Ein allgemeiner Ansatz zur effizienten Generierung von Datenbanken für perfektes Spiel

Diplomarbeit
von
Benjamin Bahnsen

14. April 2005

Betreuer: Prof. Dr. Michael Hanus
Dr. Friedemann Simon

Institut für Informatik und Praktische Mathematik der
Christian Albrechts Universität zu Kiel

Inhaltsverzeichnis

1 Einleitung

Seit den Anfängen des Computers versuchen die Programmierer den Maschinen das Spielen beizubringen. Wegen der begrenzten Leistung dieser Computer waren die Ergebnisse lange niederschmetternd. Bis Anfang der achtziger Jahre galten künstliche Spieler als äußerst schwach und leicht auszutricksen. Doch mit der rasanten Entwicklung der Hardware in den letzten zwei Jahrzehnten hat sich das Blatt gewendet. Bei fast allen populären Brettspielen haben die Computer die Überhand gewonnen. Mühle- und 4-Gewinnt-Programme sind seit Jahren unbezwingbar, der offizielle Weltmeister im Dame-Spiel ist ein Computer, Othello-Meisterspieler weigern sich geschlossen gegen Computer anzutreten und auch von einer Überlegenheit der Menschen beim Schach kann längst keine Rede mehr sein. Keinen der Wettkämpfe "Mensch gegen Maschine", die nach Kasparovs Niederlage gegen *Deep Blue* im Jahre 1997 stattfanden, konnte der Mensch für sich entscheiden. Im Jahre 2002 rang Weltmeister Kramnik *Deep Fritz*[1] ein 3 : 3 ab. Ein Jahr später trat Kasparov gegen *Deep Junior* und *Deep Fritz* an. Die Wettkämpfe endeten 3 : 3 und 2 : 2. Eine klare Niederlage mussten die drei Großmeister Ponomariov, Topalov und Karjakin in einem Mannschaftswettkampf im Oktober 2004 hinnehmen. Sie verloren gegen drei Computerprogramme mit insgesamt 3,5 : 8,5. Das letzte populäre Brettspiel, bei dem künstliche Spieler chancenlos sind, ist Go. Noch nie konnte ein Go-Programm einen asiatischen Meisterspieler bezwingen.

Oft ist es kein spielspezifisches Wissen, das hinter den Erfolgen steckt. So verfügt jeder Hobby-Schachspieler über mehr schachliches Wissen als die Computerprogramme, gegen die er verliert. Doch rohe Rechengewalt lässt diese Programme immer vorausschauender spielen, so dass sie kein Zug mehr überraschen kann. Perfekte Spieler auf Basis von Datenbanken treiben diesen "Brute Force"-Ansatz auf die Spitze. Sie verfügen über gar kein Wissen mehr. Während sich die Züge eines Schachprogramms durch das integrierte Wissen für die Stellungsbewertung noch nachvollziehen lassen, gibt es für den Zug eines perfekten Spielers keine zufriedenstellende Erklärung mehr. Er wird durch wenige Zugriffe auf die Datenbank ermittelt, deren Generierung lange zurückliegt.

Diese Form der künstlichen Spieler ist nicht unumstritten. Christian Donninger, Autor des Schachprogramms *Nimzo* und Dozent an der Universität Linz, äußert sich skeptisch gegenüber den Ambitionen, perfekte Spieler auf Basis von Datenbanken zu bauen. Seiner Meinung nach ist "... ein gelöstes Spiel uninteressant und damit tot. Es taugt bestenfalls nur mehr als Kinderspiel." ([Don99]). Ähnlich denkt Allis, der sich in [AllHer91] bereits im Titel die Frage stellt: "Which games will survive?" Er betrachtet verschiedene Spiele und prüft, ob in naher Zukunft ein perfekter Spieler zu erwarteten ist oder nicht. Obwohl Schach aufgrund seiner Komplexität nicht vom "Aussterben" bedroht ist, sind zahlreiche Spieler besorgt über die Entwicklung der Endspieldatenbanken. Der dänische Großmeister Curt Hansen fordert in einem offenen Brief ([Han93]) an das Schachmagazin *New in Chess* die "Zerstörung des Schachspiels" aufzuhalten. Er stellt die Frage, ob "... Schach in Zukunft nur das Auswendiglernen exakter Antworten zu verschiedenen Fragen ohne den Einsatz von Kreativität ..." sein wird. Die Befürchtungen der Kritiker von Datenbanken für perfektes Spiel sind nicht unbegründet. So haben Mühle und Dame durch erschöpfende Computeranalysen einen Teil ihrer Faszination verloren. Nichtsdestotrotz bieten diese Datenbanken auch Chancen, da sie viel zum Verständnis der Spiele beigetragen haben. Insbesondere beim Schach haben sich viele theoretische Abhandlungen als falsch erwiesen und konnten korrigiert werden. Der englische Großmeister John Nunn befasst sich schon länger mit der Extrahierung von Wissen aus den Endspieldatenbanken des Schachspiels ([Nun99], [Nun02]). Dies scheint auf

[1] Fritz gehört, wie auch Junior, zu den stärksten PC-Programmen. Bei der "Deep"-Variante handelt es sich um spezielle Versionen der Programme für den Multiprozessorbetrieb.

den ersten Blick eine paradoxe Aufgabe zu sein, da die Endspieldatenbanken bereits vollständiges Wissen enthalten. Leider ist dieses Wissen in einer für den Menschen schwer verständlichen Form. Das Ableiten einfacher Regeln aus der Fülle von Datenbankwissen erweist sich als sehr schwierig. Nunn steht den Endspieldatenbanken positiv gegenüber. Er sieht selbst in der erschöpfenden Analyse des Schachspiels keine Gefahr. In [SteFri95] wird er zitiert mit den Worten: "Die sportliche Seite des Schachs würde fast unverändert bleiben." Diese These wird durch die Tatsache bekräftigt, dass auch noch heute 4-Gewinnt oder Mühle gespielt wird, obwohl diese Spiele "tot" sind.

Bisherige Ansätze zum Bau eines perfekten Spielers beschränkten sich stets auf ein bestimmtes Spiel. Die größte Schwierigkeit hierbei stellt die platzsparende Speicherung der Datenbank dar, die großen Einfluss auf die Generierungsdauer hat. Zahlreiche Publikationen setzen sich allein mit dieser Problematik auseinander, allerdings immer mit Bezug auf ein einzelnes Spiel. In dieser Arbeit stelle ich einen Ansatz vor, der den Bau eines perfekten Spielers sehr vereinfachen soll. Das entstandene Programm übernimmt sämtliche Datenbankfunktionen und beinhaltet einen Algorithmus zur Generierung aller für perfektes Spiel notwendigen Informationen. Lediglich die Eingabe der Zugregeln des Spiels sowie einige Daten zum Aufbau des Spielbretts und der Spielsteine sind noch notwendig, um einen perfekten Spieler zu erschaffen. Die Idee zu dieser Arbeit entstand bei meiner Tätigkeit als wissenschaftliche Hilfskraft. Die Semesteraufgabe der von mir betreuten Vorlesung bestand im Bau eines künstlichen Spielers für das Spiel *Dodgem*. Ich stellte nach Ende des Semesters fest, dass es aufgrund der geringen Komplexität des Spiels sehr viel einfacher war, einen perfekten Spieler zu bauen, als einen, der auf Suchalgorithmen basiert. Mit der Kenntnis von perfekten Spielern zu anderen Brettspielen ist die Idee eines allgemeinen Ansatzes entstanden.

Das Spiel aus Datenbanken ist einfach und im Sinne der künstlichen Intelligenz perfekt, doch es kann nicht immer überzeugen. Aufgrund fehlender Grundlagen gibt es auch hier bisher keine allgemeinen Ansätze perfektes Spiel zu verbessern. Deshalb werden auch die Schwächen eines perfekten Spielers sowie Ansätze zur Verbesserung dessen Spiels in dieser Arbeit thematisiert. Das Resultat ist ein Algorithmus, der einen spürbaren Zuwachs an Spielstärke für perfekte Spieler bringt.

Ein weiterer Teil meiner Arbeit ist eine grafische Oberfläche zum Testen des perfekten Spielers und des Algorithmus für verbessertes perfektes Spiel.

1.1 Aufbau

Im Anschluss an diese Einleitung dient **Kapitel 2** der Klärung von Grundbegriffen, die häufig auftauchen und deren Bedeutung nicht offensichtlich ist. Neben einer genauen Definition davon, was perfektes Spiel bedeutet und voraussetzt, werden auch die Grundlagen erläutert, die zum Bau eines perfekten Spielers nötig sind.
In **Kapitel 3** stelle ich das in meiner Arbeit angewandte Verfahren zur Datenbankgenerierung vor. Hierbei handelt es sich um eine erschöpfende Rückwärtssuche. Dieses Verfahren, dass als *Retrograde Analysis* bezeichnet wird, ist nicht die einzige Möglichkeit einen perfekten Spieler zu bauen, hat jedoch mehr als andere Verfahren von der rasanten Entwicklung der Hardware profitiert und gilt deshalb heute als das leistungsfähigste. Möglichkeiten und Grenzen dieses Verfahrens werden in diesem Kapitel ebenfalls diskutiert.
Die platzsparende Speicherung von Informationen und der schnelle Zugriff auf die angesprochene Datenbank stellen das größte Problem bei der erschöpfenden Suche dar. Die existierenden Lösungen sind jeweils auf ein spezielles Brettspiel abgestimmt und nur bedingt auf andere Spiele übertragbar. Ein allgemeiner Ansatz, der die Datenbankoperationen für jedes Spiel übernehmen kann, existierte bislang nicht. In **Kapitel 4** stelle ich ein Verfahren vor, dass effizient und platzsparend arbeitet und somit eine schnelle Generierung der Datenbanken ermöglicht. Dieses Verfahren nutzt sowohl Brettsymmetrien, als auch die eingeschränkte Bewegungsfreiheit von Figuren aus.
Während der Generierung wird ununterbrochen sowohl lesend als auch schreibend auf nahezu alle Teile der Datenbank zugegriffen. Gerade bei sehr großen Datenbanken ist dies problematisch, da sie nicht mehr komplett in den Arbeitsspeicher passen und auf der Festplatte abgelegt werden müssen. Dies hat zur Folge, dass fast jeder Zugriff die besonders zeitaufwendige Neupositionierung des Schreib-/Lesekopfes der Festplatte erfordert. In **Kapitel 5** wird ein System zum Cachen der Datenbank vorgestellt, das die Zugriffe auf die Datei minimiert und den Generierungsprozess beschleunigt.
In **Kapitel 6** wird der allgemeine Ansatz auf die Probe gestellt und anhand einiger Spiele getestet. Neben ausführlichen Statistiken, die z.B. Dauer der Generierung, Größe der Datenbank, sowie die Verteilung der Gewinn- und Verlustdistanzen umfassen, werden auch einige interessante Stellungen betrachtet, die mit Hilfe der Datenbank analysiert werden, um neue strategische Erkenntnisse über das Spiel zu gewinnen. Für das Spiel *Dodgem* wird außerdem der Quellcode angefügt und erläutert, um den Aufwand zu verdeutlichen, der mit Hilfe dieser Arbeit noch nötig ist um einen perfekten Spieler zu bauen.
Perfektes Spiel im Sinne der künstlichen Intelligenz ist, wie bereits angedeutet, nicht gleichbedeutend mit gutem Spiel. Ein weiteres Ergebnis meiner Arbeit ist deshalb ein Algorithmus, der perfektes Spiel weiter verbessert. Er ist ohne die Angabe von spielspezifischen Strategien auf alle Brettspiele anwendbar. Er ist nicht an das Vorhandensein einer Datenbank gebunden, sondern kann auch künstliche Spieler verbessern, die sich anderer Verfahren bedienen um perfektes Spiel zu erzielen. All dies wird in **Kapitel 7** thematisiert.
Kapitel 8 und **9** beinhalten Informationen zur Implementierung, sowie einen Ausblick auf mögliche weitere Forschung, die in diesem Bereich betrieben werden kann.
Im **Anhang** finden sich die Daten, die für Kapitel 6 zu umfangreich waren. Hierzu gehören einige Statistiken sowie der kommentierte Quellcode für das Spiel *Dodgem*.

2 Grundlagen zum Bau eines perfekten Spielers

2.1 Betrachtete Spiele

Die in dieser Arbeit betrachteten Spiele müssen bestimmte Kriterien erfüllen, um sich zum Bau eines perfekten Spielers zu eignen. Grundsätzlich werden nur Spiele mit 2 Spielern betrachtet. Der anziehende Spieler wird im weiteren Verlauf *Spieler 1* oder *Weiß* genannt; der nachziehende Spieler heißt *Spieler 2* oder *Schwarz*. Es ist nicht zwingend notwendig, dass die Spieler abwechselnd ziehen. So kann ein Spieler durchaus mehrere Züge hintereinander ausführen. Sie werden zu einem einzigen Zug zusammengefasst. Auch ein Aussetzen ist möglich. In diesem Fall wird ein Nullzug gespielt, der den Zustand des Spiels nicht ändert, sondern nur das Zugrecht weitergibt. Ein weiteres Kriterium ist *Determinismus*. Damit scheiden all die Spiele aus in denen Zufallselemente, wie z.B. Würfel, eine Rolle spielen. Weiter müssen beide Spieler über vollständige Informationen über den aktuellen Zustand verfügen. Das schließt Kartenspiele aus, in denen ein Spieler nicht weiß, welche Karten ein anderer Spieler besitzt. Spiele, die diese Kriterien erfüllen, sind z.B. Schach, Mühle, Dame, Othello[2], 4 Gewinnt oder TicTacToe.

2.2 Grundbegriffe

Nicht alle Begriffe aus der Welt der Brettspiele sind selbsterklärend. Da einige Begriffe im Verlauf dieser Arbeit immer wieder auftauchen, soll ihre Bedeutung an dieser Stelle erläutert werden.

Der Zustand eines Spiels wird auch als *Stellung* bezeichnet. Er schließt sowohl die Konstellation von Figuren oder Steinen auf dem Spielbrett ein, als auch die Informationen, welcher Spieler am Zug ist. Im englischen wird meistens das Wort *position* verwendet. Das deutsche Wort *Position* hat jedoch sehr unterschiedliche Bedeutungen und würde in diesem Kontext oftmals für Verwirrung sorgen.

Ein *Nachfolger* einer Stellung ist ein Zustand, der durch Ausführen eines legalen Zuges aus dieser entstehen kann. Dementsprechend ist ein *Vorgänger* einer Stellung ein Zustand, aus dem diese durch Ausführen eines Zuges entstanden sein kann. Ein einzelnes Spiel zwischen zwei Kontrahenten wird im Folgenden *Partie* genannt.

Bei einigen Brettspielen, wie z.B. Schach, gilt ein Zug (engl. *move*) als die Kombination eines Zuges von Weiß und eines Zuges von Schwarz. Hier wird von dieser Terminologie abgesehen. Ein Zug bezeichnet einen einzelnen Zug von Weiß oder Schwarz. Um beim schachlich versierten Leser nicht für Verwirrung zu sorgen, wird im Zusammenhang mit Schach ausschließlich von einem Halbzug (engl. *ply*) gesprochen.

Von *Zugzwang* spricht man in Situationen, in denen es ein Nachteil ist, einen Zug ausführen zu müssen. Diese Situation kann nur dann eintreten, wenn auch Zugpflicht besteht. Zugzwang tritt bei Schach oder Go sehr selten auf. Bei anderen Spielen, wie z.B. Dame, gehören Zugzwangstellungen zum Normalfall. Bei 4-Gewinnt treten sie häufig gegen Ende der Partie auf. Auch in die englische Sprache hat der Begriff *Zugzwang* Einzug gehalten.

[2]Im deutschsprachigen Raum ist das Spiel auch unter dem Namen *Reversi* bekannt.

2.3 Arbeitsweise imperfekter künstlicher Spieler

Imperfekte Spieler zu den hier betrachteten Spielen bauen fast immer auf dem *Minimax-Algorithmus* auf. Beginnend von der Ausgangsstellung wird ein Spielbaum aufgebaut. Knoten repräsentieren eine Stellung, Kanten einen Zug, der zu einer nachfolgenden Stellung führt. Zu jedem Knoten führt eine Kante. Sie entspricht dem letzten Zug. Die Anzahl der Kanten, die von einem Knoten wegführt, entspricht der Anzahl der möglichen Züge in der entsprechenden Stellung. Abb. 1 zeigt einen solchen Spielbaum. Er wird meist bis zu einer bestimmten Tiefe (in Abb. 1 Tiefe 2) aufgebaut. An den Endknoten des Baumes, die in Abb.1 mit einer dicken Umrandung versehen sind, wird eine Stellungsbewertung durchgeführt. Sie ist eine Abschätzung, für welchen der beiden Kontrahenten die Stellung aussichtsreicher erscheint. Ist sie positiv, so ist Spieler 1 im Vorteil; ist sie negativ, hat Spieler 2 Vorteile. Der Betrag gibt dabei an, wie groß der Vorteil ist. Oftmals ist es der Fall, dass innerhalb eines Baumes eine Endstellung erreicht wird. In diesem Fall wird sie mit $+\infty$ (Weiß hat gewonnen), $-\infty$ (Schwarz hat gewonnen) oder 0 (Unentschieden) bewertet. Ist der komplette Spielbaum aufgebaut, so werden die Werte der Endknoten an ihre Vorgänger propagiert. An dieser Stelle wählt der Spieler am Zug den für ihn günstigsten Wert. Spieler 1 ist bestrebt die Stellungsbewertung zu maximieren und wird sich deshalb für den höchsten Wert entscheiden. Analog wählt Spieler 2 den minimalen Wert. Dieses Verfahren wird bis Suchtiefe 1 angewandt. War am Startknoten Spieler 1 am Zug, so wählt er den Zug, der bei Suchtiefe 1 zum höchsten Wert führt. Dieser Wert ist ihm bei optimalem Spiel beider Seiten garantiert. In Abb. 1 entspricht dies dem Wert 0. Spieler 2 am Zug würde sich für den Zug entscheiden, der zum niedrigsten Wert führt.

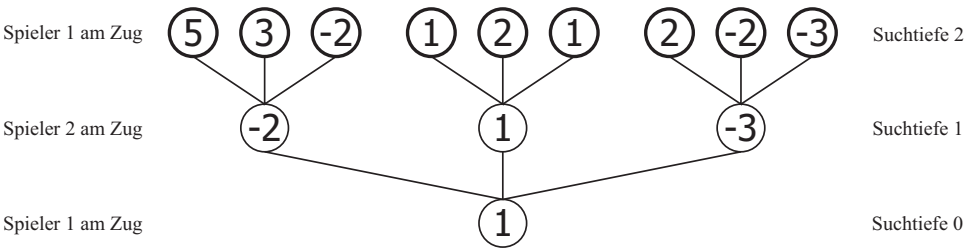


Abbildung 1: Das Minimax-Verfahren

Diese Prozedur lässt sich auch rekursiv durchführen, um den Speicherbedarf zu minimieren. In diesem Fall spricht man von einer Tiefensuche. Die hier vorgestellte Version des Minimax-Verfahrens wird nur sehr selten verwendet. Es gibt zahlreiche verbesserte Varianten des Algorithmus, die zu einem identischen Ergebnis kommen, aber eine geringere Zeitkomplexität haben und somit sehr viel schneller arbeiten. Dies sind z.B. Alpha-Beta, Negascout oder MFD(f).

Die Stellungsbewertung ist für jedes Spiel unterschiedlich. So ist es bei 4-Gewinnt von Vorteil bereits drei Steine seiner Farbe in einer Reihe zu haben. Beim Schach sind andere Kriterien entscheidend. Hier spielen Anzahl und Typ der Spielfiguren, die jede Seite noch auf dem Spielbrett hat, eine entscheidende Rolle. Die Spielstärke des künstlichen Spielers hängt sowohl von der Stellungsbewertung, als auch von der Suchtiefe ab, die die verwendete Form des Minimax-Algorithmus erreicht.

2.4 Der spieltheoretische Wert

Während eine Minimax-Suche den Wert einer Stellung nur abschätzt, ist der *spieltheoretische Wert* (engl. *game-theoretic value*) der exakte Wert einer Stellung aus Sicht des Spielers, der am Zug ist. Trotz breitem Gebrauchs in vielen Publikationen, die sich mit künstlicher Intelligenz und Brettspielen auseinandersetzen, ist dieser Begriff bisher nur grob umschrieben. Selbst Standardwerke (z.B. [RusNor95]) machen von ihm ohne eine exakte Definition Gebrauch. Allis, bekannt durch zahlreiche Publikationen im Zusammenhang mit künstlichen Spielern, beschreibt ihn lediglich als das Ergebnis, dass bei optimalem Spiel beider Seiten erzielt wird ([All94]). Genaugenommen ist er mehr als das bloße Ergebnis. Er gibt zusätzlich eine Distanz zu diesem Ergebnis an, wenn es sich um Gewinn oder Verlust handelt. An dieser Stelle soll versucht werden den Begriff etwas genauer zu spezifizieren. Basis dafür bildet eine modifizierte Minimax-Suche. Sie ist in der Lage exakte Werte einer Stellung zu liefern, wenn die Stellungsbewertung auf solche Stellungen eingeschränkt wird, deren exakter Wert bekannt ist. Dies sind genau jene Stellungen, für die folgende Funktionen definiert sind.

Sei $P_{Legal}(X)$ die Menge aller legalen Stellungen eines beliebigen, vollständig determinierten Brettspiels mit 2 Spielern. Sei weiter $P_{End}(X) \subset P_{Legal}(X)$ die Menge aller Stellungen, bei deren Erreichen das Spiel endet. Die Funktion *val* sei wie folgt definiert:

$$val(p) : P_{End}(X) \rightarrow \{lost, won, drawn\}$$

$$val(p) = \begin{cases} lost & : \text{Der Spieler am Zug hat in Stellung p verloren.} \\ won & : \text{Der Spieler am Zug hat in Stellung p gewonnen.} \\ drawn & : \text{Das Spiel ist unentschieden.} \end{cases}$$

Es folgt Pseudocode für die Funktionen MINGTV und MAXGTV. Sie rufen rekursiv ihr Gegenstück mit jedem Nachfolger der ihnen übergebenen Position auf. Genau wie bei der herkömmlichen Minimax-Suche bildet das minimale bzw. maximale Ergebnis dieser Aufrufe den Rückgabewert. Hierzu werden die Funktionen BEST-GTV-MIN und BEST-GTV-MAX benutzt. Sie unterschieden sich von herkömmlichen min- oder max-Funktionen. Stellt sich heraus, dass zwei Pfade für die Seite am Zug zu einer gewonnenen Endstellung führen, so muss der Pfad mit der niedrigeren Distanz gewählt werden, der einem schnelleren Sieg entspricht. Analog verhält es sich bei verlorenen Stellungen. Die Suchtiefe wird bei jedem Aufruf einer der Funktionen MINGTV oder MAXGTV inkrementiert. Eine Menge *history* sorgt dafür, dass keine Endlosschleifen entstehen. Ein Pfad wird erst dann abgebrochen, wenn eine Stellung erreicht ist, für die $val(position)$ definiert ist.

```
FUNCTION BEST-GTV-MAX(val1, val2)
   IF val1 > 0 AND val2> 0 THEN RETURN MIN(val1, val2)
   RETURN MAX(val1, val2)
END FUNCTION

FUNCTON BEST-GTV-MIN(val1, val2)
   IF val1 < 0 AND val2 < 0 THEN RETURN MAX(val1, val2)
   RETURN MIN(val1, val2)
END FUNCTION

FUNCTION MAXGTV(position, depth, history)
   SELECT val(position)
      CASE drawn: RETURN 0
      CASE won:   RETURN depth
      CASE lost:  RETURN -depth
      CASE undefined:
         result = -∞
         FOR ALL q ∈ succ(position) DO
            IF q ∉ history THEN
               v = BEST-GTV-MAX(v, MINGTV(q, depth + 1, history ∪ q))
            END IF
         END FOR
         RETURN result
   END SELECT
END FUNCTION

FUNCTION MINGTV(position, depth, history)
   SELECT val(position)
      CASE drawn: RETURN 0
      CASE won:   RETURN -depth
      CASE lost:  RETURN +depth
      CASE undefined:
         result = +∞
         FOR ALL q ∈ succ(position) DO
            IF q ∉ history THEN
               v = BEST-GTV-MIN(v, MAXGTV(q, depth + 1, history ∪ q))
            END IF
         END FOR
         RETURN result
   END SELECT
END FUNCTION
```

Der spieltheoretische Wert kann mit Hilfe dieser Funktionen bequem definiert werden. Sei X wiederum ein beliebiges, vollständig determiniertes Brettspiel mit 2 Spielern.

$$gtv(p) : P_{Legal}(X) \setminus P_{end}(X) \rightarrow z \in \mathcal{Z}$$

$$gtv(p) = MAXGTV(p,\ 1,\ \{p\})$$

Ist eine Stellung p bei optimalem Spiel beider Seiten unentschieden, so gilt $gtv(p) = 0$. Ist eine Stellung bei optimalem Spiel beider Seiten gewonnen für die Seite am Zug, so ist $gtv(p) = n,\ n > 0$. n gibt hier die Anzahl der Züge an, die noch nötig sind, um das Spiel siegreich zu beenden. Analog verhält es sich bei einer Stellung, die bei optimalem Spiel beider Seiten verloren ist. Hier ist lediglich das Vorzeichen umgedreht. Im weiteren Verlauf wird eine Stellung p mit $gtv(p) > 0$ als gewonnen bezeichnet und bei $gtv(p) < 0$ als verloren.

Allgemein bekannt ist der spieltheoretische Wert der Ausgangsstellug bei TicTacToe. Bei fehlerfreiem Spiel endet eine Partie stets unentschieden. Somit ist $gtv(p_{Start}(\text{TicTacToe})) = 0$. Umfangreiche Berechnungen waren für Mühle notwendig. Hier ist $gtv(p_{Start}(\text{Mühle})) = 0$ ([GasNie94]). Der spieltheoretische Wert der

Ausgangsstellung bei 4-Gewinnt wurde allein durch Mathematik gelöst. Allis kam auf diese Weise zu dem Ergebnis $gtv(p_{Start}(4 \text{ Gewinnt})) = 41$ ([All88]).

2.5 Perfektes Spiel

Mit dem Wissen über den spieltheoretischen Wert jeder Brettstellung gilt ein Spiel als *stark-gelöst*. Eine genaue Definition dieses Begriffs findet sich in [All94]. Die Übersetzung lautet:

> Ein Spiel ist ***stark-gelöst***, wenn für jede Stellung eine Strategie bekannt ist, die den spieltheoretischen Wert in annehmbarer Zeit ermittelt.

Die zeitliche Restriktion spielt eine entscheidende Rolle bei der Definition. Ohne sie wäre nahezu jedes Spiel stark-gelöst, denn eine erschöpfende Minimax-Suche leistet diese Aufgabe, wie in 2.4 ersichtlich wurde. Ihr Zeitaufwand ist jedoch exponentiell. Für die meisten Brettspiele terminiert sie selbst nach Milliarden von Jahren nicht.

Die Definition eines perfekten Spielers ist ähnlich zu der eines stark-gelösten Spiels. Ein künstlicher Spieler gilt dann als perfekt, wenn er aus jeder Stellung den spieltheoretischen Wert erreicht. Ein stark-gelöstes Spiel impliziert automatisch einen existierenden perfekten Spieler, da mit der Kenntnis des spieltheoretischen Werts jeder Stellung die Wahl des nächsten Zuges einfach ist.

