Entwicklung eines

Double Dummy Skat Solvers

mit einer Anwendung für verdeckte Skatspiele

Diplomarbeit

von

Sebastian Kupferschmid

Albert-Ludwigs-Universität Freiburg
Fakultät für angewandte Wissenschaften
Institut für Informatik

Juli 2003

Danksagung

Ein großes Dankeschön an alle, die zum Gelingen dieser Arbeit beigetragen haben. Prof. Dr. Nebel danke ich dafür, dass er diese Arbeit ermöglicht und unterstützt hat. Mein besonderer Dank gilt außerdem Malte Helmert für die ausgezeichnete Betreuung, für die unzähligen Kaffees und wöchentliche Besprechungen, für Ratschläge zu C++ und für das Durchsehen der Arbeit. Birte Krüger danke ich für die moralische Unterstützung.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe.

Freiburg im Juli 2003 Sebastian Kupferschmid

Inhaltsverzeichnis

Kapitel 1

Einleitung

Spiele, in denen vollständige Informationen vorliegen, kann man mit den bekannten Suchverfahren wie z.B. $\alpha\beta$ lösen. Allerdings funktioniert die Anwendung dieses Algorithmus in Spielen mit unvollständiger Information nicht mehr.
In dieser Arbeit wird für das Skatspiel gezeigt, wie sich dieses Problem durch den Einsatz von Simulationen umgehen läßt. Für diesen Ansatz wird ein sogenannter schneller *Double Dummy Skat Solver* (DDS) benötigt. Ein DDS ist ein Programm, das für das Ausspiel mit offenen Karten den spieltheoretischen Wert berechnet.
Der größte Teil dieser Arbeit ist der Entwicklung des DDS gewidmet. Hier werden Techniken vorgestellt, mit denen sich $\alpha\beta$ beschleunigen läßt.
Im zweiten Teil wird eine Lösung für oben beschriebenes Problem gegeben und eine Anwendung für verdeckte Skatspiele präsentiert.

Nach einer kurzen Einführung in die Grundlagen des Skatspiels wird zunächst eine theoretische Basis für die Entwicklung des Double Dummy Solvers geschaffen.
Um die Geschwindigkeit des $\alpha\beta$-Algorithmus zu erhöhen, erfolgt im Anschluß eine Modifikation dieses Verfahrens.
Darauf aufbauend werden einige Erweiterungen in diesen Algorithmus eingebettet, um die Laufzeit weiter zu verkürzen.
Im Anschluß erfolgt eine Anpassung des entwickelten Double Dummy Solvers an das sogenannte Nullspiel, das innerhalb der betrachteten Varianten des Skatspiels eine Sonderrolle einnimmt.
Schließlich folgt eine Übertragung der des entwickelten Programms auf verdeckte Spiele.
Zum Abschluß wird eine Lösung für die erste Phase des Skatspiels, das Reizen und Drücken, präsentiert.

Kapitel 2

Grundlagen des Skatspiels

Da nicht jeder mit den Regeln des Skatspiels vertraut ist, erfolgt hier eine kurze Einführung in Begriffe, Spielregeln und Spielverlauf. Es wird kein Anspruch auf Vollständigkeit erhoben, da lediglich die Begriffe und Regeln vorgestellt werden, die für das Verständnis der Arbeit notwendig sind. Ein vollständiges verbindliches Regelwerk ist die Internationale Skatordnung [5].

2.0.1 Spielmaterial und Spielziel

Skat ist ein Kartenspiel für drei Personen. Gespielt wird mit 32 Karten in den Farben ♣, ♠, ♡ und ♢. Die einzelnen Farben bestehen aus 7, 8, 9, 10, Bube (J), Dame (Q), König (K) und As (A). Jeder Karte ist ein bestimmter Punktwert zugeordnet, insgesamt ergeben alle Karten zusammen 120 Punkte Beim Skatspiel spielen immer zwei Gegenspieler gegen einen Alleinspieler, diese Rollenverteilung wird vor jedem Spiel während des Reizens (siehe Abschnitt Reizen und Drücken) neu bestimmt. Ziel des Alleinspielers ist es, mindestens 61 Punkte zu erzielen, Ziel der Mitspieler ist das Erreichen von 60 und mehr

Karte	**Wert**
As	11
10	10
K	4
Q	3
J	2
9	0
8	0
7	0

Tabelle 2.1: Kartenwerte

Punkten. Zusätzliche Punkte erhält die Siegerpartei, wenn die Gegenpartei am Spielende weniger als 30 Punkte erreicht hat. Dieses Ergebnis nennt man "Schneider".

Allgemeines

Im Skatspiel unterscheidet man nicht nur die Farben ♣, ♠, ♡ und ♢ sondern auch Trümpfe und (Fehl-)farben. Die Buben gehören immer zu den Trümpfen, zusätzlich kann der Alleinspieler zu Beginn des Spiels noch eine Farbe festlegen, die dann auch als Trumpf zählt. Die Buben bilden dann in der Rangfolge: ♣, ♠, ♡, ♢ die höchsten Trümpfe, danach kommt ggf. die Trumpffarbe in der übrigen aus Tabelle (2.1) ersichtlichen Rangfolge. Die Trümpfe sind grundsätzlich ranghöher als die übrigen Farben.
Entsprechend den vorhandenen vier Farben gibt es vier Möglichkeiten für die sogenannten Farbspiele. Zusätzlich hat der Alleinspieler noch die Möglichkeit, nur die Buben zum Trumpf zu erklären, das ist dann ein Grand. Eine weitere Variante, bei der es gar keine Trümpfe gibt, ist das Nullspiel. Dieses wird in Abschnitt über Nullspiele gesondert erklärt. Insgesamt hat der Alleinspieler also sechs Varianten zur Auswahl.
Beim Skat herrscht ein sogenannter Farb- oder Bedienungszwang, d.h. auf Trumpfkarten müssen Trumpfkarten, auf Farbkarten muss die gleiche Farbe gespielt werden. Ist dies nicht möglich, kann eine beliebige Karte ausgespielt werden. Hat ein Spieler von einer Farbe oder von den Trümpfen überhaupt keine Karte auf der Hand, so nennt man ihn blank.

Spielvorbereitung

Die Karten werden gemischt und jeweils 10 an jeden Spieler ausgeteilt. Die zwei verbleibenden Karten bilden den sogenannten Skat und bleiben zunächst verdeckt.

Spielablauf

Das eigentliche Skatspiel unterteilt sich in zwei Phasen:

1. Reizen und Drücken
2. Ausspiel

Reizen und Drücken

Nach dem Austeilen der Karten entscheidet beim Reizen jeder Spieler anhand seiner 10 Karten, ob er in diesem Spiel Alleinspieler sein will oder nicht. Entscheidet er sich dafür, mit einem bestimmten Spiel Alleinspieler sein zu können, errechnet er den Wert, den dieses Spiel hat. Dieser Wert ist identisch mit der Zahl, bis zu der maximal gereizt werden kann. Derjenige der drei Spieler, der am höchsten reizen konnte, erhält den Skat. Den Skat darf er jetzt zusätzlich zu seinen zehn Handkarten aufnehmen und sich entscheiden, mit welchen zehn seiner nun zwölf Handkarten er spielen will. Die beiden übrigen legt er wieder auf den Tisch, d.h. er drückt diese Karten. Beim Reizen kann es aufgrund der Karten im Skat zu einem als Überreizen bezeichneten Fehler kommen. Hat sich der Alleinspieler überreizt, so ist das Spiel für ihn verloren.

Ausspiel

Ein Spieler eröffnet das Ausspiel, indem er eine beliebige Karte seiner Handkarten ausspielt. Danach spielen die beiden anderen im Uhrzeigersinn eine Karte aus, unter Beachtung des Farbzwangs.
Hat jeder Spieler eine Karte ausgespielt, bilden diese drei Karten einen sogenannten Stich. Derjenige Spieler, der die ranghöchste Karte gelegt hat, erhält den Stich und damit auch die darin enthaltenen Punktwerte der Karten.
Der zweite Stich wird von dem Spieler eröffnet, der den ersten Stich gewinnen konnte. Es bedienen wieder im Uhrzeigersinn die beiden anderen Spieler. So geht es weiter, bis der 10. Stich ausgespielt ist.
Der Spieler der die erste Karte des ersten Stichs legt, wird als Vorhand bezeichnet, sein linker Nachbar als Mittelhand, der verbleibende als Hinterhand.

Spielende

Nach Beendigung des 10. Stichs endet auch das Spiel. Nun werden die Punkte des Alleinspielers bzw. der Gegenspieler gezählt um den/die Sieger zu ermitteln.

Sonderfall: Nullspiel

Ziel des Nullspiels ist für den Alleinspieler, keinen einzigen Stich zu erhalten. Folglich spielen die Punktwerte der Karten in diesem Spiel keine Rolle. Im Unterschied zu den Trumpfspielen verlieren die Buben ihre Sonderstellung, die Kartenrangfolge ändert sich wie folgt:

höchste Karte	As
	König
	Dame
	Bube
	10
	9
	8
niedrigste Karte	7

Tabelle 2.2: Reihenfolge beim Nullspiel

Da keine Trümpfe existieren, ist allein die Kartenrangfolge innerhalb einer Farbe entscheidend für den Ausgang des Spiels. Sobald der Alleinspieler gezwungen ist, eine Karte so zu spielen, dass er den aktuellen Stich erhält, auch wenn dieser null Punkte enthält, hat er das Spiel verloren. In diesem Fall endet das Spiel ggf. auch bevor alle Karten ausgespielt wurden.

Kapitel 3

Vorbereitungen

Wenn beim Skat mit offenen Karten gespielt wird, sind alle Informationen allen Spielern zugänglich. Das Spiel ist dann mit Algorithmen wie z.B. $\alpha\beta$-Suche lösbar. Im vorliegenden Kapitel werden Grundlagen für die Entwicklung eines *Double Dummy Solvers* vorgestellt. Der Name Double Dummy ist aus der Bridge-Welt entlehnt. Dort bezeichnet ein Double Dummy Solver ein Programm, das unter Kenntnis der Karten aller vier Spieler und ohne Reizung Bridge spielt [4].

Der Alleinspieler ist in dieser Arbeit immer der MAX-Spieler und die beiden Gegenspieler die MIN-Spieler. Eine Spielsituation ist eine Situation, die sich während des Spiels ergibt. Jede dieser Spielsituationen unterscheidet sich von ihren Nachfolgern durch genau eine Karte. Das Ausspielen einer Karte ist ein Zustandsübergang, der in eine neue Spielsituation führt. Spielsituationen werden auch Spielstände oder Positionen genannt.
Im Gegensatz zu Spielen wie Schach folgt auf einen MIN-Knoten nicht unbedingt ein MAX-Knoten und umgekehrt. Es kann passieren, dass auf einen Pfad von der Wurzel des Spielbaums zu einem der Blätter vier MIN Knoten hintereinander folgen. Das tritt ein, wenn MAX einen Stich eröffnet und der nachfolgende MIN-Spieler den Stich erhält. Es können auch zwei MAX-Knoten hintereinander liegen, wenn MAX den Stich mit der dritten Karte gewinnt.

Da ein Skatspiel, wie sich zeigen wird, im Mittel einen verhältnismäßig kleinen Spielbaum hat, kann dieser vollständig durchsucht werden. Das hat den Vorteil, dass nicht auf Heuristiken zurückgegriffen werden muss, die das Ergebnis der Suche beeinflussen. Dadurch erhält man den tatsächlichen, spieltheoretischen Wert des Spiels. Allerdings ist dieses Ziel in der Praxis nur zu erreichen, wenn man die $\alpha\beta$-Suche durch Erweiterungen beschleunigt. Diese

Erweiterungen werden in den nächsten Kapiteln vorgestellt und untersucht.

Die kleinen Spielbäume ermöglichen außerdem den Einsatz von Algorithmen wie z.B. *Proof Number Search* (pn-search) von V. Allis et. al. [1]. Die Idee hinter diesem Algorithmus ist den spieltheoretischen Wert des Spiels zu "beweisen". Zu diesem Zweck ordnet man der Wurzel des Spielbaums zu Beginn einen Wert zu. Im Verlauf der Suche soll nun bewiesen bzw. widerlegt werden, dass dieser Wert mit dem Minimax-Wert der Wurzel übereinstimmt. In jedem Knoten werden dafür zwei Werte gespeichert: die *proof number* und die *disproof number*. Die *proof number* gibt an, wie viele Nachfolger dieses Knotens mindestens betrachtet werden müssen, um den Wert der Wurzel zu beweisen. Die *disproof number* ist die Anzahl der Nachfolgeknoten, die mindestens examiniert werden müssen, um diesen Wert zu widerlegen. Es wird immer der Knoten zuerst expandiert, der möglicherweise den größten Einfluß auf den Minimax-Wert der Wurzel hat und mit kleinstmöglichem Aufwand zu analysieren ist. *Proof Number Search* zählt zu den *Best-First*-Suchverfahren. Diese Arbeit beschränkt sich auf Tiefensuche.

3.1 Repräsentation von Karten

Bevor nun die ersten Algorithmen angegeben werden, muss zunächst ermittelt werden, wie die einzelnen Spielstände eines Spiels repräsentiert werden können.
Da es 32 Karten gibt, bietet es sich an, jeder Karte ein festes Bit in einer 32-Bit Zahl zuzuordnen. In der Repräsentation, die in dieser Arbeit gewählt wurde, belegen Karten, die dieselbe Farbe haben, benachbarte Bits. Wenn zwei Karten die gleiche Farbe haben, belegt die höhere Karte das höherwertige Bit. Eine Ausnahme von dieser Regel bilden die vier Buben. Sie belegen die vier höchsten Bits, wobei der $\clubsuit J$ das höchste und der $\diamondsuit J$ das niedrigste belegt.

31	30	29	28	27	26	...	21	20	...	0
$\clubsuit J$	$\spadesuit S$	$\heartsuit J$	$\diamondsuit J$	$\clubsuit A$	$\clubsuit 10$	...	$\clubsuit 7$	$\spadesuit A$	...	$\diamondsuit 7$

Farben lassen sich durch die boolesche Disjunktion der einzelnen Karten repräsentieren, die zu ihr gehören. Dadurch können zwei beliebige Karten einfach miteinander verglichen werden. Ein Algorithmus, der das leistet, ist in Abbildung (3.3) zu sehen.
Um den Gewinner eines Stichs zu ermitteln muss man Karten miteinander vergleichen können. Da zwei Karten i.A. nicht miteinander vergleichbar sind,

muss man sich auf eine Farbe festlegen, auf die beide Karten projiziert werden. D.h. eine Karte, die nicht von dieser Farbe ist, wird auf Null abgebildet, wenn es sich nicht um einen Trumpf handelt. Der Vergleich zweier Karten wird so auf das Vergleichen zweier natürlicher Zahlen reduziert.
Die gewählte Repräsentation läßt sich einfach umkehren. Zuerst wird eine 32-Bit Zahl in ihre einzelnen Bits zerlegt (siehe Abbildung (3.1)), danach wird für diese Zweierpotenzen ermittelt, welche Karten sie repräsentieren (siehe Abbildung (3.2)).

```
def next_card(cards):
    return cards & (cards - 1) ^ cards
```

Abbildung 3.1: nächste Karte

```
CARD_TO_INDEX = ( -1,  0,  1, 26,  2, 23, 27, -1,  3, 16,
                  24, 30, 28, 11, -1, 13,  4,  7, 17, -1,
                  25, 22, 31, 15, 29, 10, 12,  6, -1, 21,
                  14,  9,  5, 20,  8, 19, 18 )

def get_Iidex(bits):
    return CARD_TO_INDEX[bits % 37]
```

Abbildung 3.2: Umkehrung der Repräsentation

Der Algorithmus aus Abbildung (3.2) nutzt die Eigenschaft, dass die Berechnung von $2^i \mod 37$ für jedes $i \in \{0, \ldots, 31\}$ ein anderes Ergebnis liefert. Durch vollständige Aufzählung hat sich gezeigt, dass 37 die kleinste Zahl ist, für die das gilt. In der Liste `CARD_TO_INDEX` steht an den Stellen 0, 7, 14, 19 und 28 der Wert -1. Bei der Berechnung von $2^i \mod 37$ treten diese Indizes aber nie auf.

```
def ge(card₁, card₂):
    if card₁ oder card₂ ist Trumpf:
        card₁ &= trump
        card₂ &= trump
    else:
        card₂ &= Farbe von card₁
    return card₁ > card₂
```

Abbildung 3.3: Vergleich zweier Karten

3.2 Repräsentation von Zuständen

Nachdem nun die Repräsentation für eine Menge von Karten vorgestellt wurde, kann daraus eine Repräsentation für Spielzuständen entwickelt werden. In einem Spiel werden die fixen Fakten global gespeichert. Das sind

- die Karten, die jeder Spieler zu Beginn des Spiels hat,
- der Skat,
- die Information welcher Spieler Alleinspieler ist und
- was für ein Spiel gespielt wird.

Die Spielstände, die in einem Skatspiel auftreten können unterscheiden sich nur durch folgende lokale Fakten:

- Karten, die die Spieler noch auf der Hand haben,
- Karten, die auf dem Tisch liegen,
- Spieler, der als nächstes eine Karte legen muss,
- Farbe, die bekannt werden muss und
- Anzahl der Punkte jeder Partei, die bisher erzielt wurden.

Für die Repräsentation des ersten Punktes reicht es, die bitweise Disjunktion der Kartenrepräsentationen zu verwenden. Mit Hilfe der globalen Fakten kann aus dieser Repräsentation ermittelt werden, welche Karten ein Spieler in p hat. Das soll an folgenden Beispiel illustriert werden.

Beispiel 3.1 Der Einfachheit halber gebe es nur die Karten $\diamondsuit 7$, $\diamondsuit 8$, $\diamondsuit 9$, $\diamondsuit Q$, $\diamondsuit K$, $\diamondsuit 10$ und $\diamondsuit A$. Für die Repräsentation von Karten genügen daher 7-Bit Zahlen. $\diamondsuit 7$ werde das niederwertigste und $\diamondsuit A$ das höchstwertige Bit zugeordnet.
Angenommen in einem Spiel hat Spieler 1 am Anfang die Karten $\diamondsuit K$, $\diamondsuit 10$ und $\diamondsuit A$. Diese Karten werden durch 0000111 repräsentiert. Nachdem er $\diamondsuit 10$ gespielt hat ergibt sich folgender Spielstand p.

	Karten		**Repräsentation**
Spieler 0	$\diamondsuit 7$	$\diamondsuit Q$	1001000
Spieler 1	$\diamondsuit K$	$\diamondsuit A$	0000101
Spieler 2	$\diamondsuit 8$	$\diamondsuit 9$	0110000

Die bitweise Disjunktion der drei Repräsentationen ergibt 1111101. Diese Zahl repräsentiert alle Karten, die in p noch im Spiel sind.
Die Repräsentation der Karten, die Spieler 1 in p auf der Hand hält bekommt man, indem 0000111 (das Kodifikat seiner Anfangskarten) mit 1111101 durch eine bitweise Konjunktion verknüpft wird.

