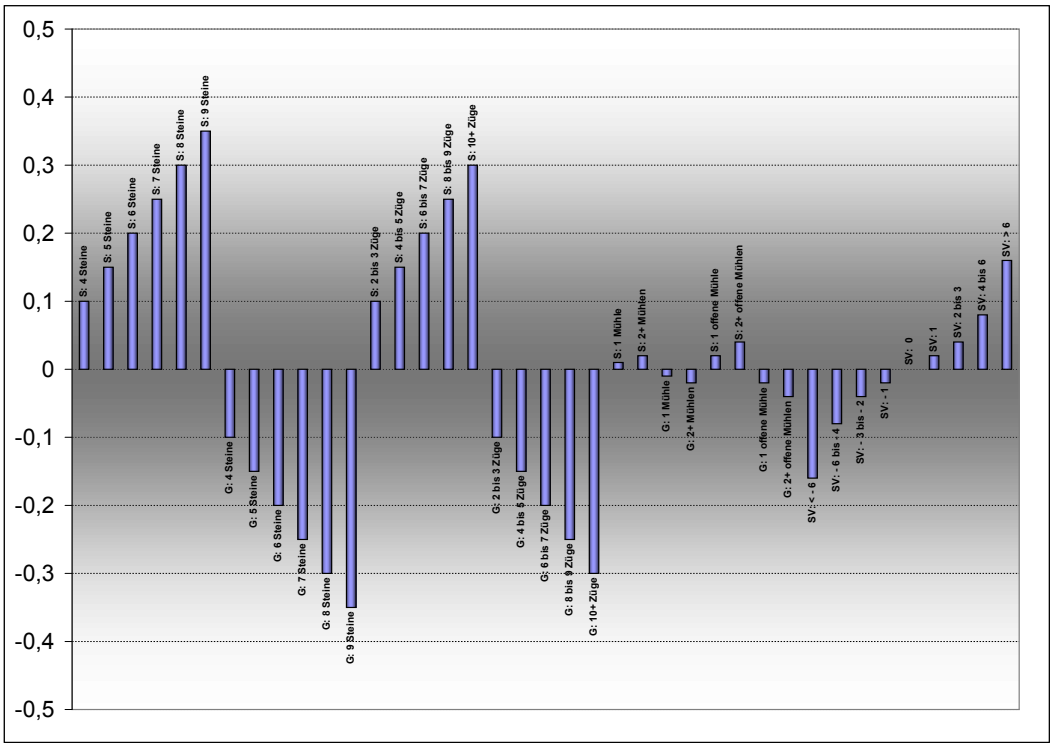


Abbildung 4.14: „Standard“-Heuristik

drei eigene Steine gewichtet sind. Eine solche Gewichtung ist auch nicht verwunderlich, da die Möglichkeit des Springens mit drei Steinen oftmals die Situation des Spielers verbessert. Für die Zugmöglichkeiten und den Situation-Value zeigen die Gewichte im Wesentlichen das gleiche Verhalten wie im „Standard“, die Bedeutung des Situation-Value wird hier jedoch noch deutlich höher eingeschätzt. Im Gegensatz zum „Standard“ werden bei der Heuristik „Single“ geschlossene und noch viel mehr offene Mühlen als entscheidend betrachtet. Dabei ist besonders bemerkenswert, dass eigene offene Mühlen stärker positiv gewichtet sind als die des Gegners negativ. Dies hängt damit zusammen, dass die Heuristik stets für den am Zug befindlichen Spieler aufgerufen wird, der damit noch die Möglichkeit besitzt, gegnerische offene Mühlen zu verbauen oder zu zerstören. Hier wird zum einen deutlich, dass Mühlen offensichtlich doch stark die Bewertung einer Spielsituation beeinflussen, auf der anderen Seite die Gewichtungen der anderen Features geeignet dazu gewählt sein müssen, da es sonst wie bei der Entstehung des „Standard“ zu schlechteren Ergebnissen kommen kann.

Die Heuristik „SinglePerf“

Die Heuristk „SinglePerf“ wurde mit den exakten Spielwerten einer vollständigen Mühle-Datenbank gelernt und schnitt in den obigen Tests sehr schwach ab. Bei Betrachtung der Gewichtungen der einzelnen Features, dargestellt in Diagramm 4.16, verwundert ein solches Abschneiden nicht. Zwar entsprechen die Gewichtungen von Zugmöglichkeiten und Mühlen im Großen und Ganzen denen des „Single“, die Gewichtung der Steinanzahl muss allerdings als chaotisch bezeichnet werden und führt wohl zu den schwachen Testergebnissen. Da sich die ungeordneten Gewichtungen nur auf die Steinanzahlen beschränken, kann man davon ausgehen, dass dies nicht auf eine zu kurze Lernphase zurückzuführen ist.
Eine Ursache für dieses Phänomen liegt möglicherweise darin, dass tatsächlich bei beliebiger Steinanzahl eines Spielers Gewinne, Verluste und Remis in vergleichbaren Verhältnissen von der Datenbank erreicht werden können, das Erreichen eines günstigen Ergebnisses jedoch