Sei X ein beliebiges, vollständig determiniertes Brettspiel für zwei Spieler. Weiter sei $p \in P_{Legal}(X) \setminus P_{end}(X)$. Drei Fälle müssen unterschieden werden:

$gtv(p) = 0$:
$\exists\, p^* \in P_{Succ(p)}$ mit $gtv(p^*) = 0$ oder $val(p^*) = draw$

$gtv(p) = n,\ n > 0$:
$\exists\, p^* \in P_{Succ(p)}$ mit $gtv(p^*) = n - 1$ oder $val(p^*) = won$

$gtv(p) = n,\ n < 0$:
$\exists\, p^* \in P_{Succ(p)}$ mit $gtv(p^*) = n + 1$ oder $val(p^*) = lost$

Ein künstlicher Spieler braucht in der Stellung p lediglich den Zug zu spielen, der zur Stellung p^* führt. Damit gilt er per Definition als perfekt. Er erreicht aus jeder Stellung nie ein schlechteres Endergebnis als den spieltheoretischen Wert. In gewonnenen Stellungen gewinnt er immer - und das sogar auf schnellstmöglichem Weg. In verlorenen Stellungen besitzt er die Fähigkeit die Niederlage so weit wie möglich hinauszuschieben. Beginnend von einer Stellung, die unentschieden ist, wird ein perfekter Spieler nie verlieren. Fehler des Gegners werden unmittelbar bestraft. Ein menschlicher Spieler, der in einer gewonnenen Stellung einen Zug macht, der in eine Stellung führt, die spieltheoretisch nur noch unentschieden ist, wird nicht mehr gewinnen können. Dass perfektes Spiel nicht automatisch gutes Spiel bedeutet, wird sich in Kapitel 7 zeigen.

2.6 Ermittlung spieltheoretischer Werte

Für den Bau eines perfekten Spielers muss ein Spiel stark gelöst werden. Es stellt sich die Frage, ob für jedes Spiel eine Strategie existiert, die in annehmbarer Zeit den spieltheoretischen Wert einer Stellung ermitteln kann. Mit der modifizierten Minimax-Suche wurde bereits ein äußerst ineffizientes Verfahren vorgestellt. Doch selbst eine optimierte Version dieser erschöpfenden Vorwärtssuche, die auf effizientere Suchalgorithmen zurückgreift, führt nur bei sehr wenigen Spielen zum Erfolg. Grundsätzlich dauert die dynamische Bestimmung dieser Werte zu lange. Um das

Zeitkriterium zu erfüllen, müssen alle Werte einmal bestimmt und abgespeichert werden. Die Dauer dieses Vorgangs kann beliebig lange dauern, denn sie steht nicht im Widerspruch zur Definition eines stark-gelösten Spiels. Sind alle Werte in einer Datenbank abgelegt, ist die Strategie zur Bestimmung dieser Werte trivial. Sie werden einfach aus der Datenbank abgerufen.

3 Retrograde Analysis

Ein effizientes Verfahren zur Ermittlung spieltheoretischer Werte ist die sogenannte *Retrograde Analysis*. Sie ist eine Kombination aus Rückwärtssuche und Breitensuche. Die wörtliche Übersetzung lautet "Rückschreitende Analyse", taucht jedoch in der Literatur nicht auf. Einzig in [SteFri95] wird dieser Algorithmus beiläufig als *retroanalytisches Verfahren* bezeichnet. Das Grundkonzept der *Retrograde Analysis* wurde erstmals von Ströhlein in [Stro70] erwähnt und in den darauffolgenden Jahren und Jahrzehnten weiter verbessert. Im folgenden Abschnitt wird das Grundprinzip am Brettspiel *Dodgem* veranschaulicht. Darauf folgt ein Streifzug durch die Anwendungen im wissenschaftlichen Kontext und das Betrachten der Möglichkeiten und Grenzen dieses Verfahrens.

3.1 Das Brettspiel Dodgem

Das Brettspiel *Dodgem* wurde von Colin Vout erfunden. Die Spielregeln wurden von ihm 1993 in [VoGr93] niedergeschrieben. Auf einem quadratischen Spielfeld der Größe `n` befinden sich `n - 1` Autos für jeden der beiden Spieler. In der Startstellung stehen die Autos von Spieler 1, in Abb. 2 als weiße Spielsteine dargestellt, am linken Rand. Die Autos von Spieler zwei, die durch schwarze Spielsteine dargestellt sind, befinden sich am unteren Rand. Das linke untere Feld bleibt leer.

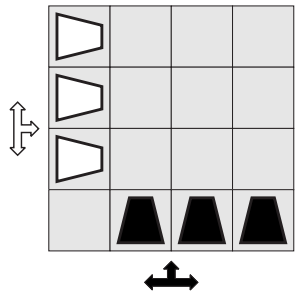


Abbildung 2: Startaufstellung

Spieler 1 kann seine Autos, wie durch den weißen Pfeil angedeutet, vorwärts (nach rechts) oder zur Seite (nach oben oder unten) bewegen, nicht aber rückwärts. Auch Spieler 2 darf seine Autos nicht rückwärts bewegen, sondern ebenfalls nur - aus seiner Sicht - vorwärts oder zur Seite. Der schwarze Pfeil deutet die Richtungen an.

Ein weißes Auto kann nur über den rechten Rand das Spielbrett verlassen; ein schwarzes Auto nur über den oberen Rand. Keinem der beiden Spieler ist es erlaubt auf ein Feld zu ziehen, auf dem bereits ein Auto steht. Es gewinnt der Spieler, der als Erster kein Auto mehr auf dem Spielbrett hat.

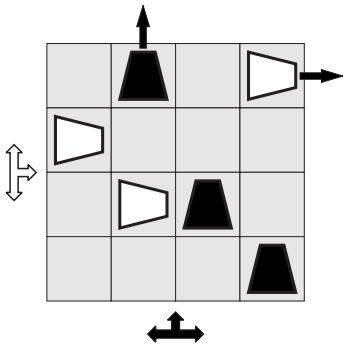
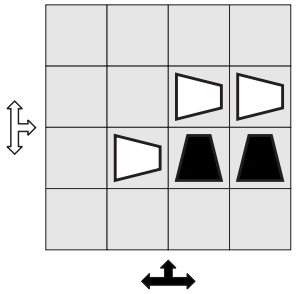


Abbildung 3: Verlassen des Spielbretts

Trotz seiner sehr einfachen Spielregeln ist Dodgem kein einfaches Spiel. Es besitzt zwar nicht die Komplexität von Schach oder Mühle, ist jedoch auch nicht so leicht durchschaubar wie TicTacToe. Als ungeübter Spieler sind selbst schwache Computerprogramme schwer zu bezwingen, woraus sich folgern lässt, dass auch Dodgem durchaus komplexere Spielstrategien und taktische Finessen beherbergt.



Es kann zu Spielsituationen kommen, in denen ein Spieler keine Zugmöglichkeit hat. Die ursprünglichen Spielregeln sehen vor, dass der Spieler ohne Zugmöglichkeit gewonnen hat. Um das Spiel nicht unnötig kompliziert zu machen, nehmen wir an, dass der Spieler in einem solchen Fall einfach aussetzt.

Abbildung 4: Spieler 2 kann nicht ziehen

3.2 Der Basisalgorithmus

Der Basisalgorithmus zur Retrograde Analysis besteht aus zwei Phasen. In der **Initialisierungsphase** werden alle Stellung mit `0` markiert, die für die Seite am Zug verloren sind. Im Falle von Dodgem sind das all jene Stellungen, in denen der Gegenspieler sein letztes Auto über den Rand gezogen hat. Der Spieler, der nun am Zug wäre, hat die Partie verloren.

Bei Dodgem existieren insgesamt

$$\binom{16}{3} + \binom{16}{2} + \binom{16}{1}$$

Stellungen, in denen Schwarz verloren hat. Da auch die weißen Verluststellungen berücksichtigt werden müssen, ist die Anzahl der Stellungen, die während der Initialisierungsphase mit einer 0 markiert werden, doppelt so hoch.

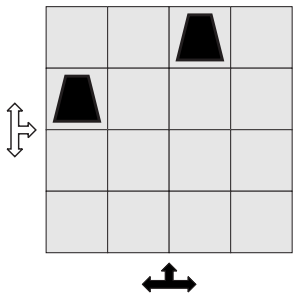
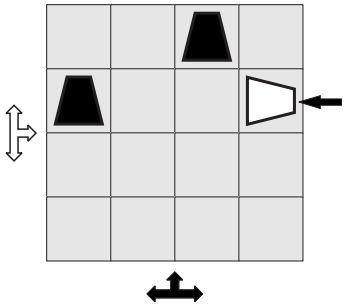


Abbildung 5: Schwarz hat verloren

Die anschließende **Iterationsphase** unterteilt sich wiederum in zwei Phasen: dem *Loss Backup* und dem *Win Backup*. In Iterationsstufe `n` wird beim *Loss Backup* nach Stellungen gesucht, die den Wert `-2 * (n - 1)` haben. Sie sind spieltheoretisch verloren. Bei der ersten Iteration sind das genau jene Stellungen, die zuvor in der Initialisierungsphase markiert wurden. Alle Vorgänger einer solchen Stellung sind gewonnen, da ein Zug existiert, der zu einer Stellung führt, die als verloren bekannt ist. Der spieltheoretische Wert dieser Vorgänger ist somit `(2 * n) - 1`. Hat diese Stellung bereits einen Wert zugewiesen bekommen, so wird dieser nicht überschrieben. In einem solchen Fall existiert ein besserer Zug, der in weniger als `(2 * n) - 1` Zügen gewinnt.



Die Stellung in Abb. 5 ist verloren für Schwarz und wurde mit einer 0 markiert. Diese Stellung zeigt einen Vorgänger, in dem Weiß am Zug ist. Er hat den spieltheoretischen Wert 1, da Weiß in einem Zug gewinnen kann.

Abbildung 6: Weiß gewinnt in einem Zug

Diese Abbildung zeigt eine Stellung, die für Schwarz am Zug verloren ist. Der Wert dieser Stellung wurde während der letzten Iterationsstufe ermittelt. Er lautet -12. Diese Stellung hat die abgebildeten sieben Vorgänger, für die alle der Wert 13 gespeichert werden kann. Die einzige Ausnahme bildet der Vorgänger mit der Nummer 3. Er kann bereits in 5 Zügen gewonnen werden, denn es ist leicht ersichtlich, dass das weiße Auto, sofern es vorwärts zieht, nicht mehr aufgehalten werden kann.

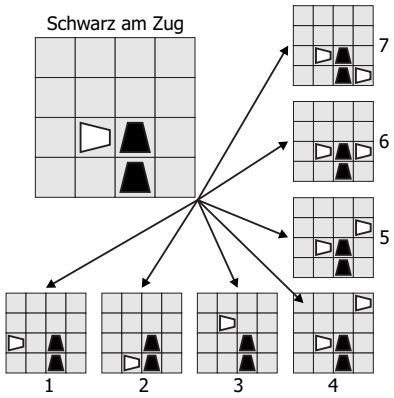
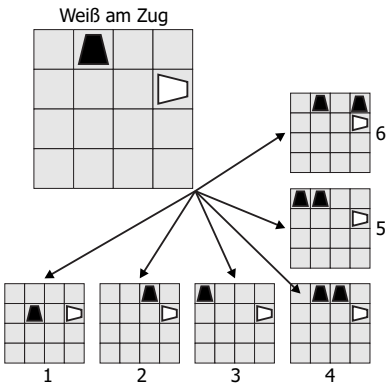


Abbildung 7: Ein nicht-triviales Beispiel

Während die Phase des *Loss Backup* dazu dient neue gewonnene Stellungen zu finden, werden beim *Win Backup* verlorene Stellungen gesucht. Dieser Prozess ist deutlich aufwendiger, denn nicht jeder Vorgänger einer gewonnenen Stellung ist automatisch verloren. Nur wenn jeder Nachfolger einer potentiell verlorenen Stellung gewonnen für die Gegenseite ist, handelt es sich tatsächlich um eine verlorene Stellung.



Weiß am Zug gewinnt in dieser Stellung in einem Zug. Nicht jeder Vorgänger von Schwarz ist verloren. Nur in den Stellungen 1, 4, 5 und 6 ist auch jeder Nachfolger gewonnen für Weiß. Sie werden mit einer -2 markiert. Mit den Stellungen 2 und 3 existieren Nachfolger, die für Schwarz unmittelbar gewinnen. Der Wert dieser Stellungen bleibt unverändert. Er wurde bereits beim *Loss Backup* mit 1 festgelegt.

Abbildung 8: Vorgänger einer gewonnenen Stellung

Aufgrund der endlichen Anzahl von Stellungen terminiert dieses Verfahren. Dies kann beim *Loss Backup* geschehen, wenn nur gewonnene Stellungen gefunden werden, die bereits einen Wert haben. Beim *Win Backup* können alle potentiell verlorenen Stellungen bereits markiert sein oder alle haben mindestens einen Nachfolger, der zu keiner gewonnenen Stellung führt. Auch in diesem Fall endet der Prozess. In Pseudocode lässt sich der Algorithmus wie folgt beschreiben:

```
FUNCTION ITERATE(n)

   // LOSS BACKUP
   FOR ALL p ∈ positions DO
      IF gtv(p) = -2 * (n - 1) THEN
         FOR ALL q ∈ pred(p) DO
            IF getValue(q) = unknown
               setValue(q) = (2 * n) - 1
            END IF
         END FOR
      END IF
   END FOR

   // WIN BACKUP
   FOR ALL p ∈ positions DO
      IF gtv(p) = (2 * n) - 1
         FOR ALL q ∈ pred(p) DO
            IF getValue(q) = unknown THEN
               lost = true
               FOR ALL r ∈ succ(q) \ p DO
                  IF NOT getValue(r) ≥ 1 THEN
                     lost = false
                     EXIT FOR
                  END IF
               END FOR
               IF lost = true THEN setvalue(q) = -(2 * n)
            END IF
         END FOR
      END IF
   END FOR

END FUNCTION
```

Mit der Terminierung des Algorithmus sind alle Stellungen mit ihrem spieltheoretischen Wert markiert. Die Stellungen, die nicht markiert sind, sind weder gewonnen

noch verloren - sie sind unentschieden.

3.3 Optimierungen

Der Algorithmus existiert in zahlreichen Formen und Variationen, häufig den spezifischen Gegebenheiten des Spiels angepasst, auf das er angewendet werden soll. Zwei Optimierungen haben sich jedoch in allen Fällen bewährt. Zu allererst ist es ratsam, *Loss Backup* und *Win Backup* in einem Schritt abzuarbeiten. So muss beim *Win Backup* nicht die gesamte Datenbank nach Werten abgesucht werden, die beim *Loss Backup* neu gespeichert wurden. Sobald also beim *Loss Backup* eine gewonnene Stellung gefunden wird, wird unmittelbar überprüft, ob einer oder mehrere Vorgänger dieser Stellung verloren sind.

Die zweite Optimierung bezieht sich allein auf das *Win Backup*. Es hat sich herausgestellt, dass es äußert ineffizient ist zu einer potentiell verlorenen Stellung stets zu prüfen, ob alle Nachfolger für die Gegenseite gewonnen sind, um so den Beweis zu liefern, dass es sich tatsächlich um eine verlorene Stellung handelt. Besser ist es, wenn eine Stellung, die gerade als gewonnen markiert wurde, ihren Wert an all ihre Vorgänger weitergibt. Sobald eine Stellung von all ihren Nachfolgern diese Nachricht empfangen hat, kann sie als verloren markiert werden. Die Information, wie viele Nachfolger bereits als gewonnen markiert sind, wird im Wert der Stellung codiert. Beim ersten Zugriff auf eine potentiell verlorene Stellung wird deren Wert mit "Anzahl ihrer Nachfolger - 1" festgelegt, denn erst ein Nachfolger ist als gewonnen bekannt. Jeder weitere gewonnene Nachfolger dekrementiert diesen Wert. Ist er bei 0 angekommen, ist die Stellung verloren. Um diese Werte nicht mit einem spieltheoretischen Wert zu verwechseln, ist ein spezielles Mapping notwendig. Dieses wird in Abschnitt 3.4 beschrieben.

Diese Abbildung zeigt eine Stellung, die für Schwarz verloren sein könnte. In den drei Nachfolgern (Stellung 1-4) ist Weiß am Zug. Stellt sich heraus, dass eine der Stellungen gewonnen ist, initialisiert sie Stellung 5 mit dem Wert 3, da der Wert der drei weiteren Nachfolger noch unbekannt ist. Wird einer dieser drei Nachfolger als gewonnen markiert, dekrementiert er den Wert von Stellung 5. Ist dieser Wert bei 0 angekommen, ist Stellung 5 verloren.

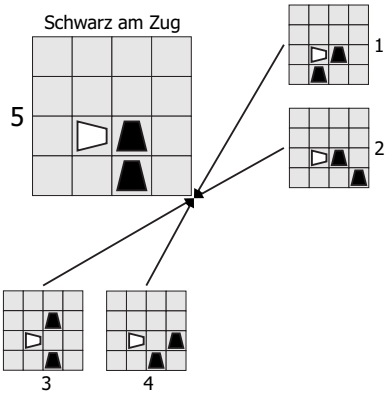


Abbildung 9: Dekrementierung durch gewonnene Nachfolger

Der Geschwindigkeitgewinn ist sofort ersichtlich. Beim Erreichen einer potentiell verlorenen Stellung muss nicht jedes Mal der Zuggenerator, der die Nachfolger liefert, gestartet werden, um anschließend jeden diesen Nachfolger auf seinen Wert in der Datenbank zu überprüfen. Dies spart zeitaufwendige Zugriffe.

Der resultierende Algorithmus sieht wie folgt aus. Er setzt voraus, dass in der Initialisierungsphase bereits das erste *Loss Backup* durchgeführt wurde, also die Stellungen, die den spieltheoretischen Wert 1 haben, bereits mit diesem markiert sind.

```
FUNCTION ITERATE(n)

   FOR ALL p ∈ positions DO
      // WIN BACKUP
      IF gtv(p) = 2 * (n - 1) THEN
         FOR ALL q ∈ pred(p) DO
            IF getValue(q) = unknown THEN
               setValue(q) = '(|succ(q)| - 1) states unknown'
            ELSE IF getValue(q) = 'x states unknown' THEN
               setValue(q) = '(x - 1) states unknown'
            END IF
            IF getValue(q) = '0 states unknown' THEN
               setValue(q) = - (2 * n)
               FOR ALL r ∈ pred(q) DO
                  // LOSS BACKUP
                  IF getValue(r) = unknown THEN
                     setValue(r) = (2 * n) + 1
                  END IF
               END FOR
            END IF
         END FOR
      END IF
   END FOR

END FUNCTION
```

Um welchen Faktor diese Optimierung den Generierungsprozess beschleunigt hängt sehr stark von der Art des Spiels ab. Je größer die mittlere Anzahl von Nachfolgern einer Stellung, desto größer ist auch der Geschwindigkeitsgewinn. Auch die Größe der Datenbank spielt eine Rolle. Selbst die maximale Gewinn- bzw. Verlustdistanz beeinflusst diese Optimierung. Werden vergleichsweise wenig Stellungen in jedem Iterationsschritt als gewonnen oder verloren markiert, so sinkt die Effizienz der Optimierung deutlich. Bei Dodgem wird die Generierung jedoch trotz einer kleinen Datenbankgröße, einer geringen Anzahl von Nachfolgern einer Stellung und einer hohen Distanz zu Endstellungen um den Faktor 2 beschleunigt. Die meisten Spiele sprechen besser auf diese Optimierung an.

3.4 Anwendungen

Es dauerte einige Zeit bis der Algorithmus zur Berechnung spieltheoretischer Werte herangezogen wurde. Dies ist in erster Linie auf die begrenzte Rechen- und Speicherkapazität zurückzuführen, wie sie in den siebziger Jahren zur Verfügung stand. Erste Anwendungen fand das Verfahren bei Schachendspielen, denen bis Anfang der neunziger Jahre das Hauptinteresse der Wissenschaft galt. Von dort an wurden auch Brettspiele wie Dame und Mühle betrachtet. Ein kleiner Überblick über die Entwicklung dieser Brettspiele im Zusammenhang mit der *Retrograde Analysis* soll den aktuellen Stand der Wissenschaft auf diesem Gebiet verdeutlichen.

Schach

Schach ist zu komplex für eine vollständige Analyse. Aus diesem Grund werden nur Endspiele mit wenigen Figuren betrachtet. Sie werden in der Literatur durch die Anfangsbuchstaben der englischen Namen der Figuren klassifiziert. K bezeichnet dabei den König (King), Q die Dame (Queen), R den Turm (Rook), B den Läufer (Bishop), N den Springer (kNight) und P den Bauern (Pawn). Das Endspiel KQ-KR bedeutet damit "König und Dame" gegen "König und Turm". Pionier auf diesem

Gebiet war Kenneth Thompson, auch bekannt als der Erfinder des Betriebssystems Unix. Er nutzte die Großrechner an den Bell Laboratories in New Jersey, um bis 1986 alle Schachendspiele mit 3 und 4 Steinen zu berechnen, so wie viele Endspiele mit 5 Steinen, die nicht mehr als einen Bauern enthielten ([Tho86]). Einige Jahre später war die Berechnung aller nicht-trivialen[3] Endspiele mit bis zu 5 Steinen abgeschlossen. Diese Sammlung war in komprimierter Form auf mehreren CDs erhältlich. Die Datenbanken hatten jedoch mehrere Nachteile. Zum einen enthielten sie nur Gewinndistanzen für die materiell stärkere Seite und die Verlustdistanzen für die schwächere Seite. Weiter war der Zugriff auf einzelne Werte zu zeitintensiv, um die Datenbanken in den Suchprozess, wie ihn jedes Schachprogramm durchführt, zu integrieren. Anfang der neunziger Jahre begann deshalb Steven Edwards mit der Generierung von Endspieldatenbanken, die Gewinn- und Verlustdistanzen für beide Seiten enthielten und einen schnelleren Zugriff garantierten. Da er dieses Programm veröffentlichte, war nun jeder mit einem handelsüblichen PC, sowie genug Speicherplatz und Geduld[4] in der Lage die Endspieldatenbanken zu generieren. Viele Schachprogramme integrierten diese Datenbanken in den Suchprozess. Ende der neunziger Jahre veröffentlichte Eugene Nalimov ein neues Programm zum Generieren von Endspieldatenbanken. Das Programm war schneller[5] und die entstandenen Datenbanken dank besserer Codierung und Komprimierung kleiner. Sie gelten heute als Standard und werden von fast jedem Schachprogramm genutzt. Unkomprimiert sind sie 22,5 Gigabyte groß; komprimiert sind es knapp 6 Gigabyte. Eine nicht-öffentliche Version des Programms von Nalimov ist auch in der Lage Endspiele mit 6 Steinen vollständig zu berechnen. Sie stehen - neben den Endspielen mit 5 oder weniger Steinen - im Internet zum Download[6] zur Verfügung. Viele von ihnen sind noch nicht verfügbar. Sie werden nach und nach ergänzt. Die Berechnung aller 6-Steiner wird voraussichtlich 2006 abgeschlossen sein. Ihr Umfang wird nach neusten Schätzungen selbst in komprimierter Form über 1000 Gigabyte betragen.

Die meisten Publikationen zum Thema Schach und Datenbanken finden sich im ICGA-Journal[7], dass vierteljährlich erscheint. Eine ausführlichere Zusammenfassung über die Entwicklung von Schachdatenbanken findet sich in [Hei99].

Der Zugewinn an Spielstärke unter Benutzung der Endspieldatenbanken ist für ein Schachprogramm nur minimal. Häufige Endspiele, wie z.B. KRP-KR, spielen Schachprogramme dank verbessertem Schachwissen heutzutage auch ohne den Einsatz von Datenbanken perfekt. Viele Spiele enden zudem lange bevor nur noch 6 Steine auf dem Brett sind. Dennoch sind Endspieldatenbanken von großem theoretischen Interesse, da sie viele Irrtümer aufgedeckt und zahlreiche in Büchern veröffentlichte Studien widerlegt haben.

[3] Ein triviales Endspiel ist z.B. KQQQ-K. In dieser Konstellation kann Weiß in jeder Stellung den schwarzen König in wenigen Zügen Matt setzen. Eine Datenbank ist in diesem Fall überflüssig, denn aufgrund der geringen Distanz zum Spielende findet auch eine herkömmliche Suche den optimalen Zug unmittelbar.

[4] Auf einem PC mit 80486-Prozessor konnte die Generierung eines Endspiels mit 5 Steinen bis zu drei Monate dauern.

[5] Ein herkömmlicher Computer mit 1Ghz benötigt für die Berechnung aller 3-, 4- und 5-Steiner ungefähr zwei Wochen.

[6] `ftp://ftp.cis.uab.edu/pub/hyatt/TB`

[7] Die International Computer Games Association ist im Internet unter `http://www.cs.unimaas.nl/icga` zu Hause.

Mühle

Ein perfekter Mühlespieler entstand in den Jahren 1993-1995 an der ETH Zürich durch die Arbeit von Ralph Gasser ([GasNie94], [Gas95], [Gas96]). Auch dieser Spieler stützte sich auf retroanalytisch generierte Datenbanken, die den spieltheoretischen Wert jeder Stellung enthielten. Sie waren insgesamt knapp 10 Gigabyte groß. Einzig während der Setzphase, in der jeder Spieler seine neun Steine auf dem Spielbrett platziert, musste eine Suche durchgeführt werden, die jedoch in jedem Pfad einen Datenbankeintrag erreichte. Dadurch wurde perfektes Spiel garantiert.

Im Jahre 2000 baute Peter Stahlhacke ebenfalls einen perfekten Mühlespieler auf Basis von Datenbanken. Dieser kam ohne eine Suche während der Setzphase aus. Da somit jede Stellung zweimal in der Datenbank enthalten sein muss, einmal für die Setz- und einmal für die Spielphase, hat deren Größe mit 17 Gigabyte ungefähr den doppelten Umfang wie die von Gasser. Gegen das Mühleprogramm von Peter Stahlhacke kann man im Internet[8] antreten.

Dame

Dame besitzt nicht die gleiche Komplexität wie Schach, ist aber dennoch für eine vollständige Analyse zu umfangreich. Deshalb existieren auch für dieses Spiel nur Endspieldatenbanken. Seit 1992 ist Jonathan Schaeffer von der University of Alberta mit deren Berechnung beschäftigt. Über seine Arbeit berichtet er in zahlreichen Publikationen (z.B. [LakSch94], [SchLak96]). 2004 war die Berechnung aller Datenbanken mit 10 Steinen oder weniger abgeschlossen. Sie enthalten die spieltheoretischen Werte zu fast 40.000.000.000.000.000 (40 Billiarden!) Stellungen. Komprimiert beträgt ihr Umfang jedoch nur wenige tausend Gigabyte, da die meisten Stellungen unentschieden sind. Das dazugehörige Dame-Programm *Chinook* nutzt diese Datenbanken innerhalb des Suchprozesses. Auch wenn der Beweis noch fehlt, dass es perfekt spielt, gilt es als nahezu unschlagbar. 1994 errang es den offiziellen Weltmeistertitel[9] und hat seit über acht Jahren keine einzige Partie mehr verloren. Die Dame-Datenbanken mit 8 Steinen und weniger stehen im Internet zum Download[10] zur Verfügung.