3.3 Minimax

Aufbauend auf diese Repräsentationen können jetzt Algorithmen zur Lösung eines vereinfachten Skatspiels angegeben werden. Bei dieser Art von Spielen bekommt jeder Spieler $k \leq 10$ Karten, zwei Karten bilden den Skat und Spieler 0 spielt alleine. Die Art des Spiels wird zufällig gewählt. Diese Spiele stimmen mit dem Teil des verdeckten Skatspiels überein, der nach dem Drücken und der Spielankündigung beginnt.
Die Basis für die in den nächsten Kapiteln vorgestellte Algorithmen bildet der Minimax-Algorithmus (siehe Abbildung (3.4)). In dieser Arbeit ist er von theoretischer Bedeutung, da Minimax den korrekten, spieltheoretischen Wert einer Position zurückliefert. Diese Eigenschaft wird ausgenutzt, um die Korrektheit anderer Algorithmen zu zeigen.

```
def minimax(p):
    if p isa Leaf:
        return T(p)
    if p isa MAX_Node:
        res = -infinity
    else:
        res = +infinity

    for q in succ(p):
        if p isa MAX_Node:
            res = max(res, minimax(q))
        else:
            res = min(res, minimax(q))

    return res
```

Abbildung 3.4: Minimax

Hierbei wird mit $T(p)$ für einen beliebigen Knoten p die von MAX bisher erzielten Punkte (Punkte aus Skat + gewonnene Stiche) bezeichnet. Für Blätter p ist $T(p)$ damit die Gesamtzahl der erzielten Punkte. Mit $t(p)$ wird im Folgenden die Zahl der Punkte bezeichnet, die MAX durch das Legen der letzten

Karte erzielt hat. Nur für jeden dritten Zug kann $t(p) \neq 0$ gelten, nämlich wenn MAX den Stich gewonnen hat. Wenn $(p_1, \ldots, p_n)$ ein Pfad von der Wurzel zu p ist, dann gilt:

$$T(p) = \sum_{i=1}^{n} t(p_i) + skat.$$

Kapitel 4

Der $\alpha\beta$-Algorithmus

Erweitert man Minimax um $\alpha\beta$-Pruning, erhält man den Algorithmus aus Abbildung (4.1). Leider ist dieser Algorithmus genau wie Minimax viel zu langsam, um ein komplettes Spiel durchzurechnen.

```
def ab(p, alpha, beta):
    if p isa Leaf:
        return T(p)

    for q in succ(p):
        if p isa MAX_Node:
            alpha = max(alpha, ab(q, alpha, beta))
        else:
            beta = min(beta, ab(q, alpha, beta))
        if alpha >= beta:
            break

        if p isa MAX_Node:
            return alpha
        else:
            return beta
```

Abbildung 4.1: $\alpha\beta$-Pruning

Um beispielsweise den Wert für folgendes ♣-Spiel mit nur 7 Karten

Spieler 0	♣J	♣10	♡8	♡K	♡10	♠7	♠K
Spieler 1	♠J	♡J	♣8	♢9	♢Q	♢10	♢A
Spieler 2	♣7	♣Q	♢7	♠8	♠9	♡7	♡A

zu berechnen, benötigt man folgende Zeiten[1]:

[1] Alle Tests wurden auf einem Athlon-XP-2100+ mit 256MB Ram und gcc-3.2 ausgeführt

Algorithmus	**Zeit**
Minimax	140,36 s
$\alpha\beta$	2,45 s

Für das folgende ♣-Spiel mit 9 Karten benötigt $\alpha\beta$ 187,88 s. Minimax benötigt für die Berechnung der ersten Karte schon über 80 min.

Spieler 0	♣7	♣8	♣Q	♣K	♣A	♢7	♢10	♢A	♠A
Spieler 1	♡J	♢8	♢9	♢Q	♠9	♡7	♡8	♡9	♡Q
Spieler 2	♢J	♣9	♡A	♠7	♠8	♠Q	♠K	♠10	♢K

Mit geeigneten Techniken kann $\alpha\beta$ erheblich beschleunigt werden. Bevor diese Erweiterungen vorgestellt werden können, müssen noch ein paar Grundlagen geschaffen werden.

Übrigens verliert der Alleinspieler (Spieler 0) das erste Spiel auf jeden Fall: Beginnt er mit ♡ oder ♠, erhält er 5, sonst 2 Punkte von 72 erreichbaren. Im zweiten Spiel kann er 78 von 92 Punkten erreichen. Das gelingt ihm, wenn er mit ♠A, ♣7 oder ♣8 das Spiel eröffnet.

4.1 Minimax entkoppelt

$\alpha\beta$ soll zuerst um eine *Transpositionstabelle* (TT) erweitert werden. Eine Transpositionstabelle ist ein assoziatives Array – meist eine Hashtabelle – in dem bei Expansion eines Knotens zuerst geprüft wird, ob die Bewertung für diesen Spielstand schon einmal berechnet wurde. Wenn dem so ist, kann das Ergebnis wiederverwendet werden. Durch diese Erweiterung kann man sich das wiederholte Traversieren gleicher Teilbäume sparen.

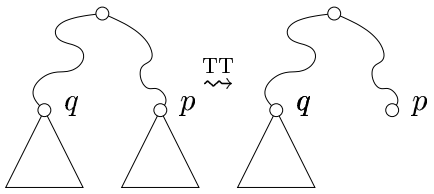


Abbildung 4.2: Funktionsweise der Transpositionstabelle

Abbildung (4.2) zeigt den Effekt, der durch den Einsatz von Transpositionstabellen zu beobachten ist. Hierbei sei $p \doteq q$. Durch das Speichern der Zwischenergebnisse kann so eine eine erneute Analyse von p eingespart werden.

Das Problem hierbei ist, dass den Blättern des Spielbaums akkumulierte Punkte zuordnet werden. Damit der Wert eines Knotens aber wiederverwertbar ist, darf er nur von den Bewertungen der Knoten des Teilbaums S abhängen, der bei ihm beginnt.

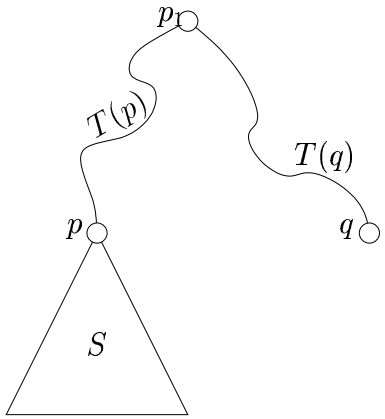


Abbildung 4.3: Problematik akkumulierter Punkte

Wenn zwei Positionen p und q gleich sind, dann ist der Pfad von der Wurzel p_1 des Spielsbaums zu p bzw. zu q i.A. verschieden (siehe Abbildung (4.3)). Würde jetzt der Wert für p von $T(p)$ abhängen, dann kann dieser Wert nicht für q verwendet werden.
Aus diesem Grund muss der Minimax-Algorithmus in eine andere, äquivalente Form gebracht werden, bei der der Wert von p nur von den Werten der Knoten in S abhängt.

Satz 4.1 *Sei P die Menge aller Spielpositionen, succ(p) die Menge der Nachfolgepositionen der Position p, $M : P \to \mathbb{R}$ die durch den Minimax-Algorithmus definierte Funktion und sei $m : P \to \mathbb{R}$ wie folgt definiert:*

$$m(p) = \begin{cases} 0 & \textit{falls } p \textit{ Blatt} \\ \max\limits_{p' \in \mathrm{succ}(p)} (t(p') + m(p')) & \textit{falls } p \textit{ MAX-Knoten} \\ \min\limits_{p' \in \mathrm{succ}(p)} (t(p') + m(p')) & \textit{falls } p \textit{ MIN-Knoten} \end{cases}$$

Dann gilt

$$\forall p \in P \quad \mathrm{M}(p) = \mathrm{m}(p) + \mathrm{T}(p).$$

Anschaulich heißt das, dass $m(p)$ für einen MAX-Knoten p die maximale Bewertung der Nachfolge-Knoten ist, zuzüglich der Punkte, die in den

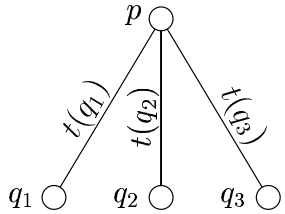


Abbildung 4.4: Funktionsweise von $m(p)$

Nachfolge-Knoten erzielt werden. Analog für MIN-Knoten.

Da beim Skatspiel erst in der dritten Spielposition Punkte erzielt werden, nämlich dann, wenn der erste Stich vorbei ist, ist $T(p) = \textit{Punkte aus dem Skat}$ für die Wurzel des Spielbaumes.

Beweis zu 4.1 *Induktion über die Höhe eines Spielbaumes*

IA: Der Baum besteht nur aus einem einzelnen Blatt p.
$M(p) = T(p) = t(p) + m(p)$

IS: Sei p ein MAX-Knoten, dann gilt

$$\begin{aligned} \mathrm{M}(p) &\overset{def}{=} \max_{p' \in \mathrm{succ}(p)} \mathrm{M}(p') \\ &\overset{IV}{=} \max_{p' \in \mathrm{succ}(p)} (\mathrm{T}(p') + m(p')) \\ &= \mathrm{T}(p) + \max_{p' \in \mathrm{succ}(p)} (\mathrm{t}(p') + \mathrm{m}(p')) \\ &\overset{def}{=} \mathrm{T}(p) + \mathrm{m}(p) \end{aligned}$$

Der Beweis für MIN-Knoten verläuft analog.

□

Mit Hilfe des Satzes kann das Ergebnis jedes Knotens p von dem zu p führenden Pfad entkoppelt werden.
Die gleiche Problematik existiert auch für den $\alpha\beta$-Algorithmus. Zusätzlich dürfen dort die Schranken α und β nicht von $T(p)$ abhängen.
Der *AB*-Algorithmus aus Abbildung (4.5) berechnet für alle $p \in P$ den gleichen Wert wie der $\alpha\beta$-Algorithmus. Abbildung (4.6) zeigt im linken Baum die Arbeitsweise von $\alpha\beta$ und im rechten die von Algorithmus (4.5). Beide Bäume repräsentieren das gleiche Spiel. In der linken Abbildung sind den Blättern

die akkumulierten Punkte zugeordnet, in der rechten stehen an den Kanten die Punkte, die erzielt werden, wenn dieser Zug ausgeführt wird. Die Algorithmen expandieren dabei die Knoten des Spielbaums von links nach rechts. Die Intervalle an jedem Knoten geben die α- und β-Werte zum Zeitpunkt des Beschneidens des rechten Knotens an. Beide Algorithmen berechnen den gleichen Wert für die Wurzel des Spielbaumes. Diese Identität läßt sich mit folgendem Satz zusammenfassen.

```
def AB(p, alpha, beta):
    if p isa Leaf:
        return 0
    for q in succ(p):
        if p isa MAX_Node:
            alpha = max(alpha, t(q) + AB(q, alpha - t(q), beta - t(q)))
        else:
            beta = min(beta, t(q) + AB(q, alpha - t(q), beta - t(q)))
        if alpha >= beta:
            break
    if p isa Max_Node:
        return alpha
    else:
        return beta
```

Abbildung 4.5: AB-Algorithmus

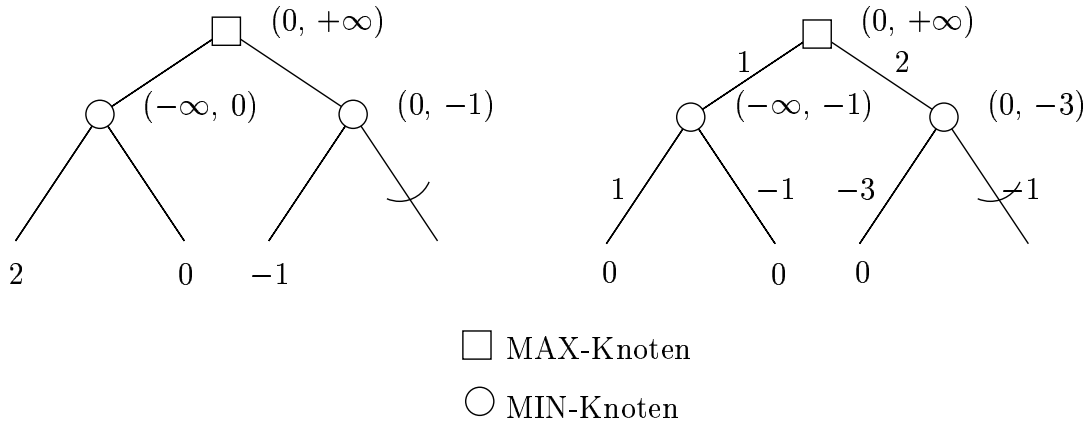


Abbildung 4.6: Arbeitsweise von $\alpha\beta$ und AB

Satz 4.2 *Sei P die Menge aller Spielpositionen, dann gilt*

$$\forall p \in P : AB(p, \alpha - T(p), \beta - T(p)) + T(p) = ab(p, \alpha, \beta)$$

Beweis zu 4.2 *Induktion über die Höhe des Spielbaums*

IA: folgt sofort aus den Definitionen von AB, T und ab.

IS: Sei p ein MAX-Knoten, dann läßt sich der Algorithmus wie folgt umschreiben:

```
def AB(p, alpha, beta):
    for q in succ(p):
        alpha = max(alpha, t(q) + AB(q, a - t(q), b - t(q)))
        if alpha >= beta:
            break
    return alpha
```

Sei $\alpha' = \alpha - T(p)$ und $\beta' = \beta - T(p)$, dann läßt sich bei einem Aufruf von $AB(p, \alpha', \beta')$ die Update-Regel für α wie folgt umschreiben:

$$\begin{aligned}
\alpha' &= \max(\alpha',\, t(q) + AB(q,\, \alpha' - t(q),\, \beta' - t(q))) \\
&= \max(\alpha',\, t(q) + AB(q,\, \alpha - T(p) - t(q),\, \beta - T(p) - t(q))) \\
&= \max(\alpha',\, t(q) + AB(q,\, \alpha - T(q),\, \beta - T(q))) \\
&\overset{IV}{=} \max(\alpha',\, t(q) + ab(q,\, \alpha,\, \beta) - T(q)) \\
&= \max(\alpha,\, ab(q,\, \alpha,\, \beta)) - T(p)
\end{aligned}$$

$$\alpha = \max(\alpha,\, ab(q,\, \alpha,\, \beta))$$

Die beiden Update-Regeln sind also äquivalent, hieraus folgt sofort die Behauptung.
Für MIN-Knoten verläuft der Beweis analog.

□

4.2 Korrektheit von $\alpha\beta$

Der $\alpha\beta$-Algorithmus liefert nicht für jedes Wertepaar $(\alpha,\, \beta)$ den spieltheoretischen Wert einer Position zurück. Das liegt daran, dass der spieltheoretische Wert einer Position nicht immer in $[\alpha,\, \beta]$ liegt. Die erhaltenen Werte lassen sich folgendermaßen in drei Gruppen einteilen:

- **valid:** $ab(p, \alpha, \beta)$ stimmt mit dem Minimax-Wert von p überein.
- **lbound:** $ab(p, \alpha, \beta)$ ist eine untere Schranke für den Minimax-Wert von p.
- **ubound:** $ab(p, \alpha, \beta)$ ist eine ober Schranke für den Minimax-Wert von p.

Satz 4.3 (Korrektheit von $\alpha\beta$) *Sei p eine beliebige Position in einem Spiel und $\alpha < \beta$. Dann gelten für den $\alpha\beta$-Algorithmus folgende drei Aussagen:*

- $ab(p, \alpha, \beta) \leq \alpha \iff m(p) \leq \alpha$
- $ab(p, \alpha, \beta) \geq \beta \iff m(p) \geq \beta$
- $\alpha < ab(p, \alpha, \beta) < \beta \iff \alpha < m(p) < \beta$.

Im dritten Fall gilt zusätzlich $ab(p, \alpha, \beta) = m(p)$.

Die dritte Eigenschaft des $\alpha\beta$-Algorithmus charakterisiert die Gruppe **valid** und die beiden anderen **ubound** bzw. **lbound**.
Um obige Behauptungen zu beweisen, wird der Algorithmus aus Abbildung (4.7), bzw. die äquivalente Form aus (4.8) (für den Fall, dass p ein Max-Knoten ist), verwendet.

Beweis zu 4.3 (Korrektheit von $\alpha\beta$) *Der Beweis erfolgt durch Induktion über die Höhe des Baumes. Der Induktionsanfang ist klar. Sei p ein MAX-Knoten, dann gilt folgendes für den Induktionsschritt:*

"$\Rightarrow$" $ab(p, \alpha, \beta) \geq \beta \iff$ *es existiert ein* $j \in \{1, \ldots, n\}$ *mit* $res_j \geq \beta$. *Da* $\forall i : res_i \leq res_{i+1}$ *und* $res_i = \max\{val_i, \ldots, val_1, \alpha\}$ *gilt* $res_j = val_j \geq \beta$, *also* $val_j = ab(q_j, res_{j-1}, \beta) \geq \beta$. *Nach Induktionsvoraussetzung gilt dann:* $m(q_j) \geq \beta$. *Da* $m(p) = \max_{q \in \mathrm{succ}(p)} m(q)$ *gilt auch* $m(p) \geq \beta$

"$\Leftarrow$"

$$
\begin{aligned}
& ab(p, \alpha, \beta) < \beta \\
\Rightarrow & res_i < \beta \text{ für alle } i = 1, \ldots, n \\
\Rightarrow & val_1, \ldots, val_n < \beta \\
\overset{IV}{\Rightarrow} & m(q_i) < \beta \\
\Rightarrow & m(p) = \max_{i=1,\ldots,n} m(q_i) < \beta
\end{aligned}
$$

Der Beweis für $ab(p, \alpha, \beta) \geq \beta \iff v(p) \geq \alpha$ *funktioniert analog. Die dritte Aussage folgt ähnlich.*

□

```
def ab(p, alpha, beta):
    if p isa Leaf:
        return v(p)
    if p isa MAX_Node:
        res = alpha
        for q in succ(p):
            val = ab(q, res, beta)
            res = max(res, val)
            if res >= beta:
                return res
    elif p isa MIN_Node:
        res = beta
        for q in succ(p):
            val = ab(q, alpha, res)
            res = min(res, val)
            if res <= alpha:
                return res
    return res
```

Abbildung 4.7: $\alpha\beta$-Algorithmus

```
def ab(p, alpha, beta):
    res_0 = alpha
    {q_1,...,q_n} = succ(p)
    for i = 1 to n:
        val_i = ab(q_i, res_{i-1}, beta)
        res_i = max(val_i,...,val_1, alpha)
        if res_i >= beta:
            return res_i
    return max(val_n,...,val_1, alpha)
```

Abbildung 4.8: $\alpha\beta$-Algorithmus für MAX-Knoten

Kapitel 5

Transpositionstabellen

In diesem Kapitel wird der Einsatz der Transpositionstabelle genauer erklärt. Durch diese Erweiterung ist der $\alpha\beta$-Algorithmus in der Lage ein offenes Spiel in akzeptabler Zeit zu berechnen.

5.1 Hashen von Zuständen

Wie bereits in der Einleitung erwähnt, benötigt man zur Repräsentation eines Spielstands 32 Bit für die Karten der Spieler. Zusätzlich müssen die Karten des aktuellen Stichs, die Information, welcher Spieler Vorhand ist und welche Farbe man bedienen muss, gespeichert werden. Zusammen sind das auf jeden Fall deutlich mehr als 32 Bit.
Sind aber zwei Spielstände gleich, in denen ein Stich noch nicht zu Ende ist, dann haben diese Positionen einen gemeinsamen Vorfahren (nämlich den Spielstand zu Beginn des Stichs). Es genügt also, in der Hashtabelle nur Positionen zu speichern, bei denen keine Karte auf dem Tisch liegt. Diese Positionen können durch 32 Bit repräsentiert werden. Dabei werden 30 Bit für die Repräsentation der Karten verwendet, die momentan noch im Spiel sind. Die beiden Bits, die den Skat repräsentieren, werden nicht benötigt, da sie global gespeichert sind. In diesen zwei Bits wird die Information gespeichert, welcher Spieler den nächsten Stich eröffnet.
Dadurch lassen sich Spielsituationen eindeutig 32-Bit Zahlen zuordnen. Auf das Hashen von Blättern (Positionen, in denen alle Karten ausgespielt wurden) wird verzichtet. Für Blätter kann nämlich schneller der spieltheoretische Wert ermittelt werden, als es durch Nachsehen in der Hashtabelle möglich wäre.
Die 32-Bit Zahl wird einer Hashtabelle als Schlüssel übergeben. Der Hashwert wird von den Informationen gebildet, die mit dieser Position gespeichert

werden sollen. Zur Implementierung wurde auf das Template `hash_map` der STL[1] zurückgegriffen.

5.2 $\alpha\beta$ mit Transpositionstabelle

Wenn man den Algorithmus aus Abbildung (4.5) um Transpositionstabellen erweitert, dann erhält man den Algorithmus aus Abbildung (5.2). Dieser Algorithmus berechnet für eine Spielposition p genau den selben Wert wie $\alpha\beta$, nur wesentlich schneller. Die Abbildungen (5.1) und (5.3) visualisieren einen Test mit 50 zufällig gewählten Spielen. In jedem dieser Spiele hat jeder Spieler 10 Karten. Der Test macht deutlich, dass im Vergleich zu $\alpha\beta$ ohne Transpositionstabelle wesentlich weniger Knoten evaluiert werden müssen.