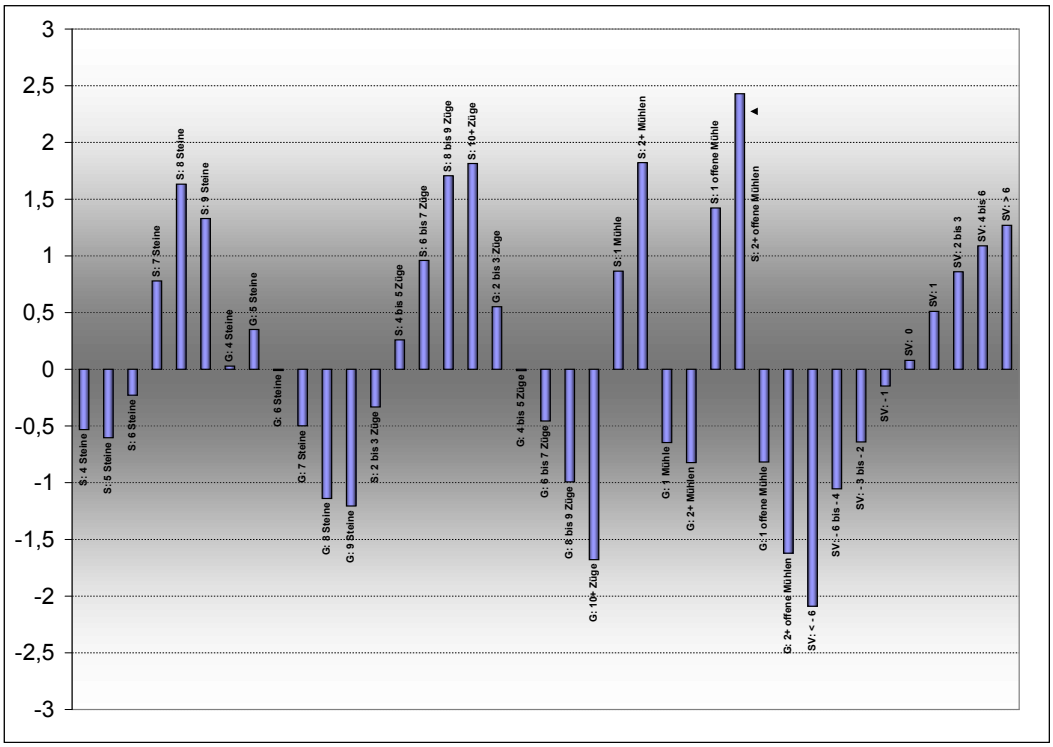


Abbildung 4.15: „Single“-Heuristik

unterschiedlich schwer ist. Geht man also von perfektem Spiel aus, so bildet wohl die Steinanzahl kein aussagekräftiges Kriterium für den Spielausgang. Für einen heuristischen Spieler dagegen spielt die Steinzahl dagegen sehr wohl eine Rolle, da sie eine Auskunft darüber gibt, wie schwer das gewünschte Ergebnis zu erreichen ist, also ob beispielsweise nur ein einziger Zug zum optimalen Ausgang führt, oder ob dieser auch bei fast beliebigem Spiel erreicht werden kann.

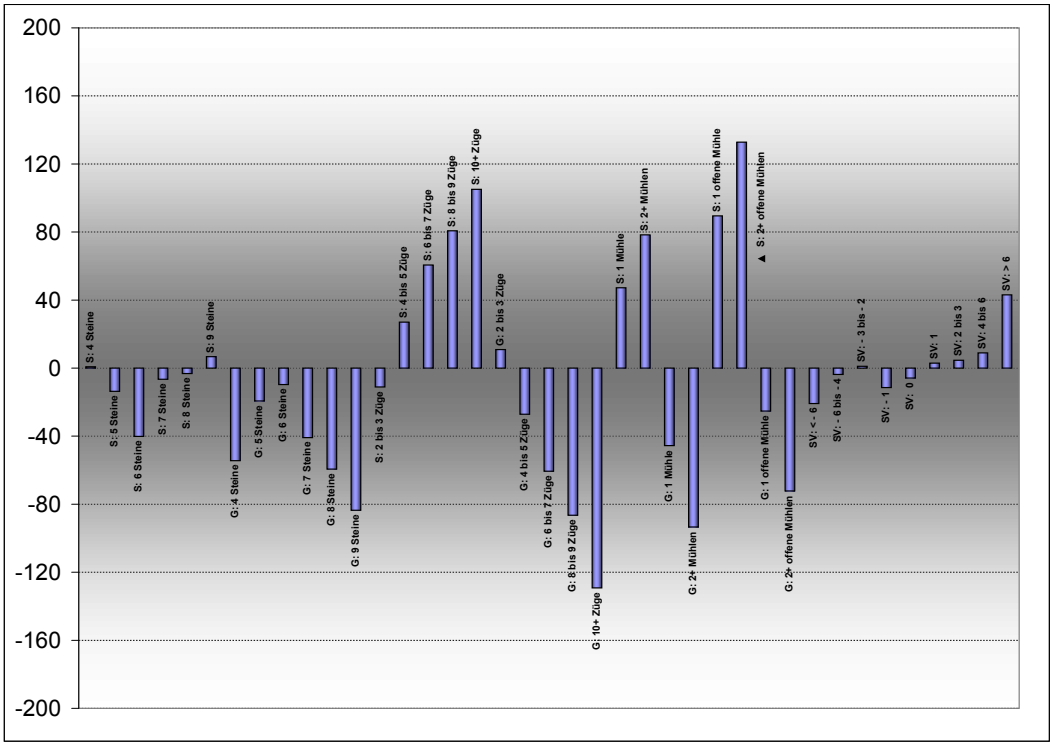


Abbildung 4.16: „SinglePerf“-Heuristik

Kapitel 5

Implementierung

5.1 Spielfeld-Repräsentation und wichtige Operationen beim Mühlespiel

Wie in der gewählten Programmiersprache JAVA üblich, wurde von uns ein objektorientierter Ansatz gewählt. Für häufig verwendete Vorgänge wie etwa dem Ausführen von Zügen auf dem Spielfeld erwies es sich allerdings als wesentlich effizienter, Operationen nicht mit Objekten sondern auf Bitebene durchzuführen. Dazu wurde die Funktionalität der Java-eigenen Bitset-Klasse genutzt. Im Folgenden sollen einige wichtige Elemente der Implementierung des Spielfeldes und wichtiger Operationen kurz beschrieben werden.

Repräsentation des Spielfeldes

Das Spielfeld wird intern durch zwei Bitsets der Länge 24 repräsentiert. In jedem dieser beiden Bitset sind die Spielsteine eines Spielers gespeichert, jede Position in den Bitsets steht für ein Spielfeld auf dem Mühlebrett. Ein gesetztes Bit bedeutet, dass sich an der entsprechenden Stelle des Mühlebrettes ein Stein dieses Spielers befindet. Das Setzen, Schlagen oder Schieben eines Steins kann damit sehr schnell durch das Setzen oder Entfernen von einem beziehungsweise zwei Bits erfolgen.

Erkennen aller legalen Züge

Das Vorgehen beim Erkennen aller legalen Züge unterscheidet sich selbstverständlich maßgeblich von Spielphase zu Spielphase.
Während der Setzphase sind Setzaktionen auf alle freien Felder möglich. Diese können sehr schnell durch logische Operationen über beide Bitsets bestimmt werden. Ein Feld i ist genau dann frei, wenn bei beiden Bitsets das Bit an Position i nicht gesetzt ist.
Sobald ein Spieler nur noch drei Steine besitzt, sind alle legalen Züge ebenfalls leicht zu bestimmen. Die mögliche Zielfelder für die Bewegung werden genauso wie in der Setzphase bestimmt, Ausgangsfelder für den Zug sind die drei vom Spieler besetzten Spielfelder.
Das Finden aller möglichen Schiebezüge für den aktiven Spieler gestaltet sich dagegen aufwendiger: Zu jedem Feld ist im Mühlespiel statisch die Position aller Nachbarfelder gespeichert. Zum Berechnen der Schiebezüge werden nun alle eigenen Steine durchlaufen und auf freie Nachbarn überprüft. Ist ein Nachbarfeld frei, so wird ein Schiebezug von der Position des eigenen Steins zu der des freien Nachbarn zur Liste aller möglichen Züge hinzugefügt.