Auf die Arbeiten von Thompson, Edwards, Nalimov, Gasser und Schaeffer wird in den weiteren Abschnitten noch mehrfach Bezug genommen. Sie verdeutlichen, dass die *Retrograde Analysis* sich zum Bau perfekter Spieler oder solchen, die zumindest einen Spielabschnitt perfekt beherrschen, bewährt hat. Aus diesem Grund bildet dieses Verfahren auch die Basis für den allgemeinen Ansatz zum Bau von perfekten Spielern, wie er in dieser Arbeit vorgestellt wird.

3.5 Speicherung der spieltheoretischen Werte von Stellungen

Was in den vorangegangenen Abschnitten als selbstverständlich betrachtet wurde, stellt in Wirklichkeit eines der größten Probleme bei der Anwendung des Algorithmus dar: die Speicherung der spieltheoretischen Werte zu den Stellungen. Die einfachste Form wäre eine Tabelle, in der Paare abgelegt werden, die aus einer eindeutigen Codierung der Stellung und dem dazugehörigen Wert bestehen.

[8] `http://www.inetplay.de/muehle.php`

[9] Anders als beim Schach können bei Dame auch nicht-menschliche Spieler den Weltmeistertitel erringen.

[10] `http://www.cs.ualberta.ca/~chinook/databases/`

Codierung Stellung 1	0
Codierung Stellung 2	-4
...	...
Codierung Stellung $\|P_{Legal}\|$	5

Tabelle 1: Abspeicherung spieltheoretischer Werte in Tabellenform

Diese Form der Speicherung hat jedoch mehrere Nachteile. Zum einen benötigt sie sehr viel Speicherplatz, da eine eindeutige Codierung der Stellung in den meisten Fällen mehrere Bytes groß ist. Zum anderen ist der Zugriff sehr aufwendig. Selbst wenn die Stellungscodierungen auf irgendeine Weise geordnet sind, ist das Auffinden einer bestimmten Stellung und ihres Wertes mit einer Suche verbunden, da die Position des Tabelleneintrags in der Datenbank nur abgeschätzt werden kann. Mehrere Zugriffe auf die Datei, um nur einen einzigen Wert zu lesen oder zu schreiben, verlangsamen den Generierungsprozess um ein Vielfaches.

Um einen spieltheoretischen Wert mit einem einzigen Zugriff lesen oder schreiben zu können, muss die Codierung der Stellung gleichzeitig der Index in der Datenbank sein. Somit besteht die Datenbank nur noch aus spieltheoretischen Werten. Geht man von einem Platzbedarf von einem Byte pro Wert aus, entsteht eine Datei von der Größe des maximalen Index. Um auf den Wert einer Stellung zuzugreifen, muss sie dann nur noch codiert und damit ihr Index bestimmt werden. An der entsprechenden Dateiposition findet sich dann der gesuchte Wert. Der Begriff *Datenbank* scheint in diesem Zusammenhang nicht mehr das geeignete Wort für derartige Dateien zu sein, hat sich jedoch in der Literatur durchgesetzt. Wie sich die angesprochene Codierung einer Stellung realisieren lässt, wird in Kapitel 4 behandelt. Die Codierung eines spieltheoretischen Werts in ein einziges Byte ist vergleichsweise einfach. Ein Byte besteht aus 8 Bit und kann damit 2^8 verschiedene Werte annehmen. Die hier gewählte Abbildung eines spieltheoretischen Werts auf den Wertebereich von 0 bis 255 orientiert sich in dieser Arbeit an der Codierung, wie sie in [Gas95] gewählt wird. Durch die Annahme, dass der Spieler, der gewinnt, auch den letzten Zug ausführt, sind nur ungerade Gewinndistanzen möglich. Umgekehrt können Verlustdistanzen nur gerade sein. Die Abbildung sieht damit wie folgt aus:

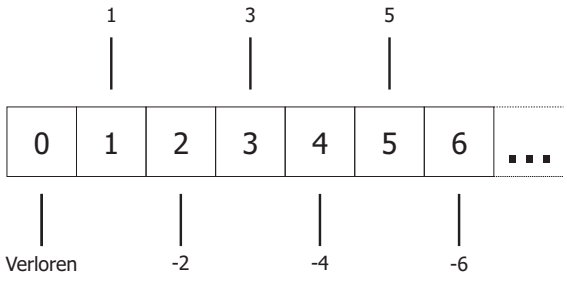


Abbildung 10: Abbildung spieltheoretischer Werte auf ein Byte

Es sei daran erinnert, dass der spieltheoretische Wert immer die Sicht des Spielers wiederspiegelt, der am Zug ist. Damit können Stellungen verloren, nicht aber gewonnen sein. Die Information zu potentiell verlorenen Stellungen, wie viele Nachfolger einen noch unbekannten Wert haben, wird auf die höchsten Werte des Bytes projeziert. Der Wert 255 ist der Initialwert einer jeden Stellung. Mit diesem Wert ist nichts über diese Stellung bekannt. Ein Wert $255 - x$ bedeutet dann, dass ei-

ne potentiell verlorene Stellung noch x Nachfolger hat, deren Gewinn noch nicht nachgewiesen wurde.

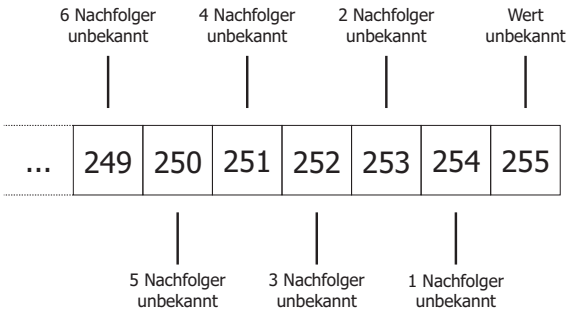


Abbildung 11: Abbildung spieltheoretischer Werte auf ein Byte

Übersteigt die maximale Gewinn- oder Verlustdistanz zuzüglich der maximalen Anzahl von Nachfolgern den Wert 255, kommt es zur Kollision. Dies ist z.B. bei Mühle der Fall, wo die maximale Gewinndistanz 169 Züge beträgt und damit - unabhängig von der Anzahl der Nachfolger - den Wertebereich von einem Byte verlässt. Aus diesem Grund lässt sich vor dem Generierungsprozess der Datenbank festlegen, ob ein oder zwei Byte zur Speicherung spieltheoretischer Werte verwendet werden sollen. Eine dynamische Anpassung ist nicht vorgesehen. Stellt sich während des Generierungsprozesses heraus, dass ein Byte nicht ausreicht, bricht das Programm die Berechnung ab.

3.6 Möglichkeiten und Grenzen

Die Möglichkeiten der Retrograde Analysis bei den Brettspielen Schach, Mühle und Dame wurden bereits aufgezeigt. Generell lässt sich dieses Verfahren auf alle deterministischen Brettspiele anwenden, in denen zwei Spieler gegeneinander antreten und beide Spieler vollständige Informationen über den aktuellen Zustand besitzen. Ob sich ein Spiel komplett analysieren lässt, um den Bau eines perfekten Spielers zu ermöglichen, oder ob nur Teile des Spiels berechnet werden können, um den Suchprozess zu unterstützen, hängt in erster Linie von der Anzahl der möglichen Stellungen ab. Je größer diese Anzahl ist, desto aufwendiger ist die Berechnung und desto größer auch der benötigte Speicherplatz. Die Anzahl der möglichen Stellungen lässt sich für die meisten Spiele kombinatorisch ermitteln oder zumindest abschätzen. Bei Dodgem sind es auf einem 4x4-Feld mit maximal 3 Autos pro Seite

$$2 * \sum_{w=1}^{3} \sum_{b=0}^{3} \left(\binom{16}{w} * \binom{16-w}{b} \right) = 555.984$$

Stellungen. Im Homecomputerbereich dürften heutzutage Berechnungen mit bis zu 10^{11} Stellungen möglich sein. Das entspricht bei einem Byte pro Stellung einer Datenbankgröße von 100 Gigabyte. Die bis heute größte Datenbank ist mit $4 * 10^{16}$ Stellungen die Endspieldatenbank zu Dame, die von einem ganzen Netzwerk von Computern mit Hilfe einer parallelen Version der Retrograde Analysis berechnet wurde. Sie profitiert allerdings von der Tatsache, dass die meisten aller Stellungen unentschieden sind. Für den allgemeinen Fall dürfte die Grenze mit heutiger Technik bei 10^{13} Stellungen liegen. Da mit stetig wachsender Prozessorleistung und Speicherdichte bei Festplatten zu rechnen ist, wird sich diese Grenze kontinuierlich nach oben verschieben. Es gibt allerdings Forscher, die der Meinung sind, dass

perfekte Spieler auf Basis von Datenbanken keine Zukunft haben, da die Speichertechnik früher oder später an ihre Grenzen stößt. Aus diesem Grund gibt es viele regelbasierte Ansätze für perfekte Spieler. [Thi99] gibt einen guten Überblick über die Forschungen bis 1999. Zur Zeit ist aber noch kein Ende bei der Entwicklung der Speichertechnik in Sicht. Die nächste Generation von Festplatten, die noch für dieses Jahr erwartet wird, verwendet eine vertikale Aufzeichnungstechnik ("perpendicular recording"), mit der eine Verzehnfachung der Datendichte möglich ist.

Neben dem verfügbaren Speicherplatz ist das Indizieren von legalen Stellungen eine weitere Voraussetzung für die Anwendung der Retrograde Analysis. Im vorangegangenen Abschnitt wurde erläutert, dass die Codierung einer Brettstellung gleichzeitig der Index in der Datenbank ist. Der maximale Index sollte nur unwesentlich größer als die Anzahl der legalen Stellungen sein, denn er ist gleichzeitig die Größe der (unkomprimierten) Datenbank in Bytes. Es wird sich zeigen, dass das Indizieren einer Stellung aus ihrer kombinatorischen Hülle schwierig, aber möglich ist. Brettspiele, deren legale Stellungen sich nicht kombinatorisch erfassen lassen, stellen ein Problem dar. Dies sind meist solche Brettspiele, in denen die möglichen Felder zum Platzieren einer Figur maßgeblich von der Konstellation der anderen Figuren abhängen. Eine Dame im Schachspiel lässt sich in fast jeder legalen Stellung auf einem beliebigen freien Feld platzieren, da die resultierende Stellung in den meisten Fällen legal bleibt. Bei einem Spiel wie *Vier Gewinnt* oder *Othello* ist dies anders. Auf welchen Feldern ein Spielstein platziert werden kann, hängt stark von der Lage der anderen Spielsteine ab. Bei *Vier Gewinnt* darf kein Spielstein "in der Luft" liegen, sondern muss auf einen bestehenden Stein fallengelassen werden. So existiert für *Vier Gewinnt* bis heute keine vollständige Datenbank, obwohl die Anzahl legaler Stellungen deutlich geringer ist als bei Mühle. Dennoch konnte das Spiel stark gelöst[11] werden. Dank seines geringen Verzweigungsfaktors und der begrenzten Suchtiefe ist dies mit einer speziellen Variante der Minimax-Suche möglich.

[11] Die Ausgangsstellung ist gewonnen für den anziehenden Spieler. Der spieltheoretische Wert beträgt 41.

4 Indizierung von Brettstellungen

4.1 Eine allgemeine Indexfunktion

Für ein beliebiges Spiel X bildet eine Indexfunktion $idx(p)$ jede Stellung auf eine Zahl $n \in \mathcal{N}$ ab. Weiter muss eine Umkehrfunktion $\overline{idx}(n)$ existieren, die zu jeder natürlichen Zahl, für die $idx(p)$ definiert ist, eine entsprechende Stellung liefert.

$$\begin{aligned} idx(p) &: P_{Legal}(X) \to n \in \mathcal{N} \\ \overline{idx}(n) &: n \in \mathcal{N}, n \leq max(idx(p)) \to P_{Legal}(X) \end{aligned}$$

Bei einem Speicherbedarf von einem Byte für jeden spieltheoretischen Wert ist $max(idx(p))$ auch gleichzeitig die Größe der resultierenden Datenbank und die Anzahl der Stellungen in derselbigen. Es ist daher erstrebenswert den Maximalwert einer Indexfunktion so gering wie möglich zu halten. Im Idealfall ist er identisch mit der Anzahl der legalen Stellungen. Je besser die Indexfunktion ist, desto niedriger ist der Quotient $\frac{max(idx(p))}{|P_{Legal}|}$.

Nicht nur der Speicherbedarf der resultierenden Datenbank ist Motivation für eine gute Indexfunktion. Auch die Dauer der Datenbankgenerierung hängt entscheidend von der Dateigröße ab. Für größere Brettspiele ist eine Datenbank mit mehreren Milliarden Werten nötig. Der Arbeitsspeicher eines Computer kann Daten dieser Dimension nicht mehr fassen. Sie müssen auf einer Festplatte gespeichert werden. Der Zugriff auf einzelne Werte in einer Datei im Gigabytebereich ist jedoch nicht unproblematisch. Ein zufälliger Zugriff erfordert fast immer eine Neupositionierung des Schreib-/Lesekopfes. Die Dauer dieses Vorgangs hängt von dem Weg ab, den der Schreib-/Lesekopf zurücklegen muss. Um vom innersten Rand an den äußersten Rand einer Speicherscheibe zu gelangen, benötigt eine Festplatte im Homecomputerbereich[12] 21 Millisekunden. Liegt eine Datei nur in einem bestimmten Abschnitt der Festplatte, so ist die Zugriffszeit entsprechend kleiner. Die minimale Zugriffszeit, die bei ungefähr 2 Millisekunden liegt, wird jedoch nie unterschritten. Für Daten, die innerhalb eines Tracks liegen, ist keine Neupositionierung des Schreib-/Lesekopfes notwendig. Die typische Größe eines Tracks liegt bei 32 Kilobyte. Der interne Cache der Festplatte nimmt ebenfalls Einfluss auf die Zugriffszeit. All diese Faktoren lassen keine genaue Schätzung über die mittlere Zeit zu, die das Lesen oder Schreiben eines Wertes tatsächlich kostet. Es ist jedoch ersichtlich, dass diese Zeit im Millisekundenbereich liegt.

Um also eine effiziente Generierung der Datenbank zu erreichen, muss die Größe der Datenbank so klein wie möglich sein. Eine defragmentierte Festplatte ist ebenfalls von Vorteil. Damit nicht für jede Operation auf die Datenbank der Schreib-/Lesekopf der Festplatte neu positioniert werden muss, ist zudem ein System zum Cachen der Daten erforderlich, dass speziell auf den Zugriff einzelner Werte in einer großen Datei angepasst ist.

4.2 Naive Indizierung von Dodgem-Stellungen

Die einfachste Methode aus einer Stellung einen Index zu gewinnen, stellt die feldweise Codierung der Stellung dar. Abb. 12 zeigt eine solche Codierung am Beispiel einer Dodgem-Stellung.
Mit insgesamt 33 Bit für jeden Index ist die resultierende Datenbank 2^{33} Bytes groß. Dies entspricht 8 Gigabyte. Bei weniger als eine Million Stellungen ist die Güte einer solchen Indexfunktion indiskutabel. Selbst eine etwas durchdachtere Codierung der Stellung wie in Abb. 13 bringt nicht die erhoffte Güte. Mit 23 Bit und damit $2^{23} =$

[12] Die angegebenen Werte beziehen sich auf eine Ultra ATA/100-Festplatte mit 250GB und 7200rpm der Marke Western Digital. Die Werte für andere Festplatten sind nahezu identisch.

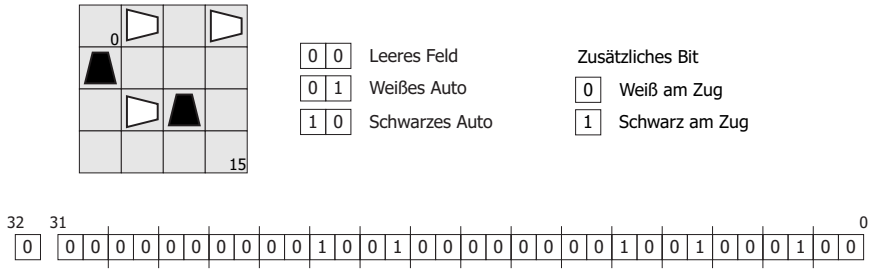


Abbildung 12: Feldweise Codierung einer Dodgem-Stellung

8388608 *Bytes* hält sich die Datenbankgröße in einem überschaubaren Rahmen und garantiert eine zügige Generierung. Dennoch übersteigt die Größe die Anzahl der legalen Stellungen um den Faktor 15. Für größere Datenbanken ist dieser Faktor unbefriedigend. Eine feldweise Indizierung ist daher nicht empfehlenswert.

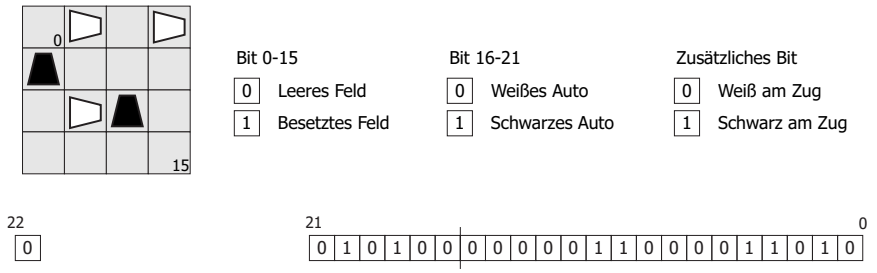


Abbildung 13: Platzsparendere Codierung einer Dodgem-Stellung

4.3 Kombinatorische Indizierung ohne Berücksichtigung gleicher Figuren

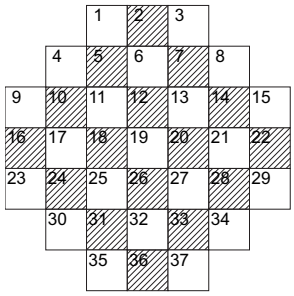


Abbildung 14: Nummerierte Felder

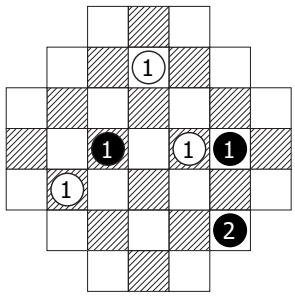


Abbildung 15: Beispiel einer Stellung

Der allgemeinste Fall einer Stellung ist ein Brett beliebig angeordneter Felder, die zeilenweise von links oben nach rechts unten durchnummeriert werden (Abb. 14). Auf diesem Brett können sich nun Figuren verschiedenen Typs befinden. Figuren vom gleichen Typ können untereinander vertauscht werden, ohne dass die Stellung

geändert wird. Abb. 15 zeigt eine solche Stellung. Weiß besitzt drei Figuren vom Typ 1; Schwarz hat zwei Figuren vom Typ 1 und eine Figur vom Typ 2.

Eine kombinatorische Indizierung ohne Berücksichtigung gleicher Figuren lässt sich durch ein Array der Dimension n interpretieren, wobei n die Anzahl der Figuren ist. Die Größe jeder Dimension startet mit der Anzahl der Felder des Spielbretts m und endet mit $m - n + 1$. Für Abb. 15 ergibt sich ein Array

$$[37][36][35][34][33][32]$$

bei dem jede Dimension stets die gleiche Figur repräsentieren muss. Der Index der Figur i sei f_i. Er ergibt sich durch erneutes Abzählen der Felder, wobei Felder, auf denen eine Figur j mit $j < i$ steht, übersprungen werden. Der absolute Index der Stellung kann dann durch die Funktion

$$idx(p) = \prod_{i=1}^{n} \left(f_i * \prod_{j=i}^{n-1} (m - j) \right)$$

berechnet werden. Der Schwachpunkt dieser Indexfunktion ist sofort ersichtlich. Während der maximale Index für die Stellung aus Abb. 15

$$\frac{37!}{31!} = 1.673.844.480$$

ist, ist die Anzahl der legalen Stellungen unter Berücksichtigung gleicher Figuren nur

$$\binom{37}{3} * \binom{34}{2} * \binom{32}{1} = 139.487.040.$$

Die Güte dieser Indexfunktion ist für Spiele mit ausschließlich unterschiedlichen Figuren optimal, für Spiele mit teilweise gleichen Figuren jedoch denkbar schlecht. Dennoch wurden bei Schachendspielen bis vor wenigen Jahren Indexfunktionen verwendet, die mit steigender Anzahl der Figuren noch dramatischer an Güte verlieren. Sowohl bei Edwards, als auch bei Thompson steigt die Größe der Datenbank bei bereits k vorhandenen Figuren durch Hinzufügen zweier gleicher Figuren nicht um den Faktor $\binom{64-k}{2}$, sondern um den Faktor 64^2. Erst Nalimov [NaHaHe00] berücksichtigt sowohl die sinkende Anzahl von möglichen Feldern für jede weitere Figur, als auch gleiche Figuren.

4.4 Bitboards

Der Begriff *Bitboard* kommt ursprünglich aus der Schachprogrammierung. Wurde früher für die interne Repräsentation einer Stellung ein Array gewählt, dessen Einträge entsprechend der Figur auf dem jeweiligen Feld des Spielbretts gewählt wurden, verwendet man heutzutage Bitboards. Ein Bitboard ist eine Binärzahl, dessen Bits angeben, ob sich auf dem entsprechenden Feld des Spielbretts eine Figur eines bestimmten Typs befindet. Bitboards machen nur dann Sinn, wenn die Rechenarchitektur Binärzahlen mit einer Länge zur Verfügung stellt, die größer oder gleich der Anzahl der Felder des Spielbretts sind. Aus diesem Grund sind Bitboards erst mit der Einführung der 32-Bit- und 64-Bit-Prozessoren populär geworden.

4.4.1 Stellungsrepräsentation durch Bitboards

Eine Brettstellung p kann als ein Array von Bitboards betrachtet werden. Jeder Eintrag entspricht dabei dem Bitboard genau einer Figur. Die Nummerierung der Figuren, die nicht kontinuierlich sein muss, startet bei 1 und endet bei *maxpiece*. Befindet sich eine bestimmte Figur nicht auf dem Spielbrett, ist ihr zugehöriges

Bitboard leer, was bedeutet, dass alle Bits gelöscht sind. Im folgenden sei p_f das Bitboard der Figur f. Alle Bitboards zusammen bilden mit der Information, welcher der beiden Spieler am Zug ist, eine eindeutige Repräsentation der Stellung. Abbildung 16 zeigt eine Dodgem-Stellung in der Bitboard-Repräsentation.

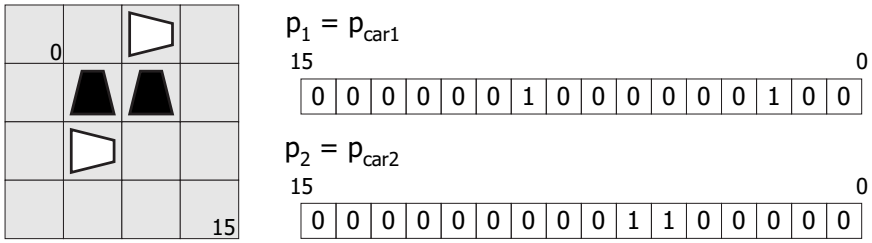


Abbildung 16: Die Bitboards einer Dodgem-Stellung

4.4.2 Indizierung eines einzelnen Bitboards

Wurde bei der einfachen kombinatorischen Indizierung jede Figur für sich betrachtet, so werden nun Figuren eines Typs zusammengefasst. Sei b ein Bitboard der Größe m. Weiter sei $b^i \in \{0,1\}$ das Bit an der Stelle i. Die Anzahl der gesetzten Bits sei mit $|b| = l$ festgelegt. Die folgende Funktion bildet b auf einen ganzzahligen Index aus dem Intervall $[0..\binom{m}{l} - 1]$ ab.

$$idx_{bitboard}(b) = \sum_{i=m-1}^{0} b^i * \binom{i}{\sum_{j=0}^{i} b^j}$$

Die Umkehrfunktion $\overline{idx}_{bitboard}$, die zu einem gegebenen Index idx das entsprechende Bitboard liefert, kann als Pseudocode beschrieben werden.

```
FOR i = m − 1 TO 0 STEP -1
   IF idx ≥ (i l) THEN
      idx = idx - (i l)
      b^i = 1
      l = l - 1
   ELSE
      b^i = 0
   END IF
END FOR
```

4.5 Indizierung einer Brettstellung

Die Indizierung einer kompletten Brettstellung baut auf der Indizierung von Bitboards auf. Um einen Index für jedes der Bitboards $p_{1..f}$ zu berechnen, sind Modifikationen notwendig. So muss bei der Berechnung eines Index für die Figuren vom Typ f das Spielbrett für alle nachfolgenden Figuren verkleinert werden. Die Bits, auf denen die Figuren vom Typ f standen, werden nicht einfach gelöscht, sondern ausgeschnitten. Höherliegende Bits rücken an ihre Position. Nur so ist gewährleistet, dass alle nachfolgend betrachteten Figuren nur noch auf Felder von 0 bis $\sum_{i=1}^{f} m - |p_i| - 1$ stehen können. Dies leistet die folgende Funktion.

$$sub(p,\ b) = p_{new}\ mit\ p^j_{new} = p^{j+\sum_{i=0}^{j} b^i}$$

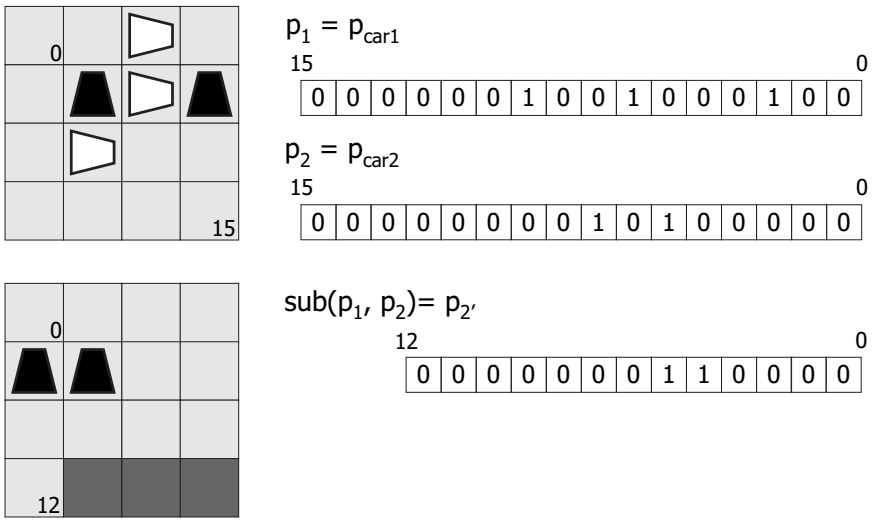


Abbildung 17: *sub*-Funktion am Beispiel einer Dodgem-Stellung

Um zu einem eindeutigen Index zu gelangen, muss jeder Index eines einzelnen Bitboards mit dem maximalen Wert zuzüglich 1 multipliziert werden, der durch die noch folgenden Figuren entstehen kann. Dieser Wert ist identisch mit der Anzahl der Möglichkeiten die Figuren vom Typ f bis *maxpiece* auf dem Spielbrett zu platzieren, wenn sich die Figuren vom Typ 1 bis $f-1$ bereits auf dem Brett befinden. Sie kann durch die folgende Funktion bestimmt werden.

$$poss(k) = \prod_{i=k}^{maxpiece} \left(\frac{\overbrace{f - \left(\sum_{j=1}^{i} |p_j| \right)}^{\substack{Anzahl\ der\ verleibenden\ Felder \\ für\ Figuren\ vom\ Typ\ i}}}{\underbrace{|p_i|}_{\substack{Anzahl\ der\ Figuren \\ vom\ Typ\ i}}} \right)$$

Aus den bisher vorgestellten Funktionen kann nun eine Indexfunktion definiert werden.

$$idx(p) = \sum_{i=1}^{maxpiece} idx_{bitboard}(sub(p_i,\ b)) \ *\ poss(i)$$

$$mit\ \ b^j = \begin{cases} 1 & wenn \sum_{k=1}^{i-1} p_k^j = 1 \\ 0 & sonst \end{cases}$$

Diese Indexfunktion hat den Nachteil, dass sie von einer festen Anzahl von Figuren ausgeht. Dies stellt jedoch kein echtes Problem dar. Für jede mögliche Kombination von Figuren wird eine Subdatenbank angelegt. Ihr Index innerhalb der Datei berechnet sich aus der Summe der Größe aller vor ihr erschaffenen Subdatenbanken. Die Beschaffenheit der Indexfunktion impliziert automatisch eine Umkehrfunktion. Da ihre Konstruktion analog verläuft, wird auf eine genaue Beschreibung an dieser Stelle verzichtet.