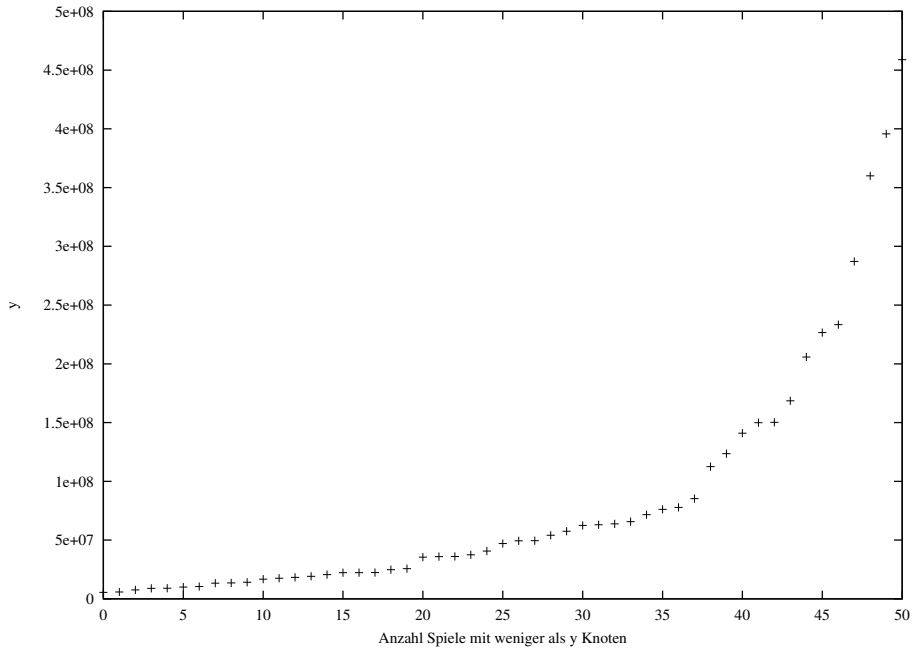


Abbildung 5.1: $\alpha\beta$ mit Transpositionstabelle (Knoten)

Die mittlere Ausführungszeit für die Berechnung eines dieser Spiele beträgt 24,21s. Im Mittel werden pro Spiel noch 84.912.121 Knoten evaluiert.
Da die Ausführungszeit von $\alpha\beta$ für ein Spiel mit 10 Karten pro Spieler extrem hoch ist, wird hier exemplarisch das Beispiel mit 9 Karten pro Spieler aus Kapitel 4 wiederholt.

[1] C++-Standardbibliothek

Spieler 0	♣7	♣8	♣Q	♣K	♣A	♢7	♢10	♢A	♠A
Spieler 1	♡J	♢8	♢9	♢Q	♠9	♡7	♡8	♡9	♡Q
Spieler 2	♢J	♣9	♡A	♠7	♠8	♠Q	♠K	♠10	♢K

Algorithmus	**Zeit**	**Knoten**	**Einträge in TT**
$\alpha\beta$	187,88 s	2.021.578.764	-
$\alpha\beta\_tt$	0,40 s	2.539.062	67.873

Durch die Erweiterung um Transpositionstabellen ist es jetzt möglich "komplette Spiele" zu berechnen. Dass $\alpha\beta$ und $\alpha\beta\_tt$ die gleichen Ergebnisse liefern, läßt sich mit folgendem Satz zusammenfassen:

Satz 5.1 *Sei $\alpha < \beta$ und p ein beliebiger Knoten im Spielbaum, dann gilt:*

$$ab(p,\, \alpha,\, \beta) = ab\_tt(p,\, \alpha,\, \beta)$$

Mit dem Korrektheitsbeweis für den $\alpha\beta$-Algorithmus aus Kapitel 4 kann nun gezeigt werden, dass die Erweiterung um Transpositionstabellen ebenfalls zu korrekten Werten führt.

Beweis zu 5.1 *Bevor ein Wert aus der Hashtabelle verwendet wird, sind die beiden Algorithmen offensichtlich gleich. Sei q der erste Knoten, der ein zweites mal analysiert wird. Beim ersten Besuch von q wurde dieser mit einem Aufruf von $ab(q,\, \alpha',\, \beta')$ analysiert. Es lassen sich dann folgende drei Fälle unterscheiden:*

1. *$val \leq \alpha'$ (kann nur bei MIN-Knoten auftreten)*
2. *$\alpha' < val < \beta'$*
3. *$\beta' \leq val$ (kann nur bei MAX-Knoten auftreten)*

Es ist klar, dass im zweiten Fall der Einsatz der Hashtabelle zu korrekten Werten führt, da hier $val = M(q)$ gilt.
Sei q ein MAX-Knoten und $ab(q,\, \alpha',\, \beta') = val \geq \beta$. In die Hashtabelle wurde dann das Tripel $(q,\, val, \mathbf{lbound})$ eingetragen. Für den Nachfolger p von q, der das Abschneiden verursacht hat, gilt also $ab(p,\, res,\, \beta') = val \geq \beta'$ mit $res \geq \alpha$.
Evaluiert man bei einem erneuten Besuch von q zuerst p, dann liefert der Aufruf von $ab(p,\, \alpha,\, \beta)$ den Wert val zurück. Danach würde α angepaßt werden. Diese Vorgehensweise ist äquivalent dazu, dass statt einer erneuten Analyse dieser Wert aus der Hashtabelle verwendet wird.
Nach dem Zugriff auf die Hashtabelle und dem Anpassen von α sind $\alpha\beta$ und $\alpha\beta\_tt$ wieder gleich.
*Der Beweis für **ubound**-Einträge verläuft analog.* □

5.3 Abschätzung der Größe des Spielbaums

Mit dem Einsatz einer Hashtabelle ergibt sich sofort auch die Frage nach dem benötigten Speicherplatz für die Tabelle.
Die Anzahl der verschiedenen Positionen, die im Verlauf eines beliebigen Skatspiels auftreten können, lässt sich folgendermaßen noch oben abschätzen. Da ein Spieler das Spiel eröffnet, gibt es genau eine Position, in der 30 Karten im Spiel sind, d.h. in den Händen der Spieler gehalten werden. Die Zahl der Positionen mit 29 Karten, ist durch $3 \cdot \binom{10}{9}\binom{10}{10}^2$ beschränkt. Da Farbe bekannt werden muss, ist der Verzweigungsgrad i.A. kleiner als die Anzahl der Karten, die ein Spieler noch auf der Hand hat. Da sich in jedem dritte Stich die Ausspielreihenfolge ändern kann, ist die Anzahl dieser Positionen durch $3 \cdot \binom{10}{i}^3$ beschränkt. Die Zahl der verschiedenen Positionen in einem Spiel ist also beschränkt durch

$$1 + \underbrace{3 \cdot \sum_{i=0}^{9} \binom{10}{i}\binom{10}{i+1}^2}_{\substack{\#\text{Positionen mit einer} \\ \text{Karte auf Tisch}}} + \underbrace{3 \cdot \sum_{i=0}^{9} \binom{10}{i}^2\binom{10}{i+1}}_{\substack{\#\text{Positionen mit zwei} \\ \text{Karten auf Tisch}}} + \underbrace{3 \cdot \sum_{i=1}^{9} \binom{10}{i}^3 + 1}_{\substack{\#\text{Positionen ohne} \\ \text{Karten auf Tisch}}} .$$

Die Auswertung dieses Terms ergibt 316.581.176. Da in der Hashtabelle nur Positionen gespeichert werden, in denen keine Karten auf dem Tisch liegen, kann man die Anzahl der Einträge in der Tabelle durch $3 \cdot \sum_{i=1}^{9} \binom{10}{i}^3 + 1 = 114.495.775$ nach oben abschätzen. Da man schon für die Schlüssel vier Byte benötigt, würde diese Tabelle über 400 MB groß werden. Allerdings ließ sich so etwas nie beobachten. Abbildung (5.4) zeigt für 300 zufällig gewählte Spiele mit 10 Karten pro Spieler die Anzahl der Einträge in der Hashtabelle, die $\alpha\beta\_tt$ anlegt. Der Durchschnitt liegt in diesem Test bei 1.498.831 Knoten pro Spiel, das Maximum bei 7.761.778 Knoten.
$\alpha\beta\_tt$ wurde für diesen Test gewählt, da die in den folgenden Kapiteln besprochene Algorithmen weniger Knoten evaluieren und so auch weniger Einträge anlegen.

```
def ab_tt(p, alpha, beta):
    if p isa Leaf:
        return 0

    if hash.lookup(p, val, flag):
        if flag == VALID:
            return val
        elif flag == LBOUND:
            alpha = max(alpha, val)
        elif flag == UBOUND:
            beta = min(beta, val)
        if alpha >= beta:
            return val

    if p isa MAX_Node:
        res = alpha
    else:
        res = beta

    for q in succ(p):
        if p isa MAX_Node:
            succVal = t(q) + ab_tt(q, res - t(q), beta - t(q))
            res = max(res, succVal)
            if res >= beta:
                hash.add(p, res, LBOUND)
                return res
        elif p isa MIN_Node:
            succVal = t(q) + ab_tt(q, alpha - t(q), res - t(q))
            res = min(res, succVal)
            if res <= alpha:
                hash.add(p, res, UBOUND)
                return res
    hash.add(p, res, VALID)
    return res
```

Abbildung 5.2: $\alpha\beta$ mit Transpositionstabelle

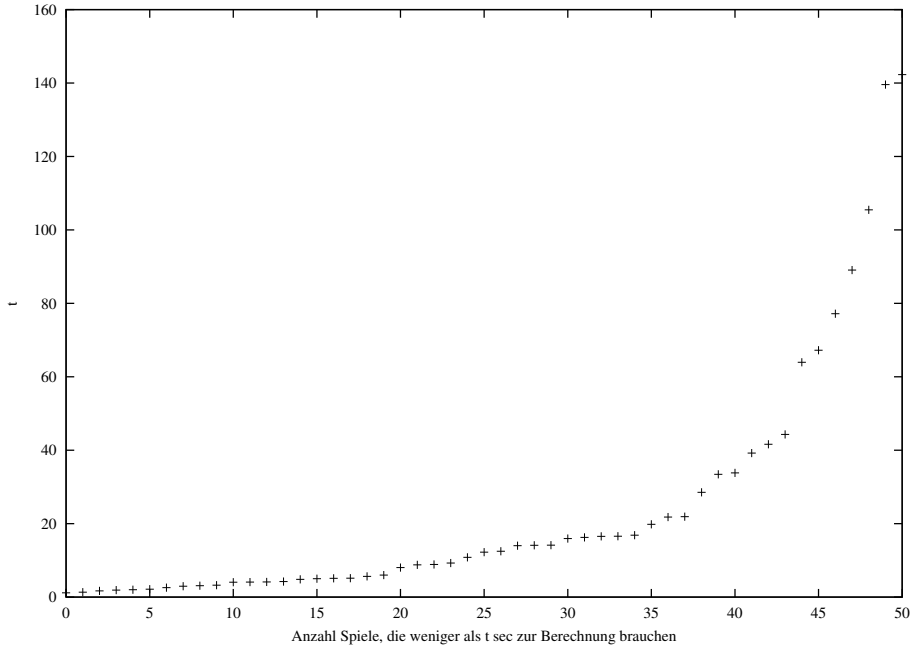


Abbildung 5.3: $\alpha\beta$ mit Transpositionstabelle (Zeiten)

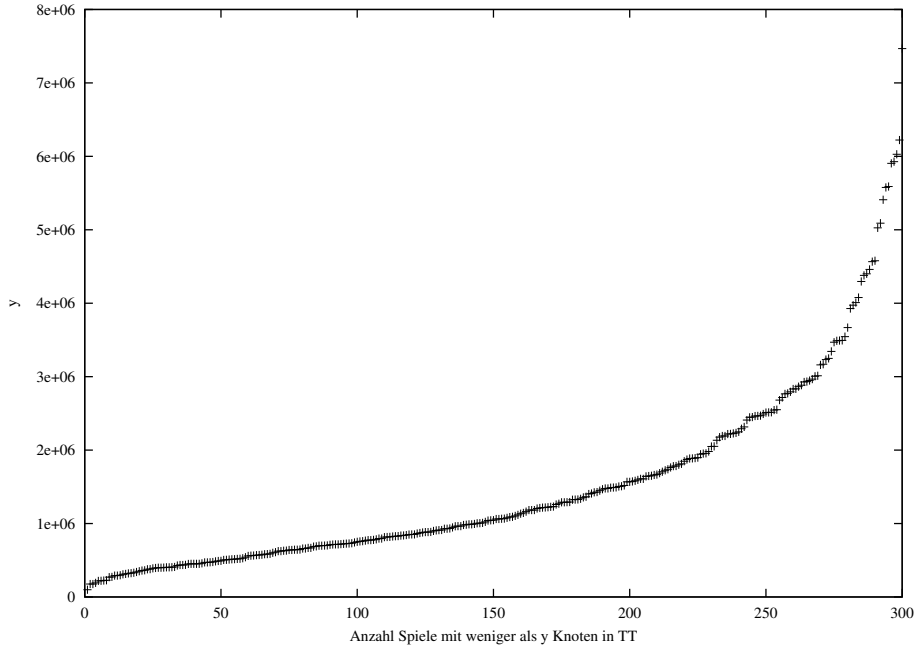


Abbildung 5.4: Zahl der Knoten in der Transpositionstabelle

Kapitel 6

Vorzeitiger Spielabbruch

Skatspiele werden i.A. nicht erst mit der letzten Karte entschieden. In den meisten Spielen erreicht eine Partei schon vor Spielende die nötigen 60 bzw. 61 Punkte. In diesem Kapitel wird der bisher entwickelte Algorithmus um einen Test erweitert, der sich dies zu Nutze macht. Der daraus resultierende Algorithmus ist in gewisser Weise mit *Minimal Window Search* oder auch *Null Windows Search* verwandt.

6.1 Minimal Window Search

Der $\alpha\beta$-Algorithmus durchsucht einen Spielbaum T mit einem anfänglichen Fenster $F = [-\infty, +\infty]$. Dieses Fenster garantiert, dass der spieltheoretische Wert der Wurzel p von T auch wirklich in F liegt. Die Suche kann aber effizienter gestaltet werden, indem das Suchfenster verkleinert wird. Das liegt daran, dass durch eine Verkleinerung des Suchfensters die Wahrscheinlichkeit wächst, dass Knoten beschnitten werden. Beim Skatspiel z.B. liegt der Wert eines Knotens immer in [0, 120].
Ein Fenster $F = [\alpha, \beta := \alpha + 1]$ ist ein sogenanntes *Zero Window* oder *Minimal Window*. Es ist ein kleinstmögliches Intervall mit $\alpha < \beta$, wenn man sich auf $\alpha, \beta \in \mathbb{N}$ beschränkt. Für das Skatspiel ist das keine Einschränkung, da hier jedes mögliche Ergebnis in $\mathbb{N}$ liegt.
Wählt man etwa das initiale Fenster $F = [60, 61]$, dann liefert ein Aufruf von $ab(p, 60, 61)$ genau dann einen Wert ≥ 61, wenn der Alleinspieler von p aus gewinnen kann. Analog verliert er, wenn das Ergebnis ≤ 60 ist. Ein Ergebnis zwischen 60 und 61 kann nie auftreten. Dadurch ist garantiert, dass man immer eine Aussage erhält. Diese Eigenschaft folgt sofort aus dem Satz über die Korrektheit des $\alpha\beta$-Algorithmus (siehe Kapitel 4).
Der Algorithmus, den man so aus $\alpha\beta$ erhält heißt *Minimal Window Search*.

6.2 Minimal Window Search verbessert

Minimal Window Search nutzt die Tatsache, dass ein Skatspiel schon vor Erreichen eines Blattes beendet sein kann, nicht aus. Ein Spiel kann abgebrochen werden, sobald eine der Parteien die nötigen 60 bzw. 61 Punkte erzielt hat. Erweitert man Minimal Window Search um diesen Test auf vorzeitigen Abbruch, dann erhält man Algorithmus *mws* aus Abbildung (6.1).

```
def mws(p, bound):
    if p.good_points > bound:
        return True
    elif p.bad_points >= 120 - bound:
        return False

    if p isa MAX_Node:
        result = False
    else:
        result = True

    for q in succ(p):
        succ_win = mws(q)
        if p isa MAX_Node:
            if succ_win == True:
                return True
        elif p isa MIN_Node:
            if succ_win == False:
                return False

    return result
```

Abbildung 6.1: *mws*-Algorithmus

Hierbei bezeichnet *p.good_points* die bisher erzielte Punktzahl des Alleinspielers in p und *p.bad_points* die der Gegenspieler. Ein Aufruf von z.B. $mws(p, 60)$ liefert dasselbe Ergebnis, wie ein Aufruf von $ab(p, 60, 61)$. In der Implementierung dieses Algorithmus fällt auf, dass es keine gesonderte Behandlung mehr für Blätter des Spielbaumes gibt. Wie sich herausstellen wird, ist das aber auch nicht notwendig.
Vor dem Beweis dieser Eigenschaft, wird zuerst die Korrektheit dieses Algorithmus bewiesen.

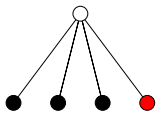
Satz 6.1 (Korrektheit des mws Algorithmus) *Sei p eine beliebige Position in einem Skatspiel, das kein Nullspiel ist, dann gilt*

$$mws(p) = \text{True}$$
$$\Longleftrightarrow$$
der Alleinspieler kann ausgehend von p einen Sieg erzwingen.

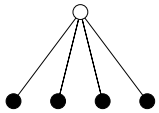
Beweis zu 6.1 *Eine Position p ist gewonnen, wenn der Alleinspieler in p mindestens 61 Punkte hat. Analog ist p verloren, wenn die Gegner mindestens 60 Punkte haben.*
Der Algorithmus (6.1) unterscheidet zwei Fälle mit je zwei Unterscheidungen. Die Bäume sollen dabei die Situation graphisch beschreiben. Ein roter Knoten entspricht dabei eine gewonnenen Position und ein schwarzer einer verlorenen.

1. *p ist ein MAX-Knoten*

   - *Wenn einer der Nachfolger von p eine Position ist, in der der Alleinspieler gewonnen hat, dann hat er auch in p gewonnen.*

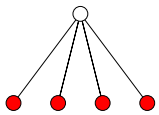


   - *Wenn alle Nachfolger von p verloren sind, dann ist auch p verloren.*

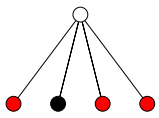


2. *p ist ein MIN-Knoten*

   - *Wenn alle Nachfolger von p Positionen sind, in denen der Alleinspieler gewonnen hat, dann hat er auch in p gewonnen.*



   - *Wenn ein Nachfolger von p eine verlorene Position ist, dann ist auch p eine verlorenen Position.*



Das sind alle Fälle, die auftreten können. Daraus folgt die Behauptung. Die totale Korrektheit des Algorithmus ergibt sich daraus, dass auf jedem Pfad von der Wurzel zu einem Blatt eine Position erreicht wird, in der eine Partei mindestens 60 bzw. 61 Punkte hat.