Erkennen geschlossener Mühlen

Zum Erkennen von geschlossenen Mühlen wurden alle 16 potentiellen Mühlen beim Mühlespiel in einem 24-bit Bitset mit drei gesetzten Bits gespeichert. Wird nun untersucht, ob

ein Spielzug eine Mühle geschlossen hat, so werden die Mühlenmasken der beiden Mühlen verwendet, die das Zielfelder des Spielzuges beinhalten. Auf diese Mühlemasken wird anschließend eine logische AND-Operation mit dem Bitset der Steine des aktiven Spielers angewendet. Sind bei einem der beiden Bitset anschließend noch drei Bits gesetzt, so wurde eine Mühle geschlossen. Ganz ähnlich lassen sich zur Verwendung in einer Heuristik die Anzahl der geschlossenen Mühlen zählen: Hierzu wird auf alle 16 Mühlemasken der AND-Operator mit den eigenen Steinen angewandt und daraufhin abgezählt, bei wievielen Bitset noch immer drei Bits gesetzt sind.

Erkennen offener Mühlen

In vielen von uns verwendeten Heuristiken und für die Ruhesuche wird die Anzahl geöffneter Mühlen benötigt. Diese zu berechnen ist allerdings mit einigem Rechenaufwand verbunden. Am zeiteffektivsten erwies sich die folgende Implementierung: Wie beim Erkennen geschlossener Mühlen wird auf alle Mühlenmasken der logische AND-Operator mit dem Bitset der eigenen Steine angewendet. Eine offene Mühle kann offenbar nur dann vorliegen, wenn im Ergebnis-Bitset genau zwei Bits gesetzt sind. Kann der aktive Spieler noch Steine setzen oder springen, so ist diese Bedingung auch hinreichend für eine offene Mühle. Beim Schieben jedoch muss das freie Feld in der Mühle durch Vergleich mit der ursprünglichen Maske für diese Mühle festgestellt werden. Daraufhin wird bei allen Nachbarn dieses Feldes überprüft, ob sie zugleich von einem eigenen Stein besetzt sind und sich selbst nicht in der Mühle befinden, die gerade betrachtet wird. Treffen beide Bedingungen zu, so wurde auch für die Schiebephase eine offene Mühle gefunden.

Zusammenfassung

Im Vergleich zu einer naiven Implementierung konnte deutlich an Rechenzeit eingespart werden. Als wichtigste Punkte bei der Verbesserung der Implementierung stellten sich die Verwendung von logischen Operationen für die Repräsentation und Berechnung von Spielzügen und Mühlen auf dem Spielfeld und die Speicherungen von möglichst vielen Informationen - etwa der Position von Mühlen und der Nachbarn eines Feldes - heraus.

5.2 UML-Diagramme

Im Folgenden werden wichtige Aspekte unserer Implementierung durch UML-Diagramme dargestellt. Im Sequenzdiagramm (Abbildung: 5.1) ist die Ablaufstruktur eines kompletten Mühlespiels zwischen zwei Computerspielern abgebildet. Die beiden Klassendiagramme (Abbildungen 5.2 und 5.3) zeigen den Aufbau und Zusammenhang wichtiger Programmteile.