4.6 Spiegelsymmetrie

Bei der Spiegelsymmetrie möchte man sich zu Nutze machen, dass es Stellungen gibt, die auch mit vertauschten Farben und der Gegenseite am Zug in der Datenbank sind. Diese Stellungen sind zwar grundsätzlich verschieden, können jedoch das gleiche Problem darstellen und deshalb den gleichen spieltheoretischen Wert haben. Abb. 18 zeigt einen solchen Fall.

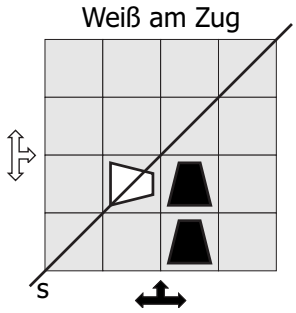


Abbildung 18: Stellung 1

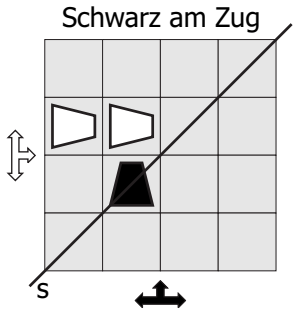


Abbildung 19: Stellung 2

Es ist egal, ob in Abb. 18 Weiß am Zug ist oder in Abb. 19 Schwarz. Der spieltheoretische Wert ist identisch. Die automatische Erkennung, ob ein Spiel eine Achse zur Spielsymmetrie besitzt, ist zu aufwendig. Bei der Spezifikation eines Spiels muss deshalb angegeben werden, ob eine solche Achse existiert. Ist dies der Fall, muss eine Abbildungsvorschrift in Form eines Arrays formuliert werden. Beim Reservieren von Speicherplatz für die Datenbank ist es dann einfach die Spiegelsymmetrie zu berücksichtigen. Soll eine neue Subdatenbank angelegt werden, so wird vorher überprüft, ob bereits eine Subdatenbank mit vertauschten Farben und der Gegenseite am Zug existiert. In diesem Fall ist keine neue Subdatenbank nötig. Wird beim Generierungsprozess oder im späteren Spiel versucht auf eine Subdatenbank zuzugreifen, die nicht existiert, so werden die Farben der Figuren getauscht, der Platz jeder Figur gemäß der Abbildungsvorschrift geändert und das Zugrecht an die Gegenseite übergeben. Der spieltheoretische Wert der resultierenden Stellung ist dann gleich dem spieltheoretischen Wert der ursprünglichen Stellung.

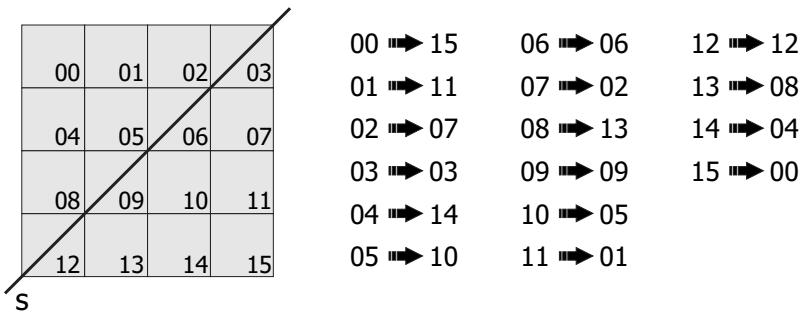


Abbildung 20: Abbildungsvorschrift für Dodgem

Bei Brettspielen wie Dodgem oder Mühle wird die Größe der Datenbank dadurch auf genau die Hälfte reduziert, da nur noch die Subdatenbanken für eine Seite am Zug vorhanden sein müssen. Die Datenbankgenerierung wird dadurch mehr als doppelt so schnell. Mühle hat zudem die Besonderheit, dass gar keine Abbildungsvorschrift

nötig ist. Ein einfaches Vertauschen der Farben genügt. Die Spiegelsymmetrieachse beim Schach ist in Abb. 21 eingezeichnet. Befindet sich kein Bauer auf dem Spielbrett, so ist ebenfalls keine Abbildungsvorschrift nötig.

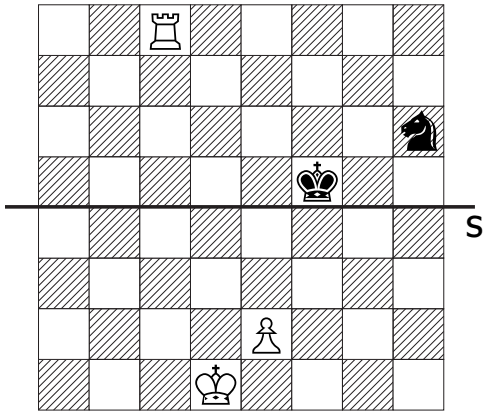


Abbildung 21: Spiegelsymmetrie beim Schachspiel

4.7 Berücksichtigung von Brettsymmetrien

Das Ausnutzen von Brettsymmetrien zum Verkleinern der Datenbank ist deutlich schwieriger als bei der Spiegelsymmetrie. Bei Brettsymmetrien werden lediglich die Figuren umgestellt. Es erfolgt kein Austausch der Farben oder Wechseln des Zugrechts. Deshalb befinden sich zwei Stellungen, die unter Berücksichtigung von Symmetrie identisch sind, stets in der gleichen Subdatenbank. Da es bei Dodgem keine Symmetrieachsen gibt, wird an dieser Stelle das Schachspiel zur Veranschaulichung benutzt. Wenn sich noch Bauern auf dem Spielbrett befinden, gibt es genau eine Symmetrieachse (Abb. 22). Ohne Bauern auf dem Spielbrett sind es vier Symmetrieachsen, wobei eine redundant ist.

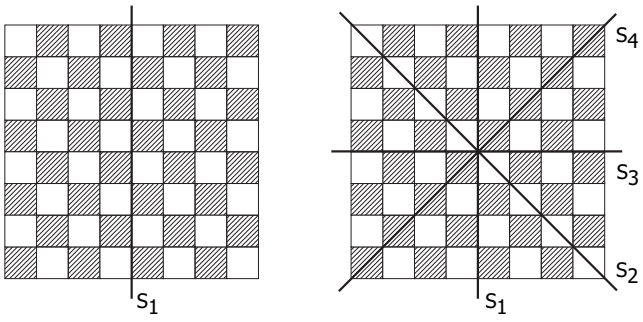


Abbildung 22: Symmetrieachsen beim Schachspiel mit und ohne Bauern

Ein guter Ansatz zur Platzersparnis durch Symmetrie findet sich bei den Schachendspielen von Edwards. Dort wird eine Referenzfigur gewählt, die in jeder Stellung vorhanden ist. Dies ist z.B. der weiße König. Für ihn wird eine *Symmetrietabelle* angelegt. Dazu wird versucht den weißen König nacheinander auf jedes der 64 Felder

zu platzieren. Bei jedem Schritt werden die Felder gestrichen, die durch Spiegelung einer Kombination von Symmetrieachsen erreicht werden. Sie dürfen während dieser Iteration nicht mehr betreten werden. Nur neue Felder werden in die Tabelle mit aufgenommen.

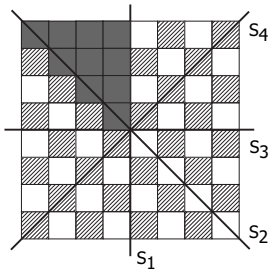


Abbildung 23: Mögliche Felder für den weißen König

0	0
1	1
2	2
3	3
4	9
5	10
6	11
7	18
8	19
9	27

Abbildung 24: Symmetrietabelle

Die entsprechende Subdatenbank wird nun in 10 Teile aufgeteilt. Jeder Teil ist von der Größe des maximalen Index ohne den weißen König. Wird nun auf eine Stellung zugegriffen, so werden alle möglichen Kombinationen von Symmetrien auf sie angewandt, bis der weiße König auf einem der 10 Felder steht. Anhand der Tabelle wird festgestellt, um welchen Tabellenindex es sich handelt. Für eine effiziente Bestimmung ist es nötig die Tabelle nach den Werten der rechten Spalte zu ordnen. Das Ergebnis ist eine Zahl aus dem ganzzahligen Intervall [0..9]. Nun ist bekannt in welchen Teil der Subdatenbank gesprungen werden muss.

Im allgemeinen Fall kann nicht von einer Figur ausgegangen werden, die sich in jeder Stellung auf dem Brett befindet. Deshalb muss für jede Subdatenbank eine eigene Referenzfigur für die Symmetrie gewählt werden. Um die Tabelle klein zu halten wird die Figur gewählt, deren Vorkommen auf dem Brett am geringsten sind. Handelt ist sich hierbei um mehr als eine Figur, so sind deutlich mehr Tabelleneinträge notwendig. In der rechten Spalte wird nicht mehr das Feld der Figur gespeichert, sondern das entsprechende Bitboard. Dies ist z.B. bei Mühle notwendig, wo stets mehrere Steine einer Farbe auf dem Spielbrett sind. Wie Gasser in [Gas95] die Symmetrie ausnutzt, wird nicht sehr detailliert beschrieben. Die Resultate lassen jedoch darauf schließen, dass er genauso verfahren ist.

Diese Form der Berücksichtigung von Brettsymmetrien ist nicht optimal. Es gibt weiterhin - symmetrisch betrachtet - identische Stellungen, die mehrmals in der Datenbank vorhanden sind. Dieser Fall tritt immer dann auf, wenn die Referenzfigur direkt auf der Symmetrieachse liegt. Abbildung 25 zeigt zwei dieser Stellungen.

Dieses harmlos wirkende Phänomen verfälscht den Generierungsprozess und führt zu inkorrekten Ergebnissen. War die Anzahl der Nachfolger, die bei der optimierten Version der Retrograde Analysis eine wichtige Rolle spielt, bisher leicht zu bestimmen, muss sie nun auf die Symmetrie Rücksicht nehmen. Auch die Markierung eines Vorgängers als verlorene Stellung wird zum Problem, da nicht klar ist, ob sich vielleicht ein symmetrisch identischer Vorgänger in der Datenbank befindet, der ebenfalls markiert werden muss. Der Algorithmus in seiner ursprünglichen Form funktioniert aus diesen Gründen nicht mehr. Weder Thompson und Edwards, noch Gasser beschreiben in ihren Arbeiten eine Lösung des Problems. Ihre Ergebnisse zeigen jedoch, das sie alle dieses Problem gelöst haben müssen. An dieser Stelle kann nur darüber spekuliert werden, auf welche Weise dies geschah. Bei Nalimov ist bekannt, dass er auf die vorgestellte Optimierung bei der Retrograde Analysis verzichtet [Nal05], um diese Probleme zu umgehen. Da die Datenbankgenerierung bei Edwards und Thompson langsamer bzw. ähnlich schnell wie bei Nalimov ist,

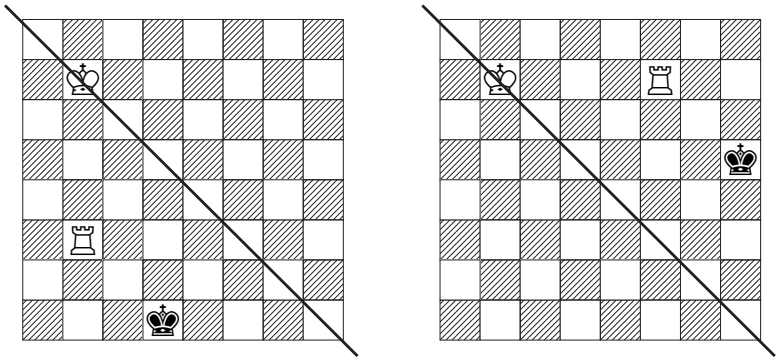


Abbildung 25: Symmetrisch identische Stellungen in der Datenbank

liegt die Vermutung nahe, dass auch sie auf die Optimierung verzichtet haben. Eine Möglichkeit, die die Symmetrie berücksichtigt ohne auf die Optimierung zu verzichten, soll im Folgenden erläutert werden.

Das in dieser Arbeit angewandte Verfahren basiert auf der alleinigen Betrachtung der symmetrisch identischen Stellung, die den niedrigsten Index hat. Während der Initialisierungphase müssen deshalb die Stellungen als illegal markiert werden, zu denen eine symmetrisch identische Stellung mit niedrigerem Index existiert. Auf die so markierten Stellungen wird in der Folge nicht mehr zugegriffen. Der niedrigste Index einer Stellung kann schnell ermittelt werden, da symmetrische identische Stellungen einfach auszumachen sind. Schwieriger wird es bei den Vor- und Nachfolgern einer Stellung. Hier müssen die symmetrisch identischen Stellungen herausgefiltert werden. Dazu muss zu jeder Stellung der zugehörige Index ermittelt werden. Stellungen mit gleichem Index werden gelöscht. Dies ist mit etwas Zeitaufwand verbunden. Die Effizienz der Optimierung aus Kapitel 3 sinkt mit diesen notwendigen Maßnahmen deutlich. Im Gegensatz zu einer nicht-optimierten Version des Algorithmus ist sie jedoch deutlich schneller und rechtfertigt den zusätzlichen Aufwand.

4.8 Unerreichbare Felder

Bei vielen Brettspielen gibt es Figuren, die nicht jedes Feld erreichen können. So kann z.B. ein Bauer im Schachspiel nicht auf den beiden Grundreihen stehen. Die Kombinatorik des vorgestellten Algorithmus berücksichtigt diese Einschränkung bisher nicht. So enthält ein Schachendspiel mit Bauern jede Stellung; auch solche, in denen ein Bauer auf der Grundreihe steht. Dies führt zu vielen illegalen Stellungen in der Datenbank, die aus den bekannten Effizienzgründen eliminiert werden sollen.

Bei Dame und Mühle existieren keine Figuren, deren Bewegungsfreiheit auf dem Feld eingeschränkt ist. Die einzigen im wissenschaftlichen Kontext betrachteten Spiele, in denen diese Beschränkungen auftauchen, sind Schach und *Chinese Checkers*. Chinese Checkers ist ein in Europa relativ unbekanntes Brettspiel. Hier gibt es deutlich mehr Figuren, die nicht jedes Feld erreichen können. In [WuBel02] wird eine Methode vorgestellt diese Eigenschaften auszunutzen. Sie basiert allerdings auf spielspezifischen Annahmen und bietet daher keine Grundlage für einen allgemeinen Ansatz. Beim Schach ist die einzige Figur mit Bewegungseinschränkung der Bauer. Zwar kann sich auch ein Läufer nur auf den weißen oder schwarzen Feldern, also genau auf der Hälfte der Felder des Spielbretts bewegen; da aber jeder Spieler sowohl weiß- als auch schwarzfeldrige Läufer besitzen kann, macht es keinen Sinn, die Felder einzuschränken, auf denen sich ein Läufer aufhalten kann. Dann

nämlich müssten beide Arten von Läufern als unterschiedliche Figuren behandelt werden, was unnötig aufwendig wäre. Ohne Bewegungseinschränkung deckt das Endspiel *KR-KB* (König und Turm gegen König und Läufer) sowohl alle Stellungen mit einem weißfeldrigen Läufer als auch alle Stellungen mit einem schwarzfeldrigen Läufer ab. Die Einbindung in den Algorithmus zur Indizierung ist beim Schach einfach. Der Bauer, der 48 Felder erreichen kann, wird zuerst auf dem Spielfeld platziert. Dadurch sinkt die Anzahl der kombinatorisch möglichen Stellungen von $64 * x$ auf $48 * x$, wobei x die kombinatorisch möglichen Stellungen der restlichen Figuren auf 63 Feldern sind.

Voraussetzung dafür ist, dass die Figuren aufsteigend nach der Anzahl der Felder geordnet werden, die sie erreichen können. Der Grund dafür wird am Beispiel von Schach deutlich. Steht bereits ein Bauer auf dem Spielbrett, so gibt es für eine nachfolgende Figur, z.B. einen König, nur noch 63 Möglichkeiten. Umgekehrt gilt dies nicht. Steht der König bereits auf dem Feld, so kann keine Aussage darüber getroffen werden, auf wie vielen Feldern ein Bauer platziert werden kann. Dies können sowohl 47, als auch 48 sein. Da stets der ungünstigste Fall angenommen werden muss, bleiben 48 Möglichkeiten für den Bauern, so dass keine Einsparung an Stellungen stattgefunden hat.

Im allgemeinen Fall muss davon ausgegangen werden, dass es mehrere bewegungsbeschränkte Figuren gibt. In den folgenden Bitboards sind die Felder markiert, die eine Figur betreten darf. Sie werden in der Folge *Movement-Bitboards* bezeichnet. Die Movement-Bitboards der Figuren 1 bis i seien m_1 bis m_i.

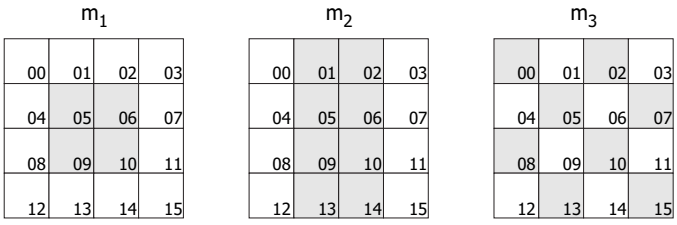


Abbildung 26: Erreichbare Felder der Figuren 1 bis 3

Sind 2 Figuren vom Typ 1 auf dem Spielfeld und 3 Figuren vom Typ 2, so ist leicht ersichtlich, dass eine Figur vom Typ 3 nicht mehr auf jedem der acht Felder stehen kann. Selbst wenn die Figuren vom Typ 1 auf den Feldern 6 und 9 stehen, die die Figur vom Typ 3 nicht erreichen kann, so müssen die Figuren vom Typ 2 mindestens ein Feld besetzen, dass die Figur vom Typ 3 nicht erreichen kann. Die normalerweise $\binom{4}{2} * \binom{6}{3} * \binom{8}{1}$ möglichen Stellungen lassen sich demnach auf $\binom{4}{2} * \binom{6}{3} * \binom{7}{1}$ mögliche Stellungen beschränken. Die Zahl der tatsächlichen Stellungen ist zwar immer noch kleiner, lässt sich jedoch mit reiner Kombinatorik nicht mehr erfassen.

Bei der Bestimmung der Anzahl der Felder, auf die eine Figur bei bereits vorhandenen Figuren auf dem Brett platziert werden kann, handelt es sich um ein Problem, dessen Lösung exponentiellen Zeitaufwand benötigt. Soll die Anzahl möglicher Felder einer Figur i bestimmt werden, so muss über alle Kombinationen von Movement-Bitboards der Figuren 1 bis $i-1$ iteriert werden. Sie werden mit der OR-Operation verknüpft. Das invertierte Movement-Bitboard der Figur i ($\overline{m_i}$) gibt an wo Figuren platziert sein dürfen, die keinen Einfluss auf m_i nehmen. Ist die Anzahl der Figuren auf den verknüpften Movement-Bitboards größer als die Anzahl der gesetzten Bits bei der AND-Verknüpfung dieses Bitboards mit ($\overline{m_i}$), so müssen einige Figuren auf Feldern stehen, die die Figur m_i betreten darf.

Sei b_i das Bitboard der Figur i. Weiter sei x eine Binärzahl und x^i das i-te Bit von x. Mathematisch kann die Anzahl der möglichen Felder v_i für die Figur i wie folgt berechnet werden:

$$v_i = min\left(|m_i| - \sum_{j=0}^{i-1} |b_j| + |z \;\; AND \;\; \overline{m_i}|\right)$$

$$mit\ z = \left(x^0 * m_1 \;\; OR \;\; x^1 * m_2 \;\; OR \;\; ... \;\; OR \;\; x^{i-2} * m_{i-1}\right) \;\; für\ x \in [1 \,...\, 2^m - 1]$$

Für das Beispiel aus Abb. 26 ergibt sich die folgende Rechnung:

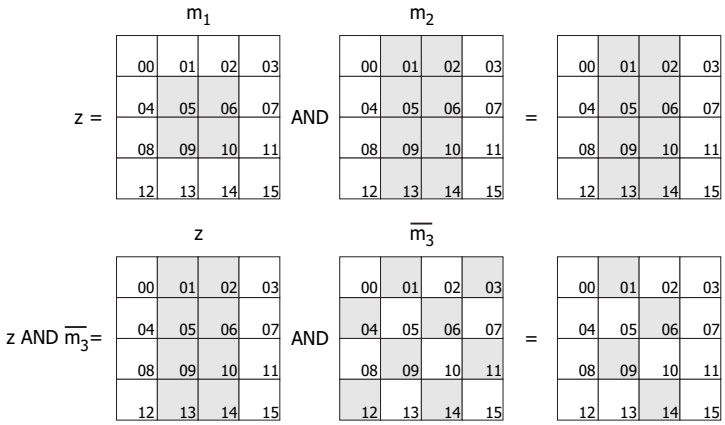


Abbildung 27: Berechnung von v_3 für das Beispiel aus Abb. 26

Mit $|b_1| = 2$ und $|b_2| = 3$ folgt

$$v_3 = |m_3| - \sum_{j=0}^{2} |b_j| + |z \;\; AND \;\; \overline{m_3}| \;\; = \;\; 8 \;\; - \;\; 5 \;\; + \;\; 4 \;\; = \;\; 7$$

da andere Kombinationen aus m_1 und m_2[13] keinen besseren Wert ergeben.

Um die Zeit zur Berechnung aller v_i gering zu halten, ist die Zahl der unterschiedlichen Figuren, die nicht alle Felder erreichen können, auf 16 beschränkt. Damit sind im ungünstigsten Fall $2^{17} - 1$ Iterationsschritte nötig, um alle v_i zu berechnen. Aufgrund der wenigen und einfachen Bitoperationen, die in jedem Schritt nötig sind, dauert dieser Prozess nur den Bruchteil einer Sekunde.

Die Einbindung von unerreichbaren Feldern in die Indizierungsfunktion geschieht bei der *sub*-Funktion, die zur Indexbestimmung eines einzelnen Bitboards dient. Sie wird dahingehend geändert, dass nicht mehr nur die Felder ausgeschnitten werden auf denen Figuren eines Typs niedrigerer Nummer stehen, sondern auch diejenigen, die von der betreffenden Figur nicht erreicht werden können. Da die Zahl der tatsächlichen Stellungen häufig kleiner als der reservierte Indexbereich ist, kann das Problem entstehen, dass zu einem Index keine Stellung existiert. Ein Beispiel soll diese Tatsache verdeutlichen. Seien die Movement-Bitboards der Figuren vom Typ 1 und 2 wie in Abb. 28. Auf dem Spielbrett sollen sich drei Figuren vom Typ 1 und eine Figur vom Typ 2 befinden.

[13] In diesem Beispiel sind andere Kombinationen nur die beiden Movement-Bitboards allein.

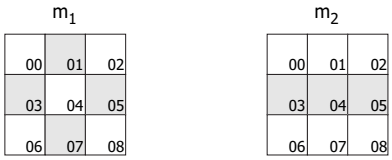


Abbildung 28: Erreichbare Felder der Figuren vom Typ 1 und Typ 2

Es gibt $\binom{4}{3}$ Möglichkeiten die 3 Figuren des Typs 1 auf dem Spielfeld zu platzieren. Da mindestens eine Figur ein erreichbares Feld von Figur 2 blockiert, beschränken sich die Möglichkeiten von Figur 2 auf $\binom{2}{1}$. Damit liegt der Indexbereich im Intervall [0..7], da $\binom{4}{3} * \binom{2}{1} = 8$. Blockieren jedoch die 3 Figuren vom Typ 1 die beiden Felder der mittleren Reihe, so gibt es für die Figur vom Typ 2 nur noch eine Möglichkeit. Insgesamt gibt es damit nur 6 mögliche Stellungen. Die zwei Indizes (in diesem Beispiel 1 und 7), zu denen keine Stellung existiert, werden während der Initialisierungsphase als illegal markiert und von weiteren Zugriffen ausgeschlossen. Sie sind leicht auszumachen, da bei ihrer Umrechnung in eine Brettstellung stets der Fehler auftritt eine Figur auf ein Feld setzen zu wollen, welches nicht vorhanden ist.

4.9 Die Güte der allgemeinen Indexfunktion

Die beiden folgenden Tabellen vergleichen den maximalen Index der allgemeinen Indexfunktion mit den Indexfunktionen, die für ein bestimmtes Spiel entworfen wurden. Die optimale Stellungsanzahl bezieht sich dabei auf alle legalen Stellungen eines Spiels abzüglich solcher, die durch Spiegelung an den Symmetrieachsen erzielt werden können. Es zeigt sich, dass die allgemeine Indexfunktion bei Schachendspielen ohne Probleme mit den speziellen Indexfunktionen von Edwards und Thompson konkurrieren kann. Bei Nalimov fließt sehr viel Schachwissen in die Indexfunktion mit ein. So können die beide Könige nicht nebeneinander stehen. Auch die Dame des Spielers, der am Zug ist, darf nicht direkt neben dem König des Kontrahenten stehen. Nalimovs Indexfunktion ist damit bei jedem Endspiel den anderen überlegen. Trotz dieser und einiger weiterer Verbesserungen ist sie jedoch noch weit vom Optimum entfernt. Eine genaue Beschreibung der Indexfunktionen von Edwards, Thompson und Nalimov findet sich in [NaHaHe00].

Endspiel	Optimal	Edwards	Thompson	Nalimov	Bahnsen
KR-K	49.811	81.920	59.136	55.674	78.120
KRR-K	1.362.431	5.242.880	3.784.704	1.747.284	2.382.600
KQR-K	2.392.019	5.242.880	3.784.704	3.247.560	4.765.320
KP-K	165.676	232.144	196.608	165.676	187.488

Abbildung 29: Anzahl der Stellungen verschiedener Indexfunktionen bei Schachendspielen

Bei Mühle liegt das Minimum der allgemeinen Indexfunktion über dem der Indexfunktion von Gasser, die speziell für dieses Spiel entwickelt wurde. Auch hier ist spielspezifisches Wissen der Grund. In der Subdatenbank mit je 9 Spielsteinen auf jeder Seite gibt es zahlreiche Stellungen, in denen eine Mühle geschlossen ist. Solche Stellungen sind illegal, denn bei einer geschlossenen Mühle können nicht mehr

alle Spielsteine auf dem Brett sein. Gasser schafft es einen Teil dieser Stellungen zu eliminieren. Prozentual betrachtet ist die allgemeine Indexfunktion jedoch nur geringfügig schlechter. Alle Werte beziehen sich auf die Datenbanken der *Spielphase*. Sie beginnt, wenn beide Spieler ihre 9 Steine auf dem Spielbrett platziert haben. Der optimale Wert wurde von Gasser selbst errechnet und ist aus [Gas95] übernommen.