□

Satz 6.2 *Sei* $\delta \in \{0, \ldots, 120\}$ *und sei* p *ein beliebiger Spielstand in einem Skatspiel, dann gilt:*

$$mws(p, \delta) = True \iff ab(p, \delta, \delta + 1) \geq \delta + 1$$

Beweis zu 6.2

"$\Rightarrow$"

$$\begin{aligned} mws(p, \delta) = True \Rightarrow & \textit{der Alleinspieler kann ausgehend von } p \textit{ eine} \\ & \textit{Position erreichen mit } p.good\_points \geq \delta + 1 \\ \Rightarrow & v(p) \geq \delta + 1 \\ \Rightarrow & ab(p, \delta, \delta + 1) \geq \delta + 1 \end{aligned}$$

"$\Leftarrow$" *In jedem Spielstand* p *gilt:*

$$p.good\_points + p.bad\_points + p.rest = 120$$

hierbei bezeichnet $p.rest$ *die Punkte, die in* p *noch nicht vergeben sind.*

$$\begin{aligned} mws(p, \delta) = False \Rightarrow & \textit{Der Alleinspieler kann ausgehend von } p \textit{ keine} \\ & \textit{Position erreichen mit } p.good\_points \geq \delta + 1 \\ \Rightarrow & \textit{Der Alleinspieler erreicht nur Positionen } p \textit{ mit} \\ & p.good\_points \leq \delta \\ \Rightarrow & v(p) \leq \delta \\ \Rightarrow & ab(p, \delta, \delta + 1) \leq \delta \end{aligned}$$

□

6.3 *mws* mit Transpositionstabelle

Auch dieser Algorithmus läßt sich um Transpositionstabellen erweitern. Dazu werden für jeden besuchten Knoten p eine obere (max_value) und eine untere Schranke (min_value) in die Hashtabelle geschrieben. min_value gibt

an, dass der Alleinspieler ausgehend von p noch mindestens min_value viele Punkte erzielt. max_value gibt entsprechend das Maximum der von p ausgehend erreichbaren Punkte an. Dadurch ergibt sich der Algorithmus mws_tt aus Abbildung (6.2).
Hierbei wird beim Aufruf von `hash.lookup(p, min_value, max_value)` (für den Fall dass p in der Hashtabelle ist) min_value eine untere und max_value eine obere Schranke für $m(p)$ zugewiesen. Ansonsten wird min_value auf 0 und max_value auf die Zahl der noch erzielbaren Punkten ($120 - p.good\_points - p.bad\_points$) gesetzt.
Die Korrektheit für den Einsatz der Transpositionstabelle ergibt sich daraus, dass zu jedem Zeitpunkt min_value die Anzahl der Punkte ist, die der Alleinspieler mindestens noch erzielen wird und max_value die Anzahl der Punkte ist, die er höchstens noch erreichen kann.
Die Abbildungen (6.3) und (6.4) zeigen das Ergebnis einer Wiederholung des Tests aus Kapitel 5. Zum Vergleich werden die Meßergebnisse von $\alpha\beta\_tt$ noch einmal dargestellt.

```
def mws_tt(p, bound):
    if p.good_points > bound:
        return true
    if p.bad_points >= 120 - bound:
        return false

    if hash.lookup(p, min_value, max_value):
        if p.good_points + min_value > bound:
            return true
        if p.good_points + max_value <= bound:
            return false

    if p isa MAX_Node:
        result = false
    else:
        result = true

    for q in succ(p):
        succ_win = mws_tt(q, bound)

        if p isa MAX_Node:
            if succ_win:
                result = True
                break
        else:
            if not succ_win:
                result = False
                break

    if result:
        min_value = bound + 1 - p.good_points
    else:
        max_value = bound - p.good_points

    hash.add(p, min_value, max_value)
    return result
```

Abbildung 6.2: *mws*-Algorithmus mit Transpositionstabelle

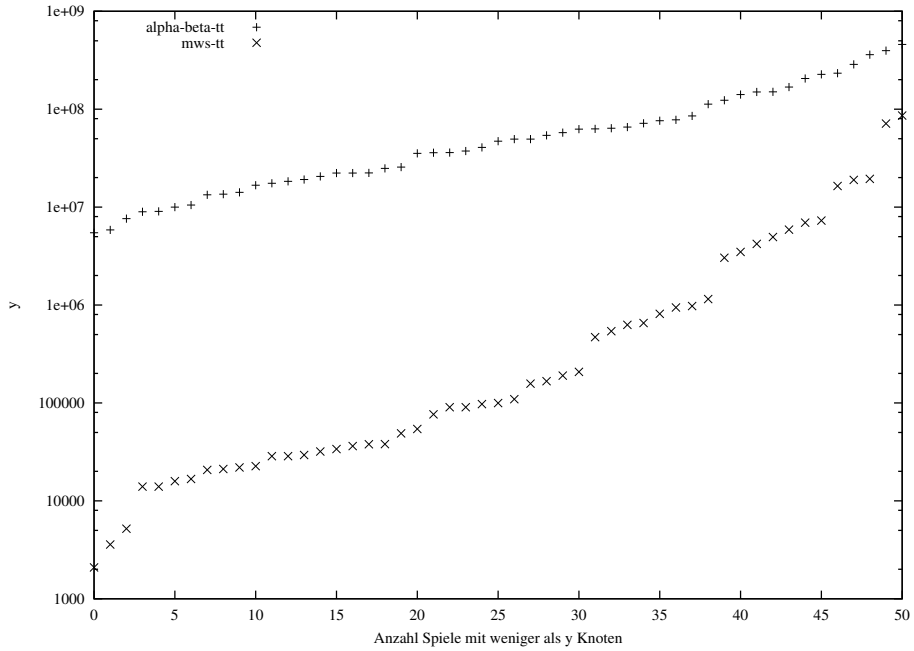


Abbildung 6.3: Vergleich von $\alpha\beta\_tt$ und mws_tt (logarithmische Skala)

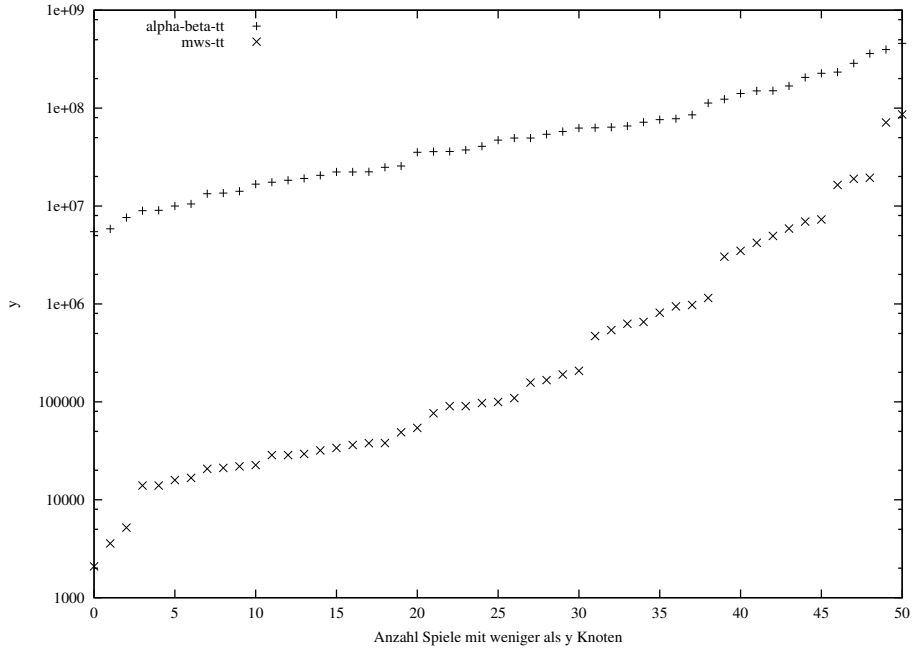


Abbildung 6.4: Laufzeiten von $\alpha\beta\_tt$ und mws_tt (logarithmische Skala)

Kapitel 7

Äquivalenzklassen

Die bisher betrachteten Algorithmen berechnen immer nur den Wert für eine Spielposition. Die Werte solcher Positionen werden zwar in der Transpositionstabelle gespeichert, doch kann man diese Ergebnisse nur dann wiederverwerten, wenn man bei der Suche genau diese Positionen ein zweites Mal analysiert.
Es läßt sich allerdings zeigen, dass man mit diesen Ergebnisse den Wert für andere, noch nicht besuchte Positionen nach oben und unten abschätzen kann. Dadurch wird die Suche erheblich beschleunigt. Solche Situationen treten z.B. auf, wenn ein Spieler in einer Position $\diamondsuit$ bedienen muss. Nun kann er sich u.a. für $\diamondsuit 8$ oder $\diamondsuit 9$ entscheiden. Die bisherigen Algorithmen hätten dann zuerst den Wert für $\diamondsuit 8$ und dann für $\diamondsuit 9$ ausgerechnet. Mit Hilfe des Satzes über Äquivalenzklassen, der in diesem Kapitel aufgestellt und bewiesen wird, wird eine dieser beiden Berechnungen überflüssig, da hier genau der gleiche Wert erzielt wird.
Bevor nun dieser Mechanismus genauer erklärt wird, bedarf es einiger Definitionen.

Definition 7.1 *Auf Spielkarten läßt sich wie folgt eine partielle Ordnung definieren: Haben zwei Karten c und c' die gleiche Farbe, dann gilt $c \prec c'$ falls c von c' gestochen werden kann. Eine Karte c kann von einer Karte c' gestochen werden, falls c' die gleich Farbe hat wie c, oder Trumpf ist und nach den Regeln des aktuellen Skatspiels die höhere Karte ist.*

Das ist im Prinzip die natürliche Ordnung, die durch die Regeln des Skatspiels induziert wird.

Definition 7.2 *Sei C eine Menge von Karten und die disjunkte Vereinigung von C_1 und C_2. Eine Äquivalenzklasse $E_{C_1}^{C_2}(c)$ auf C, ist die größte Teilmenge von C_1 mit folgender Eigenschaften:*

- $c \in E_{C_1}^{C_2}(c)$
- $\forall c' \in E_{C_1}^{C_2}(c)$ *gilt: c' hat dieselbe Farbe wie c und für jede Karte $d \in C_2$ mit derselben Farbe wie c und c' gilt: $c \prec d \iff c' \prec d$*

Zwei Karten c und c' sind äquivalent bezüglich der Mengen C_1 und C_2, in Zeichen $c \overset{C_2}{\underset{C_1}{\sim}} c'$, falls $c' \in E_{C_1}^{C_2}(c)$. Da diese Äquivalenzklassen repräsentantenunabhängig sind, ist das gleichbedeutend mit $c \in E_{C_1}^{C_2}(c')$.

Diese Definition soll an folgendem Beispiel illustriert werden.

Beispiel 7.1 Sei $C_1 = \{\diamondsuit 8, \diamondsuit Q, \diamondsuit A\}$ und $C_2 = \{\diamondsuit 7, \diamondsuit K, \diamondsuit 10\}$. Die Äquivalenzklasse $E_{C_1}^{C_2}(\diamondsuit Q)$ ergibt sich zu $\{\diamondsuit Q, \diamondsuit 8\}$. Beide Karten, $\diamondsuit 8$ und $\diamondsuit Q$, können die gleichen Karten aus C_2 stechen, nämlich $\diamondsuit 7$. Von den restlichen Karten aus C_2 werden beide Karten gestochen. Das ist auch die größte Menge dieser Art, denn $\diamondsuit A$ kann von keiner Karte aus C_2 gestochen werden.

Der spieltheoretische Wert für einen Repräsentanten einer Äquivalenzklasse E liefert zusammen mit dem nächsten Satz eine obere und untere Schranke für die Werte der anderen Karten aus E.
In diesem Kapitel wird oft nicht zwischen einzelnen Karten und Positionen unterschieden. Allerdings ist das auch nicht notwendig, denn beide Begriffe bezeichnen hier das gleiche: Eine Karte c bezeichnet gleichzeitig eine Position p, wenn man weiß in welcher Position q sie ausgespielt wird. Hierbei ist p der Nachfolger von q, in den man durch Ausspielen von c gelangt.

7.1 Satz über Äquivalenzklassen

Skat kann auch als 2-Personenspiel aufgefaßt werden. Hierbei ist Spieler 1 der Alleinspieler und Spieler 2 wird von den beiden Gegnern gebildet. Im Folgenden sei $G = (X, Y, A)$ ein Skatspiel in strategischer Form im Sinne der Spieltheorie [7] mit beliebiger, aber fester Kartenverteilung. Das Spiel sei ein Farb- oder Grandspiel. Hierbei sei X die Menge der Strategien für Spieler 1 und Y die Menge der Strategien für Spieler 2. Eine Strategie $x \in X$ ist eine Funktion $x : P \to C$. Hierbei ist P die Menge aller Spielpositionen, und C ist die Menge der Spielkarten. Analog ist $y \in Y$ definiert. Sei weiter $A : X \times Y \to \mathbb{R}$ die Bewertungsfunktion. A gibt den Wert des Spiels aus Sicht

von Spieler 1 an, d.h. $A(x, y) = r$ bedeutet, dass wenn Spieler 1 mit x und Spieler 2 mit y spielt, Spieler 1 r Punkte erzielt. Wieviele Punkte Spieler 2 erreicht kann man sich leicht ausrechnen, da immer 120 Punkte im Spiel sind.

Satz 7.1 (Satz über Äquivalenzklassen) *Sei p eine beliebige Position in einem Skatspiel. C_1 bezeichne die Menge der Karten, die Spieler 1 in p auf der Hand hält und C_2 die Menge der restlichen Karten, die in p noch im Spiel sind, einschließlich der Karten im aktuellen Stich. Spieler 1 habe die Karten c und c' mit $c \overset{C_2}{\underset{C_1}{\sim}} c'$ auf der Hand. Sei weiter $v(p, c) = \max\limits_{\substack{x \in X \\ x(p)=c}} \min\limits_{y \in Y} A(x, y)$ Dann gilt:*

$$|v(p, c) - v(p, c')| \leq d(c, c')$$

Hierbei ist $d(c, c') = |val(c) - val(c')|$ und $val(c)$ ist der Wert der Karte z.B. $val(\clubsuit K) = 4$.

Anschaulich ist $v(p, c)$ der beste Wert, den Spieler 1 erreichen kann, wenn Spieler 1 in p die Karte c spielt. Für den Beweis dieses Satzes werden folgende Lemmata verwendet.

Lemma 7.1 *Sei $G = (X, Y, A)$ ein Skatspiel in strategischer Form, in dem kein Nullspiel gespielt wird, mit beliebiger aber fester Kartenverteilung. C_i bezeichne die Karten von Spieler $i \in \{1, 2\}$ einschließlich der Karten im aktuellen Stich. In dieser Kartenverteilung habe Spieler 1 die beiden Karten c und c' mit $c \overset{C_2}{\underset{C_1}{\sim}} c'$ auf der Hand. Sei $G' = (X, Y, A')$ das Spiel, das man erhält, wenn man in G die Werte der beiden Karten c und c' vertauscht (es werden nur die Werte vertauscht, wenn aber vor der Vertauschung $c \prec c'$ galt, dann gilt das auch nach der Vertauschung). Sei f die Funktion, die $x(p)$ auf $x(p')$ abbildet. Hierbei ist p' diejenige Position, die man erhält, wenn man in p die Karten c und c' miteinander vertauscht. Sei weiter*

$$g(x)(p) = \begin{cases} c' & falls\ x(p) = c \\ c & falls\ x(p) = c' \\ x(p) & sonst \end{cases}$$

Dann gilt:

$$A((g \circ f)(x), f(y)) = A'(x, y) \quad \forall x \in X, \forall y \in Y$$

Beweis zu Lemma 7.1 *Ohne Einschränkung spielt Spieler 1 in G' c vor c' aus. Spieler 1 spiele G' mit Strategie x und G mit $(g \circ f)(x)$. Spieler 2 spielt G' mit y und G mit $f(y)$ (Abbildung (7.1)). Sei p diejenige Position, in der Spieler 1 c ausspielt. Bis zu dieser Position verlaufen die Stiche in beiden Spielen gleich.*
In Spiel G' wird c ausgespielt. Daraus ergibt sich die Position p_1. Werden in p_1 die Karten c und c' vertauscht, so erhält man Position p'_1. Diese Position wird auch erreicht, wenn Spieler 1 in G die Karte c' spielt. In beiden Spielen wird dieser Stich vom gleichen Spieler gewonnen. Danach ist die Situation in beiden Spielen wieder gleich. D.h. in beiden Spielen eröffnet der gleiche Spieler den nächsten Stich und auch die erzielten Punkte der Spieler sind identisch.
Die Stiche, an denen die Karte c' nicht beteiligt ist, verlaufen in G und G' parallel. Sei nun p_n die Position, in der Spieler 1 im Spiel G' die Karte c' spielt. Durch Vertauschen von c' mit c erhält man Position p'_n. Auch hier gewinnt derjenige Spieler den Stich mit c in G, der den Stich mit c' in G' gewinnt. Der Rest des Spiels G ist dann identisch mit dem Rest von G'.

□

Anschaulich bedeutet das, dass beide Spieler so spielen, als gäbe es die Wertevertauschung in G' gar nicht. Hierbei beeinflußt f die Wahrnehmung der Spieler und g die Aktionen, die Spieler 1 ausführt. Die Spieler reagieren also auf eine Situation, in der c gespielt wurde, so, als ob c' gespielte wurde und umgekehrt. Wegen der Äquivalenz von c und c' wird somit ein legaler Zug ausgeführt. Zudem wird dieser Stich in beiden Spielen vom gleichen Spieler gewonnen.
Da die Werte beider Karten zweimal vertauscht wurden, nämlich in p und in q, erhält man am Ende des Spiels natürlich in beiden Spielen das gleiche Ergebnis.

Lemma 7.2 *Unter den gleichen Voraussetzungen wie im letzten Lemma gilt:*

$$|A'(x, y) - A(x, y)| \leq d(c, c') \quad \forall x \in X, \forall y \in Y$$

Beweis zu Lemma 7.2 *Das ist sehr leicht zu sehen. Das Spiel G' verläuft genau wie G nur mit der Tatsache, dass die Werte der Karten c und c' vertauscht worden sind. Sei $val(c) > val(c')$, dann gibt es folgende drei Fälle:*

1. *Die beiden Stiche mit c und c' vom gleiche Spieler gewonnen. Hier ändert das Vertauschen der Werte nichts an der Punktzahl der einzelnen Spieler.*

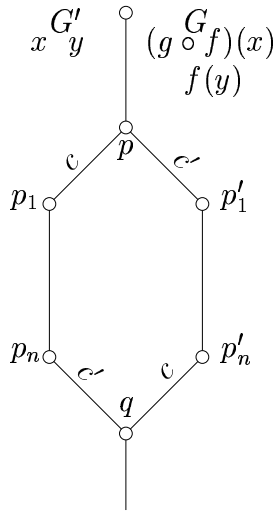


Abbildung 7.1: Spielverlauf

2. *Wenn Spieler 1 den Stich mit c gemacht hat, dann wurden ihm $d(c, c')$ Punkte zuwenig angerechnet.*

3. *Analog, wenn Spieler 1 den Stich mit c' gemacht hat, werden im $d(c, c')$ zuviel angerechnet.*

□

Beweis zu 7.1 (Satz über Äquivalenzklassen) *Sei $\tilde{x}$ optimale Strategie in G für Spieler 1 unter den Strategien, die in p die Karte c spielen, und $\tilde{x}'$ optimale Strategie in G für Spieler 1 unter den Strategien, die in p die Karte c' spielen. Sei ferner $\tilde{y}$ optimale Strategie in G für Spieler 2. Nach dem* Satz von Kuhn *(siehe etwa [7] S. 99) gibt es solche Strategien (d.h. Strategien, die auf alle Gegenstrategien beste Antwort sind), da G ein Nullsummenspiel mit vollständiger Information ist.*
Dann gilt:

$$\begin{aligned} |v(p, c) - v(p, c')| &= |\max_{\substack{x \in X \\ x(p)=c}} \min_{y \in Y} A(x, y) - \max_{\substack{x \in X \\ x(p)=c'}} \min_{y \in Y} A(x, y)| \\ &= |A(\tilde{x}, \tilde{y}) - A(\tilde{x}', \tilde{y})| \end{aligned}$$

Annahme:

$$|A(\tilde{x}, \tilde{y}) - A(\tilde{x}', \tilde{y})| > d(c, c')$$

Ohne Einschränkung sei $A(\tilde{x},\, \tilde{y}) \geq A(\tilde{x}',\, \tilde{y})$*, der Beweis würde ansonsten analog verlaufen. Somit gilt:*

$$\begin{aligned} & A(\tilde{x},\, \tilde{y}) - A(\tilde{x}',\, \tilde{y}) > d(c,\, c') \\ \Rightarrow & A(\tilde{x},\, \tilde{y}) < A(\tilde{x}',\, \tilde{y}) - d(c,\, c') \end{aligned} \qquad (*)$$

Betrachte die Strategie

$$\bar{x} := (g \circ f)(\tilde{x})$$

$$\begin{aligned} A(\bar{x},\, \tilde{y}) &= A((g \circ f)(\tilde{x}),\, \tilde{y}) \\ &\geq A((g \circ f)(\tilde{x}), f(\tilde{y})) && \textit{($\tilde{y}$ optimal)} \\ &= A'(\tilde{x},\, \tilde{y}) && \textit{(Lemma 6.1)} \\ &\geq A(\tilde{x},\, \tilde{y}) - d(c,\, c') && \textit{(Lemma 6.2)} \end{aligned}$$

Mit $(*)$ *folgt* $A(\bar{x},\, \tilde{y}) > A(\tilde{x}',\, \tilde{y})$ *im Widerspruch zur Definition von* $\tilde{x}'$*, da* $\bar{x}(p) = c'$*.*

□

Mithilfe dieses Satzes lassen sich die bisher betrachteten Algorithmen weiter beschleunigen. Statt für jede Karte den spieltheoretischen Wert auszurechnen, kann man somit auf schon errechnete Werte zurückgreifen und Abschätzungen vornehmen.