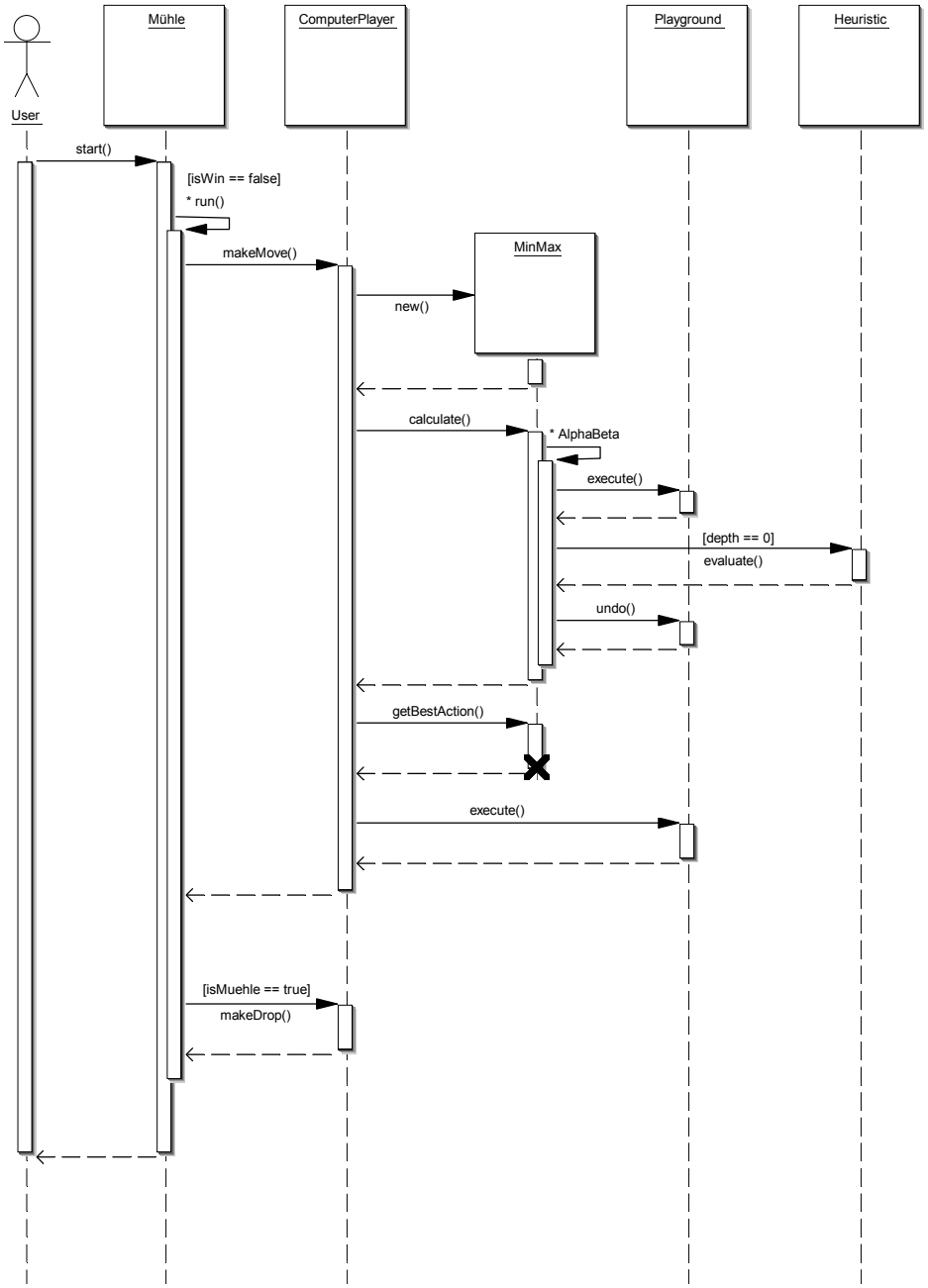


Abbildung 5.1: UML-Sequenzdiagramm

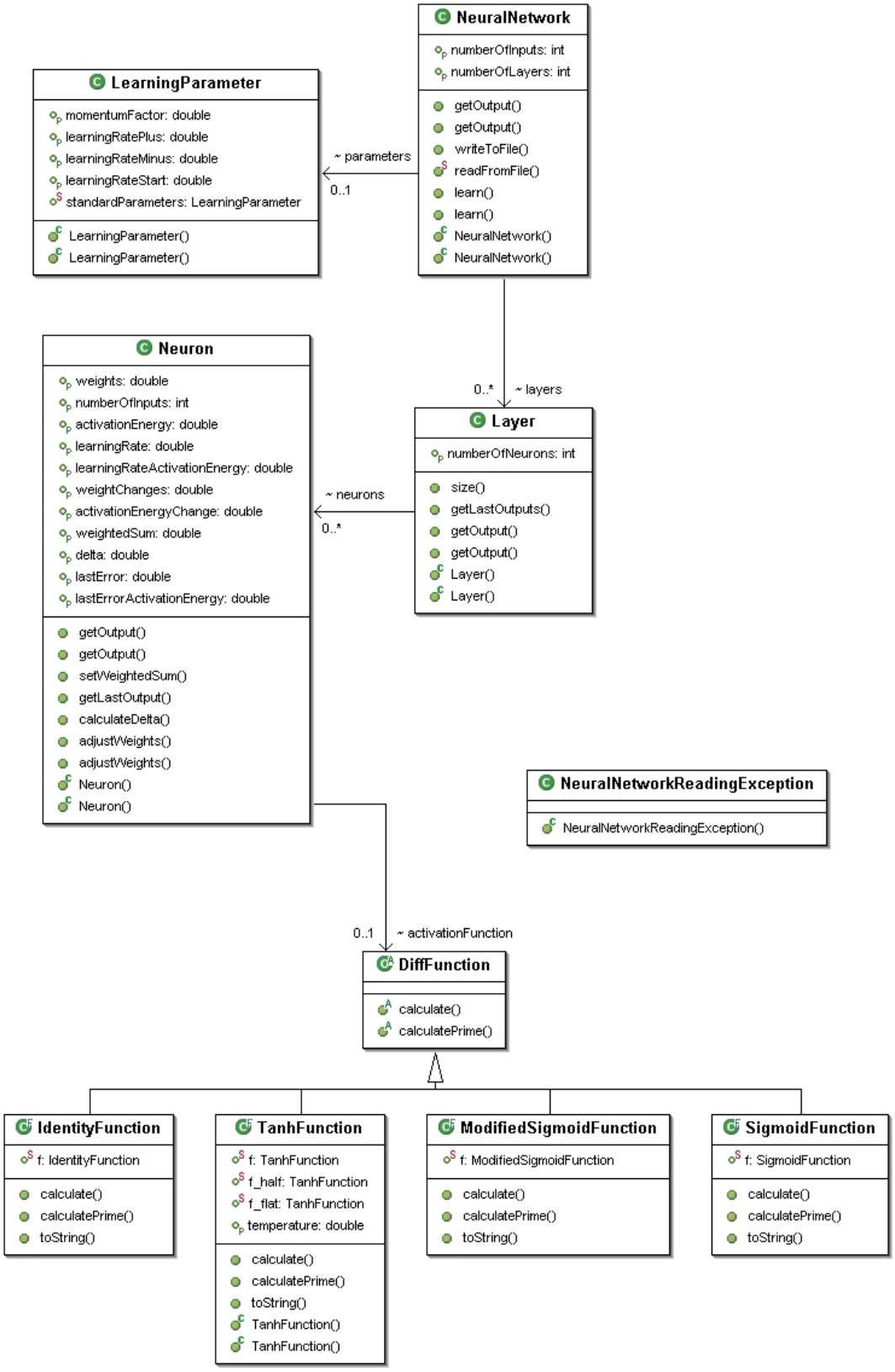


Abbildung 5.2: UML-Klassendiagramm des Neuronalen Netzes

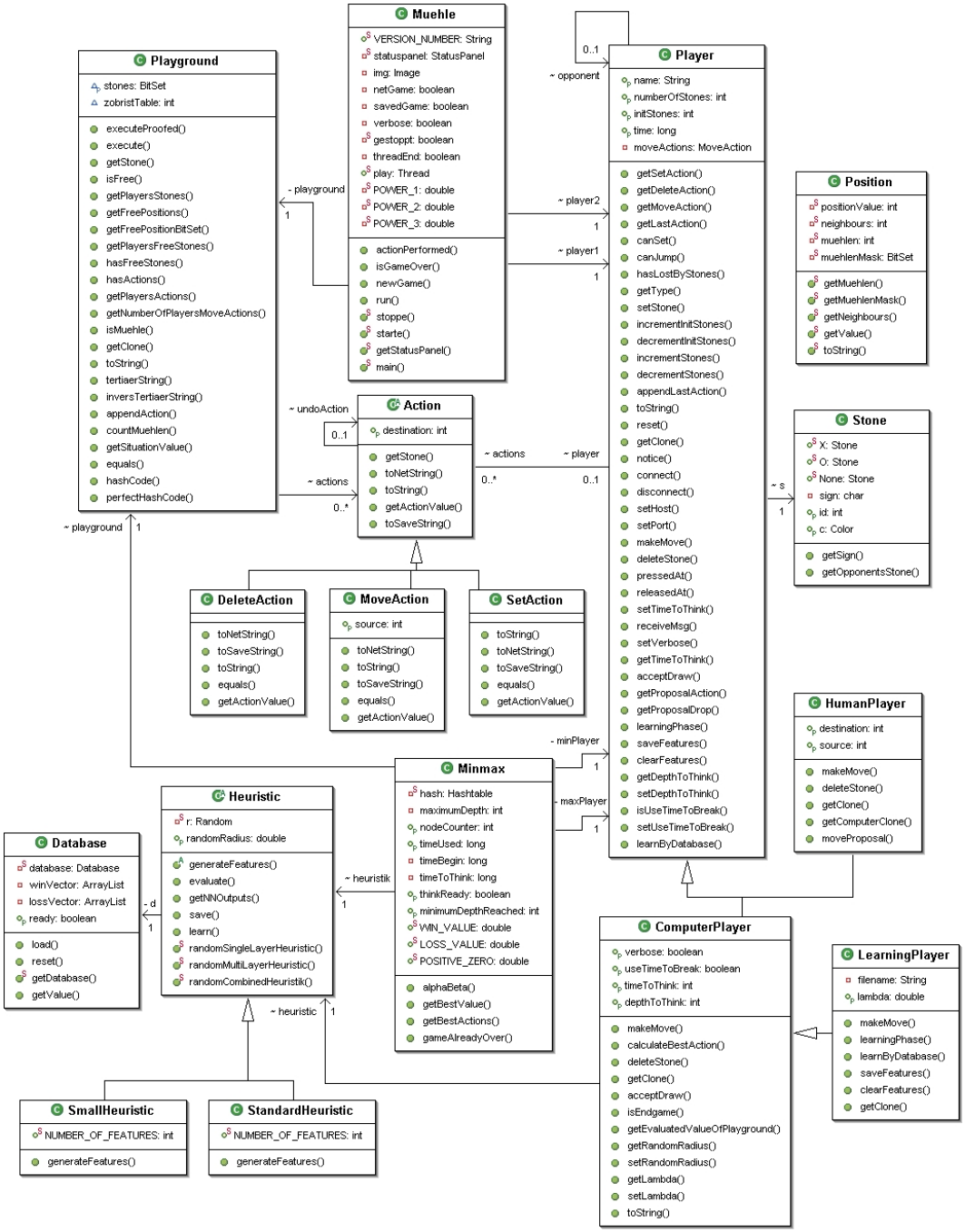


Abbildung 5.3: UML-Klassendiagramm der Muehle-Applikation

Kapitel 6

Bedienungsanleitung

6.1 Installation

Unser Mühleprogramm verwendet die Programmiersprache JAVA. Zum Ausführen des Programms wird somit eine JAVA Runtime Enviroment ab Version 1.4 benötigt. Diese lässt sich zum Beispiel auf der Webeseite „http://java.sun.com“ herunterladen und muss zuerst installiert werden. Nähere Instruktionen zur Installation von JAVA finden sich ebenfalls auf dieser Seite.
Die Mühle-Software selbst liegt gepackt in Form einer .zip-Datei vor. Zur Installation muss diese Datei lediglich an einem beliebigen Platz auf der Festplatte entpackt werden. Das entpackte Archiv sollte dann die Dateien „muehle.exe“ und „muehle.jar“ enthalten, sowie vier weitere Ordner mit der Endspieldatenbank (falls heruntergeladen), den Heuristiken, einen für gespeicherte Spiele sowie einen für weitere Ressourcen.

6.2 Programmstart

Zum Start des Mühlespiels bieten sich mehrere Möglichkeiten.
Unter dem Betriebssystem Windows lässt sich das Programm durch Öffnen der Datei „muehle.exe“ starten. Bei anderen Betriebssystemen (wie zum Beispiel Linux) muss das Spiel wie folgt aus einer bash per Kommandozeile gestartet werden:

```
bash:/> java -jar muehle.jar
```

Bei einem derartigen Aufruf per Kommandozeile hat man noch die Option, den „Verbose-Mode“ einzuschalten, der die aktuelle Spieleinschätzung eines aktiven Computer-Spielers, der sich am Zug befindet, zurückgibt. Dieser Modus lässt sich mit dem Parameter „-v“ aktivieren:

```
bash:/> java -jar muehle.jar -v
```

Die Bedienung des Spiels wird weiter unten beschrieben.

Neben dem eigentlichen Spiel ist es auch möglich eine Lern- und Testumgebung aufzurufen, die zwei oder mehrere (selbst erstellte) Heuristiken lernen und testen lassen kann. Diese wird ausschließlich per Kommandozeile aufgerufen.

```
bash:/> java -cp muehle.jar muehle.LernUmgebung [-options] heuristic_1