Optimal	Gasser	Stahlhacke	Bahnsen
7.673.759.269	8.102.965.583	9.074.932.579	9.074.932.579

Abbildung 30: Anzahl der Stellungen verschiedener Indexfunktionen bei Mühle

Weitere Vergleichsmöglichkeiten gibt es nicht. Da es bei den Endspielen zu Dame keine Symmetrieachsen gibt, erübrigt sich eine Betrachtung. Hier erreicht die allgemeine Indexfunktion, wie auch die von Schaeffer, den optimalen Wert. Gleiches gilt für Dodgem.

5 Dateicaching in der Generierungsphase

Bereits im vorangegangenen Kapitel wurden die Probleme verdeutlicht, die bei anhaltenden zufälligen Zugriffen auf eine große Datei entstehen. Nahezu jeder Zugriff erfordert die Neupositionierung des Schreib-/Lesekopfes der Festplatte. Eine Sequenz von Datenbankzugriffen bezieht sich meistens auf eine Reihe von Vorgängern oder Nachfolgern einer Stellung. Ihre Lage hängt sowohl von der Indexfunktion, als auch vom Spiel selbst ab. Die Komplexität dieses Zusammenspiels lässt keine Rückschlüsse darüber zu wo der nächste Zugriff stattfindet, so dass in diesem Fall durchaus von Zufälligkeit gesprochen werden kann, obwohl es sich um ein deterministisches System handelt.

Ein kleines Rechenbeispiel soll verdeutlichen wie abhängig die Dauer der Datenbankgenerierung von einem schnellen Zugriff auf die Werte einzelner Stellungen ist. Man denke sich eine Datenbank mit 2 Milliarden Werten. Bei nur 3 Zugriffen auf jeden Wert während der gesamten Generierung, was einen sehr niedriger Wert darstellt, sind insgesamt 6 Milliarden Zugriffe nötig. Dauert jeder Zugriff durchschnittlich nur eine Millisekunde, würden allein die Dateioperationen über zwei Monate in Anspruch nehmen. Eine Zwischenspeicherung der Werte im Arbeitsspeicher ist aus diesem Grund unumgänglich. Üblicherweise geschieht diese Zwischenspeicherung, auch Caching genannt, in Blöcken bestimmter Größe. Beim Zugriff auf einen Wert der Datenbank wird ein ganzer Block von der Festplatte gelesen, mit der Hoffnung, dass spätere Zugriffe denselben Block betreffen. Da aber die Zugriffe scheinbar willkürlich erfolgen, ist die Wahrscheinlichkeit groß, dass der Block ohne weitere Verwendung aufgrund des begrenzten Arbeitsspeichers durch einen anderen ersetzt werden muss. Das Cachen einzelner Werte allein ist jedoch auch keine gute Lösung. Muss ein einzelner Wert von der Festplatte gelesen werden, so ist mit der Neupositionierung des Schreib-/Lesekopfes der Festplatte bereits die meiste Arbeit getan. Das eigentliche Lesen bzw. Schreiben der Daten dauert nur ein Bruchteil dieser Zeit, so dass es kaum einen Unterschied macht, ob 1 Byte oder 1000 Bytes gelesen oder geschrieben werden. Aus diesen Gründen wird ein zweistufiges System zum Cachen der Daten verwendet, das in Abb. 31 dargestellt ist.

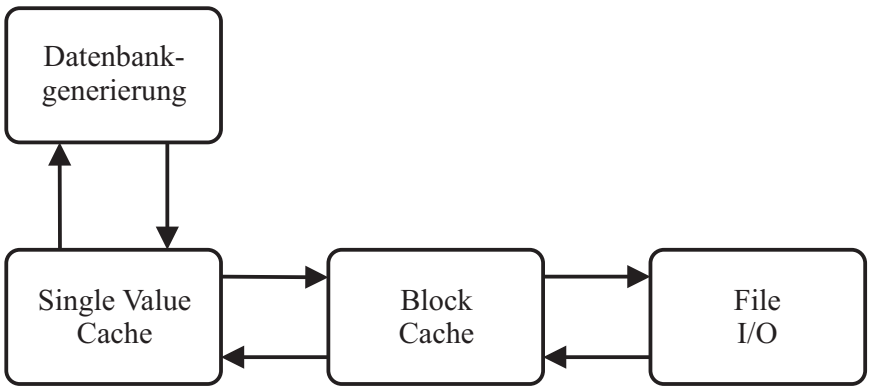


Abbildung 31: Schematische Darstellung des Caching-Systems

Die Anfrage für eine Leseoperation wird stets an den *Single Value Cache* gestellt. Er speichert die Werte einzelner Stellungen. Ist der gewünschte Eintrag nicht vorhanden, so wird die Anfrage an den *Block Cache* weitergeleitet. Findet sich auch dort der passende Eintrag nicht, so muss er von der Festplatte gelesen werden. Ein Schreibvorgang funktioniert ähnlich. Gehört der vorhandene Wert zu derselben Stel-

lung wie der neue Wert, so kann er überschrieben werden. Ist dies nicht der Fall, so wird der vorhandene Wert vor dem Überschreiben an den *Block Cache* weitergeleitet. Ist der passende Block vorhanden, wird dort der Wert geändert, andernfalls muss ein alter Block auf die Festplatte geschrieben werden, damit der passende Block gelesen und geändert werden kann.

Der *Single Value Cache* besteht aus 64-Bit-Werten, deren 16 obere Bits den spieltheoretischen Wert beinhalten und die 48 unteren Bits den Index der zugehörigen Brettstellung. Je nach Größe des Cache-Speichers geben nur die ersten n Bits des Index die Position im Cache-Speicher an. Bei 1024 Werten wären dies 10 Bits, da $2^{10} = 1024$. Damit teilen sich mehrere Stellungen einen Eintrag im Cache-Speicher. Um festzustellen, ob es sich um den richtigen Eintrag handelt, müssen noch die restlichen (in diesem Fall 38) Bits verglichen werden. Nur wenn sie identisch sind, gehört der eingetragene spieltheoretische Wert tatsächlich zu der betrachteten Stellung.

Der *Block Cache* besteht aus Blöcken von 32 KB. Eine Gewichtung einzelner Blöcke, die die Häufigkeit der Zugriffe berücksichtigt, findet nicht statt, da die Unterschiede nur gering sind. Sie hängen von der Anzahl der Vorgänger und Nachfolger einer Stellung ab, die - über einen ganzen Block betrachtet - relativ konstant ist. Einem Block im Cache-Speicher werden deshalb mehrere Dateiblöcke zugeordnet. Beim Zugriff wird überprüft, ob sich der korrekte Dateiblock im Speicher befindet. Dieser wird bei Bedarf von der Festplatte gelesen.

Die Größe von *Single Value Cache* und *Block Cache* muss bei Programmstart festgelegt sein. Die folgende Tabelle zeigt den Geschwindigkeitsgewinn am Beispiel das Spiels *Losing Tic Tac Toe*, das später noch genauer vorgestellt wird. Die resultierende Datenbank hat eine Größe von 1318 KB. Die Prozentangaben der Größe des Cache-Speichers beziehen sich auf diese Zahl.

Cachegröße	Zeit
100%	01m 22s
50%	02m 09s
40%	03m 05s
20%	03m 47s
10%	04m 06s
5%	04m 22s
2,5%	04m 29s

Tabelle 2: Generierungsdauer der *Losing Tic Tac Toe* Datenbank in Abhängigkeit von der Größe des Caches

Aufgrund der minimalen Cachegröße von 32 KB ist eine Messung mit einer geringeren Prozentzahl bei dieser Datenbank nicht möglich. Messungen mit anderen Datenbanken zeigen, dass die Generierungsdauer bei weniger als 2% Cachespeicher nicht weiter sinkt. Bei derart kleinen Werten hebt der Aufwand für die Verwaltung der gecachten Daten die eigentliche Wirkung wieder auf.

6 Ausgewählte Spiele

In diesem Kapitel wird das Programm zum Generieren von Datenbanken an einigen Spielen getestet. Alle diese Spiele haben eine relativ geringe Komplexität. Ursache dafür ist die Zeit, die bei der Erstellung dieser Arbeit nicht in unbegrenztem Umfang zur Verfügung stand. Aus diesem Grund beschränken sich die ausgewählten Spiele auf einfache, aber repräsentative Beispiele.

Neben einer ausführlichen Statistik der Datenbank werden zu jedem Spiel, mit Ausnahme der Schachendspiele, auch einige interessante Spielsituationen aufgezeigt, die mit Hilfe der Datenbank analysiert wurden. Für *Dodgem* wird zusätzlich der Quellcode vorgestellt, der den geringen Aufwand verdeutlichen soll, der zum Bau eines perfekten Spielers mit Hilfe dieses Ansatzes noch nötig ist.

Alle Zeiten beziehen sich auf ein 32-Bit-Betriebssystem mit einem 800 Mhz-Prozessor vom Typ Athlon und 1024 MB Arbeitsspeicher, der mit 133Mhz getaktet ist. Die Cache-Größe wurde mit jeweils 100% der Größe der resultierenden Datenbank festgelegt. Die Datenbanken wurden also ohne Zugriff auf die Festplatte erzeugt.

6.1 Das Schachendspiel KR-K

König und Turm gegen König ist eines der einfachsten Endspiele im Schach. Seine Besonderheiten sind bereits seit Jahrhunderten bekannt. Die Datenbank birgt daher keinerlei Neuigkeiten. Die Ergebnisse sind nahezu identisch mit denen von Nalimov. Sie unterscheiden sich lediglich in der Anzahl der Stellungen, die unentschieden sind. Die Ursache liegt in den Unterendspielen (in diesem Fall das Endspiel K-K), die bei diesem Ansatz automatisch generiert werden. Bei dem Datenbankgenerator von Nalimov müssen alle Unterendspiele separat erzeugt werden.

Die Tabelle zeigt die Datenbankstatistik ohne Ausnutzung der Symmetrie und mit Ausnutzung der Symmetrie. Die deutlich geringere Anzahl der Stellungen und die dadurch kleinere Datenbankgröße sowie kürzere Berechnungszeit machen den Vorteil deutlich, der bei der Ausnutzung der Symmetrie entsteht.

Stellungen	**Ohne Symmetrie**		**Mit Symmetrie**	
Gesamt	508.032		79.380	
Illegal	101.696 (20.45%)		15.932 (20.1%)	
Symmetrisch identisch	0 (0.00%)		12.509 (15.8%)	
Tatsächlich verwendet	406.336 (79.98%)		50.939 (64.1%)	
	wtm	btm	wtm	btm
Verloren in 0	0	216	0	27
Gewonnen in 1	1.512	0	189	0
Verloren in 2	0	624	0	78
Gewonnen in 3	4.676	0	587	0
Verloren in 4	0	1.948	0	246
Gewonnen in 5	3.852	0	484	0
Verloren in 6	0	648	0	81
Gewonnen in 7	1.900	0	238	0
Verloren in 8	0	1.584	0	198
Gewonnen in 9	4.848	0	607	0
Verloren in 10	0	3.768	0	471
Gewonnen in 11	8.708	0	1.091	0
Verloren in 12	0	4.728	0	592
Gewonnen in 13	11.320	0	1.418	0
Verloren in 14	0	5.444	0	683
Gewonnen in 15	17.172	0	2.149	0
Verloren in 16	0	11.448	0	1.433
Gewonnen in 17	20.088	0	2.514	0
Verloren in 18	0	13.672	0	1.712
Gewonnen in 19	2	9	2.382	0
Verloren in 20	123	345	0	1.985
Gewonnen in 21	20.476	0	2.565	0
Verloren in 22	0	22.788	0	2.854
Gewonnen in 23	21.480	0	2.691	0
Verloren in 24	0	28.732	0	3.597
Gewonnen in 25	17.824	0	2.234	0
Verloren in 26	0	33.516	0	4.194
Gewonnen in 27	16.136	0	2.027	0
Verloren in 28	0	36.372	0	4.553
Gewonnen in 29	5.244	0	662	0
Verloren in 30	0	17.284	0	2.166
Gewonnen in 31	916	0	121	0
Verloren in 32	0	3.056	0	390
Summe Gewonnen	175.168 (97.98%)	0 (0.00%)	21.959 (97.94%)	0 (0.00%)
Summe Verloren	0 (0.00%)	201.700 (88.64%)	0 (0.00%)	25.260 (88.58%)
Summe Unentschieden	3.612 (2.02%)	25.856 (11.36%)	462 (2.06%)	3.258 (11.42%)
Gewonnen gesamt	175.168 (43.11%)		21.959 (43.11%)	
Verloren gesamt	201.700 (49.64%)		25.260 (49.59%)	
Unentschieden gesamt	29.468 (7.25%)		3.720 (7.30%)	
Generierungsdauer	8m 46s		74s	
Größe	496 KB		77 KB	

Tabelle 3: Spielstatistik zum Schachendspiel KR-K

Abbildung 32: Screenshot der Spieloberfläche

6.2 Das Schachendspiel KR-KN

Das Schachendspiel *Turm und König gegen Springer und König* setzt sich aus insgesamt acht Subdatenbanken zusammen. Dies sind K-K, KR-K, K-KN und KR-KN mit jeweils Weiß und Schwarz am Zug. Diese Subdatenbanken sind auch hier verantwortlich für die leichten Abweichungen zu den Werten, die Nalimov ermittelt hat. Dieses Endspiel wurde bereits Anfang der achtziger Jahre von Thompson erstmals erschöpfend analysiert. Die so entstandene Datenbank zeigte, dass in der vorhandenen Endspieltheorie, die über Jahrhunderte durch menschliche Analysen entstanden ist, zahlreiche Fehler waren.

Abbildung 33 zeigt die Stellung mit der maximalen Gewinndistanz für Weiß. Der weiße König muss sich aus dem Schach befreien. Um zu gewinnen, muss er den Weg nehmen, der durch den Pfeil gekennzeichnet ist. Jeder andere Zug gibt Schwarz die Möglichkeit ein Unentschieden zu erzielen. Die Eroberung des schwarzen Springers, die für den Gewinn der Partie notwendig ist, findet erst 54 Halbzüge später statt. Abbildung 34 zeigt die einzige Möglichkeit den weißen König Matt zu setzen.

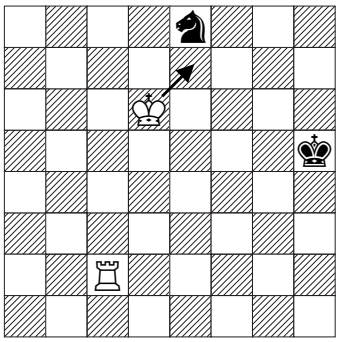


Abbildung 33: Matt in 79 Halbzügen

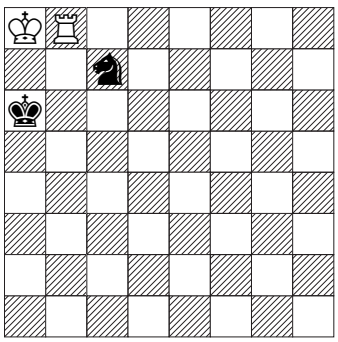


Abbildung 34: Weiß ist Matt

Die folgende Tabelle zeigt nur einen Ausschnitt der Spielstatistik. Eine vollständige Verteilung der Stellungen über die Gewinn- und Verlustdistanzen findet sich im Anhang. Auch hier zeigt sich der Vorteil bei der Ausnutzung von Brettsymmetrien.

Stellungen	**Ohne Symmetrie**		**Mit Symmetrie**	
Gesamt	31.506.048		4.922.820	
Illegal	7.354.288 (23.34%)		1.153.108 (23.43%)	
Symmetrisch identisch	0 (0.00%)		749.839 (15.23%)	
Tatsächlich verwendet	24.151.760 (76.66%)		3.019.873 (61.34%)	
	wtm	btm	wtm	btm
Verloren in 0	8	9.544	1	1.193
Gewonnen in 1	65.736	32	8.217	4
Verloren in 2	0	5.952	0	744
Gewonnen in 3	36.996	0	4.630	0
Verloren in 4	0	5.308	0	666
Gewonnen in 5	38.696	0	4.848	0
Verloren in 6	0	5.384	0	673
Gewonnen in 7	25.568	0	3.197	0
Verloren in 8	0	3.920	0	491
Gewonnen in 9	32.848	0	4.109	0
Verloren in 10	0	6.136	0	767
Gewonnen in 11	70.984	0	8.876	0
Verloren in 12	0	9.456	0	183
Gewonnen in 13	94.820	0	11.858	0
Verloren in 14	0	12.124	0	1.518
Gewonnen in 15	119.740	0	14.973	0
Verloren in 16	0	18.572	0	2.326
⋮	⋮	⋮	⋮	⋮
Verloren in 76	0	72	0	9
Gewonnen in 77	80	0	10	0
Verloren in 78	0	16	0	2
Gewonnen in 79	8	0	1	0
Verloren in 80	0	8	0	1
Summe Gewonnen	5.386.088 (48.16%)	32 (0.00%)	673.451 (48.16%)	4 (0.00%)
Summe Verloren	8 (0.00%)	1.566.500 (12.08%)	1 (0.00%)	195.932 (12.08%)
Summe Unentschieden	5.797.356 (51.84%)	11.401.776 (87.92%)	724.931 (51.84%)	1.425.554 (87.92%)
Gewonnen gesamt	5.386.120 (22.30%)		673.455 (22.30%)	
Verloren gesamt	1.566.508 (6.49%)		195.933 (6.49%)	
Unentschieden gesamt	17.199.132 (71.21%)		2.150.485 (71.21%)	
Generierungsdauer	1h 52m 41s		17m 02s	
Größe	30.768 KB		4.807 KB	

Tabelle 4: Spielstatistik zum Schachendspiel KR-KN

6.3 Losing Tic Tac Toe 4x4

Stellungen	**Ohne Symmetrie**		**Mit Symmetrie**	
Gesamt	10.165.779		1.349.657	
Illegal	3.982.385 (39.17%)		520.702 (38.58%)	
Symmetrisch identisch	0 (0.00%)		54.043 (4.00%)	
Tatsächlich verwendet	6.183.394 (60.83%)		774.912 (57.42%)	
	wtm	btm	wtm	btm
Gewonnen in 0	1.084.159	1.635.676	135.867	204.836
Verloren in 1	24.717	13.548	3.131	1.717
Gewonnen in 2	67.841	106.892	8.531	13.400
Verloren in 3	186.312	86.156	23.356	10.809
Gewonnen in 4	248.818	407.412	31.177	51.008
Verloren in 5	417.088	145.560	52.238	18.253
Gewonnen in 6	265.622	491.844	33.292	61.577
Verloren in 7	314.206	93.144	39.395	11.679
Gewonnen in 8	121.726	210.648	15.270	26.406
Verloren in 9	81.814	25.344	10.267	3.182
Gewonnen in 10	24.378	31.740	3.068	3.996
Verloren in 11	7.258	2.320	935	296
Gewonnen in 12	1.758	1.516	228	197
Verloren in 13	192	44	26	6
Gewonnen in 14	32	16	5	3
Verloren in 15	1	0	1	0
Summe Gewonnen	1.814.334 (62.57%)	2.885.744 (87.88%)	227.438 (62.56%)	361.423 (87.86%)
Summe Verloren	1.031.588 (35.57%)	366.116 (11.15%)	129.349 (35.58%)	45.942 (11.17%)
Summe Unentschieden	53.876 (1.86%)	31.736 (0.97%)	6.775 (1.86%)	3.985 (0.97%)
Gewonnen gesamt	4.700.078 (76.01%)		588.861 (75.99%)	
Verloren gesamt	1.397.704 (22.61%)		175.291 (22.62%)	
Unentschieden gesamt	85.612 (1.38%)		10.760 (1.39%)	
Generierungsdauer	3m 40s		1m 22s	
Größe	9.928 KB		1.318 KB	

Tabelle 5: Spielstatistik zu Losing Tic Tac Toe 4x4

Das Spiel *Losing Tic Tac Toe* ist dem Spiel *Tic Tac Toe* sehr ähnlich. Der einzige Unterschied besteht darin, dass der Spieler, der drei seiner Spielsteine in einer Reihe, Spalte oder Diagonale platziert hat, nicht gewinnt, sondern verliert. Dieses Spiel endet demnach meist mit einer Zugzwang-Situation: Ein Spieler ist gezwungen einen Spielstein so auf das Spielbrett zu platzieren, dass drei seiner Spielsteine verbunden sind. Gespielt wird üblicherweise auf einem quadratischen Spielfeld der Dimension 4. Dieses Spiel unterscheidet sich in der Art des Partieausgangs von den anderen Beispielen. Der Spieler, der die Partie gewonnen hat, hat nicht den letzten Zug ausgeführt. Dieser wurde von seinem Kontrahenten gemacht, der entweder aus Zugzwang oder aus Unachtsamkeit drei seiner Spielsteine verbunden hat. Dieses Beispiel zeigt, dass das Programm auch in so einem Fall korrekt arbeitet. Die Statistik lässt bereits zahlreiche Rückschlüsse auf das Spielgeschehen zu. Der prozentuale Anteil der gewonnenen Stellungen für Weiß und Schwarz ist sehr unterschiedlich. Da viel mehr Stellungen gewonnen sind in denen Schwarz am Zug ist, hat Weiß das

schwierigere Spiel. Tatsächlich ist bereits die Ausgangsstellung für Weiß verloren, da genau eine Stellung mit dem spieltheoretischen Wert -15 existiert, in der Weiß am Zug ist. Dies entspricht 8 weißen Zügen, die nur vom leeren Spielbrett aus möglich sind. Da nur ein sehr niedriger Prozentsatz der Stellungen unentschieden ist, wird kaum eine Partie mit diesem Ergebnis enden.

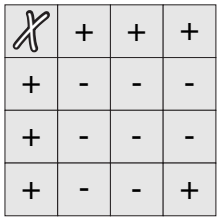


Abbildung 35: p_1

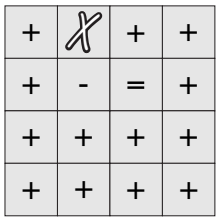


Abbildung 36: p_2

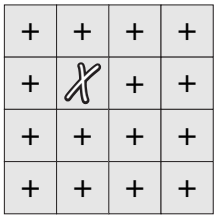
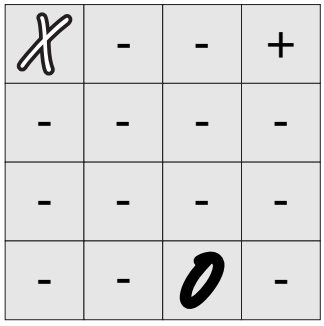


Abbildung 37: p_3

Die Stellungen p_1, p_2 und p_3 zeigen die drei Spielsituationen, wie sie nach dem ersten Zug entstehen können. Alle anderen 13 Möglichkeiten sind symmetrisch identisch zu diesen Stellungen. Weiß, hier durch das weiße "X" dargestellt, hat in Stellung p_1 in die Ecke gesetzt. Schwarz ("O") muss nun vorsichtig sein. Setzt er auf eines der Felder, die mit einem Minuszeichen versehen sind, kippt der spieltheoretische Wert und plötzlich ist es Weiß, der gewinnen kann. Alle mit einem Pluszeichen versehen Felder halten Schwarz auf der Siegerstraße. Der Beginn in einer der Ecken ist der aussichtsreichste Weg für Weiß die Partie entgegen des spieltheoretischen Werts doch noch zu gewinnen. Der Beginn mit einem "X" auf einem der Zentrumsfelder leistet am wenigsten Widerstand; Schwarz hat die freie Auswahl, da jede Fortsetzung gewinnt. Stellung p_2 zeigt eine der wenigen Möglichkeiten, die zu einem Unentschieden führen. Setzt Schwarz auf das Feld mit dem Gleichheitszeichen, so endet die Partie bei optimalem Spiel beider Seiten mit einem Remis.



Dass Weiß selbst in spieltheoretisch gewonnenen Stellungen einen schweren Stand hat, zeigt Abb. 38. Weiß hat mit einem Kreuz in der linken oberen Ecke begonnen, worauf Schwarz einen Fehler begangen hat, so dass Weiß in der Lage ist das Spiel zu gewinnen. Allerdings ist ein "X" hierzu in der oberen rechten Ecke die einzige Möglichkeit - ein keineswegs intuitiver Zug.

Abbildung 38: Gewinnstellung für Weiß

Diese Beispiele und viele weitere Stellungen aus der Datenbank lassen darauf schließen, dass es ratsam ist, zuerst die Ecken zu besetzen und anschließend die Randfelder. Die Zentrumsfelder sollten erst besetzt werden, wenn alle anderen Felder besetzt sind oder zum direkten Verlust führen.

Ein perfekter Spieler zu Losing Tic Tac Toe lässt sich auch mit herkömmlichen Suchalgorithmen leicht verwirklichen, da die Suchtiefe durch 16 Züge beschränkt ist und der Verzweigungsgrad mit jeder Suchtiefe sinkt.

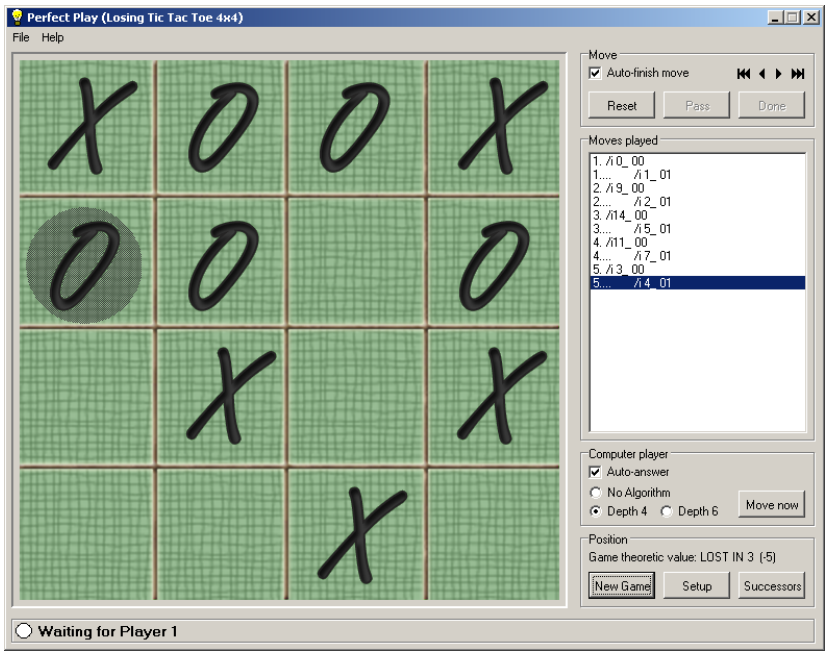


Abbildung 39: Screenshot der Spieloberfläche

6.4 Dodgem

Da bei *Dodgem* jede Stellung ein spiegelsymmetrisches Pendant besitzt, werden nur die spieltheoretischen Werte der Stellungen berechnet, in denen Weiß am Zug ist. Da alle Werte für Schwarz identisch sind, werden sie in der Tabelle nicht aufgeführt.