7.2 Anwendung des Satzes

Sei p ein MAX-Knoten in einem Spielbaum und $v(p,\, c) \leq 60$. D.h. der Alleinspieler hat in Position p verloren, wenn er dort die Karte c ausspielt. Spielt er in p stattdessen die Karte c', dann gilt im günstigsten Fall $v(p,\, c') = v(p,\, c) + d(c,\, c')$. Es genügt also zu prüfen, ob $v(p,\, c) + d(c,\, c') \leq 60$ ist. Wenn dies der Fall ist, dann verliert der Alleinspieler auch beim Ausspielen der Karte c' in p. Folglich ist es nicht mehr notwendig, diesen Teil des Spielbaumes zu durchsuchen.
Analog kann man in einem MIN-Knoten verfahren. Hier werden dann im Gegensatz zum MAX-Knoten nicht die Verlierer, sondern die Gewinner eliminiert.
In der Implementierung des Double Dummy Solvers wird dieser Satz nur auf Karten c und c' angewendet mit $d(c,\, c') = 0$. D.h. es kommen nur die vier Buben, alle 7er, 8er und 9er in Betracht. Erste Versuche mit allgemeineren Äquivalenzen zeigten nicht den gewünschten Effekt.

7.3 Ergebnisse

Erweitert man Algorithmus *mws_tt* aus Kapitel 6 um den im letzten Abschnitt vorgestellten Test, dann erhält man Algorithmus *mws_equiv*. Dieser Algorithmus evaluiert wesentlich weniger Knoten als *mws_tt*. Das hat zur Folge, dass die Ausführungszeit weiter sinkt.
Die Abbildungen (7.2) und (7.3) visualisieren die Ergebnisse eines Tests aus 1.000 zufälligen Spielen. Folgende Tabelle zeigt die Durchschnittswerte der beiden Algorithmen.

Algorithmus	**durchschnittliche #Knoten pro Spiel**	**durchschnittliche Zeit pro Spiel**
mws_tt	2.491.071	2,38s
mws_equiv	1.090.127	1,40s

Die Ausnutzung der Äquivalenzklassen reduziert die Anzahl der zu betrachtenden Knoten im Vergleich zu *mws*. Trotz einer Ersparnis von über 56% bei den Knoten, sinkt die Ausführungszeit nicht in gleichem Maß. Das liegt daran, dass in diesem Fall zusätzlich noch die Äquivalenzklassen berechnet werden müssen. I.A. ist eine Erweiterung immer zweischneidig: zwar reduziert sie die Zahl der zu betrachtenden Knoten, sie benötigt aber zusätzliche Rechenzeit. Der Einsatz einer Erweiterung ist nur dann gerechtfertigt, wenn dadurch die Ausführungszeit insgesamt sinkt.

7.4 Partition Search

Die Idee hinter dem Satz über Äquivalenzklassen geht auf *Partition Search* von M. Ginsberg [3] zurück. Im Prinzip ist Partition Search ein $\alpha\beta$-Algorithmus mit Transpositionstabelle. Der Unterschied besteht darin, dass in der Transpositionstabelle nicht einzelne Positionen, sondern Mengen von Positionen gespeichert werden, die im Sinne der Klassen $E_{C_1}^{C_2}$ äquivalent sind. Positionen, die in solch einer Menge gespeichert werden, haben den gleichen Minimax-Wert.
Partition Search überprüft zuerst für einen Knoten p, ob in der Hashtabelle eine Menge S existiert, mit $p \in S$. Wenn das der Fall ist, so wird die Bewertung für Positionen aus S zurückgeliefert. Für den Fall, dass keine solche Menge in der Hashtabelle existiert, prüft der Algorithmus, ob es einen Zug gibt, der in einen Zustand führt, der in der Tabelle gespeichert ist.
Ginsberg beobachtete durch den Einsatz dieser Technik einen Effizienzgewinn, der mit dem von $\alpha\beta$ gegenüber Minimax vergleichbar ist.

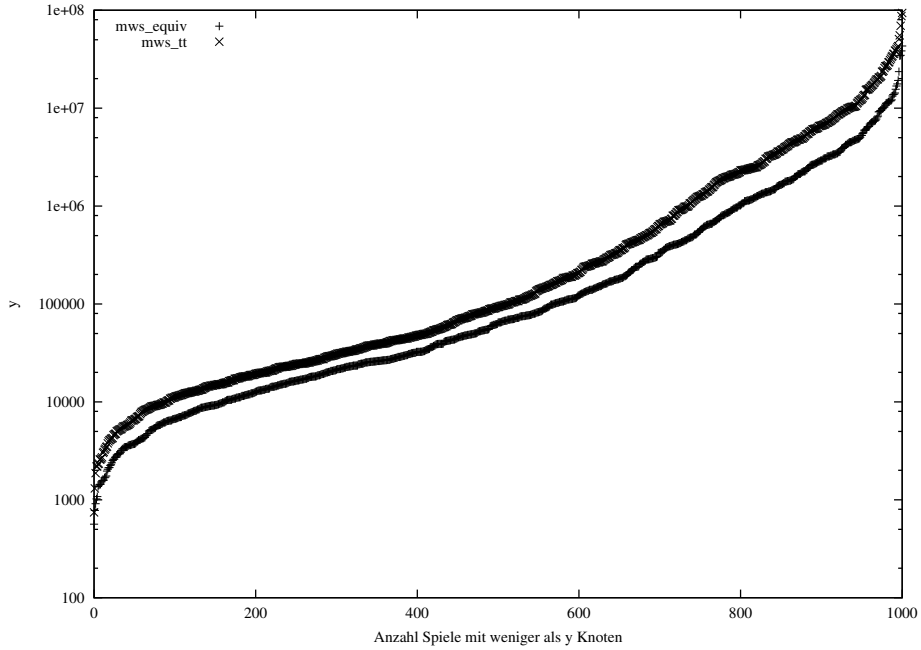


Abbildung 7.2: Evaluierte Knoten

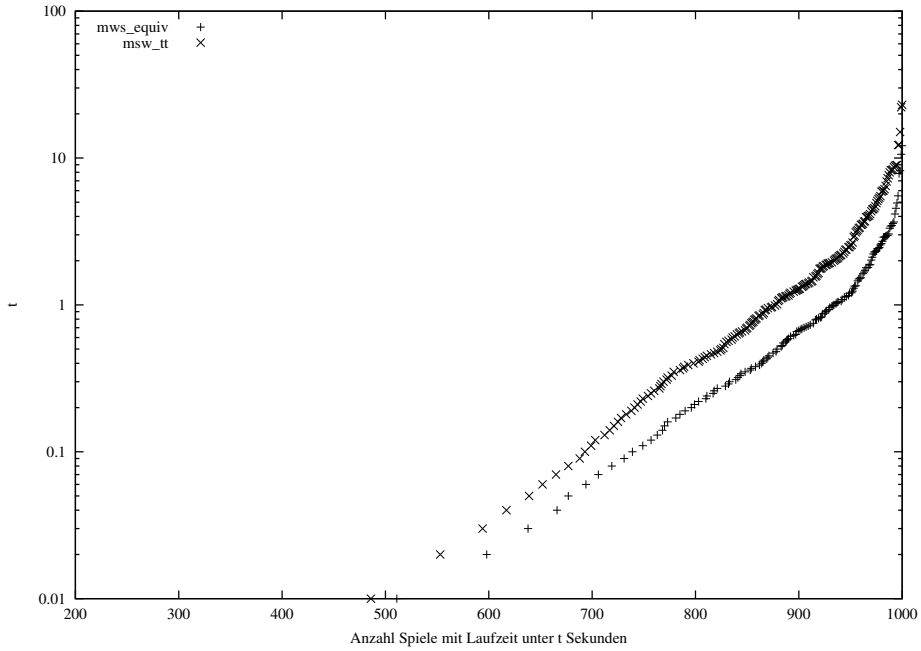


Abbildung 7.3: Laufzeiten

Kapitel 8

Memory Test Driver

In den beiden letzten Kapiteln hat unter anderem ein Verzicht auf die exakten numerischen Werte bei der Berechnung von Spielen zu einer Performance-Steigerung beigetragen.
Der offensichtliche Nachteil dieser Herangehensweise betrifft die Qualität der so erhaltenen Informationen. Diese geben zwar an, mit welchen Karten man gewinnen kann, aber nicht, wie viele Punkte durch das Ausspielen einer Karte erzielt werden können. Kann in einer Position mit mehreren Karten gewonnen werden, dann ist unklar mit welcher bzw. welchen dieser Karten die höchste Punktzahl erzielt werden kann. Folglich ist nicht garantiert, dass man mit diesem Algorithmus optimal spielt, z.B. wenn man die "Schneider"-Regel berücksichtigt.
Dieses Problem läßt sich mit einigen Änderungen des zuletzt vorgestellten Algorithmus lösen. Zunächst wird ein allgemeines Schema zur Lösung dieses Problems präsentiert und anschließend auf den Algorithmus aus dem letzten Kapitel eingegangen.

8.1 MTD(f)

MTD ist ein Framework, in dem Algorithmen wie z.B. $\alpha\beta\_tt$ mit *Zero Window*, den spieltheoretischen Wert einer Position p durch wiederholtes Suchen berechnen.
Die einfache Gestalt dieses Frameworks zusammen mit $\alpha\beta\_tt$ ist in Abbildung (8.1) zu sehen. Da in jeder Iteration das gleiche Spiel berechnet wird, kann man bei geeigneter Implementierung von $\alpha\beta\_tt$ auf bereits berechnete Ergebnisse in der Hashtabelle zurückgreifen. Hierdurch kann der spieltheoretische Wert von p in kürzerer Zeit berechnet werden, als das mit $\alpha\beta\_tt$ möglich wäre.

Der Name MTD(f) geht auf J. Pearls Test-Prozedur zurück [8]. Plaat, Schaeffer et. al. [10] entwickelten daraus MTD (Memory-enhanced Test Driver).
Der Algorithmus *mws_equiv* ist für den Einsatz mit MTD geeignet. In die Hashtabelle werden von *mws_equiv* wie angesprochen obere und untere Schranken für die Punkte, die der Alleinspieler noch erzielen wird geschrieben. Diese Schranken sind unabhängig davon, ob der Alleinspieler ein Spiel mit mehr als 60 oder z.B. erst mit mehr als 90 Punkten gewonnen hat. Eine Möglichkeit, *mws_equiv*(p, *bound*) in dieses Framework einzubinden, wird in Abbildung (8.2) gezeigt. Dabei wird *bound* in jedem Schritt herabgezählt, bis *mws_equiv*(p, *bound*) = $True$ gilt. Im Fall des Skatspiels kann man *bound* anfangs auf 120 setzten.

8.2 Der beste Wert für *bound*

Die Suche nach dem Wert der Wurzel kann auf verschiedene Arten angegangen werden. Die Herangehensweise, in der *bound* in jeder Iteration verkleinert wird, wird in [10] mit MTD($+\infty$) bezeichnet. Alternativ kann man sich aber auch von unten an den Wert der Wurzel annähern. Dieser Ansatz wird mit MTD($-\infty$) bezeichnet. Man könnte auch eine binäre Suche verwenden, die Schrittgröße, in der *bound* verändert wird, variieren etc.
In gewisser Weise ist MTD($+\infty$) optimistisch, da ein hoher Wert erwartet wird, der den Algorithmus schneller abbricht. MTD($-\infty$) kann so als pessimistisch bezeichnet werden. Einen realistischen Ansatz bildet MTD(f). Hier versucht man *bound* $\approx$ *minimax*(p) zu wählen. Exemplarisch werden hier diese drei Ansätze miteinander verglichen.
In einem Test werden 1.000 zufällig erzeugte Grand- und Farbspiele gespielt. In jedem Spiel wird zufällig eine Spielfarbe vorgegeben und der Skat wird festgelegt. Spieler 0 spielt immer allein und eröffnet jedes Spiel. Die Abbildungen (8.3) und (8.4) zeigen die Ergebnisse dieses Tests. Sie zeigen, dass im gegebenen Fall der Ansatz MTD($+\infty$) dem Ansatz MTD($-\infty$) vorzuziehen ist. Die Ausführungszeit ist gegenüber MTD($-\infty$) 4,5-mal so schnell. Bleibt noch die Frage, ob die optimistische Herangehensweise besser als die realistische ist. Zu diesem Zweck wurde mit der exakten, spieltheoretischen Punktzahl gerechnet. Die realistische Suche ist 2,5-mal schneller als die optimistische. Das ist auch nicht verwunderlich, da hier die wenigsten verschiedenen Werte von der Wurzel bis zum spieltheoretischen Wert auftauchen. Somit ist in diesem Fall die schnellste Konvergenz zu erwarten. Da eine gute Vorhersage des zu erwartenden Wertes schwierig ist, wurde in der Implementierung des Double Dummy Solvers der optimistische Zugang gewählt.

8.3 Interpretation der Arbeitsweise von MTD($+\infty$)

Man könnte die Vorgehensweise von MTD($+\infty$) mit iterierter Tiefensuche (IDS) vergleichen. Der Spielbaum wird immer nur bis zu einer vorgegebenen Schranke durchsucht, ist die Suche erfolglos, dann wird die Schranke herabgesetzt. Auf diese Art und Weise wird dann ein größerer Teil des Baumes durchsucht. Die vorhandenen Einträge in der Hashtabelle dienen dabei als Wegweiser für die Suche.

```
def mtd(p, f):
    g = f
    upper_bound = +infinity
    lower_bound = -infinity
    while lower_bound < upper_bound:
        if g == lower_bound:
            beta = g + 1
        else:
            beta = g
        g = ab_tt(p, beta - 1, beta)
        if g < beta:
            upper_bound = g
        else:
            lower_bound = g
    return g

def mtd_mws(p, f):
    g = f
    upper_bound = 120
    lower_bound = 0
    while lower_bound < upper_bound:
        if g == lower_bound:
            bound = g + 1
        else:
            bound = g

        if mws(p, bound):
            g += 1
            lower_bound = g
        else:
            g -= 1
            upper_bound = g
    return g
```

Abbildung 8.1: MTD(f)

```
def mtd_mws(p, bound):
    while mws_equiv(p, bound) == False:
        bound -= 1
    return bound
```

Abbildung 8.2: MTD mit *mws_equiv*

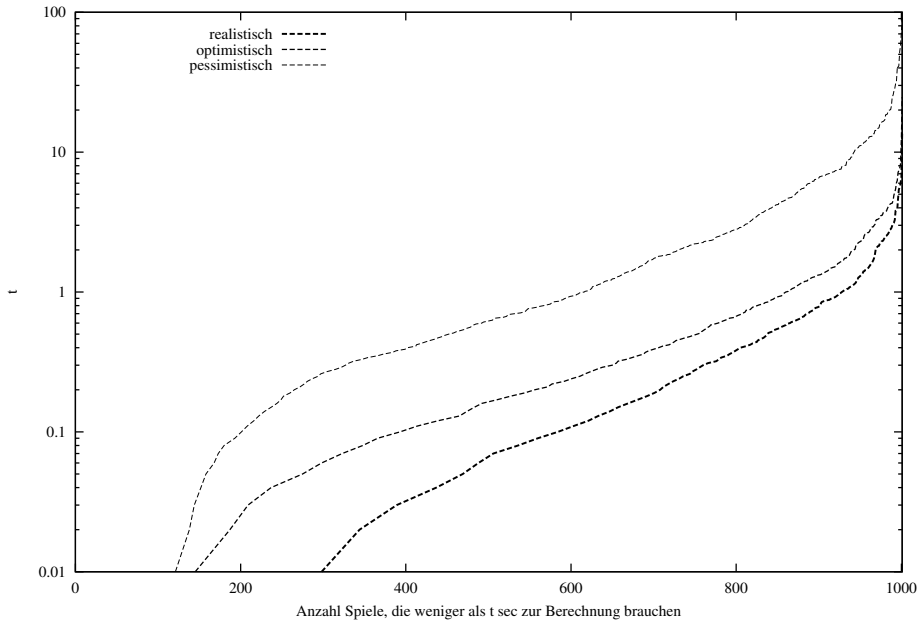


Abbildung 8.3: Laufzeiten MTD(f) (logarithmische Skala)

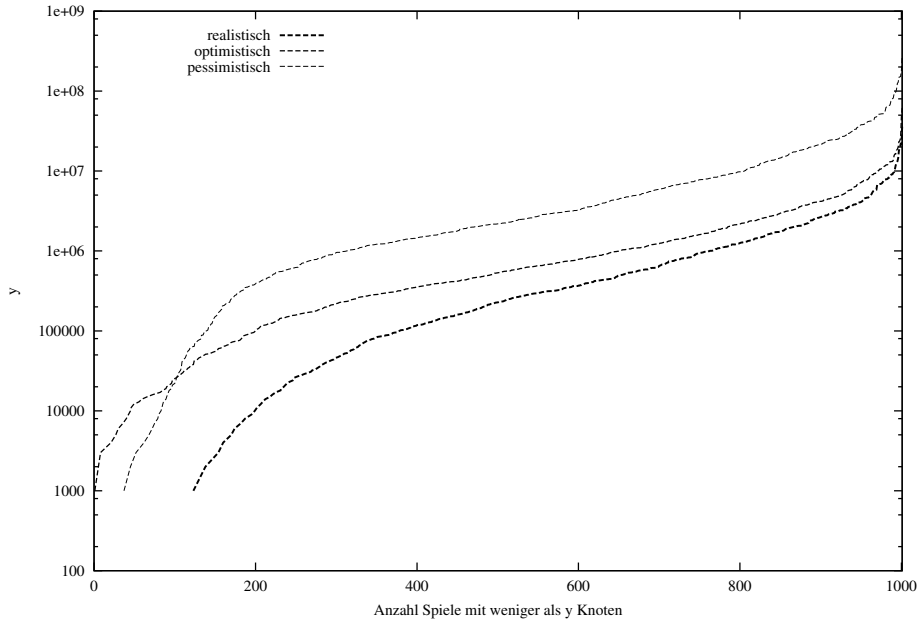


Abbildung 8.4: Knoten MTD(f) (logarithmische Skala)

Kapitel 9

Sichere Stiche

In bestimmten Situationen kann man Aussagen über den Ausgang des Spiels treffen, bevor eine Partei die nötigen 60 bzw. 61 Punkte für den Sieg erzielt hat.
Hat der Alleinspieler in der Spielposition p z.B. die höchsten Trümpfe auf der Hand, dann wird er damit jeden Stich gewinnen. Kann man nun abschätzen, wie viele Punkte s dadurch mindestens erzielt werden können, dann hat der Alleinspieler in p auch dann gewonnen, wenn $p.goodPoints + s \geq 61$. Analog natürlich auch für die Gegner.
In diesem Kapitel wird ein Mechanismus beschrieben, mit dem man diese "sicheren" Stiche nach unten abschätzen kann.

9.1 Der Alleinspieler

Die Vorgehensweise ist wie folgt: in einer Position p, in der der Alleinspieler (Spieler 0) den Stich eröffnet spielt er wie in Abbildung (9.1) beschrieben.

```
def one_vs_two():
    while Spieler 0 besitzt höchsten Trumpf:
        Spieler 0 spielt höchsten Trumpf

    for each suit s:
        while Alleinspieler hat höchste Karte von s and
              keiner der Gegner, der s nicht hat, besitzt Trumpf:
            spiele diese Karte
```

Abbildung 9.1: Einer gegen Zwei

Die Trümpfe werden deshalb als erstes ausgespielt, weil dadurch u.U. auf Grund des Bedienzwanges erreicht werden kann, dass bei den anschließenden

Farbstichen die Gegner nicht mehr mit Trumpf stechen können.
Es werden nur Stiche gezählt, die der Alleinspieler ausgehend von p gewinnt, ohne dass ein Gegner davon einen Stich bekommt. Wenn man dieser forcierenden Spielweise eine optimale Strategie der Gegner entgegensetzt, dann erhält man eine untere Schranke für die Punkte, die der Alleinspieler im Restspiel noch erreichen wird. Die Strategie für Gegner $i \in \{1, 2\}$ sei folgende:

```
def ans(i):
    if Spieler i kann Farbe bekennen:
        lege niedrigste Karte dieser Farbe
    else:
        count_i += 1
```

Abbildung 9.2: Spielweise der Gegner

Diese Regeln genügen, da das Spiel vorbei ist, wenn Spieler 0 keinen Stich mehr gewinnen kann. I.A. ist nicht klar, welche Karte abgeworfen werden muss, wenn die Farbe nicht bekannt werden kann. Aus diesem Grund wird statt dem Ausspielen solch einer Karte für jeden Gegner gezählt, wie oft er nicht bekennen konnte. Wenn Spieler 0 keinen Stich mehr gewinnen kann, dann werden die niedrigsten $count_i$ Karten von Spieler i zum Gewinn von Spieler 0 gezählt. Auf diese Weise wird sichergestellt, dass die Gegner so wenig Punkte wie möglich abgeben.
Zur Illustration sei folgende Situation eines ♡-Spiels gegeben.