 [-options] heuristic_2 ...
```

Folgende Optionen (Tabelle 6.1) können hierzu verwendet werden:

-v	setzt den Verbose-Mode, der die Spieleinschätzung des lernenden oder zu testenden Spielers ausgibt
-test	folgende Heuristiken werden nur getestet und lernen nicht
-d=xx	setzt die Denktiefe auf den festen Wert xx. Bis zu dieser Ebene wird unabhängig von der Zeit gedacht. Standard ist d=6
-t=xxxx	setzt die Denkzeit auf xx Millisekunden, die dem Computer-Spieler für einen Zug gegeben wird. Standard ist t=0
-r=x.xx	setzt einen Random-Radius, bis zu dem der eigentliche Spielwert zufällig verändert werden kann. Dies dient dazu, dass nicht immer wieder ein komplett identisches Spiel entsteht. Standard ist r=0.05
-l=x.xx	setzt den Lernparameter λ. Wie dieser Parameter zu wählen ist, wird im Kapitel „Lernen beim Mühlespiel“ beschrieben.
-q=xx	setzt die zusätzliche Denktiefe der Ruhesuchesuche auf den Wert xx. Standard ist q=0

Tabelle 6.1: Optionen zum Aufruf der Lern- und Testumgebung

Folgender Beispielaufruf lässt einen Computerspieler mit der Heuristik „max.xml“ im Heuristikordner Partien gegen einen Computerspieler mit der Heuristik „joe.xml“ im Heuristikordner spielen. Die Denkzeit des ersten Spielers darf 5 Sekunden betragen, der zweite Spieler soll immer 6 Ebenen weit denken. Beide Heuristiken werden nach jedem Spiel mit dem Lernparameter $\lambda = 0.7$ gelernt. Die Quiescent Search ist ausgeschaltet und als Random Radius wird r=0.1 verwendet.

```
bash:/> java -cp muehle.jar muehle.LernUmgebung -r=0.1 -l=0.7 -q=0
 -t=5000 max -d=6 joe
```

Ein Aufruf für das Testen dieser zwei Heuristiken nach der Lernphase könnte dann so aussehen:

```
bash:/> java -cp muehle.jar muehle.LernUmgebung -test -r=0 -t=5000 max joe
```

Der Output der Lernumgebung umfasst jeweils die Gewinne, die Verluste und die Unentschieden pro Spieler in folgender Form:

```
...
***********************************
max: Won: 153  Lost: 228  Remis: 133
joe: Won: 228  Lost: 153  Remis: 133
***********************************
max: Won: 154  Lost: 228  Remis: 133
joe: Won: 228  Lost: 154  Remis: 133
***********************************
```