Die Ausgangsstellung bei Dodgem ist unentschieden. Diesen spieltheoretischen Wert teilt sie jedoch nur mit einem kleinen Prozentsatz anderer Stellungen, so dass ein spannendes Spiel garantiert ist. Die maximale Gewinndistanz ist mit 63 Halbzügen im Vergleich zur Entfernung der Autos zum Ziel verhältnismäßig hoch. Da die Entfernung aller Autos eines Spielers zum Ziel maximal 12 Felder betragen kann, was bei bloßem Vorwärtsziehen einer Gewinndistanz von 23 Halbzügen entspricht, gibt es Stellungen, in denen ein Spieler seine Autos häufiger zur Seite als vorwärts ziehen muss[14] um zu gewinnen. Diese Tatsache zeigt, dass Dodgem keineswegs ein triviales Spiel ist, dessen Züge stets intuitiv sind.

Allgemeingültige Regeln für das Spiel zu formulieren ist nicht möglich, da es stets Ausnahmesituationen gibt, in denen entgegen bekannter Muster gehandelt werden muss. Dennoch gibt es wichtige Punkte, die ein Spieler beachten muss. Die folgenden Regeln sind durch Recherchieren in der Datenbank und unzählige Partien gegen den perfekten Spieler entstanden.

1. Ein Spieler hat verloren, wenn das letzte seiner Autos in der rechten oben Ecke eingeklemmt (Abb. 40) ist. Sein Gegenspieler muss dann ein eventuell vorhandenes drittes Auto über die Ziellinie bringen und in der Situation, wie sie in Abb. 40 zu sehen ist, den Zug ausführen, der durch den Pfeil dargestellt ist.

2. Ein eingeklemmtes Auto in der rechten oberen Ecke bedeutet nicht automatisch den Verlust der Partie. Es ist jedoch auch nicht mehr möglich die Par-

[14] Dies sind alle Stellungen mit einem spieltheoretischen Wert, der größer als 47 ist. Hierbei handelt es sich insgesamt um 140 Stück.

tie noch zu gewinnen. Der Gegenspieler braucht nur sein verbleibendes Auto wahllos über das Spielbrett zu ziehen (Abb. 41).

3. Es ist kein Nachteil mehr Autos als der Kontrahent auf dem Spielbrett zu haben. So ist eine "3-gegen-1-Situation" fast immer für die Seite mit den 3 Autos gewonnen, wenn das einzelne Auto nicht unmittelbar über die Ziellinie fahren kann.

4. Mobilität ist ein wichtiger Faktor. Sind alle eigenen Autos eingeklemmt oder stehen so, dass sie einen vom Gegenspieler eingegrenzten Bereich nicht verlassen können, ist dies ein großer - meist schon spielentscheidender - Nachteil.

5. Die Autos sollten nicht gleichmäßig vorwärts gezogen werden, sondern auf die Spalten (Spieler 1) bzw. Reihen (Spieler 2) verteilt sein. Da nicht rückwärts gezogen werden kann, wird so vermieden, dass Schlupflöcher für gegnerische Autos hinter den eigenen Autos entstehen.

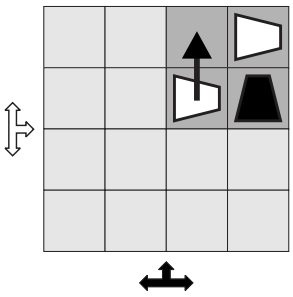


Abbildung 40: Schwarz verliert

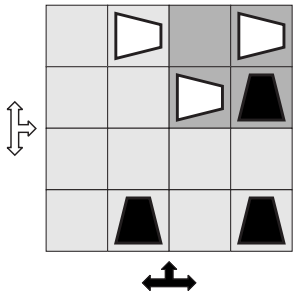


Abbildung 41: Weiß ist sicher

Eine interessante Spielsituation ist in Abb. 42 dargestellt. Schwarz hat bereits ein Auto über die Ziellinie gebracht, während Weiß sich passiv verhalten hat. Dass dies keine schlechte Strategie war, zeigt die Tatsache, dass Weiß am Zug gewinnen kann. Dafür sind allerdings sehr genaue Züge notwendig. Weiß muss im weiteren Verlauf der Partie versuchen die schwarzen Autos einzuklemmen, um so den Gegner in Zugzwangstellungen zu treiben. Dies ist nur schwer möglich, wenn die schwarzen Autos zusammenarbeiten können. Nur ein einzelnes Auto lässt sich wirkungsvoll einklemmen. Weiß muss deshalb wie in Abb. 43 ziehen. Bei allen anderen Zügen kippt der spieltheoretische Wert und das Spiel endet bei optimalem Spiel unentschieden oder geht verloren. Zieht Schwarz dasselbe Auto erneut nach oben, muss Weiß ihm folgen, bis es zur Situation in Abb. 44 kommt. Schwarz darf das Auto nicht über die Ziellinie ziehen, da so eine klassische "3-gegen-1-Situation" entsteht, die für Weiß leicht gewonnen ist. Das untere schwarze Auto kann jedoch leicht von den beiden bewegungsfreien weißen Autos an den rechten Rand getrieben werden. Es entsteht die Situation in Abb. 45, in der Schwarz unter Zugzwang steht. Er muss sein oberes Auto über die Ziellinie ziehen, wodurch die Stellung aus Abb. 40 entsteht, die bekanntlich für Weiß gewonnen ist.

Abbildung 42:

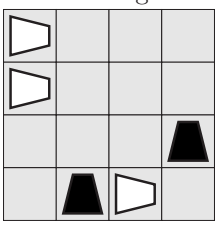


Abbildung 43:

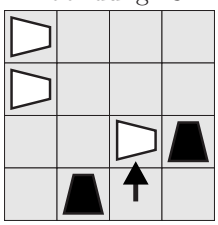
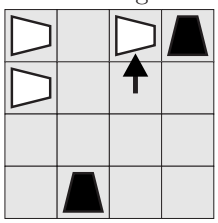


Abbildung 44:



Auch die Alternative, die Schwarz in Abb. 43 zur Verfügung steht, führt zum Verlust. Zieht Schwarz sein anderes Auto (Abb. 46), so sperrt Weiß den Weg ab (Abb. 47). Schwarz muss zur Seite ausweichen (Abb. 48), worauf Weiß ihn auf Spalte 1 festhält (Abb. 49). Zieht Schwarz sein Auto auf Spalte 1 vor, so zieht Weiß das zuletzt gezogene Auto nach oben, um das schwarze Auto einzuklemmen. Nun ist Schwarz gezwungen sein Auto am rechten Rand nach oben über die Ziellinie zu ziehen. Weiß ist also auch in dieser Variante siegreich.

Abbildung 45:

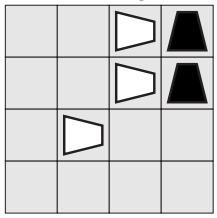


Abbildung 46:

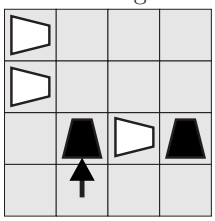


Abbildung 47:

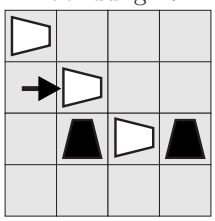


Abbildung 48:

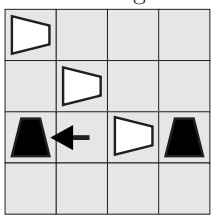


Abbildung 49:

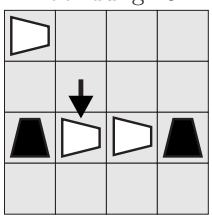
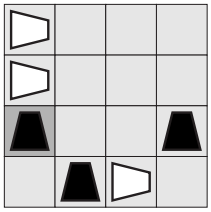


Abbildung 50:



Die Bedeutung von Zugzwangsituation wird in Abb. 50 deutlich. Im Gegensatz zur Ausgangsstellung aus Abb. 42 hat Schwarz ein zusätzliches, aber völlig deplatziertes Auto. Was auf den ersten Blick wie ein Nachteil aussieht, entpuppt sich als Rettungsanker. Schwarz kann viele Zugzwang-Situationen umschiffen und ein Unentschieden erreichen. Dabei spielt es keine Rolle, welcher der beiden Kontrahenten in Abb. 50 am Zug ist.

Stellungen	**Mit Spiegelsymmetrie**		
Gesamt	277.992		
Illegal	0 (0.00%)		
Symmetrisch identisch	0 (0.00%)		
Tatsächlich verwendet	277.992 (100.00%)		
	wtm		wtm
Verloren in 0	696	Gewonnen in 1	2.300
Verloren in 2	2.285	Gewonnen in 3	4.655
Verloren in 4	4.301	Gewonnen in 5	8.871
Verloren in 6	7.500	Gewonnen in 7	14.523
Verloren in 8	11.843	Gewonnen in 9	17.795
Verloren in 10	12.903	Gewonnen in 11	19.177
Verloren in 12	12.395	Gewonnen in 13	17.871
Verloren in 14	10.890	Gewonnen in 15	14.896
Verloren in 16	8.808	Gewonnen in 17	12.018
Verloren in 18	6.461	Gewonnen in 19	8.821
Verloren in 20	4.819	Gewonnen in 21	6.261
Verloren in 22	3.180	Gewonnen in 23	4.112
Verloren in 24	2.053	Gewonnen in 25	2.635
Verloren in 26	1.304	Gewonnen in 27	1.861
Verloren in 28	866	Gewonnen in 29	1.305
Verloren in 30	690	Gewonnen in 31	1.021
Verloren in 32	491	Gewonnen in 33	739
Verloren in 34	388	Gewonnen in 35	568
Verloren in 36	319	Gewonnen in 37	463
Verloren in 38	254	Gewonnen in 39	368
Verloren in 40	208	Gewonnen in 41	338
Verloren in 42	169	Gewonnen in 43	231
Verloren in 44	110	Gewonnen in 45	121
Verloren in 46	58	Gewonnen in 47	61
Verloren in 48	34	Gewonnen in 49	39
Verloren in 50	26	Gewonnen in 51	23
Verloren in 52	17	Gewonnen in 53	22
Verloren in 54	22	Gewonnen in 55	19
Verloren in 56	12	Gewonnen in 57	15
Verloren in 58	13	Gewonnen in 59	9
Verloren in 60	7	Gewonnen in 61	7
Verloren in 62	4	Gewonnen in 63	6
Gewonnen gesamt	141.151 (50.78%)		
Verloren gesamt	93.126 (33.50%)		
Verloren gesamt	43.715 (15.73%)		
Generierungsdauer	13s		
Grösse	217 KB		

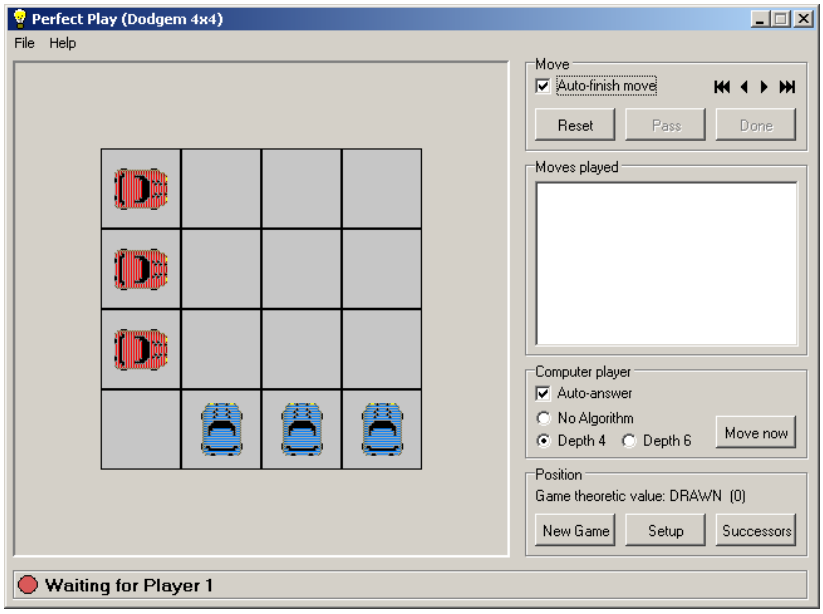


Abbildung 51: Screenshot der Spieloberfläche

7 Perfektes Spiel in der Praxis

Ken Thompson, der als Erster zahlreiche Endspieldatenbanken für das Schachspiel berechnete, bezeichnete die Spielstärke des resultierenden perfekten Spielers als "as good as god". Noch heute sind diese Datenbanken auf seiner Internetseite[15] bei den Bell Laboratories unter der Rubrik "Play chess with god" zu finden. Da über die Spielstärke eines möglichen Gottes nur spekuliert werden kann, lässt sich diese Aussage nicht entkräften. Dennoch zeigt sich rasch, dass ein perfekter Spieler in vielen Situation eklatante Schwächen zeigt, die es dem Gegenspieler viel zu leicht machen ein Unentschieden zu erzielen. Donniger kommt in [Don00] zu dem Schluss, dass der Thompson-Gott übermäßig "lieb und gütig" ist.

Dieses Kapitel soll die Schwächen eines herkömmlichen perfekten Spielers verdeutlichen und einen Ansatz liefern, wie sich perfektes Spiel verbessern lässt. Als Basis dienen dabei lediglich die Informationen aus der Datenbank und keine weiteren spielspezifischen Strategien.

7.1 Das Schachendspiel KR-KN

Am einfachsten lassen sich die Probleme eines perfekten Spielers und die Ansätze, diese Probleme zu lösen, am Beispiel von Schach verdeutlichen. Insbesondere ist hier das Endspiel König und Turm gegen König und Springer interessant. Im englischen werden die Figuren mit King (K) für den König, Rook (R) für den Turm und Knight (N) für den Springer bezeichnet. Hieraus ergibt sich die Abkürzung KR-KN, wobei Weiß den Turm hat und Schwarz den Springer. Die meisten Stellungen sind spieltheoretisch unentschieden, einige sind gewonnen für Weiß und nur eine einzige ist gewonnen für Schwarz. Letztere ist nicht von Bedeutung. Durch das Schlagen entstehen Unterendspiele. Das Endspiel KR-K ist stets für Weiß gewonnen, es sei denn, Schwarz kann unmittelbar den weißen Turm schlagen. Die beiden anderen Endspiele K-K und K-KN beenden das Spiel sofort. Da keine Seite genügend Material besitzt, um den gegnerischen König Matt zu setzen, wird die Partie als unentschieden gewertet. Unentschieden endet das Spiel auch dann, wenn 50 Züge keine Figur geschlagen wurde oder eine Stellung sich dreimal wiederholt.
Der spieltheoretische Wert des Endspiels KR-KN kann bei bloßem Betrachten einer Stellung recht gut abgeschätzt werden. Stehen der schwarze König und der schwarze Springer dicht beieinander ist der Wert der Stellung unentschieden. Sind diese beiden Figuren weit voneinander entfernt und ist der weiße König dichter am Springer als der schwarze König, ist die Stellung gewonnen für Weiß. Eine allgemeingültige Regel ist dies nicht; sie genügt jedoch für die strategischen Betrachtungen in diesem Kapitel.

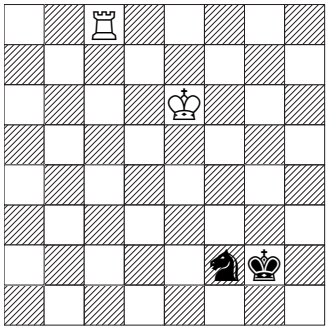


Abbildung 52: Unentschieden

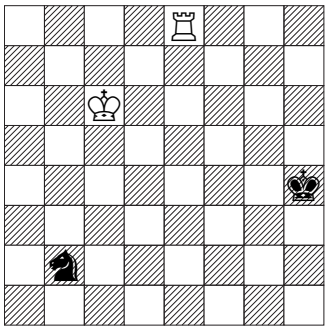


Abbildung 53: Weiß am Zug gewinnt

[15] http://cm.bell-labs.com/cm/cs/who/ken

Auch wenn die Stellung in Abb. 52 unentschieden ist, muss Weiß das Ziel verfolgen die Partie zu gewinnen. Dazu müssen die beiden schwarzen Figuren auseinander getrieben werden. Eine andere Möglichkeit besteht in der Eroberung des Springers, was auch bei geringer Distanz zum König möglich ist. Es ist offensichtlich, dass ein perfekter Spieler dieses Ziel kaum erreichen wird. Er wählt den Folgezug zufällig aus den Zügen aus, die das Unentschieden halten, was zur Folge hat, dass er planlose Züge mit Turm oder König macht ohne die schwarzen Figuren in Bedrängnis zu bringen. Selbst Züge, die zum sofortigen Partieende führen, sind möglich. In Abb. 54 ist ein solcher Zug gekennzeichnet. Schwarz kann den Turm schlagen (Abb. 55) und die Partie unmittelbar beenden. Er hätte damit das für ihn bestmögliche Ergebnis erreicht.

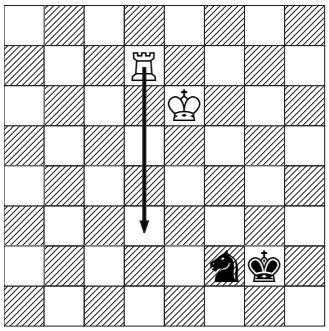


Abbildung 54: Möglicher Zug eines perfekten Spielers

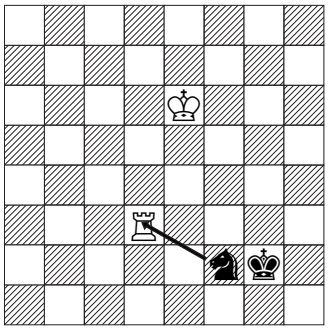


Abbildung 55: Mit dem Schlagen des weißen Turms endet das Spiel unentschieden

7.2 Allgemeine Probleme perfekter Spieler

Künstliche Spieler auf der Basis von Suchalgorithmen haben ein grundlegendes Problem. Sie reagieren zwar auf die Züge des Gegners, nehmen aber bei der Berechnung einer Antwort an, dass sie gegen sich selbst spielen. Beim Aufbau eines Suchbaums halten sie auf jeder Ebene den Zug für den wahrscheinlichsten, den sie selbst spielen würden. Ein ähnliches Problem haben auch perfekte künstliche Spieler. Sie gehen ebenfalls von einem ebenbürtigen Gegner aus. Im Falle einer Stellung, die spieltheoretisch unentschieden ist, gibt es meistens mehrere Züge, die dieses Ergebnis halten. Einer dieser Züge wird zufällig ausgewählt. Gegen einen perfekten Gegner ist diese Strategie gerechtfertigt. Da das Endergebnis bereits feststeht, ist die Wahl des Zuges bedeutungslos. Generell hat es ein perfekter Spieler jedoch mit einem fehlerbehafteten Kontrahenten zu tun. Dies kann ein Computerspieler sein, der mit bekannten Suchalgorithmen und einer Stellungsbewertung arbeitet, oder aber ein menschlicher Gegner, von dem grundsätzlich angenommen werden kann, dass er fehlerbehaftet ist. Gegen diese Gegner sind nicht alle Züge gleichwertig, sondern müssen differenzierter betrachtet werden. Einige Züge können es einem menschlichen Gegner schwer machen einen Zug zu finden, der das Unentschieden hält. Die Gewinnwahrscheinlichkeit erhöht sich in so einem Fall. Im Folgenden wird stets von einem menschlichen Gegner ausgegangen.

Ein künstlicher Spieler, der Züge im Falle einer ausgeglichenen Stellung zufällig auswählt, ist der perfekte Mühlespieler von Peter Stahlhacke. Selbst ein Amateurspieler hat keinerlei Probleme ein Unentschieden gegen dieses Programm zu erzielen. Bei einem Mühleturnier würde es nur einen der mittleren Plätze belegen. Eine Serie von Unentschieden reicht nicht aus, da es viele Teilnehmer mit einer positiven Spielbilanz geben wird. Auch Schaeffer erkannte die Probleme von perfekten Spielern bei

der Entwicklung seines Damespielers. Je umfangreicher die Endspieldatenbanken in der Anzahl ihrer Steine wurden, desto einfacher wurde es für Menschen ein Unentschieden zu erzielen. Schaeffer führt aus diesem Grund in [SchLak96] eine weitere Klasse von Spielen ein, die *ultra-stark gelösten* Spiele:

> Ein Spiel ist *ultra-stark gelöst*, wenn es *stark gelöst* ist und für jede Brettstellung eine Strategie existiert, die die Wahrscheinlichkeit erhöht, gegen einen fehlerbehafteten Gegner mehr als den spieltheoretischen Wert zu erzielen.

Für Mühle existieren zahlreiche Programme, die sehr stark spielen, aber nicht perfekt. Die Kombination der Suchalgorithmen, die in diesen Programmen verwendet werden, mit der Datenbank, die Mühle stark löst, ließe sich das Spiel ultra-stark lösen. Der Suchalgorithmus müsste an der Wurzel des Suchbaums nur die möglichen Züge streichen, die der Datenbank als Fehler bekannt sind. In [SchLak96] beschreibt Schaeffer ein ähnliches Verfahren für sein Damespiel, das sich auf umfangreiche Endspieldatenbanken stützt. Dort wird eine statische Analyse der Position durchgeführt und der Wert einer Stellung, die unentschieden ist, leicht gesenkt oder erhöht. Beide dieser Verfahren haben den Nachteil, dass sie spezifische Kenntnisse über das jeweilige Spiel benötigen. Einen allgemeinen Ansatz, der ohne spielspezifisches Wissen auskommt, sondern sich allein der Information aus der Datenbank bedient, gibt es bisher nicht. Der Grund dafür ist einfach. Da bisher kein allgemeiner Ansatz für den Bau perfekter Spieler existierte, fehlte sowohl der Anreiz, als auch die Grundlage auf diesem Gebiet zu forschen. Im folgenden Abschnitt wird ein Algorithmus vorgestellt, der sich zum Ziel setzt perfektes Spiel zu verbessern. Es handelt sich hierbei um einen experimentellen Ansatz, dessen Ergebnisse aber bereits vielversprechend sind.

7.3 Ein Algorithmus für verbessertes perfektes Spiel

Verbessertes perfektes Spiel muss dort ansetzen, wo die Ausgangsstellung spieltheoretisch unentschieden ist. In seltenen Fällen kann es Sinn machen auch bei verlorenen Stellungen andere Wege zu gehen. Ein perfekter Spieler zögert in diesen Fällen den Verlust so weit wie möglich hinaus. Dies muss nicht immer die beste Strategie sein. Es ist denkbar, dass ein Zug existiert, der zwar schneller verliert, dem Gegenspieler aber weitaus mehr Probleme bereitet. Auch können mehrere Züge eine identische Verlustdistanz aufweisen. Die Ausgangsstellung von Losing Tic Tac Toe ist ein Beispiel dafür. Spielsituationen, in denen die Differenzierung von gleichwertigen Verlustzügen ein Zuwachs an Spielstärke bringt, sind eher selten. Ein perfekter Spieler wird mit eigenen Zügen ohnehin nie in eine solche Situation gelangen. Nur wenn er bereits im ersten Zug mit einer Verlustsituation konfrontiert wird, die Voraussetzungen also denkbar unfair sind, muss er sich zwischen Verlustzügen entscheiden. Aus diesen Gründen kommt der hier vorgestellte Algorithmus nur bei Stellungen zum Einsatz, die spieltheoretisch unentschieden sind.

In diesem Kapitel wird häufig von Prozenten und Wahrscheinlichkeiten gesprochen. Um die Begriffe besser in Zusammenhang bringen zu können, beziehen sich beide auf den Wertebereich einer Wahrscheinlichkeit. Wenn von einer Prozentzahl gesprochen wird, ist also stets ein Wert aus dem Intervall $[0, 1]$ gemeint.

7.3.1 Zugauswahl

Bei herkömmlichen Suchalgorithmen erfolgt die Auswahl des besten Zuges auf Basis der Bewertung von Stellungen. Alle Nachfolger einer Stellung werden mit Hilfe eines Algorithmus, der in den meisten Fällen auf dem Minimax-Verfahren basiert, bewertet. Der Zug, der aus Sicht des am Zug befindlichen Spielers zu der besten

Stellung führt, wird gewählt. Der Wert einer Stellung ist einem perfekten Spieler für alle potentiellen Nachfolger bekannt - er ist 0. Potentielle Nachfolger sind die Stellungen, die spieltheoretisch unentschieden sind. Züge, die zu einer verlorenen Stellung führen, werden von vornherein ausgeschlossen. Der perfekte Spieler muss seine Zugauswahl also auf andere Kriterien als die Stellungsbewertung stützen. Die Idee ist die Einführung des Begriffs der *Gewinnwahrscheinlichkeit*. Sie soll ausdrücken, wie hoch die Wahrscheinlichkeit bei Ausführung des entsprechenden Zuges ist doch noch zu gewinnen. Diese Wahrscheinlichkeit muss für jeden Zug berechnet werden. Anschließend wird der Zug mit der höchsten Gewinnwahrscheinlichkeit gespielt.

Die Frage nach der Gewinnwahrscheinlichkeit eines Zuges führt unmittelbar zu der Frage wie wahrscheinlich es ist, dass der menschliche Gegner im weiteren Verlauf dieses Pfades einen Fehler begeht. Um diese Frage zu beantworten, sind einige psychologische Überlegungen notwendig.

7.3.2 Die Schwächen von menschlichen Spielern

Um perfektes Spiel zu verbessern müssen die menschlichen Schwächen gezielt ausgenutzt werden. Der Begriff Schwäche soll die kognitiven Leistungen nicht abwerten, die der Mensch beim Spielen von klassischen Brettspielen vollbringt. Jedoch besitzt er zahlreiche Nachteile im Vergleich zu Computerprogrammen, die letztendlich deren Überlegenheit ausmachen.

Computerspieler betrachten bei der Wahl eines geeigneten Zuges viele Millionen von Pfaden. Hierbei werden auch die scheinbar unsinnigsten Züge in Betracht gezogen. Ein menschlicher Spieler konzentriert sich auf einige wenige Fortsetzungen, die er weiter analysiert. Viele mögen dieses sehr selektive Verfahren als eine Stärke bezeichnen, da unnötige Berechnungen vermieden werden. Es ist jedoch fehleranfällig und aus Sicht eines künstlichen Spielers eine Schwäche. Probleme bekommt ein menschlicher Spieler, wenn eine Stellung so kompliziert ist, dass die Selektion schwer fällt. Dies machen sich insbesondere Schachprogramme zu Nutze. Sie werden für den Wettkampf gegen Großmeister speziell präpariert. So wurden bei den Wettkämpfen Barejew gegen Hiarcs und Fritz gegen den amtierenden Weltmeister Kramnik die Programme so modifiziert, dass sie die Damen auf dem Brett behalten, auch wenn dies zu einer schlechteren Stellung führt [Ree03]. Durch dieses Vorgehen bleibt das Spiel kompliziert und wird nicht zum Vorteil des Menschen vereinfacht. Von Haus aus werden Computerprogramme nicht mit diesen Eigenschaften ausgestattet. Sie sind auf das Spiel gegen andere Computerprogramme optimiert, so dass die erwähnten Modifikationen keinen Vorteil bringen.