Spieler 0	♣J	♡10
Spieler 1	♢J	♡A
Spieler 2	♠8	♠9

Der Spielbaum zu diesem Spiel ist auszugsweise in Abbildung (9.3) dargestellt.

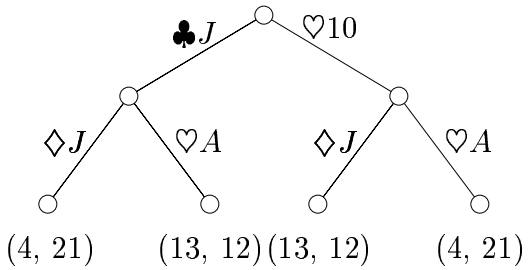


Abbildung 9.3: Spielbaum

Die erste Komponente der Wertepaare gibt an, wie viele Punkte der Alleinspieler in diesem Teilspiel erzielt, die zweite analog für die Gegner.
Der Satz über Äquivalenzklassen (siehe Kapitel 7) sagt aus, dass die Reihenfolge in der Spieler 2 seine Karten ausspielt, unerheblich für den Ausgang des Spiels ist. Da Spieler 2 nie einen Stich bekommt und auch keine wertvollen Karten hat, wurden seine Karten im Spielbaum nicht berücksichtigt. Statt des Ausspielens einer Karte von Spieler 2, würde jedes mal $count_2$ inkrementiert werden.
Der Alleinspieler gewinnt in diesem Spiel den Stich mit $\clubsuit J$. Nach der oben vorgestellten Strategie würde Spieler 1 dann $\heartsuit A$ spielen. Diese Vorgehensweise führt dazu, dass der Alleinspieler 13 Punkte erzielt und die Gegner 12 (siehe Abbildung (9.3)). Diesen Wert kann der Alleinspieler allerdings nie erzwingen.
Man erhält aber korrekte Schranken, wenn die Werte der Karten gemäß folgender Wertevertauschung modifiziert werden.

Definition 9.1 (Wertevertauschung) *Sei T_i die Menge der Trümpfe, die Gegner $i \in \{1, 2\}$ auf der Hand hat. Durch folgenden Algorithmus ist die Wertevertauschung definiert:*

- *Sei V_i die absteigend sortierte Folge der Kartenwerte aus T_i*
- *Sortiere T_i nach absteigender Kartenhöhe*
- *Der Wert der j-ten Karte aus T_i ist der j-te Wert aus V_i*

Angewandt auf das Beispiel ergeben sich folgende Werte:

Spieler 0	($\clubsuit J$, 2)	($\heartsuit$10, 10)		Spieler 0	($\clubsuit J$, 2)	($\heartsuit$10, 10)
Spieler 1	($\diamondsuit J$, 2)	($\heartsuit A$, 11)	$\rightsquigarrow$	Spieler 1	($\diamondsuit J$, 11)	($\heartsuit A$, 2)
Spieler 2	($\spadesuit$8, 0)	($\spadesuit$9, 0)		Spieler 2	($\spadesuit$8, 0)	($\spadesuit$9, 0)

Jede Karte ist hier als Paar dargestellt. Die erste Komponente bezeichnet die Karte selbst, die zweite gibt den Wert der Karte an: links ohne und rechts mit Wertevertauschung.
Jetzt bekommt der Alleinspieler nur 4 Punkte und die Gegner die restlichen 21 Punkte.
Diese Vorgehensweise kann aber auch unterschätzen, wie folgendes Beispiel zeigt:

Spieler 0	$\clubsuit J$	$\heartsuit A$
Spieler 1	$\diamondsuit J$	$\heartsuit$10
Spieler 2	$\spadesuit$8	$\spadesuit$9

Im Vergleich zum oberen Spiel wurden hier $\heartsuit 10$ und $\heartsuit A$ vertauscht. Auch hier spielt Spieler 0 $\clubsuit J$ und Spieler 1 antwortet mit $\heartsuit 10$. Der zweite Stich geht dann an die Gegner. Wenn jetzt noch die Werte von $\heartsuit 10$ und $\diamondsuit J$ vertauscht werden, so erhält der Alleinspieler 4 Punkte. Dieser Wert taucht aber im Spielbaum (siehe 9.4) nie auf.

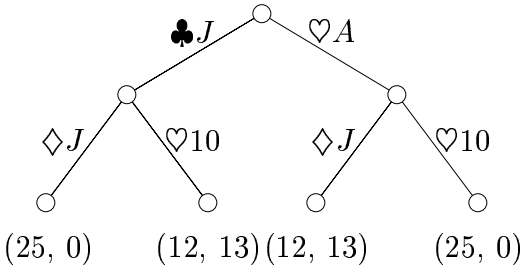


Abbildung 9.4: Spielbaum 2

Satz 9.1 (sichere Stiche für den Alleinspieler) *Sei p eine Position, in der der Alleinspieler einen Stich eröffnet. Dann läßt sich die Anzahl Punkte r, die er im Restspiel, ausgehend von p, noch erzielen wird wie folgt abschätzen:*
Spieler 0 spielt forcierend, die Werte der Trümpfe werden mit obiger Wertvertauschung vertauscht und die Gegner spielen wie oben beschrieben. Die Punkte, die von Spieler 0 dadurch gemacht werden, sind eine untere Schranke von r.

Beweis zu 9.1 *Betrachte das Spiel, in dem der Alleinspieler, ausgehend von p, so viele Stiche wie möglich machen muss. Die Punktzahl der einzelnen Karten spielt in diesem Spiel keine Rolle. Das Spiel ist vorbei, wenn er keinen Stich mehr für sich entscheiden kann. Angenommen $(t_1, \ldots t_n)$ sind die Stiche, die der Alleinspieler gewinnt. Das Ziel des Alleinspielers ist es, n zu maximieren, das der Gegner, n zu minimieren. Es ist offensichtlich, dass der Alleinspieler jede Farbe von oben nach unten spielen muss. D.h. zuerst spielt er solange Trumpf, wie er Stiche gewinnen kann, dann in beliebiger Reihenfolge die restlichen Farben. Die Gegner können n minimieren, indem sie immer die niedrigste Karte legen. Diese Strategie ist für dieses Spiel optimal.*

Nun betrachte man das gleiche Spiel aber mit Kartenwerten, die nach der Wertevertauschung vertauscht wurden. Ziel von Spieler 0 ist es nun, die Punktzahl zu maximieren. Durch die Wertevertauschung gilt für 2 Karten c und c': wenn $c \prec c'$, dann auch $val(c) < val(c')$. Die Punkte des Allein-

spielers werden genau dann minimiert, wenn er so wenig Stiche wie möglich gewinnt.

□

9.2 Die Gegenspieler

Für den Fall, dass einer der Gegner den Stich eröffnet, ergibt sich eine andere Situation. Hier spielen die Gegenspieler forcierend. Anders als beim Alleinspieler genügt es hier aber, dass einer der beiden Gegner den Stich gewinnt. Dadurch kann es vorkommen, dass nicht jeder Stich vom gleichen Spieler eröffnet wird. In diesem Fall müssen die Rollen dann getauscht werden. Spieler 0 spielt hier genau wie vorher die Gegner. Daraus ergibt sich der Algorithmus aus Abbildung (9.5).

```
def two_vs_one():
    repeat:
        until play_trick() == False
```

Abbildung 9.5: Zwei gegen Einen

Die Vorgehensweise wird an einem Beispiel beschrieben. Sei durch folgende Kartenverteilung ein ♠-Spiel gegeben.

Spieler 0	♢9	♢A	♣9	♣10	♣A	♡7	♠Q	♢J
Spieler 1	♢K	♣7	♣Q	♡8	♡9	♠7	♠9	♠J
Spieler 2	♢10	♣8	♣K	♡Q	♡10	♡A	♠K	♡J

Spieler 1 fängt an und spielt mit Spieler 2 zusammen. Nach obigem Algorithmus ergeben sich dann der Reihe nach folgende Stiche:

	Stiche			**count**	**Regel**
1.	♠J	♠K	♠Q	0	(Regel 1)
2.	♠9	♡J	♢J	0	(Regel 1)
3.	♡A	♡7	♡9	0	(Regel 2a)
4.	♡10	♡8	$*$	1	(Regel 2a)
5.	♡Q	$*$	♢K	2	(Regel 2b)
6.	♢10	♠7	$*$	3	(Regel 3)

Hierbei gibt "Regel" an, welche Regel zur Anwendung kam. Auch hier werden die Werte der Karten gemäß Definition (9.1) vertauscht, allerdings nur die des Alleinspielers. Der Alleinspieler muss am Ende noch seine 3 niedrigsten

Karten abgeben (♣9, ♣10, ♢9). Die Wertevertauschung kommt in diesem Beispiel nicht zum Tragen, da die Gegner alle Trümpfe des Alleinspielers gewonnen haben. Dadurch erzielen die Gegner $51 + 10 = 61$ Punkte.

9.3 Die dritte Regel

Interessant ist in diesem Beispiel der letzte Stich. Die Kartenverteilung vor diesem Stich ist folgende:

Spieler 0	♢9	♢A	♣9	♣10	♣A	count = 2
Spieler 1	♣7	♣Q	♠7			
Spieler 2	♢10	♣8	♣K			

Spieler 2 spielt nun ♢10, Spieler 1 übernimmt mit Trumpf-7. Über die Handkarten von Spieler 0 ist nur bekannt, dass er von den fünf angegebenen Karten noch drei besitzt (*count* = 2 bedeutet, dass Spieler 0 von seinen Karten zwei abgeworfen hat, aber nicht bekannt ist, welche dies sind). In jedem Fall kann Spieler 0 den Stich nicht gewinnen, da er kein Trumpf besitzt. Der Algorithmus erhöht in dieser Situation *count*, d.h. Spieler 0 spielt eine beliebige, nicht bekannte Karte, was die tatsächlichen legalen Möglichkeiten für Spieler 0 höchstens überschätzt, also aus Sicht der Gegenpartei pessimistisch ist.

Satz 9.2 (sichere Stiche für die Gegner) *Sei p eine beliebige Position in einem Skatspiel. Auf dem Tisch liegen keine Karten und einer der Gegner (Spieler 1 oder Spieler 2) eröffnet den Stich.*
Die Anzahl der Punkte, die die Gegner im Restspiel ausgehend von p noch erzielen, läßt sich wie folgt abschätzen: Die drei Spieler spielen wie in Algorithmus aus Abbildung (9.9) beschrieben, wobei die Werte der Trümpfe des Alleinspielers mit der Wertevertauschung vertauscht werden.

Beweis zu 9.2 *Der Beweis verläuft analog zum Beweis für den Satz über die sicheren Stiche des Alleinspielers.*

□

9.4 Ergebnisse

Ein Test mit 1.000 zufälligen Farb- und Grandspielen hat ergeben, dass die durchschnittliche Differenz von sicheren Punkten für eine Anfangsposition und dem tatsächlichen Wert 8,67 beträgt. Hierbei wurde die Vorhandposition zufällig an einen der drei Spieler vergeben. Dieser Test ist in Abbildung

(9.6) dargestellt.
Die Rechenzeit für alle Beispiele zusammen lag bei 0,03 s.

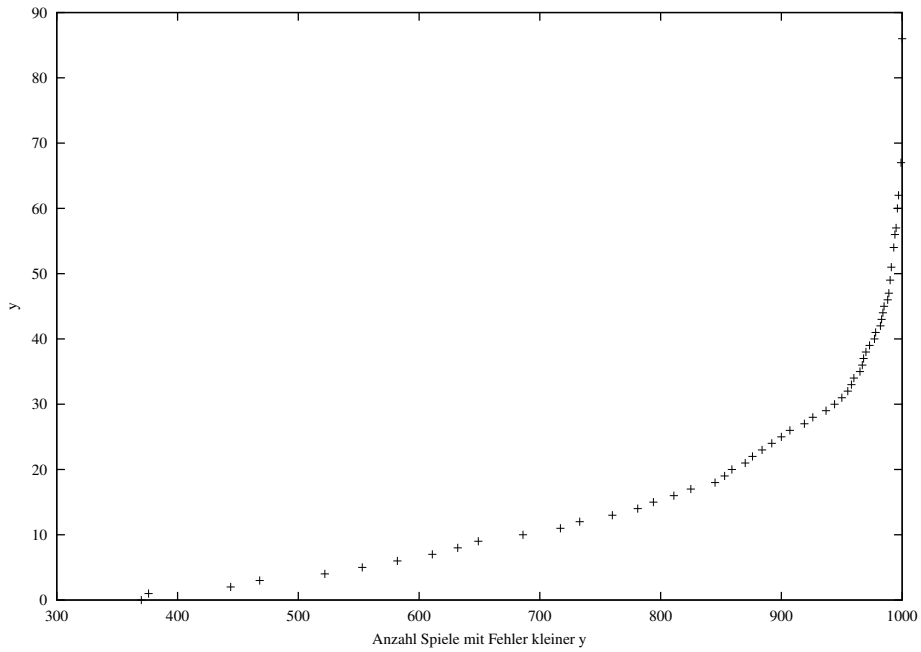


Abbildung 9.6: Sichere Stiche unterschätzen den spieltheoretischen Wert

Es ist offensichtlich, dass sich dieser Algorithmus in *mws_equiv* einbetten läßt. Der daraus entstandene Algorithmus wird mit *mws_ss* bezeichnet. Ein Test aus 1.000 zufälligen Spielen zeigt die Performance-Steigerung im Vergleich zu *mws_equiv*. Die Resultate sind in den Abbildungen (9.7) und (9.8) zu sehen.

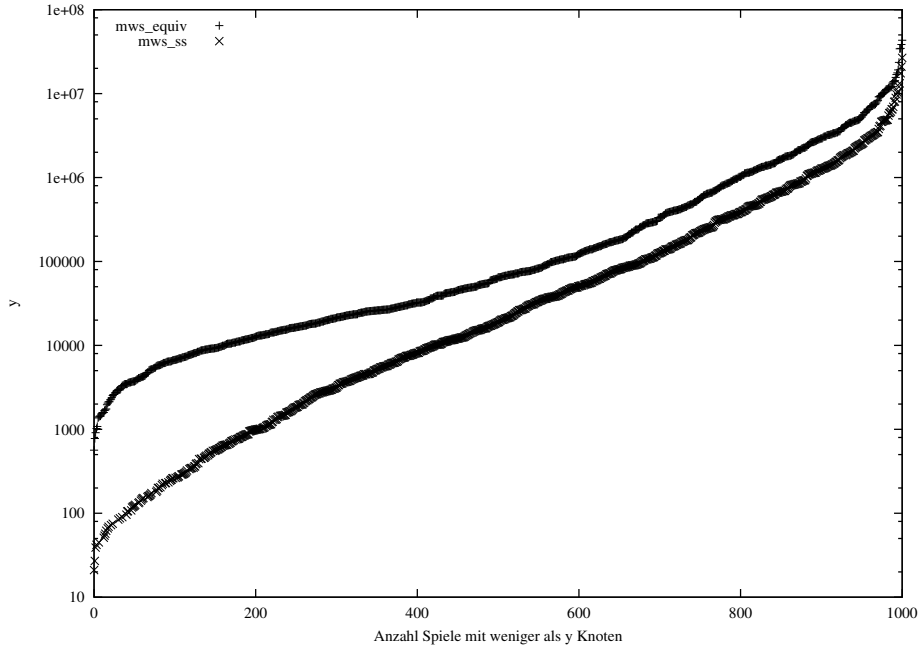


Abbildung 9.7: Evaluierte Knoten von *mws_equiv* und *mws_ss* (logarithmische Skala)

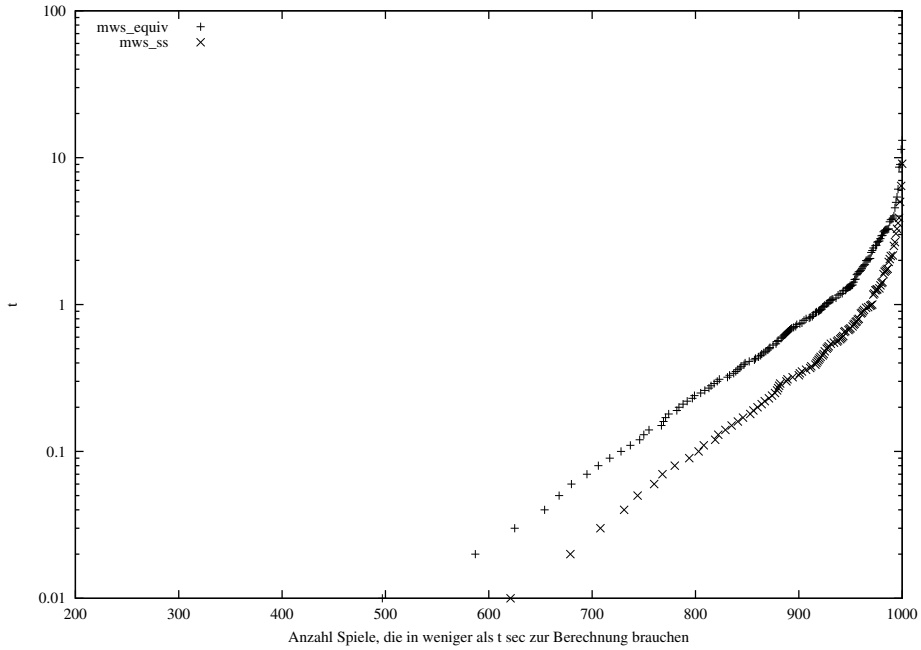


Abbildung 9.8: *mws_equiv* und *mws_ss* im Vergleich (logarithmische Skala)

```
def play_trick():
    # 1. Regel
    if Auspieler hat Trumpf and die Partner haben den höchsten Trumpf:
        Der Spieler der gegenpartei mit dem höchsten Trumpf spielt
        diesen aus.Der andere Spieler der Gegenparte spielt den Trumpf
        mit dem höchsten Wert aus, bzw. seine höchstwertige Karte,
        falls er keinen Trumpf besitzt

        if Alleinspieler hat Trumpf:
            Alleinspieler spielt niedrigsten Trumpf
        else:
            count++
        return True

    # 2. Regel
    if es gibt Farbe s mit folgender Eigenschaft:
        Ausspieler hat s and
        (Die Partner haben die höchste Karte von s or
         der Mitspieler ist blank auf s, hat aber Trumpf)
         and Alleinspieler kann nicht einstechen:

            Spieler spielt höchste Karte dieser Farbe
            if Alleinspieler hat Farbe:
                Alleinspieler spielt niedrigste Karte
            else:
                count++

            # 2a
            if Partner hat Farbe:
                Partner spielt höchste Karte dieser Farbe
            # 2b
            elif es ist nicht notwendig, dass Partner Trumpf spielt:
                Partner spielt Karte mit maximalem Wert (Farbe egal)
            # 2c
            elif Partner muss Trumpf spielen:
                Partner spielt Trumpf mit maximalem Wert
        return True

    # 3. Regel
    if Ausspieler hat eine Farbe, die der Mitspieler nicht hat and
       der Mitspieler hat den höchsten Trumpf
        Spieler spielt Farbe an, die der Partner nicht hat.
        Von dieser Farbe legt er die wertvollste Karte.
        Partner spielt höchsten Trumpf
        count++
        return True

    return False
```

Abbildung 9.9: Sichere Stiche für die Gegner

Kapitel 10

Zuganordnung

Der $\alpha\beta$-Algorithmus schreibt nicht vor, in welcher Reihenfolge die Nachfolger eines Knotens evaluiert werden müssen. Wenn jedoch immer der "beste" Zug als erstes ausprobiert wird, dann beschneidet der Algorithmus die meisten Knoten. Diese Tatsache macht sich die Zuganordnung zu Nutze. Es wird dabei versucht, mit einer Heuristik denjenigen Zug vorherzusagen, welcher der beste ist.
Durch diesen Mechanismus können größere Teile des Spielbaumes ausgeschlossen werden, was wiederum eine Reduzierung der Suchzeit bewirkt.