Der Output beim Einschalten des Verbose-Mode sieht folgendermaßen aus:

```
*****************************************************************
Ebenen: 10   Maximal: 10
Zeit: 3484 ms
Anzahl beste: 1
Knoten durchsucht: 149057
Spieleinsch"atzung: 0.047893886778407
NetworkOutput: [0.6130141761393355], [0.6318124177361016], [0.5]
*****************************************************************
```

Abzulesen sind die Anzahl der gedachten Ebenen, die Anzahl der maximalen Ebenen, die durch Verwendung der Ruhesuche erreicht wurden, und die Denkzeit für den letzten Zug. Außerdem wird die Anzahl der als gleichstark berechneten besten Züge ausgegeben. Bei mehreren besten Zügen wird ein zufälliger gewählt. Eine weitere Information ist die Anzahl der durchsuchten Knoten im Spielbaum. Schließlich wird die Spieleinschätzung der Partie nach einer Minmaxsuche mit der angegebenen Anzahl an Ebenen und die direkte Heuristische Einschätzung der aktuellen Spielsituation durch das Neuronale Netzwerk ohne Minmax-Suche ausgegeben.

6.3 Spielbedienung

Nach dem Starten des Mühle-Programms sieht man im Programmfenster drei Bereiche: Der Spielbereich mit dem Mühlebrett, der Status-Bereich (unten). Hier werden die vergangenen Züge angezeigt, Systemmeldungen ausgegeben und bei Netzwerkspielen Nachrichten des Partners angezeigt. Der Bereich rechts am Rand gibt Auskunft, welcher Spieler an der Reihe ist und wieviele Steine die Spieler noch setzen können beziehungsweise auf dem Spielfeld haben. Folgender Screenshot zeigt das Programmfenster nach dem Start (Abbildung 6.1).

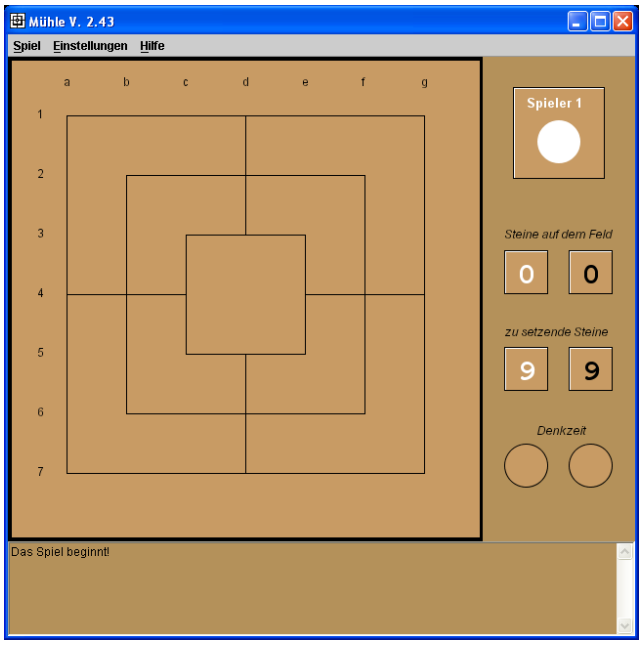


Abbildung 6.1: Screenshot: Spielfenster

Nach Programmstart beginnt automatisch ein Standard-Spiel gegen einen voreingestellten Computer-Spieler. Man kann nun also entweder ein Standardspiel beginnen oder aber vor dem Spiel noch einige Einstellungen festlegen, wie zum Beispiel den eigenen Namen, die Denkzeit und Heuristik des Gegners und weitere Einstellungen. Hierzu wählt man aus dem Menü-Eintrag „Einstellungen“ und den Menüpunkt „Spieler“.
In dem erscheinenden Dialog hat man nun zahlreiche Möglichkeiten das Spiel zu konfigurieren. Der Dialog ist unterteilt in zwei Bereiche, den linken für den ersten Spieler, den rechten für den zweiten Spieler. Der erste Spieler besitzt die weißen Steine und beginnt das Spiel. Für jeden Spieler kann man einen Namen eingeben und wählen, ob ein Mensch, ein gewöhnlicher Computerspieler oder ein lernender Computerspieler spielen soll. Im Falle eines (lernenden) Computerspielers lässt sich desweiteren entweder eine feste Denktiefe oder eine feste Denkzeit festlegen, die der Computer für einen Zug verwenden darf. Auch die Tiefe der Ruhesuche

lässt sich hier einstellen, wobei „0“ die Ruhesuche ausschaltet. Für Computer-Spieler lässt sich dazu optional eine Heuristik auswählen, die anstatt der Standard-Heuristik verwendet werden soll. Im Falle eines lernenden Computerspielers ist die Angabe einer Heuristik obligatorisch. Existiert eine angegebene Heuristik nicht, wird eine zufällige neue Heuristik mit entsprechendem Namen erstellt. Ein Screenshot dieses Dialogs ist in Abbildung 6.2 dargestellt.

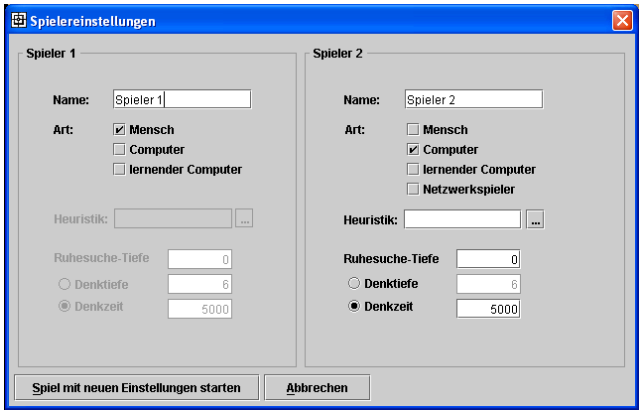


Abbildung 6.2: Screenshot: Spieler Dialogfenster

Für den zweiten Spieler ist zusätzlich die Option „Netzwerkspieler“ möglich. Diese kann verwendet werden, um im Netzwerk oder über Internet Partien gegen einen anderen Menschen, der sich nicht am selben Computer befindet, zu spielen. In diesem Fall muss man sich entscheiden, ob man für das Spiel die Funktion des Servers übernehmen will, zu dem sich ein Spielgast (Client) verbinden kann, oder man sich selbst als Client zu einem anderen Server verbinden möchte. Im Falle des Serverbetriebs muss man noch einen beliebigen Port wählen, auf dem die Verbindung aufgebaut werden soll. Der Standard-Port liegt auf 9000. Im Clientbetrieb muss zu der Portnummer noch die Host-Adresse (IP-Adresse oder DNS-Name) des Servers eingetragen werden. Abbildung 6.3 zeigt diesen Zusammenhang.