Betrachtet man die Niederlagen menschlicher Spieler, so stellt man fest, dass es häufig die unauffälligen Züge des Computers sind, die der Mensch nicht berücksichtigt hat. Züge, die unmittelbar zum Figurenverlust führen, werden von einem Menschen nur selten übersehen. Generell sorgt sich ein menschlicher Spieler zuerst um die Existenz seiner Spielsteine und setzt sich erst dann mit der Frage auseinander, wie er sie gewinnbringend einsetzen kann. Jeder Hobbyspieler wird bestätigen, dass er beim Mühle-Spiel erst schaut, ob sein Gegner eine Mühle schließen kann und wie dies zu verhindern ist. Ähnliches gilt bei Dame. Der erste Gedanke bei der Betrachtung eines möglichen Zuges ist meistens: "Kann mein Gegenspieler einen meiner Spielsteine überspringen und ihn vom Spielbrett nehmen?" Auch beim Schach stellt sich den meisten Menschen als erstes die Frage, welche Figuren angegriffen und vom Gegner geschlagen werden können. Erst danach fließen strategische Überlegungen in den Denkprozess mit ein. Professionelle Spieler mögen bei der Auswahl ihres Zuges anders vorgehen. Die Mehrheit der Spieler wird jedoch nach dem hier beschriebenen Muster vorgehen.

Gesicherte empirische Befunde zu den Schwächen menschlicher Spieler existieren nicht. Jahrelange Erfahrungen, insbesondere bei Schachprogrammen, haben jedoch

gezeigt, dass diese Annahmen durchaus ihre Gültigkeit haben. Wie ein perfekter Spieler davon profitieren kann, wird in den folgenden Abschnitten betrachtet.

7.3.3 Wahrscheinlichkeit für die Ausführung eines Verlustzuges

Um die Wahrscheinlichkeit für die Ausführung eines Verlustzuges zu ermitteln, wird zunächst eine solche Situation betrachtet. In Abb. 56 sind Stellungen durch Kreise dargestellt. Die enthaltene Nummer stellt den spieltheoretischen Wert dar. Die Kanten zwischen den Stellungen sind als Züge zu verstehen. p_1 und p_2 sind in diesem Beispiel Verlustzüge, denn sie führen zu einer für den Computer gewonnenen Stellung. Die Züge p_3 und p_4 halten das Unentschieden. Gewinnzüge gibt es selbstverständlich keine für den Menschen. Dann nämlich wäre der spieltheoretische Wert der Stellung positiv und nicht 0.

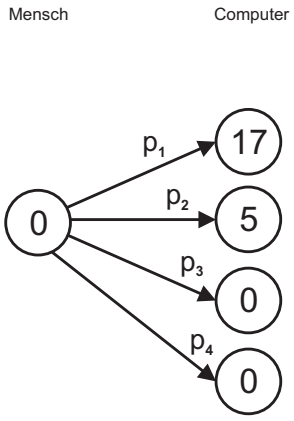


Abbildung 56: Ein Beispiel mit noch unbekannten Zugwahrscheinlichkeiten

Der Mensch sieht sich in einer Stellung, die unentschieden ist. Allgemein hat er k Züge zur Auswahl, die verlieren und $n - k$ Züge, die das Unentschieden halten. Gesucht sind die Wahrscheinlichkeiten $A(p_1)$ bis $A(p_4)$. Die wichtigste Information zur Bestimmung dieser Größen scheint zunächst die Distanz zum Gewinn aus Sicht des perfekten Spielers, die bei Zug p_1 mit 17 festgelegt ist, und das Verhältnis von k und $n - k$ zu sein. Vorherige Überlegungen haben gezeigt, dass der Mensch beim Vorausschauen von Spielzügen eine sehr begrenzte Suchtiefe hat und aus diesem Grund zu p_1 verleitet werden könnte, da er 17 Züge kaum vorhersehen kann. Des Weiteren hat es den Eindruck, dass die Anzahl der k Verlustzüge im Verhältnis zu den korrekten Zügen ein entscheidender Faktor ist. Je mehr Züge zum Verlust führen und je weniger zum Unentschieden, desto höher ist die Wahrscheinlichkeit, dass der Mensch einen Verlustzug wählt. Beide Annahmen sind generell nicht richtig. Ein Beispiel verdeutlicht dies:

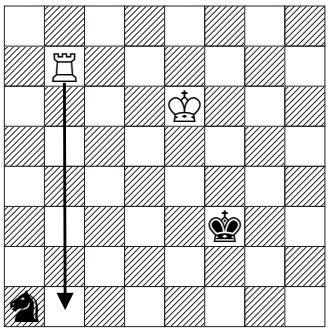


Abbildung 57: Vor Ausführung des Zuges

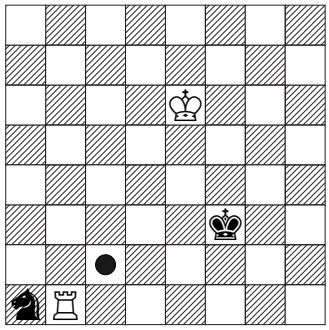


Abbildung 58: Nach Ausführung des Zuges

Unter alleiniger Berücksichtigung der beiden Kriterien spielt ein perfekter Spieler den eingezeichneten Zug. Der Springer hat daraufhin nur ein einziges Fluchtfeld, was durch einen Kreis gekennzeichnet ist. Dies ist der einzige Zug, der das Unentschieden hält. Alle weiteren 9 Züge, die Schwarz zur Verfügung stehen, verlieren den Springer und damit die Partie. Trotzdem ist die Wahrscheinlichkeit groß, dass Schwarz seinen Springer rettet. Und das obwohl die Verlustdistanz hoch ist und Schwarz unmöglich das Spielende sehen kann. Selbst wenn der Springer erobert ist, sind immer noch 30 bis 35 Halbzüge notwendig, bis Weiß den schwarzen König Matt setzt. Dieses Beispiel zeigt, dass eine genauere Interpretation der Informationen aus der Datenbank nötig ist.

Der vorgestellte Algorithmus unterscheidet zwischen drei Kriterien, die die Wahrscheinlichkeit der Ausführung eines Verlustzuges p_i bestimmen. Dies sind die angepasste Verlustdistanz $v(p_i)$, die Gewinnzugvarianz $z(p_i)$ und die Änderung der Figurenverteilung $f(p_i)$. Alle liefern als Ergebnis eine Zahl aus dem Intervall $[0, 1]$. Die Multiplikation dieser drei Zahlen liefert eine vorübergehende Wahrscheinlichkeit für die Ausführung eines Verlustzuges. Sie muss nur noch in Verhältnis zu den anderen Zügen gesetzt werden.

Die angepasste Verlustdistanz

Die Verlustdistanz allein kann ein verfälschtes Bild liefern, wie es das letzte Beispiel gezeigt hat. Sie muss die Beschaffenheit der Subdatenbank berücksichtigen, in der sie zum ersten Mal auftritt. In der Stellung in Abb. 58 ist dies nach Schlagen des Springers das Endspiel KR-K. Eine Analyse dieser Subdatenbank zeigt, dass alle Stellungen für den Spieler, der den Turm besitzt und der am Zug ist, gewonnen sind. Es ist unwahrscheinlich, dass der menschliche Spieler von einer dermaßen einfachen Regel keine Kenntnis besitzt. Er wird wissen, dass er dieses Endspiel verliert. Die Distanz zum Verlust der Partie spielt in diesem Fall keine Rolle. Deshalb wird zuerst der prozentuale Anteil der Stellungen bestimmt, die der Computer bei Zugrecht in der entsprechenden Datenbank gewonnen hat. Dieser Wert von 1 subtrahiert bildet den ersten Faktor der angepassten Verlustdistanz. Bei dem Endspiel KR-KN ergibt sich unabhängig von den anderen Kriterien eine Wahrscheinlichkeit von 0, dass der Mensch seinen Springer nicht wegzieht, wenn dieser angegriffen wird.

Der zweite Faktor wird durch eine Funktion d bestimmt, die eine Verlustdistanz auf das Intervall $[0, 1)$ abbildet. Für diese Funktion gilt:

$$\begin{aligned} d(1) &= 0 \\ d(w_1) &= 0.5 \\ d(w_2) &= 0.9 \end{aligned}$$

$$\lim_{x \to \infty} d(x) = \quad 1$$

Eine solche Funktion lässt sich mit Hilfe des Logarithmus leicht konstruieren.

$$d(x) = \quad r_2^{\frac{1}{x^{r_1}}}$$

$$mit \;\; r_1 = \quad \frac{\log\left(\frac{\log(0.5)}{\log(0.9)}\right)}{\log(\frac{w_2}{w_1})}$$

$$und \;\; r_2 = \quad 0.5^{\left(w_1^{r_1}\right)}$$

Die Werte w_1 und w_2 sollten auf einen menschlichen Spieler angepasst werden. In diesem Fall wurden $w_1 = 7$ und $w_2 = 12$ gewählt. Dahinter steckt die Annahme, dass bei einer Distanz von 7 Zügen 50% der Gegner diesen Verlust übersehen. Bei einer Distanz von 12 Zügen sind es 90%. Bei einer Distanz von nur einem Zug zum Verlust der Partie wird angenommen, dass kein Mensch den Verlust übersieht und diesen Zug spielt.

Die Gewinnzugvarianz

Die Gewinnzugvarianz untersucht die Gewinnzüge, die dem Computerspieler zur Verfügung stehen, wenn der Mensch einen Verlustzug ausführt. Ist die Anzahl dieser Gewinnzüge hoch, so ist die Wahrscheinlichkeit gering, dass der Mensch alle diese Züge übersieht. Gibt es aber nur einen einzigen Gewinnzug für den Computer steigt die Chance, dass der Mensch diesen Zug nicht bedacht hat. Auch die Häufigkeit eines Gewinnzuges auf einer Ebene spielt eine Rolle. Gewinnt stets der gleiche Zug für den Computer, wie dies z.b. in Abb. 58 der Fall ist, so steigt die Wahrscheinlichkeit, dass der Mensch ihn sieht. Zur Bestimmung der Gewinnzugvarianz z werden für alle Verlustzüge, die der Mensch ausführen kann, die Gewinnzüge gesammelt. Zur Bestimmung der Gewinnzugvarianz eines Verlustzuges wird zunächst der prozentuale Anteil c der dadurch möglichen Gewinnzüge des Computers im Vergleich zu allen ihm zur Verfügung stehenden Zügen bestimmt. Nun wird die Häufigkeit des Auftretens eines jeden Gewinnzuges auf dieser Ebene bestimmt. Sie werden multipliziert. Anschließend wird p mit der Wurzel dieses Resultats potenziert. Ein Beispiel soll dieses Verfahren verdeutlichen:

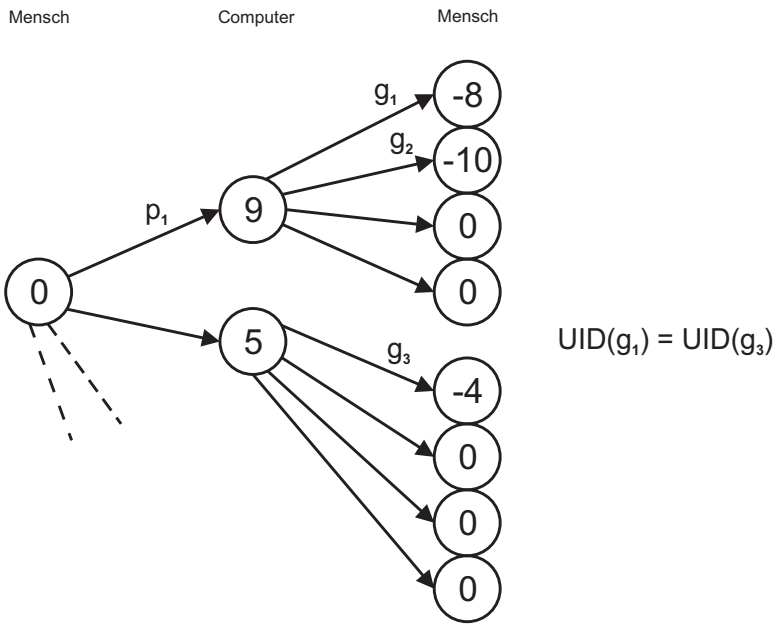


Abbildung 59: Beispiel zur Bestimmung der Gewinnzugvarianz für p_1

Gesucht ist die Gewinnzugvarianz für den Zug p_1. Der Mensch hat in einer ausgeglichenen Stellung einen Verlustzug begangen. Dem Computer stehen nach diesem Verlustzug zwei Gewinnzüge zur Auswahl. Damit ist c=0.5. Nun werden die beiden Gewinnzüge genauer untersucht. g_1 hat den gleichen eindeutigen Identifikator wie g_3. Damit tritt dieser Zug doppelt in der Ebene des Computerspielers auf. Der Gewinnzug g_2 ist nur einmal vorhanden. Somit ergibt sich

$$z(p_i) \;=\; 0.5^{\sqrt{2*1}} \;=\; 0.38\,.$$

Die Änderung der Figurenverteilung

Unter dem Gesichtspunkt, dass der Mensch einer Änderung der Figurenverteilung sehr viel Aufmerksamkeit schenkt, muss diese Eigenschaft in den Algorithmus einfließen. Hierzu werden einfach die letzten 5 Züge betrachtet. Seien h_1 bis h_n die bisher ausgeführten Züge. Die Funktion D bilde einen Zug auf die Menge $\{0, 1\}$ ab. Es gilt $D(h_i) = 1$, falls mit dem Zug h_i eine Änderung der Figurenverteilung stattgefunden hat. Andernfalls ist $D(h_i) = 0$. Für die Züge h_j mit $j < 1$ gelte ebenfalls $D(h_j) = 0$. Die Änderung der Figurenverteilung f für eine Verlustzug h_i berechnet sich nun wie folgt:

$$f = 1 - |D(h_i) - \frac{\sum_{j=i-5}^{i-1} D(h_j)}{5}|$$

Hat sich die Figurenverteilung innerhalb der letzten fünf Züge nicht geändert, so führt eine Änderung der Figurenverteilung mit dem Zug h_i zu $f = 0$. Unabhängig von den anderen Kriterien resultiert daraus eine Wahrscheinlichkeit für die Ausführung dieses Verlustzuges $A(h_i) = 0$. Es wird also angenommen, dass einfache Schlagzüge vom Menschen nicht übersehen werden. Erst wenn diesem Zug ein

oder mehrere Schlagzüge vorausgingen, besteht eine Wahrscheinlichkeit, dass der Mensch fehlerhaft spielt.

Die Wahrscheinlichkeit eines Verlustzuges p_i, wie er beispielsweise in Abb. 59 zu sehen ist, kann nun durch $A(p_i) = v(p_i) * z(p_i) * f(p_i)$ berechnet werden.

7.3.4 Rekursive Ermittlung der Gewinnwahrscheinlichkeit

Mit dem Wissen über die Wahrscheinlichkeit eines Verlustzuges durch den Menschen ist noch nicht die Frage der Gewinnwahrscheinlichkeit für den Computer geklärt. Hier kommt ein rekursiver Algorithmus zum Einsatz, der am Beispiel einer Abbildung erläutert werden soll.

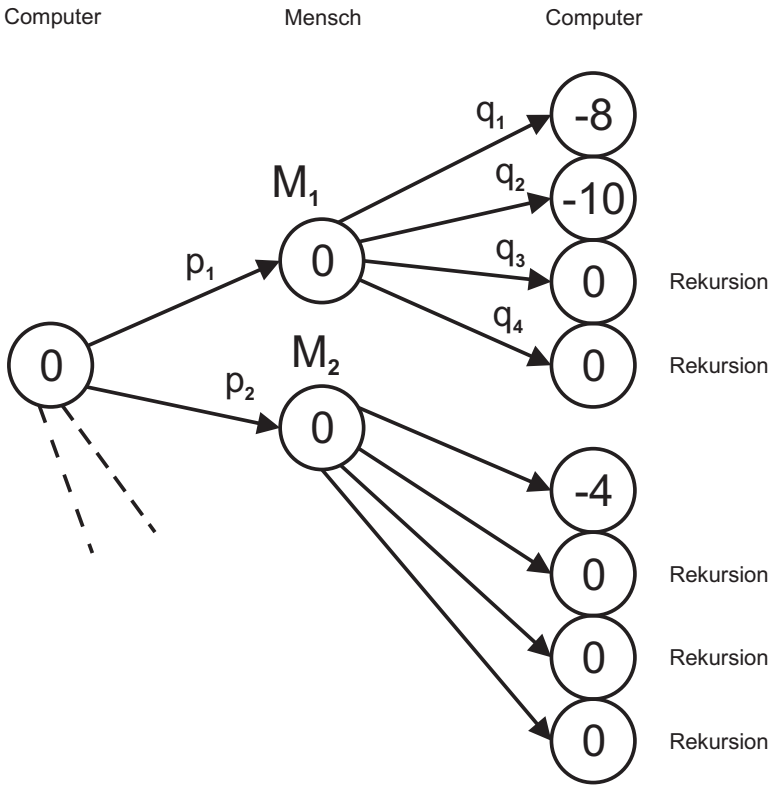


Abbildung 60: Bestimmung der Gewinnwahrscheinlichkeit für den Computer

Der Algorithmus startet immer dann, wenn sich der Computer in einer Stellung mit dem spieltheoretischen Wert 0 befindet und mehrere Züge zur Auswahl hat, die diesen Wert halten. Im Beispiel in Abb. 60 sind dies die Züge p_1 und p_2. Gesucht sind nun die Gewinnwahrscheinlichkeiten für die beiden Züge, die mit $G(p_1)$ und $G(p_2)$ bezeichnet werden. Um $G(p_1)$ zu bestimmen, wird der Knoten M_1 betrachtet. Die Züge q_1 und q_2 verlieren für den Menschen. Ihre Wahrscheinlichkeit kann durch das im vorherigen Abschnitt erläuterte Verfahren ausgerechnet werden. Damit sind $A(q_1)$ und $A(q_2)$ bekannt. Bei den beiden Zügen q_3 und q_4 wird Rekursion angewandt, um $G(q_3)$ und $G(q_4)$ zu bestimmen. Das gesuchte Ergebnis $G(p_1)$ wird nun durch

$$G(p_1) = \left(\prod_{i=1}^{n} A(q_i) \right)^{\frac{1}{n}}$$

bestimmt. n ist hier die Anzahl der Züge des entsprechenden Knotens, in diesem Beispiel M_1. Mit dieser Formel ist gewährleistet, dass $G(p_1) = 0$ ist, wenn eine der Wahrscheinlichkeiten $A(q_1)$ bis $A(q_n)$ ebenfalls 0 ist. Dieser Fall tritt unter anderem dann auf, wenn einer dieser Züge die Partie unmittelbar beendet. Die Verbindung der Wahrscheinlichkeiten A und G mag zunächst fremd erscheinen, ist aber zu rechtfertigen. Ist $A(q_i)$ hoch, steigt auch die Gewinnwahrscheinlichkeit des Zuges, der zum Knoten M_1 geführt hat. Umgekehrt sinkt die Gewinnwahrscheinlichkeit, wenn $A(q_i)$ niedrig ist.

Geklärt werden muss noch die Abbruchbedingung der Rekursion. Trifft der Algorithmus bei der Bestimmung der Gewinnwahrscheinlichkeit eines Zuges q auf einen Knoten, an dem das Spiel unentschieden endet, so folgt $G(q) = 0$, da die Wahrscheinlichkeit ein Spiel mit dessen Ende noch zu gewinnen 0 ist. Weiter muss der Algorithmus in der Tiefe beschränkt werden. Ist der Algorithmus bei der festgelegten maximalen Suchtiefe angelangt, so wird die Gewinnwahrscheinlichkeit nur noch durch das Verhältnis von Verlustzügen zu den korrekten Zügen bestimmt und durch 2 geteilt. Wäre der linke Knoten in Abb. 60 ein Endknoten im Sinne der Suchtiefe, so wäre $G(p_1) = \frac{0.5}{2} = 0.25$, da der Mensch zwei Verlust- und zwei korrekte Züge zur Verfügung hat. Die Gewinnwahrscheinlichkeit des linken Knotens ist dann das Maximum aus $G(p_1)$ und $G(p_2)$.

7.3.5 Ergebnisse

Wie stark der Algorithmus perfektes Spiel verbessert lässt sich nur schwer nachweisen. Die Spielstärke von Computerspielern wird normalerweise in zahlreichen Testpartien gegen andere Computerprogramme ermittelt. Bei dem hier vorgestellten Algorithmus müssen imperfekte Spieler die Rolle des Kontrahenten übernehmen. Eine umfangreiche Testserie im Spiel gegen Menschen könnte wertvolle Ergebnisse liefern. Dies allerdings führt an dieser Stelle zu weit. Deshalb beschränken sich die Ergebnisse auf subjektive Eindrücke und einige Teststellungen.

Die Änderung der Figurenverteilung als Faktor bei der Wahrscheinlichkeit für die Ausführung eines Verlustzuges hat kaum einen Einfluss bei den hier betrachteten Spielen. Beim Schachendspiel KR-KN sorgt bereits die angepasste Verlustdistanz dafür, dass die Wahrscheinlichkeit, einzügig eine Figur zu verlieren, 0 ist. Bei Dodgem ist dieser Faktor ebenfalls bedeutungslos, da die Figurenverteilung wenig über eine Stellung aussagt. Verfälscht werden die Ergebnisse dadurch nicht. Die Gewinnwahrscheinlichkeiten sinken zwar deutlich, bleiben aber relativ zueinander gleich. Um realistischere Wahrscheinlichkeiten zu bekommen, wurde dieser Faktor ausgegrenzt.

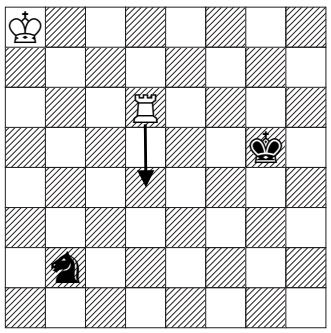


Abbildung 61: Teststellung beim Schach

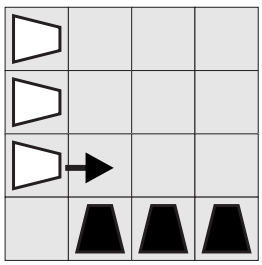


Abbildung 62: Teststellung bei Dodgem

Die Spielstärke beim Schachendspiel KR-KN ist nur leicht gestiegen. Der Computer

bietet seinen Turm nicht mehr zum Schlagen an, was bereits ein guter Fortschritt ist. Die Versuche den Springer des menschlichen Spielers zu erobern sind jedoch eher zaghaft. Eine leichte Bedrohung ist zeitweise vorhanden, doch listige Fallen oder raffinierte Tricks vermisst man weiterhin. Dies kann mit der geringen Suchtiefe zusammenhängen, die der Algorithmus beim Schach erzielt. Aufgrund des hohen Verzweigungsgrades dauert die Suche mit einer Tiefe von 6 schon zu lange für ein angenehmes Spiel. Wie beim Minimax-Algorithmus werden auch bei diesem Algorithmus alle Stellungen bis zu der gegebenen Suchtiefe betrachtet. Zudem sind die Berechnungen mit Fließkommazahlen sehr zeitaufwendig. Hier ist sicherlich noch viel Spielraum für Verbesserungen. Abb. 61 zeigt einen guten Zug des Computerspielers. Dem schwarzen Springer wird jedes Feld zur Flucht genommen. Der schwarze König muss nun zur Hilfe kommen.

Bei Dodgem ist der Gewinn an Spielstärke gegenüber einem üblichen perfekten Spieler mit zufälliger Zugauswahl deutlich spürbar. War es vorher ein Kinderspiel ein Remis zu erzielen, ist es nach Aktivierung des Algorithmus sehr schwierig geworden. Dodgem lässt sich damit als ultra-stark gelöst bezeichnen. In der Startaufstellung bei Dodgem (Abb. 62) wird das untere Auto nach vorn gezogen. Dies ist keine schlechte Wahl. Zieht der menschliche Spieler nun sein Auto am rechten Brettrand nach vorne hat er bereits verloren. Der Computer gibt auf Suchtiefe 6 eine Gewinnwahrscheinlichkeit von 46% für diesen Zug an. Dieser Wert ist durchaus realistisch. Die Gewinnwahrscheinlichkeiten für Spieler 2 werden in den ersten Zügen des Spiels deutlich geringer eingeschätzt.

Inwieweit dieser Algorithmus perfekte Spieler zu anderen Brettspielen verbessern kann, muss noch untersucht werden. Von einer allgemeinen Methode aus einem stark gelösten Spiel ein ultra-stark gelöstes Spiel zu machen kann noch nicht gesprochen werden.

8 Implementierung

Für die Implementierung des Programms wurde die Programmiersprache C++ gewählt. Ausschlaggebend hierfür war die Geschwindigkeit. Im Unterschied zu Interpretersprachen oder Programmiersprachen, die Bytecode generieren, erzeugt C++ Maschinencode, wodurch Programme um ein Vielfaches schneller laufen. Die flexible Speicherverwaltung und schnelleren Zugriffe auf die Peripherie sind ebenfalls Argumente, die für C++ sprechen.

Der Quellcode besteht aus 16 Klassen mit 156 Methoden und etwa 60 zusätzlichen Funktionen. Die Anzahl der Zeilen beträgt ungefähr 8000. Als Entwicklungsplattform diente Windows. Der größte Teil des Quellcodes nutzt jedoch keine Funktionen dieses Betriebssystems, so dass er ohne Schwierigkeiten auf andere Plattformen übertragen werden kann. Hierzu zählen das Programm zum Generieren der Datenbanken sowie eine Bibliothek zum Zugriff auf diese. Einzig die grafische Oberfläche ist nicht portabel. Da sich die Programmierung von GUI[16]-Anwendungen in C++ sehr aufwendig gestaltet, ist die grafische Oberfläche zum Testen der perfekten Spieler in *RapidQ* geschrieben, einem frei verfügbaren Compiler dessen Sprache als objektorientiertes Basic bezeichnet werden kann. Zwar ist *RapidQ* neben Windows auch für Linux, HP-Unix und Solaris verfügbar, in diesem Fall wurde aber mit DirectX eine windowseigene Technik benutzt. Die Voraussetzungen zum Bau einer ähnlichen Oberfläche auf anderen Betriebssystemen sind dank der vorhandenen Bibliothek zum Zugriff auf die Datenbank jedoch gegeben. Der Quellcode des *RapidQ*-Programms umfasst ungefähr 1000 Programmzeilen, von denen 400 Zeilen das GUI beschreiben und automatisch generiert wurden. Die grafische Oberfläche zum Spielen beschränkt sich auf die notwendigen Funktionen. Die gespielten Züge können vor- und zurückgeblättert werden, eine beliebige Stellung kann aufgebaut werden und der Computerspieler kann wahlweise die Rolle von Spieler 1 oder Spieler 2 übernehmen. Weiter wird der spieltheoretische Wert jeder Stellung wird automatisch angezeigt.

Die Erweiterung um perfekte Spieler zu anderen Brettspielen geschieht durch eine abstrakte Klasse. Ihre Methoden, insgesamt 19 Stück, müssen implementiert werden. Die meisten dieser Methoden sind nur wenige Zeilen lang. Im Anhang findet sich mit dem Brettspiel Dodgem ein Beispiel für einen perfekten Spieler.