10.1 Niedrigster Verzweigungsgrad

Beim Skat hat sich folgende Zuganordnung als geeignet erwiesen: Expandiere immer die Farbe zuerst, die den kleinsten Verzweigungsgrad verursacht. Diese Idee läßt sich durch den in Abbildung (10.1) dargestellten Algorithmus ausdrücken.
Aufwendige Zuganordnung verursacht u.U. mehr Beschneidungen, dafür ist sie berechnungsintensiver. Die hier vorgestellte Zuganordnung ist nur für jeden dritten Knoten etwas rechenintensiver, nämlich dann wenn ein Stich zu Ende ist.
A. Plaat et. al. ordnen in [9] die Züge in ähnlicher Weise. Zusammen mit Schaeffers *History-Heuristik* [11] nennen sie diese Vorgehensweise *Exploiting Irregular Branching Factor* (EIB). Die History-Heuristik in $\alpha\beta$-Algorithmen bewertet Züge nach ihrer Güte. Dazu wird für jeden Zug gezählt, wie oft er ein sogenannter guter Zug war. Ein Zug ist ein guter Zug, wenn er entweder das Abschneiden eines Teilbaums verursacht oder den besten Minimax-Wert erzielt. In jeder Position werden die Züge gemäß ihrer Bewertung aus der History-Heuristik ausgeführt.

Züge, die einen kleineren Verzweigungsgrad als andere Züge verursachen bekommen einen Bonus, der auf die Bewertung der History-Heuristik angerechnet wird.

```
def next_ordered_move(cards):
    if Stich nicht zu Ende:
        return höchste Karte von cards
    branch = +∞
    c = 1
    for each suit s:
        if Auspieler hat s:
            for i in {0, 1, 2} \ {Ausspieler}:
                if Spieler i hat s:
                    c *= #Karten von Spieler i mit Farbe s
                else:
                    c *= #Karten von Spieler i
            if branch > c:
                branch = c
                bestSuit = s
    return höchste Karte von cards mit Farbe bestSuit
```

Abbildung 10.1: Zuganordnung für Farbspiele

10.2 Die beste Karte zuerst

Ähnlich wie bei EIB wird auch in der hier gewählten Anordnung der Züge die beste Karte gespeichert. Dazu wird für jede Position, in der eine Karte das Abschneiden eines Teilbaums verursacht, diese Karte zusammen mit den anderen Informationen über diese Position in die Transpositionstabelle eingefügt.
Wird ein Knoten ein zweites Mal besucht, wird diese Karte zuerst analysiert. Die Idee dahinter ist einfach: Wenn eine Karte einmal gut ist, dann ist sie auch in ein zweites Mal gut. Tatsächlich läßt sich durch dieser Erweiterung der Suchbaum noch weiter verkleinern.
Die Abbildungen (10.2) und (10.3) zeigen das Ergebnis eines Tests mit 1.000 zufälligen Spielen. Trotz des höheren Berechnungsaufwandes erzielt die Version mit Zuganordnung wesentlich bessere Ergebnisse. *mws_mo* ist dabei der Suchalgorithmus *mws_ss* zusammen mit der hier vorgestellten Zuganordnung.

10.3 Interpretation

Die Performance des Algorithmus aus Abbildung (10.1) sinkt, wenn man statt der höchsten Karte immer die niedrigste Karte zurückliefert. Vermutlich ist das ein Indiz dafür, dass man beim Skat – soweit es geht – forcierend spielen sollte. Das wäre auch eine Erklärung dafür, warum man mit den sicheren Stichen (siehe Kapitel 9) passable Schranken erhält.

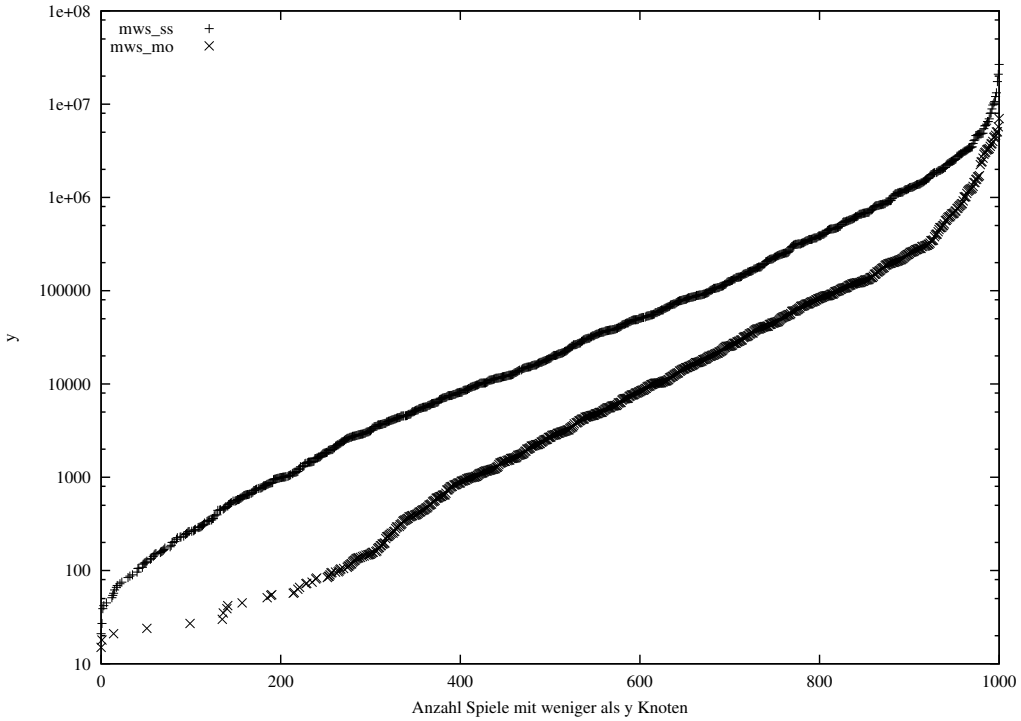


Abbildung 10.2: Knotenzahl mit und ohne Zuganordnung (logarithmische Skala)

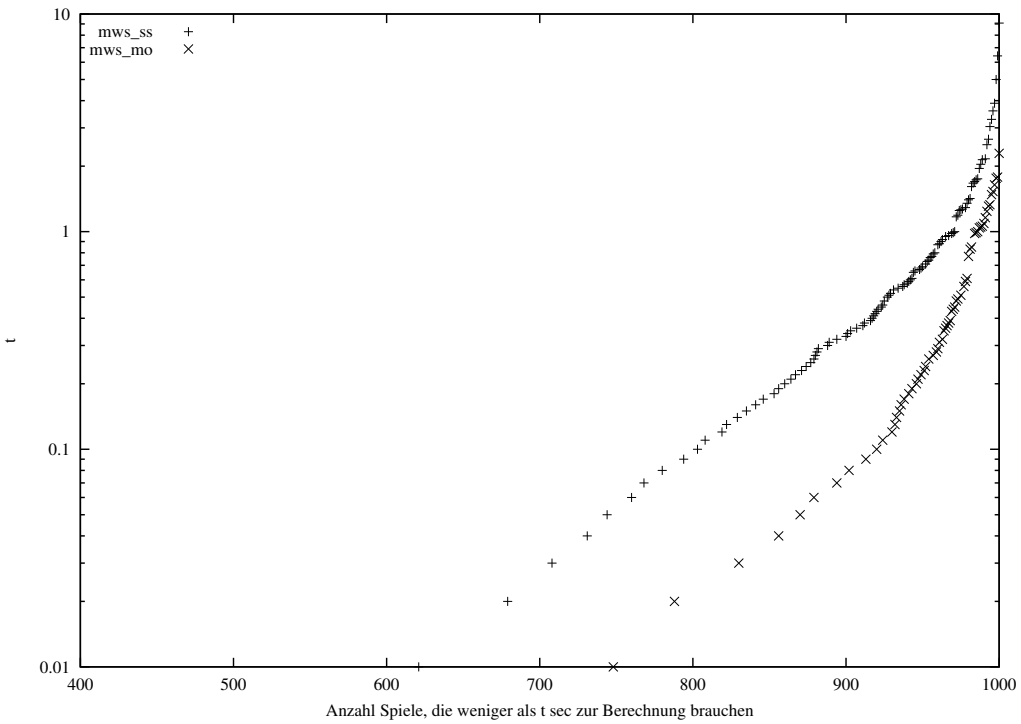


Abbildung 10.3: Laufzeit mit Zuganordnung (logarithmische Skala)

Kapitel 11

Nullspiele

Ziel des Nullspiels ist es für den Alleinspieler, überhaupt keinen Stich zu gewinnen. Die Punkte, die in einem Stich erzielt werden, sind also uninteressant. Der Alleinspieler muss versuchen jeden Stich zu verlieren. Da hier die Punkte nicht wichtig sind, darf man ein Spiel im Gegensatz zu den anderen Varianten auch dann nicht abbrechen, wenn eine Partei mehr als 60 Punkte erreicht hat.
Doch auch bei Nullspielen kann die Suche vor dem Erreichen von Blättern im Spielbaum abgebrochen werden. Die erste Situation, in der man ein Spiel abbrechen kann, folgt direkt aus den Spielregeln, nämlich dann, wenn der Alleinspieler einen Stich gewonnen hat.
Die zweite Situation, in der das Spiel vorzeitig beendet ist, tritt ein, wenn der Alleinspieler ein *sicheres Blatt* auf der Hand hat. Ein sicheres Blatt ist ein Blatt, mit dem man bei optimalem Spiel und beliebiger Kartenverteilung bei der Gegnern immer gewinnt. Mit dieser kleinen Modifikation läßt sich der $mws_t t$-Algorithmus aus Kapitel 6 auch für Nullspiele verwenden.
Im Folgenden wird nun genau definiert, unter welchen Voraussetzungen der Alleinspieler das Spiel gewinnt und gezeigt, wie in diesem Fall die optimale Spielweise des Alleinspielers aussieht.

11.1 Sicheres Blatt

Von bestimmten Karten einer Farbe kann man a priori sagen, daß ein Nullspiel damit nicht verloren wird. Hat der Alleinspieler z.B. nur ♠7, ♠9 und ♠J auf der Hand, dann ist ♠ sicher, außer er muss damit anspielen. Sicher heißt in diesem Zusammenhang, dass der Alleinspieler keinen Stich mit dieser Farbe gewinnt, falls er optimal spielt – egal welche Karten die Gegenspieler auf der Hand haben.

Im Folgenden wird nun ein Test entwickelt, mit dem man feststellen kann, ob die Karten des Alleinspielers sicher sind.
Um diesen Test zu entwickeln und die Korrektheit davon zu beweisen, bedarf es folgender Definitionen:

Definition 11.1 (Sichere Karten) *Eine Menge C von Karten derselben Farbe ist sicher, wenn für alle $k \in \{1, \ldots, |C|\}$ die Karten $\{c_1, \ldots, c_k\}$ unter den $2k-1$ kleinsten Karten der Farbe sind.*

Wenn $C \neq \emptyset$, dann muss C somit die niedrigste Karte beinhalten (betrachte $k = 1$). Aus der Definition für sicher Karten, ergibt sich unmittelbar ein Test, mit dem die sicheren Karten des Alleinspielers bestimmt werden können. Dieser Test wird in Abbildung (11.1) gezeigt. Im abgebildeten Algorithmus wird mit *cards* eine Menge von Karten bezeichnet, für die die Teilmenge der sicheren Karten (*result*) bestimmt werden soll. *played_cards* bezeichnet dabei die Karten, die in früheren Stichen ausgespielt wurden und deshalb nicht mehr berücksichtigt werden müssen.

```
def sichere_karten(cards, played_cards):
    result = ∅
    counter = 0
    for s in (♢, ♡, ♠, ♣):
        for c in (s7, s8, s9, s10, sJ, sQ, sK, sA) \ played_cards:
            if c in cards:
                counter += 1
                result ∪= {c}
            else:
                if counter == 0:
                    break
                counter -= 1

    return result
```

Abbildung 11.1: Algorithmus Sichere-Karten

Satz 11.1 *Wenn der Alleinspieler nicht die erste Karte eines Stichs legen muss, dann verliert er jeden Stich (d.h. er gewinnt das Nullspiel), wenn er nur sichere Karten auf der Hand hat.*

Beweis zu 11.1 *Der Alleinspieler habe nur sichere Karten und sei nicht am Anspiel. Er spielt dann nach folgender Strategie:*

a) Wenn er die angespielte Farbe nicht bedienen kann, spielt er eine Karte maximalen Rangs.

b) Wenn er die angespielte Farbe bedienen kann, legt er eine Karte maximalen Rangs von den Karten, die unter der bzw. den bereits gespielten Karten liegen.

Zu zeigen ist:

1. *Es ist möglich dieser Strategie zu folgen, wenn der Alleinspieler nur sichere Karten hat.*

2. *Der Alleinspieler hat nach dem Ende des Stichs wieder nur sichere Karten auf der Hand.*

Damit ist diese Strategie erfolgreich. 1. und 2. werden mit vollständiger Induktion über die Anzahl der Stiche gezeigt.

zu 1. Regel a) kann immer befolgt werden, wenn die angespielte Farbe nicht bedient werden kann. Andernfalls kann immer Regel b) befolgt werden, da der Alleinspieler nach der Definition von sicheren Karten die Karte niedrigsten Ranges in der angespielten Farbe halten muss.

zu 2. Zunächst ist klar, dass nur die angespielte Farbe kritisch ist. Wirft der Alleinspieler ab, bleiben seine Karten in der abgeworfenen Farbe sicher. Wirft ein Gegenspieler ab, kann das nicht zum Nachteil des Alleinspielers sein. Es genügt also nur die angespielte Farbe zu betrachten.
Sei c diejenige von den Gegnern gespielte Karte, die der Alleinspieler mit seiner Karte c' nach Regel b) unterbietet. Die dritte Karte im Stich spielt keine Rolle. Nach Regel b) gilt $c \prec c'$ und der Alleinspieler hält keine Karte c'' mit $c \prec c'' \prec c'$. Seien $c_1 \prec c_2 \prec \cdots \prec c_k$ die Karten des Alleinspielers in dieser Farbe und sei $c = c_i$ für $i \in \{1, \ldots, k\}$. Zu überprüfen ist, dass nach dem Stich die Menge $\{c_1, \ldots, c_k\} \setminus \{c_i\}$ sicher ist. Es muss also gelten:

   (a) Für alle $1 \leq j < i$: $\{c_1, \ldots, c_j\}$ sind unter den niedrigsten $2j-1$ Karten der Farbe

   (b) Für alle $i \leq j \leq k-1$: $\{c_1 \ldots, c_{j+1}\} \setminus \{c_i\}$ sind unter den niedrigsten $2j-1$ Karten der Farbe.

   Ersteres gilt auf jeden Fall, da $\{c_1, \ldots, c_j\}$ diese Eigenschaft bereits vor dem Stich hatte und durch die von den Gegnern gespielten Karten die Rangposition höchstens sinken kann.
   Für $\{c_1, \ldots, c_{j+1}\} \setminus \{c_i\}$ mit $j \geq i$ gilt nach Voraussetzung, dass diese Karte vor dem Stich unter den $2(j+1)-1 = 2j+1$ niedrigsten Karten waren. Da c und c' gespielt wurden und $c \prec c_{j+1}$, $c' \prec c_{j+1}$, sind diese Karten also nach dem Stich unter den $2j-1$ niedrigsten Karten.

□

11.2 Zuganordnung beim Nullspiel

Um die Performance zu steigern, wird die Zuganordnung aus Abbildung (11.2) verwendet. Diese wird nur für den Alleinspieler angewendet.

```
def next_null_move(p, cards):
    c = höchste Karte der aktuellen Spielfarbe auf dem Tisch
    if Alleinspieler ist Mittelhand:
        H = Karten mit aktueller Spiefarbe von Hinterhand
        d = niedrigste Karte aus H
        c = max(c, d)
    # Alleinspieler muss unter c bleiben

    A = Karten mit aktueller Spiefarbe des Alleinspielers
    if A == ∅: # Alleinspieler hat Farbe nicht
        f = sichere Karten des Alleinspielers
        A = A \ f
        if A == ∅:
            A = cards
        return höchste Karte von A

    else: # Alleinspieler hat die Farbe
        if es gibt Karte d ∈ A mit d < c:
            return höchste Karte mit dieser Eigenschaft
        else: # das Spiel ist verloren
            return irgendeine Karte aus A
```

Abbildung 11.2: Zuganordnung für Nullspiele

Man könnte natürlich auch eine Zuganordnung für die Gegenspieler entwerfen, doch Tests haben gezeigt, dass dadurch zwar die Zahl der Knoten gesenkt werden kann, aber die Ausführungszeit deutlich höher liegt.
Die Abbildungen (11.3) und (11.4) zeigt einen Test mit 1.000 zufälligen Nullspielen, die durchschnittliche Kontenzahl pro Spiel liegt bei dabei bei 20.993. Die durchschnittliche Laufzeit bei 0,019 s. Weitere Verbesserungen des Algorithmus sind möglich. So gibt es z.B. viele Situationen, in denen nicht alle Züge des Alleinspielers betrachtet werden müssen – so sollte der Alleinspieler z.B. nie $\diamondsuit 9$ abwerfen, wenn er auch $\diamondsuit K$ auf der Hand hält. Aufgrund der ohnehin sehr guten Laufzeitergebnisse wurde jedoch auf solche Verbesserungen verzichtet.

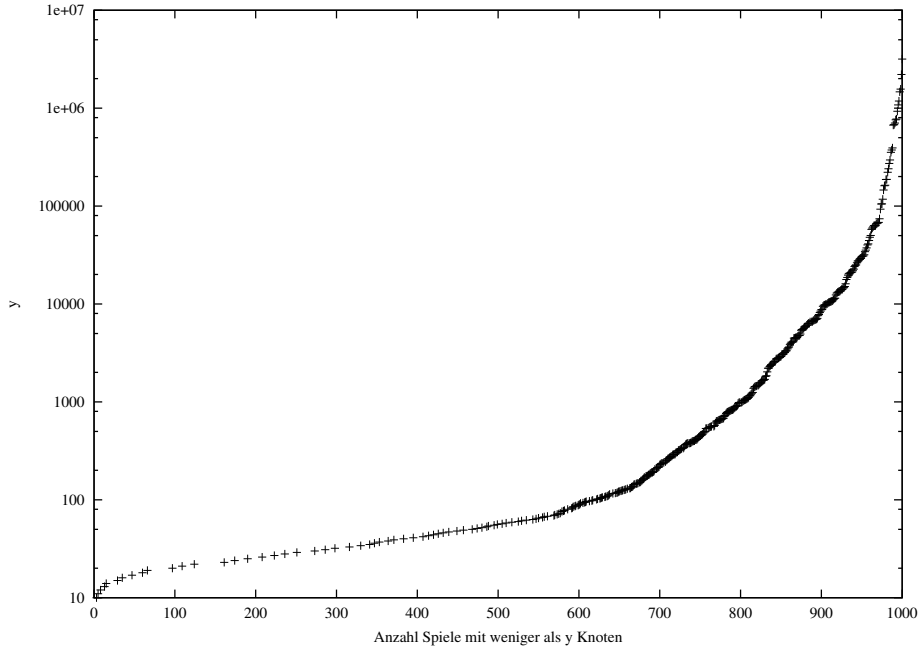


Abbildung 11.3: Knoten bei Nullspielen

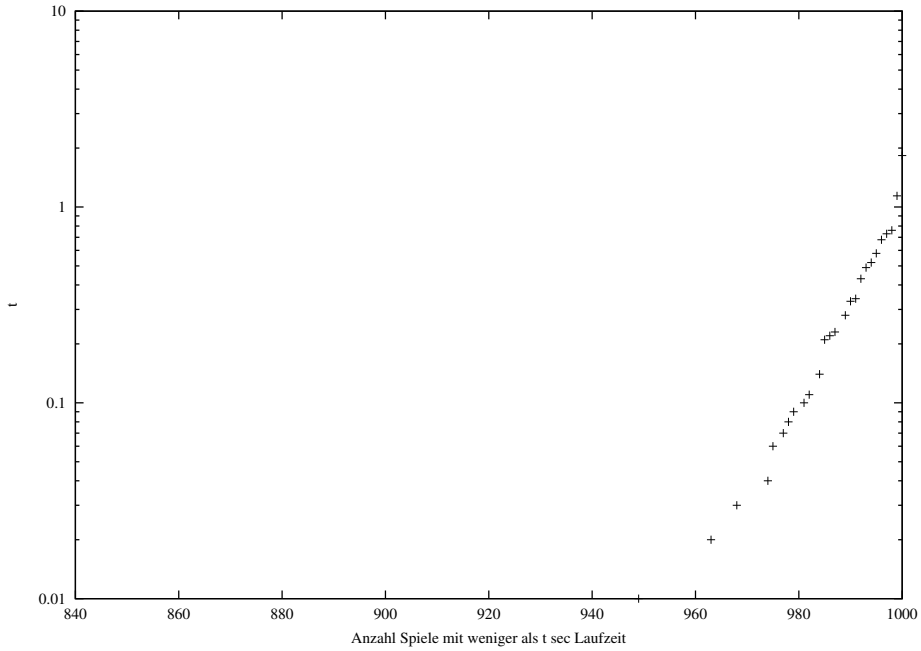


Abbildung 11.4: Laufzeiten für Nullspiele

Kapitel 12

Verdeckte Skatspiele

Bis jetzt wurde immer von vollständigem Weltwissen ausgegangen, bei einem Skatspiel hat man das aber i.A. nicht. In diesem Kapitel wird eine Möglichkeit beschrieben, wie man den in dieser Arbeit entwickelten Double Dummy Solver in verdeckten Skatspielen verwenden kann. Reizen und Drücken wird in diesem Zusammenhang noch nicht berücksichtigt. Diesem Thema ist das nächste Kapitel gewidmet.