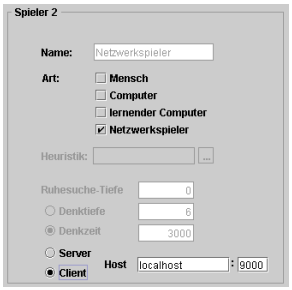


Abbildung 6.3: Screenshot: Netzwerk Dialogfenster

Durch einen Klick auf den Button „Spiel starten“ wird ein neues Spiel mit den hier getroffenen Einstellungen gestartet. Die Spielstäre eines Computer-Spielers, welche natürlich auch schon von der festegelegten Denkzeit abhängt, lässt sich unabhängig davon unter dem Menüpunkt „Einstellungen → Spielstärke“ noch auf drei Stufen einstellen: Anfänger, Fortgeschrittener und Profi.

Der Spielablauf selbst ist intuitiv. Wenn man an der Reihe ist, wird durch das Klicken mit der linken Maustaste auf ein freies Spielfeld ein eigener Stein dorthin gesetzt. Wenn durch

das Setzen eines eigenen Steines eine Mühle geschlossen wird, kann durch Klicken auf einen Stein des Gegeners dieser aus dem Spiel genommen werden. Die jeweils letzte Aktion wird zur besseren Orientierung auf dem Spielbrett farbig markiert. Diese Einstellung lässt sich unter dem Menüpunkt „Einstellungen“ ändern. Wenn die Setzphase vorüber ist, kann kein Stein mehr gesetzt werden. Nun wird per Drag&Drop ein eigener Stein auf ein benachbartes Feld geschoben. Das Schmeißen eines gegnerischen Steines geschieht weiterhin so wie oben beschrieben. Im Endspiel (nur noch drei eigene Steine) kann ein eigener Stein entsprechend den Regeln per Drag&Drop auf ein beliebiges freies Feld gezogen werden.
Während der Partie besteht außerdem die Möglichkeit sich einen Zugvorschlag einzuholen. Dieser wird auf Basis derselben Heuristik erstellt, die der entsprechende Gegner verwendet, und bekommt maximal 5 Sekunden Zeit zur Berechnung. Beim Spiel Mensch gegen Mensch und im Netzwerkspiel ist ein Zugvorschlag nicht möglich.
Schließlich besteht noch die Möglichkeit dem Mitspieler ein Remis anzubieten oder die Partie aufzugeben. Beide Optionen befinden sich unter dem Menüpunkt „Spiel“. Dort befindet sich auch die Option „Neues Spiel“, die eine neue Partie mit denselben Einstellungen beginnt. Bei Netzwerkspielen darf so lange kein neues Spiel begonnen werden, bis das aktuelle Spiel beendet ist, sei es durch reguläres Ende der Partie oder durch Aufgabe eines Spielers.
Abbildung 6.4 zeigt einen Screenshot mitten aus einem Spiel. Die grüne Markierung bedeutet, dass der schwarze Spieler von Feld a1 auf Feld a4 gezogen ist. Durch das Schließen der Mühle darf schwarz also einen weißen Stein aus dem Feld nehmen. Die rote Markierung zeigt, dass sich der Spieler für den Stein auf Feld b6 entschieden hat. Diesen Sachverhalt gibt auch die Ausgabe im Statusbereich zurück: „Zug von Spieler2: a1-a4 xb6“.

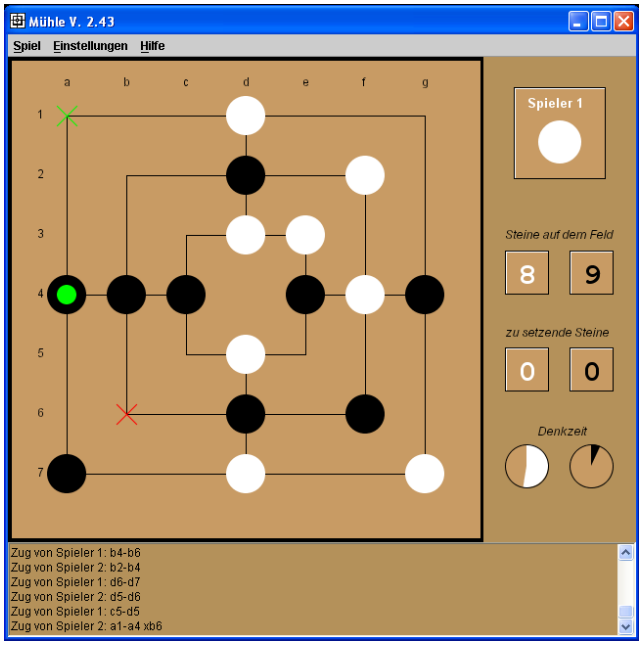


Abbildung 6.4: Screenshot: Beispielpartie

Unser Mühleprogramm erlaubt auch das Speichern und Laden von Partien, sowie das Drucken des aktuellen Spielbretts mit den Aktionen, die zu diesem geführt haben, zu jedem Zeitpunkt der Partie. Diese Optionen sind ebenfalls unter dem Menüpunkt „Spiel“ aufzurufen. Ein Beispielausdruck einer Partie ist in Abbildung 6.5 zu sehen.