[16] GUI steht für *Graphical User Interface*

9 Ausblick

Der in dieser Arbeit vorgestellte Ansatz zum Bau perfekter Spieler kann die Basis für weitere Forschung sein. Eine Verbesserung der Indexfunktion ist erstrebenswert, erfordert aber ein hohes Maß an Einarbeitung in die Materie. Nalimov liefert im Zusammenhang mit Schach einige Ansätze, die sich noch verallgemeinern ließen. So schränkt er die Position der Figuren relativ zueinander ein. Könige können damit nicht mehr nebeneinander stehen, ganz so wie es das Schachspiel auch verbietet. Andere Spiele, bei denen sich die Berücksichtigung dieser Eigenschaft bemerkbar macht, dürften aber eher selten sein. Eine Indexfunktion, die eine gewisse Lokalität garantiert, wäre ebenfalls ein Ziel, das verfolgt werden kann. Lokalität bedeutet in diesem Zusammenhang, dass Stellungen, die in einem möglichen Suchbaum durch Kanten miteinander verbunden sind, dicht in der Datenbank zusammenliegen. Hierdurch ließe sich die Geschwindkeit noch einmal verbessern. Dass so eine Indexfunktion für ein bestimmtes Spiel möglich ist, zeigen die Arbeiten von Nalimov. Ob dies auch für den allgemeinen Fall machbar ist, lässt sich nur schwer einschätzen.

Ebenfalls denkbar ist eine verteilte Variante der optimierten Retrograde Analysis. So könnte die Generierung der Datenbank auf mehrere Prozessoren oder mehrere Computer innerhalb eines Netzwerks aufgeteilt werden. Die Forschung auf diesem Gebiet ist bereits weit vorangeschritten. Insbesondere Schaeffer sowie Bal und Allis stellen in ([LakSch94]) und ([BalAll95]) bewährte Techniken vor. Hier ist lediglich ein Transfer dieser Algorithmen zu leisten.

Der vorgestellte Algorithmus zur Verbesserung von perfektem Spiel ist sicher der Teil dieser Arbeit, der am meisten Potential für eine Weiterentwicklung hat. Eine umfangreichere Auseinandersetzung mit diesem Thema war im Rahmen dieser Diplomarbeit nicht möglich, so dass nur ein experimenteller Ansatz entstanden ist. Trotz positiver Ergebnisse ist er stark verbesserungsfähig. Basis hierfür müssen perfekte Spieler zu weiteren Brettspielen sein, die sich mit Hilfe dieser Arbeit leicht erstellen lassen. Die Bestimmung der Spielstärke sollte nicht auf subjektiven Eindrücken basieren, sondern auf zahlreichen Tests im Spiel gegen Menschen oder imperfekte künstliche Spieler. Ein weiterer Anreiz für die Forschung auf diesem Gebiet ist die Tatsache, dass es bisher keine weiteren Ergebnisse zu diesem Thema gibt.

A Quellcode für das Spiel Dodgem

Dieser Abschnitt enthält den kommentierten Quellcode aus der Datei *Dodgem.cpp*, der alle Funktionen bereitstellt, die für die Implementierung des Spiels notwendig sind. Hierbei handelt es sich um die folgenden 19 Funktionen:

- char *cGetName();
- char *cGetFileName(int iColor, int iPiece);
- int iGetBoardSize();
- void GetStartPosition(board *b);
- int iBoardAppearance(int iTyp);
- unsigned int64 i64GetFields(int iPiece, int iColor);
- int iGetMinPieceCount(int iPieceTyp, int iColor);
- int iGetMaxPieceCount(int iPieceTyp, int iColor);
- bool bLegalPieceCombination(int *iPiecesW, int *iPiecesB, int iColorToMove);
- bool bLastMoveWins();
- bool bLost(board *position, int iColorToMove);
- bool bWon(board *position, int iColorToMove);
- bool bDrawn(board *position, int iColorToMove);
- bool bIsLegal(board *position, int iColorToMove);
- bool bMirrorable(int *iPiecesW, int *iPiecesB);
- int* iMirrorArray(unsigned int64 i64PiecesW, unsigned int64 i64PiecesB);
- int iGetSymmetryAxis(int *iPiecesW, int *iPiecesB, int **iMirrorArray);
- int iSuccessors(int iColor, board *position, MoveGenerator *movegen);
- int iPredecessors(int iColor, board *position, MoveGenerator *movegen);

Eine genaue Erklärung, welche Aufgaben die einzelnen Funktionen erfüllen, ist in die Kommentare des Quellcodes eingebettet. Die meisten Funktionen umfassen nicht mehr als eine Zeile. Lediglich die Funktionen *iSuccessors(...)* und *iPredecessors(...)* sind etwas umfangreicher.

```
// Die einzige Spielfigur, das Auto, erhält den Wert 0
# define CAR 0


char *cGetName() {

   // An dieser Stelle wird der Name des Spiels sowie der
   // Dateiname der Datenbank festgelegt.
   return ''Dodgem 4x4.rasd'';

}


char *cGetFileName(int iColor, int iPiece) {

   // Diese Funktion legt die Dateinamen für die verwendeten
   // Grafiken fest. FIELDUSED ist die Grafik für ein Feld,
   // das bei dem letzten Zug benutzt wurde. Dies hilft bei der
   // visuellen Erkennung von Zügen. BOARDMASK symbolisiert ein
   // modifiziertes Spielbrett (BOARD), auf dem die einzelnen
   // Felder in fest definierten Farben dargestellt sind. Dadurch
   // erkennt das Programm, welcher Teil des Bretts zu welchem Feld
   // gehört.

   if (iColor == FIELDUSED) return ''field.bmp'';
   if (iColor == BOARD) return ''board.bmp'';
   if (iColor == BOARDMASK) return ''boardmask.bmp'';
   if (iColor == WHITE && iPiece == 0) return ''redcar.bmp'';
   if (iColor == BLACK && iPiece == 0) return ''bluecar.bmp'';

   return NULL;

}


int Dodgem :: iGetBoardSize() {

   // Das Spielfeld besteht aus 16 Feldern.

   return 16;

}


void Dodgem :: GetStartPosition(board *b) {

   // Die Spielfelder sind von links oben nach rechts unten durch-
   // nummeriert. Die Startaufstellung sieht folgendermaßen aus:
   // X...
   // X...
   // X...
   // .OOO
   // Das X markiert hier ein Auto von Spieler 1, das 'O' ein Auto
   // von Spieler 2.

   ClearBoard(b);
   bInsertPiece(b, WHITE, 0, CAR);
   bInsertPiece(b, WHITE, 4, CAR);
   bInsertPiece(b, WHITE, 8, CAR);
   bInsertPiece(b, BLACK, 13, CAR);
   bInsertPiece(b, BLACK, 14, CAR);
   bInsertPiece(b, BLACK, 15, CAR);

}
```

```
int Dodgem :: iBoardAppearance(int iTyp) {

   // Die grafische Oberfläche benötigt noch drei weitere
   // Informationen über das Spielbrett. Die Farbe der
   // Spielsteine der beiden Kontrahenten, sowie die maximale
   // Anzahl der Felder in einer Reihe.

   switch (iWhat) {
      case WHITECOLOR: return 0x5959D9;
      case BLACKCOLOR: return 0xDA8C58;
      case FIELDSPERROW: return 4;
   }

}


unsigned int64 Dodgem :: i64GetFields(int iPiece, int iColor) {

   // Als Rückgabewert dieser Funktion wird eine Bitmaske erwartet,
   // die angibt, welche Felder die Figur, die durch die Parameter
   // spezifiziert ist, betreten darf. Autos beider Seiten dürfen
   // alle 16 Felder betreten, weshalb die ersten 16 Bits gesetzt
   // werden.

   return 0xFFFF;

}


int Dodgem :: iGetMinPieceCount(int iPieceTyp, int iColor) {

   // Ein Spieler hat gewonnen, wenn er alle Autos von dem
   // Spielbrett entfernen konnte. Die minimale Anzahl von Autos
   // jedes Spielers ist demnach 0. Dies gilt ebenfalls für
   // die Figuren mit den Zahlen 1-15, die in diesem Spiel nicht
   // benötigt werden.

   return 0;

}


int Dodgem :: iGetMaxPieceCount(int iPieceTyp, int iColor) {

   // Die maximale Anzahl an Autos jedes Spielers ist 3. Dies ist
   // gleichzeitig auch die maximale Anzahl der Figuren jedes Spielers.

   return (iPieceTyp == CAR || iPieceTyp == PIECEALL) ? 3 : 0;

}


bool Dodgem :: bLegalPieceCombination(int *iPiecesW, int *iPiecesB, int iColorToMove) {

   // Mit dieser Funktion können Beschränkungen der Figurenkonstellation
   // festgelegt werden. Bei Dodgem ist es nicht möglich, dass beide
   // Spieler keine Autos mehr auf dem Spielfeld haben.

   return (iPiecesW[CAR] == 0 && iPiecesB[CAR] == 0) ? false : true;

}
```

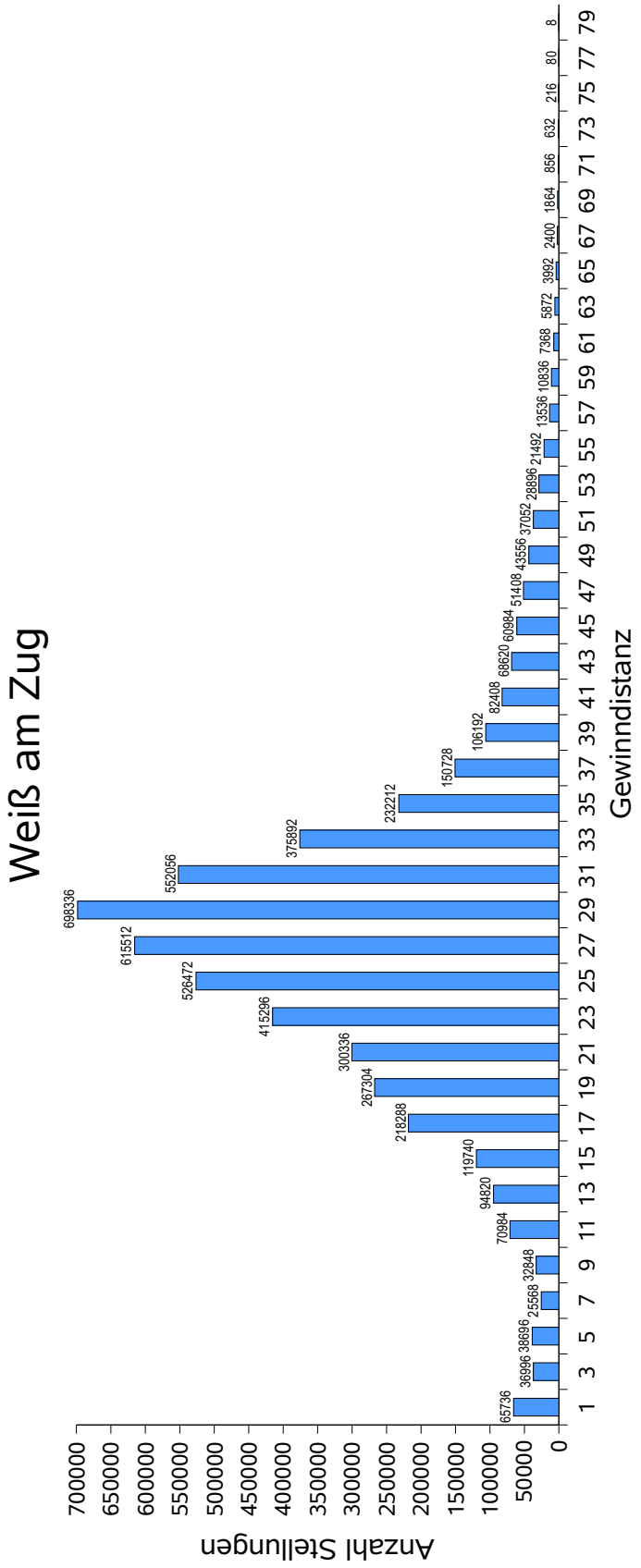
```
bool Dodgem :: bLastMoveWins() {

   // Macht bei diesem Spiel der Gewinner auch den letzten Zug?

   return true;

}


bool Dodgem :: bLost(board *position, int iColorToMove) {

   // Diese Funktion muss TRUE liefern, wenn die Seite am Zug die Partie
   // verloren hat. Sie muss bei den Spielen implementiert
   // werden, bei denen die Partie mit dem letzten Zug gewonnen wird.
   // Bei Dodgem ist die Partie verloren, wenn der Gegenspieler keine Autos
   // mehr auf dem Spielfeld hat.

   return bIsEmpty(position, iColorToMove ^ 1);

}


bool Dodgem :: bWon(board *position, int iColorToMove) {

   // Eine Gewinnbedingung ist nur bei den Spielen nötig, bei denen
   // der Spieler gewinnt, der nicht den letzten Zug ausgeführt hat.

   return false;

}


bool Dodgem :: bDrawn(board *position, int iColorToMove) {

   // Ein Unentschieden gibt es in dieser Form bei Dodgem nicht.
   // Zwar ließe sich - wie beim Schach - eine Regel einführen,
   // dass bei einer dreifachen Stellungswiederholung die Partie
   // mit einem unentschieden beendet wird, dies allerdings kann
   // bei der Retrograde Analysis nicht berücksichtigt werden.

   return false;

}


bool Dodgem :: bIsLegal(board *position, int iColorToMove) {

   // Bei Dodgem gibt es keine illegalen Stellungen. Jede mögliche
   // Verteilung der Autos entspricht einer legalen Stellung.

   return true;

}


bool Dodgem :: bMirrorable(int *iPiecesW, int *iPiecesB) {

   // Diese Funktion legt fest, ob die Nutzung von Spiegelsymmetrie möglich
   // ist. Dies kann in Abhängigkeit der Figurenkonstellation geschehen.
   // Bei Dodgem ist stets eine Spiegelsymmetrieachse existent.

   return true;

}
```

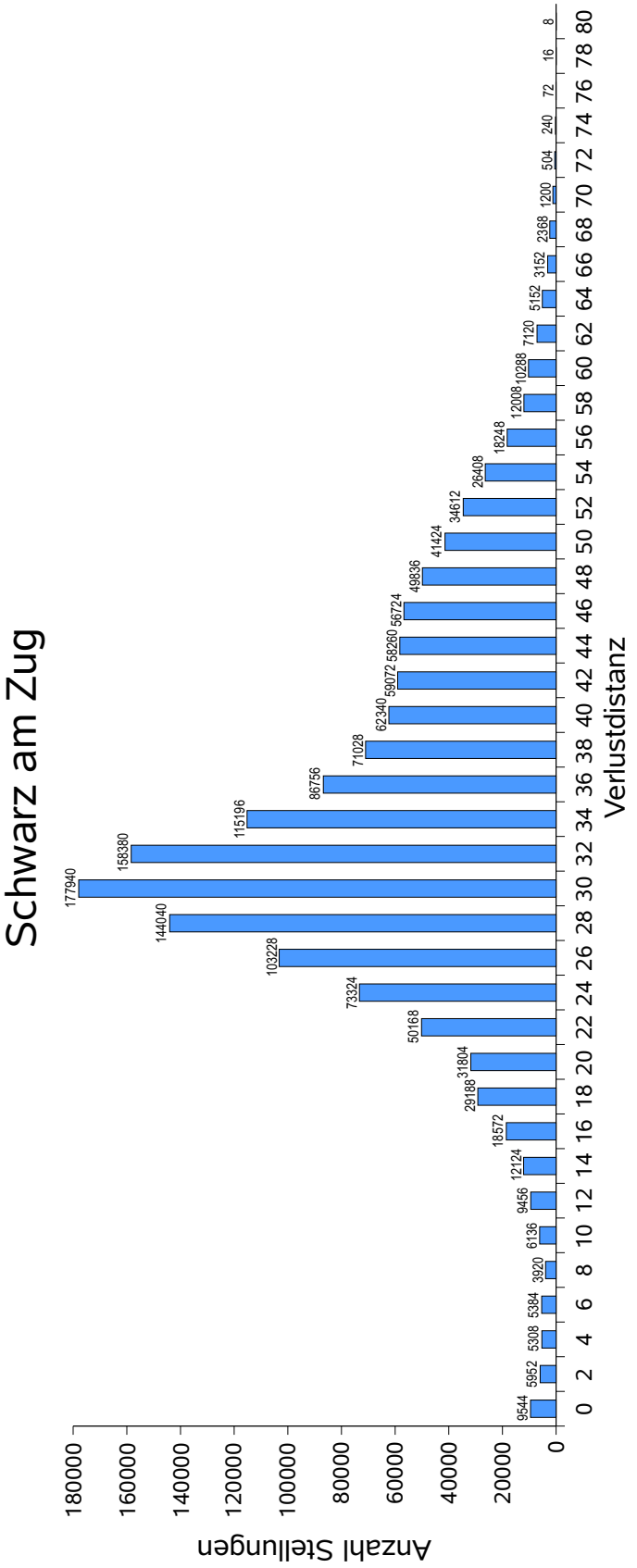
```
int* Dodgem :: iMirrorArray(unsigned int64 i64PiecesW, unsigned int64 i64PiecesB) {

   // Die Abbildung zur Spiegelsymmetrie ist der Rückgabewert dieser Funktion. Dies
   // geschieht in Form eines Arrays. An dieser Stelle wird auf eines der zahlreichen
   // vordefinierten Arrays zurückgegriffen.

   return iMirror[4][4][DIAGONAL_UP];

}


int Dodgem :: iGetSymmetryAxis(int *iPiecesW, int *iPiecesB, int **iMirrorArray) {

   // An dieser Stelle werden die Symmetrieachsen in Abhängigkeit der
   // Figurenkonstellation festgelegt. Sie müssen in **iMirrorArray abgelegt
   // werden. Rückgabewert ist die Anzahl der Symmetrieachsen.

   return 0;

}


int Dodgem :: iSuccessors(int iColor, board *position, MoveGenerator *movegen) {

   // Die Funktion zur Bestimmung der Nachfolger einer Stellung ist die einzige
   // nicht triviale Funktion bei der Implementierung von Dodgem. Der Zuggenerator
   // läuft über eine Klasse ''MoveGenerator'', die alle nötigen Funktionen zur
   // Modifikation der Brettstellung zur Verfügung stellt.

   int iCarCount, i, iCurrentPosition;

   // Die Spielfeldgröße, sowie die Länge einer Reihe werden dem Zuggenerator
   // mitgeteilt. Dadurch können die Spielfeldränder erkannt werden.

   movegen->Reset(16, 4);

   // Nun wird die Ausgangsstellung festgelegt.

   movegen->SetBasePosition(position, iColor);

   // Es folgt die eigentliche Zuggenerierung. Eine Methode 'bSetEvent' legt
   // fest, was geschehen soll, wenn eine Figur sich nicht wie gewünscht
   // in die vorgegebene Richtung bewegen kann. Die Standardeinstellung sieht
   // vor nichts zu tun. Mögliche Events sind unter anderem:
   // 1) Auto trifft auf eigenes Auto
   // 2) Auto trifft auf gegnerisches Auto
   // 3) Auto trifft auf ein Hindernis (z.B. ein nicht erreichbares Feld)
   // 4) Auto trifft auf Spielfeldrand
   // Bei Dodgem ist nur letzteres Event interessant. Trifft ein Auto von
   // Spieler 1 auf den rechten Rand, fährt es aus dem Spielfeld. Es wird vom
   // Spielfeld entfernt. Gleiches gilt für Spieler 2 bei dem oberen Rand.

   if (iColor == WHITE) { // Weiß am Zug
      movegen->bSetEvent(RIGHTBORDER, REMOVE);
      iCarCount = iGetWeight64(position->i64BoardW);
      for (i = 1; i <= iCarCount; i++) {
         iCurrentPosition = iGetPiece64(i, position->i64BoardW);
         movegen->iMove(iCurrentPosition, RIGHT);
         movegen->iMove(iCurrentPosition, UP);
         movegen->iMove(iCurrentPosition, DOWN);
      }
   } else { // Schwarz am Zug
      movegen->bSetEvent(TOPBORDER, REMOVE);
      iCarCount = iGetWeight64(position->i64BoardB);
      for (i = 1; i <= iCarCount; i++) {
         iCurrentPosition = iGetPiece64(i, position->i64BoardB);
         movegen->iMove(iCurrentPosition, UP);
```

```
         movegen->iMove(iCurrentPosition, LEFT);
         movegen->iMove(iCurrentPosition, RIGHT);
      }
   }

   // Wurden keine Züge generiert, so ist der Spieler am Zug eingeklemmt.
   // In diesem Fall wird ein Nullzug erstellt. Der Spieler setzt aus.

   if (movegen->iGetMoveCount() == 0)
      movegen->iPass();

   // Rückgabewert ist die Anzahl der generierten Züge

   return movegen->iGetMoveCount();

}


int Dodgem :: iPredecessors(int iColor, board *position, MoveGenerator *movegen) {

   // Die Generierung der Vorgänger läuft analog zur Generierung der Nachfolger.
   // Einziger Unterschied ist hier die Behandlung von Situationen, in denen ein
   // Spieler nicht ziehen kann bzw. ziehen konnte. Hat die gegebene Stellung für
   // die Seite am Zug keine Nachfolger, so ist das Aussetzen ein möglicher
   // Vorgänger. Für die Bestimmung dieses Spezialfalls wird eine Variable
   // ''bNoSuccessor'' eingeführt. Weiter muss in Betracht gezogen werden, dass mit
   // dem letzten Zug ein Auto aus dem Spielfeld gefahren ist. Dieser Umstand macht
   // die Bestimmung der Vorgänger zu einer Stellung weitaus komplizierter.

   int iCarCount, i, iCurrentPosition;
   bool bNoSuccessor = true;

   movegen->Reset(16, 4);
   movegen->SetBasePosition(position, iColor);

   if (iColor == WHITE) { // Weiß am Zug
      movegen->bSetEvent(RIGHTBORDER, REMOVE);
      iCarCount = iGetWeight64(position->i64BoardW);
      if (iCarCount == 0)
         bNoSuccessor = false;
      for (i = 1; i <= iCarCount; i++) {
         iCurrentPosition = iGetPiece64(i, position->i64BoardW);
         movegen->iMove(iCurrentPosition, LEFT);
         if (movegen->iMove(iCurrentPosition, UP) != IMPOSSIBLE)
            bNoSuccessor = false;
         if (movegen->iMove(iCurrentPosition, DOWN) != IMPOSSIBLE)
            bNoSuccessor = false;
         if (movegen->bCanMove(iCurrentPosition, RIGHT))
            bNoSuccessor = false;
      }
      if (iCarCount < 3) {
         for (i = 0; i < 4; i++) {
            if (bEmptyPosition(position, i * 4 + 3)) {
               movegen->bInsertPieceS(WHITE, i * 4 + 3, 0);
               movegen->bAddBoard();
            }
         }
      }
   } else { // Schwarz am Zug
      movegen->bSetEvent(TOPBORDER, REMOVE);
      iCarCount = iGetWeight64(position->i64BoardB);
      if (iCarCount == 0)
         bNoSuccessor = false;
      for (i = 1; i <= iCarCount; i++) {
         iCurrentPosition = iGetPiece64(i, position->i64BoardB);
         movegen->iMove(iCurrentPosition, DOWN);
         if (movegen->iMove(iCurrentPosition, LEFT) != IMPOSSIBLE)
```

```
            bNoSuccessor = false;
         if (movegen->iMove(iCurrentPosition, RIGHT) != IMPOSSIBLE)
            bNoSuccessor = false;
         if (movegen->bCanMove(iCurrentPosition, UP))
            bNoSuccessor = false;
      }
      if (iCarCount < 3) {
         for (i = 0; i < 4; i++) {
            if (bEmptyPosition(position, i)) {
               movegen->bInsertPieceS(BLACK, i, 0);
               movegen->bAddBoard();
            }
         }
      }
   }

   if (bNoSuccessor)
      movegen->iPass();

   return movegen->iGetMoveCount();

}
```

B Statistiken des Schachendspiels KR-KN




Schwarz am Zug
Anzahl Stellungen
Verlustdistanz
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80
9544 5952 5308 5384 3920 6136 9456 12124 18572 29188 31804 50168 73324 103228 144040 177940 158380 115196 86756 71028 62340 59072 58260 56724 49836 41424 34612 26408 18248 12008 10288 7120 5152 3152 2368 1200 504 240 72 16 8

Literatur

[All88] L.V. Allis. *A Knowledge-based Approach of Connect-Four - White Wins*. Vrije Universität Amsterdam, 1988.

[AllHer91] L.V. Allis und H.J. van den Herik., I.S. Herschberg. *Which games will survive?*. Heuristic Programming in Artificial Intelligence 2 - The second Computer Olympiad, Ellis Horwood Publishers, 1991.

[All94] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. University of Limburg, Doctorial Thesis, 1994.

[BalAll95] Henri Bal und L.V. Allis. *Parallel Retrograde Analysis on a Distributed System*. Vrije Universität Amsterdam, 1995.

[Don99] Christian Donninger. *Der Fortschritt ist nicht aufzuhalten*. Kaissiber (Ausgabe unbekannt), 1999.

[Don00] Christian Donninger. *W.Steinitz gegen L.Gott*. Kaissiber Ausg. 15, 2000.

[GasNie94] Ralph Gasser und Jürg Nievergelt. *Es ist entschieden: Das Mühlespiel ist unentschieden*. Informatik Spektrum 17(5), 1994.

[Gas95] Ralph Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search*. Dissertation ETH Zürich No.10927, 1995.

[Gas96] Ralph Gasser. *Solving Nine Men's Morris*. MSRI Publications Vol. 29, 1996.

[Han93] Curt Hansen. *Should the mysteries of the chess game be solved?* Offener Brief an das Schachmagazin *New In Chess*, 1993.

[Hei99] Ernst A. Heinz. *Endgame Databases and Efficient Index Schemes*. Universität Karlsruhe, 1999.

[LakSch94] Robert Lake, Jonathan Schaeffer and Paul Lu. *Solving Large Retrograde Analysis Problems Using a Network of Workstations*. University of Alberta, 1994.

[NaHaHe00] E.V. Nalimov, G.MC. Harworth and E.A. Heinz. *Space-Efficient Indexing of Chess Endgame Tables*. ICGA Journal, September 2000.

[Nal05] Persönlicher Schriftverkehr mit E.V. Nalimov, Januar 2005.

[Nun99] John Nunn. *Secrets of Rook Endings*. Gambit Publications, 1999.

[Nun02] John Nunn. *Secrets of Pawnless Endings*. Gambit Publications, 2002.

[Ree03] Eric van Reem. *Großmeister Barejew gegen Hiarcs*. Computerschach und Spiele Ausg. 2, 2003.

[RusNor95] Stuart J. Russel and Peter Norvig. *Artificial Intelligence. A Modern Approach*. Prentice-Hall, 1995.

[SchLak96] Jonathan Schaeffer and Robert Lake. *Solving the Game of Checkers*. University of Alberta, 1996.

[SteFri95] Dieter Steinweder und Frederic A. Friedel. *Schach am PC*. Markt&Technik Verlag, 1995.

[Stro70] T. Ströhlein. *Untersuchungen über Kombinatorische Spiele*. Dissertation TH München, 1970.

[Thi99] Ulrich Thiemonds. *Ein regelbasiertes Spielprogramm für Schachendspiele*. Diplomarbeit RFW-Universität Bonn, 1999.

[Tho86] Ken Thompson. *Retrograde Analysis of Certain Endgames*. ICGA Journal, Dezember 1986.

[Tho96] Ken Thompson. *6-Piece Endgames*. ICGA Journal, Dezember 1996.

[VoGr93] Colin Vout and Gordon Gray. *Challenging Puzzles*. Cambridge University Press, 1993.

[Win87] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, 1987.

[WuBel02] Ren Wu and Donald F. Beal. *A Memory Efficient Retrograde Algorithm and Its Application To Chinese Chess Endgames*. MSRI Publications Vol. 42, 2002.