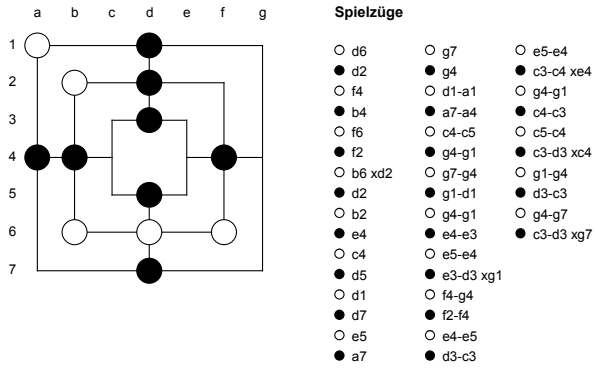


Abbildung 6.5: Beispielausdruck aus einer Mühlepartie

Zu guter Letzt lassen sich mit dem Dialog „Heuristik erstellen“ unter dem Menüpunkt „Einstellungen“ eigene Heuristiken anlegen, die dann im Mühlespiel oder mit der Lernumgebung trainiert werden können. Folgende Einstellungen sind hier möglich:
Layer bezeichnet die Anzahl der Layer aus denen das Neuronale Netzwerk bestehen soll. Dabei bezeichnet ein Layer ein Single-Layer-Netzwerk. Bei mehr als einem Layer wird ein Multi-Layer-Netzwerk mit entsprechender Anzhahl an Hidden-Layern erstellt. Unter Features kann man das Feature-Set wählen, das zu heuristischen Einschätzung herangezogen werden soll. Die Beschreibung der verschieden Feature-Sets sind in der Ausarbeitung unter dem Kapitel „Heuristiken im Spielbaum“ nachzulesen. Für jeden Layer lässt sich nun noch die Anzahl der Knoten und der Typ der Aktiverungsfunktion festlegen. Für den ersten Layer gibt es allerdings stets drei Knoten, da sie den Output des Neuronalen Netzwerkes, also Gewinn-, Verlust- und Remis-Anteil bei der Einschätzung einer Spielsituation, darstellen. Darüber hinaus können hier auch die weiteren Parameter für den Lernvorgang im Neuronalen Netzwerk (siehe Kapitel „Lernen im Multilayer Neural Networks“) gesetzt werden. Abbildung 6.6 zeigt diesen Dialog.

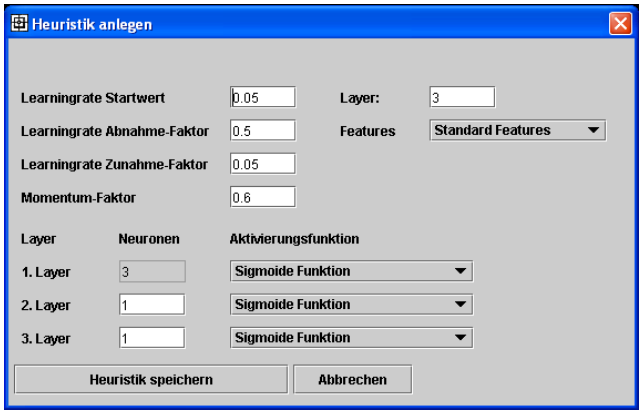


Abbildung 6.6: Screenshot: „Heuristik erstellen“ Dialogfenster

Kapitel 7

Zusammenfassung

Abschließend zu dieser Arbeit können wir sagen, dass die meisten auch in der Literatur vorgeschlagenen Methoden, die Suche im Minmax-Baum zu optimieren, wirklich Verbesserungen brachten. Lediglich die Quiscent Search, die im Schach gute Ergebnisse erzielt, konnte beim Mühlespiel nicht überzeugen.
Insgesamt lässt sich festhalten, dass eine gute und schnell berechenbare Heurstik ebenso wichtig ist wie die Denktiefe, die durch die beschriebenen Minmax-Verbesserungen erhöht wird. Denn wie aus den durchgeführten Tests ersichtlich ist, genügt oftmals nur eine einzige zusätzliche Ebene in der Denktiefe, um gegen einen Spieler mit gleicher Heuristik zu gewinnen.
Die spielstärkste von uns geschaffene Heuristik basiert auf einem Multilayer-Netzwerk. Dies bestätigte unsere Annahme, dass sich mit einem Singlelayer-Netzwerk die Lernfunktion nicht vollständig ausdrücken lässt. So haben wir mit der Version 2.43 unseres Mühleprogramms in Verbindung mit der Heuristik „Multi“ ein spielstarkes Computerprogramm entwickelt, das das Spielniveau eines starken menschlichen Spielers erreicht und auch gegen eine perfekte Datenbank gute Ergebnisse erzielt.

Literaturverzeichnis

Allgemeines

[1] S. Russel / P.Norvig: Artificial Intelligence. A Modern Approach, 2. Auflage, New Jersey, 2003.

[2] S. Russel / P.Norvig: AIMA Slides, 1998.
http://www.it.lut.fi/kurssit/02-03/010595002/block_b6/block_b6_4.pdf.

alpha-beta und seine Verbesserungen

[3] J. Schaeffer: The History Heuristic and Alpha-Beta Search Enhancements in Practice. in: IEEE Transactions on Pattern Analysis and Machine Intelligence 11, 11, 1989, S. 1203-1212. http://citeseer.ist.psu.edu/schaeffer89history.html.

[4] P. Pantel: Intelligent Adversary Searches, University of Manitoba, 1997.
http://citeseer.ist.psu.edu/pantel97intelligent.html.

[5] http://www.brucemo.com/compchess/programming/index.htm, Stand November 2004.

Mühledatenbank

[6] R. Gasser: Applying Retrograde Analysis to Nine Men's Morris. in: Heuristic Programming in Artificial Intelligence 2, 1990, S. 161-173.
http://citeseer.ist.psu.edu/gasser90applying.html.

[7] Gasser, R: Solving Nine Men's Morris. in: Computational Intelligence 12(1),S. 24-41, 1996. http://citeseer.ist.psu.edu/gasser96solving.html.

[8] T. Lincke: Perfect Play using Nine Men s Morris as an example, Diploma thesis, Eidgenössische Technische Hochschule, Zürich, Schweiz, 1994. http://citeseer.ist.psu.edu/lincke94perfect.html.

[9] A. L. Samuel: Some Studies in Machine Learning Using the Game of Checkers, IBM Journal of Research and Development 3, S. 210-229.

[10] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, 1998. http://www.cs.ualberta.ca/ sutton/book/the-book.html.

[11] Sutton, R.S.: Learning to predict by the methods of temporal differences. An Introduction. in: Machine Learning 3: S. 9 - 44., MIT Press, 1988.

[12] R. Levinson, R. Weber: Chess Neighborhoods, Function Combination, and Reinforcement Learning, Lecture Notes In Computer Science, Revised Papers from the Second International Conference on Computers and Games, 2000, S. 133 - 150, 2000.

[13] G. Tesauro: Temporal Difference Learning and TD-Gammon, originally published in Communications of the ACM, March 1995 / Vol. 38, No. 3, 1992.
http://www.research.ibm.com/massive/tdl.html.

[14] S. Thrun: Learning to play the game of chess, Advances in Neural Information Processing Systems (NIPS) 7, MIT Press, 1995.

[15] Zell, A., Simulation Neuronaler Netze, Bonn, 1994.

Abbildungsverzeichnis

Die folgenden Abbildungen wurden aus der Literatur entnommen:

Abbildung 2.1 aus [1], S. 164.
Abbildung 2.2 aus [1], S. 168.
Abbildung 3.1 aus [7], S. 26.
Abbildung 4.2 aus [1], S. 737.
Abbildung 4.3 aus [2], S. 9.
Abbildung 4.4 aus [1], S. 745.
Abbildung 4.8 aus [15], S. 113.
Abbildung 4.9 aus [15], S. 113.

Die übrigen Abbildungen wurden selbst erstellt.