12.1 Gewinnen von Informationen

Am Anfang eines Spiels kennt jeder Spieler nur seine eigenen Karten. Der Alleinspieler kennt – durch die Aufnahme des Skats – 12 von 32 Karten. Es bleiben dann noch $\binom{20}{10} = 184.756$ mögliche Kartenverteilungen für die Gegner übrig.
Im weiteren Spielverlauf erhält man durch das Ausspielen von Karten immer mehr Informationen darüber, welcher Spieler welche Karten zu Beginn des Spiels hatte.
Aus dem Ausspielen einer Karte c eines Spielers $i \in \{0, 1, 2\}$, können folgende Informationen gezogen werden:

- Spieler i hatte die Karte c zu Beginn des Spiels
- Wenn Spieler i nicht Farbe bekannt hat, dann hat i diese Farbe nicht (mehr)

Diese Informationen sind auf jeden Fall richtig. Man könnte auch noch aus anderen Situationen Informationen gewinnen. Sei z.B. ♣ Trumpf und der Alleinspieler muss die letzte Karte des Stichs spielen. Auf dem Tisch liegen

$\heartsuit As$, $\heartsuit 10$. Wenn der Alleinspieler nicht Farbe bekennt bzw. Trumpf spielt, dann kann mit großer Wahrscheinlichkeit davon ausgegangen werden, daß er keinen Trumpf (mehr) hat. Allerdings muß das nicht notwendigerweise so sein.
In dieser Arbeit werden nur sichere Informationen verwendet. Für den Fall, dass auch unsichere Informationen gesammelt werden würden, würde man ein Maß benötigen, das den Grad der Sicherheit dieser Informationen bewerten. Aus Gründen der Einfachheit habe ich mich für den Ansatz, nur sichere Informationen zu verwenden, entschieden.

12.2 Mögliche Kartenverteilungen

Jeder Spieler sammelt die im letzte Abschnitt benannten sicheren Informationen. Durch folgende Vorgehensweise kann für den Alleinspieler ausgerechnet werden, welche verschiedenen Kartenbelegungen es mit seinen gesammelten Informationen gibt und wie viele es sind:
$rest_i$ bezeichne die Anzahl der Karten, die Gegner i auf der Hand hat. Die unbekannten Karten für den Alleinspieler sind alle Karten, die er noch nicht gesehen hat.
Wenn z.B. Gegner 1 auf einer Farbe blank ist, dann muß Gegner 2 die unbekannten Karten dieser Farbe auf der Hand haben. In diesem Fall kann $rest_2$ um die Anzahl dieser Karten herabgesetzt werden. Das gleiche gilt auch, wenn Spieler 2 auf einer Farbe blank ist.
Nachdem alle diese Informationen ausgenutzt worden sind, ergibt sich die Anzahl der möglichen Kartenbelegungen zu $\binom{rest_1+rest_2}{rest_1}$. Diese Belegungen erhält man, indem alle Permutationen der noch unbekannten Karten konstruiert werden. Die ersten $rest_1$ Karten bekommt Gegner 1, den Rest Gegner 2.
Für die Gegner wird ein anderer Algorithmus benötigt, um alle mit den bisher erhaltenen Informationen konsistente Kartenbelegungen zu konstruieren. Das liegt daran, dass für die Gegner der Skat unbekannt ist. Allerdings läßt sich dieses Problem auf das obige zurückführen.
Wählt man eine noch mögliche Belegung für den Skat, dann ergibt sich für einen Gegner eine ähnliche Situation wie für den Alleinspieler. Allerdings muss darauf geachtet werden, dass eine so konstruierte Kartenbelegung überhaupt möglich ist. Folgende zwei Fälle können z.B. für Spieler 1 auftreten. Wenn Spieler 0 blank auf einer Farbe ist, dann muss Spieler 2 die unbekannten Karten dieser Farbe haben.

1. Spieler 2 ist auch blank auf dieser Farbe. Das kann aber nur sein, wenn die restlichen Karten dieser Farbe im Skat liegen. D.h. mit dem gewähl-

ten Skat gibt es keine Kartenbelegung, die konsistent mit den bisher gesammelten Informationen ist.

2. Spieler 2 ist nicht blank, aber dadurch dass er die restlichen Karten dieser Farbe bekommen hat, ist $rest_2 < 0$ was nicht sein kann. Auch in diesem Fall gibt es keine konsistente Belegung mit dem gewählten Skat.

Zu den Skatbelegungen, bei denen keiner dieser beiden Fälle eintritt, können wie beim Alleinspieler alle möglichen Kartenverteilungen konstruiert werden. Die Anzahl der möglichen Kartenverteilungen ist die Summe der einzelnen Kartenverteilungen, die sich für jeden möglichen Skat ergeben.

12.3 Konstruktion der Kartenverteilungen

Mit Hilfe eines Zufallszahlengenerators soll uniform eine mögliche Kartenbelegung gezogen werden. Die Idee hierbei ist, eine Zahl i zwischen 0 und der Anzahl der möglichen Kartenbelegungen P zu wählen. Dieser Zahl wird dann die i-te $rest_1$-elementige Teilmenge der noch unbekannten Karten zugeordnet. Diese Karten bekommt der erste Spieler, den Rest bekommt Spieler 2. Das Problem, das sich hierbei ergibt ist folgendes: Konstruiere zu einer n-elementigen Menge M unterscheidbarer Objekte die i-te k-elementige Teilmenge, mit $k \in \{0, \ldots, n\}$, $i \in \{0, \ldots, \binom{n}{k} - 1\}$ Ein Algorithmus, der das leistet ist in Abbildung (12.1) zu sehen.

```
def choose_sample(M, k, i):
    n = len(M)
    result = []
    while k > 0:
        if i < bin(n-1, k-1):
            result.append(M[n-1])
            k -= 1
        else:
            i -= bin(n-1, k-1)
        n -= 1
    return result
```

Abbildung 12.1: i-te k-elementige Teilmenge von M

Hierbei bezeichnet `bin(n, k)` den Binomialkoeffizienten $\binom{n}{k}$. Die Idee hinter diesem Algorithmus ist folgende: Die Menge der Teilmengen, die das Element c_n nicht beinhalten, werde mit $T_{\widehat{c_n}}$ bezeichnet, die übrigen Teilmengen mit

T_{c_n}. Die Teilmengen werden dann rekursiv nach dem Schema aus Abbildung (12.2) numeriert:

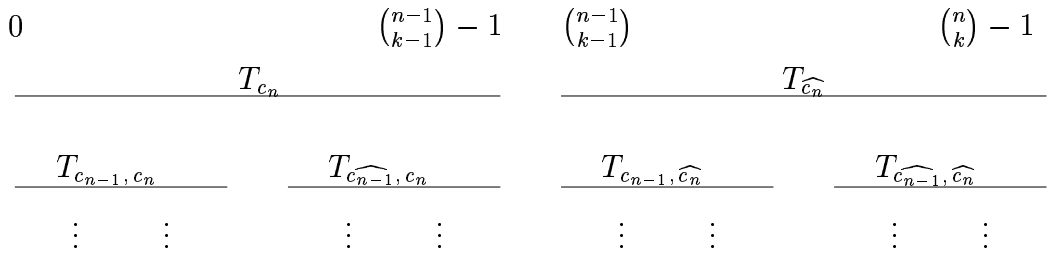


Abbildung 12.2: Numerierung von Teilmengen

Bei der Konstruktion der i-ten k-elementigen Teilmenge T_i^k vom M muss für jedes Element $c \in M$ entschieden werden, ob es zu T_i^k gehört oder nicht. Zuerst wird das für das Element c_n entschieden. Ist $i \leq \binom{n-1}{k-1} - 1$, dann ist $c_n \in T_i^k$. Eine Zahl $i \leq \binom{n-1}{k-1} - 1$ bezeichnet nämlich eine Teilmenge, die c_n enthält (vgl. Abbildung (12.2)). Analog wird für $i \geq \binom{n-1}{k-1}$ eine Teilmenge gewählt, die c_n nicht enthält.
Gilt $c_n \notin T_i^k$ muss der Index von i angepaßt werden, damit er wieder im Bereich $0 \leq i < \binom{n-1}{k}$ liegt. Ansonsten liegt er bereits wie gewünscht im Bereich $0 \leq i < \binom{n-1}{k}$ und es kann rekursiv fortgefahren werden.

12.4 Monte Carlo

Mit Hilfe des DDS sollen nun die Teile des Skatspiels gelöst werden, in denen man keine vollständige Information besitzt. Dies geschieht mit dem im Folgenden beschriebenen Monte-Carlo-Ansatz.

1. Konstruiere eine Menge D von Kartenbelegungen, die konsistent mit den bereits gesammelten Informationen ist.

2. Rechne für jede Kartenbelegung $d \in D$ und jede spielbare Karte c, den Wert des Spiels aus, wenn diese Karte gespielt wird.

3. Spiele die Karte, die die meisten Spiele gewonnen hat.

4. Haben mehrere Karten die selbe Anzahl an Spielen gewonnen, dann wird diejenige Karte davon gespielt, die die höchste kumulative Punktzahl in diesen Spielen erreicht hat.

Ohne den letzten Punkte kann sich folgende pathologische Situation ergeben: alle Karten haben die selbe Anzahl an Spielen gewonnen. Jetzt ist unklar,

welche Karte gespielt werden soll. Unter Umständen wird dann eine wertvolle Karte ausgespielt, die aber von der anderen Partei gestochen wird. Mit dieser Spielweise kann zwar das Spiel gewonnen werden, aber nur mit knappem Ergebnis.
Rationales Vorgehen würde in so einer Situation versuchen, die erreichbare Punktzahl zu maximieren und deshalb eine niedrigere Karte ausspielen.

12.5 Probleme mit Monte Carlo

Dieser Ansatz bietet die Möglichkeit, Algorithmen, wie z.B. $\alpha\beta$ in Domänen anzuwenden, in denen nur unvollständige Informationen vorliegen. Allerdings produziert diese Herangehensweise auch Probleme.
Der offensichtlichste Nachteil ist, dass auf diese Weise nie ein Zug gemacht wird, durch den nur Informationen gesammelt werden sollen. In gewissen Situationen würde z.B. ein Skatspieler eine Karte ausspielen – und möglicherweise verlieren – um Informationen zu bekommen. Solche Züge würde der DDS vermeiden. Das liegt daran, dass das reale Spiel mit unvollständiger Information auf den Fall mit vollständigem Weltwissen reduziert wird. Sind alle Informationen bekannt, dann muss so ein Zug nämlich nie gemacht werden.
Ein weiterer Nachteil, den diese Herangehensweise mit sich bringt, betrifft die Optimalität der Züge. Selbst wenn erschöpfende Suche angewendet wird, kann diese zu inkorrekten Ergebnissen führen, wie D. Basin und I. Frank in [2] gezeigt haben.
Allerdings spielt Ginsbergs GIB [4] mit einem Monte-Carlo-Ansatz auf hohem Niveau Bridge. GIB konstruiert zu diesem Zweck in jedem Schritt 100 konsistente Kartenbelegungen. Reduziert man diese Anzahl auf 50, dann spielt GIB signifikant schlechter. Aus diesem Grund werden auch in dieser Arbeit 100 Belegungen gewählt.

Kapitel 13

Reizen und Drücken

Ein Skatspiel beginnt mit dem Reizen. Hierbei muss ein Spieler entscheiden, ob er mit seinen 10 Karten und den 2 unbekannten Karten, die im Skat liegen, gegen die beiden anderen Spieler gewinnen kann. Ein Ansatz, um dieses Problem mit dem DDS zu lösen wäre folgender:

- Konstruiere eine Menge D von möglichen Kartenbelegungen für die anderen Spieler und den Skat.
- Berechne für jede dieser Kartenbelegungen und für jedes Spiel, ob mit diesen Karten gewonnen werden kann.
- Reize auf das Spiel, das die meisten dieser Spiele gewinnt.

Leider müssen für diesen Ansatz sehr viele Spiele berechnet werden. Angenommen D habe nur fünf Elemente. Für jede dieser Kartenverteilungen müssen alle sechs möglichen Spiele berechnet werden. In einem Spiel hat der Alleinspieler $\binom{12}{10} = 66$ Möglichkeiten zwei Karten zu drücken. Insgesamt müßten demnach 1.980 Spiele berechnet werden.
Aus diesem Grund wurde in dieser Arbeit für das Reizen und Drücken ein anderer Ansatz gewählt.

13.1 Kleinste Fehlerquadrate

Mit einem Mechanismus, der ungefähr vorhersagen kann, ob ein Spiel gewonnen oder verloren wird, kann man obigen Ansatz beibehalten, vorausgesetzt, dieser Mechanismus ist sehr schnell. Mit einem *Least Mean Square* Algorithmus (LMS) (siehe Abbildung (13.1)) wurden Regeln erstellt, mit denen man, gegeben 10 Karten, vorhersagen kann, ob mit diesen ein bestimmtes Spiel gewonnen werden kann. Der daraus resultierende Klassifikator kann in etwa

5 s für 10.000 mal 10 Karten vorhersagen, welches Spiel man damit gewinnen könnte.
Im Folgenden wird kurz beschrieben, wie LMS für dieses Problem eingesetzt werden kann. Eine genaue Beschreibung dieses Algorithmus findet man in [6]. Dieser Algorithmus minimiert die Summe der Fehlerquadrate E^s mit $s \in \{\diamondsuit, \heartsuit, \spadesuit, \clubsuit, grand, null\}$.

$$E^s = \sum_{\langle b, V^s_{train}(b)\rangle \in T} (V^s_{train}(b) - V^s(b))^2$$

Hierbei bezeichnet T die Trainingsmenge. Ein Trainingsbeispiel $t \in T$ ist ein Paar $\langle b, V^s_{train}(b)\rangle$, in dem b die zehn Karten des Alleinspielers bezeichnet und

$$V^s_{train}(b) = \begin{cases} 1 & \text{falls der Alleinspieler das Spiel } s \text{ gewinnt} \\ 0 & \text{sonst} \end{cases}.$$

Die Bewertungsfunktion V^s ist wie folgt definiert:

$$V^s(b) = \sum_{i=0}^{6} f_i(p) \cdot w^s_i.$$

$w^s \in \mathbb{R}^6$ ist ein sogenannte Gewichtsvektor. Diese wurden anfangs mit zufälligen Werten aus [0, 1] initialisiert und danach gemäß der Update-Regel in Abbildung (13.1) angepaßt. Hierbei ist c die sogenannte *Learning Rate*. Für diese erwies sich die Zahl 0,0001 als geeignet. Mit diesem Vektor werden die Merkmale $f_i(b)$ gewichtet. Die Bedeutung dieser Merkmale ist in der folgende Tabelle beschrieben.

	Grand- und Farbspiele	**Nullspiele**
$f_0(b)$	Anzahl der Buben in b	Anzahl 7er und 8er in b
$f_1(b)$	Anzahl der Trümpfe in b	Anzahl der unsicheren Farben in b
$f_2(b)$	Anzahl 10er und Asse in b	Anzahl Könige und Asse in b
$f_3(b)$	Punktzahl aller Karten in b	Summe über die "Ränge" aller Karten in b, d.h. nach der Abbildung $\{7 \mapsto 0, 8 \mapsto 1, \ldots, As \mapsto 7\}$.
$f_4(b)$	Anzahl verschiedener Farben in b	Anzahl verschiedener Farben in b
$f_5(b)$	Anzahl blanker 10er in b	Anzahl der Farben aus b, zu denen keine 7 in b ist
$f_6(b)$	durchschnittliche Punktzahl der "sicheren Stiche" aus 30 zufälligen Spielen	Anzahl "sicherer Karten" in b

Insgesamt standen 180.000 Testbeispiele zur Verfügung. Auf 126.000 davon wurde trainiert, der Rest diente der Validation.

```
def lms():
    for each example <b, V_train(b)>:
        error = V_train(b) - V(b)
        for each weight w_i:
            w_i = w_i + c * f_i * error
```

Abbildung 13.1: LMS Algorithmus

13.2 Resultate

Für jedes Spiel (♢, ♡, ♠, ♣, Grand (G) und Null (N)) lassen sich drei Gruppen von Klassifikatoren unterscheiden, je nachdem, welcher Spieler Vorhand ist. Für jede dieser Gruppen wurden 100 Klassifikatoren gelernt. Von diesen wurden diejenigen Klassifikatoren gewählt, die die wenigsten Beispiele der Validationsmenge falsch positiv und falsch negativ klassifiziert haben. Die Ergebnisse sind in folgenden Tabellen zu sehen. Der Index eines Klassifikators gibt dabei an, welcher Spieler Vorhand ist. In den Zeilen der Tabellen stehen die Ergebnisse der Klassifikatoren, die Spalten geben die korrekten Werte an.

Spieler 0 ist Vorhand

$\diamondsuit_0$	+	-
+	89	158
-	139	2614

$\heartsuit_0$	+	-
+	96	158
-	57	2689

♠$_0$	+	-
+	68	187
-	141	2604

♣$_0$	+	-
+	76	165
-	183	2576

G_0	+	-
+	88	201
-	128	2583

N_0	+	-
+	10	110
-	240	2640

Spieler 1 ist Vorhand

$\diamondsuit_1$	+	-
+	13	169
-	39	2779

$\heartsuit_1$	+	-
+	33	157
-	217	2593

$\spadesuit_1$	+	-
+	8	183
-	49	2760

$\clubsuit_1$	+	-
+	29	164
-	208	2599

G_1	+	-
+	10	169
-	44	2777

N_1	+	-
+	31	89
-	461	2419

Spieler 2 ist Vorhand

$\diamondsuit_2$	+	-
+	41	137
-	388	2434

$\heartsuit_2$	+	-
+	20	169
-	159	2652

$\spadesuit_2$	+	-
+	36	150
-	212	2602

$\clubsuit_2$	+	-
+	9	177
-	152	2662

G_2	+	-
+	11	165
-	179	2645

N_2	+	-
+	12	108
-	191	2689

Um nun zu erfahren, welches Spiel man mit gegebenen 10 Karten und gegebener Spielposition gewinnen kann, werden die entsprechenden 6 Klassifikatoren auf diese Karten angewendet. Es wird dann das Spiel zurückgeliefert, dessen Klassifikator die höchste positive Bewertung abgegeben hat. Räumt kein Klassifikator diesen Karten eine Gewinnchance ein, dann wird kein Spiel zurückgeliefert.
An den Ergebnissen der Klassifikatoren fällt auf, dass sehr viele Spiele verloren werden. Das liegt daran, dass bei zufälliger Kartenverteilung und zufälligem Spiel mit offenen Karten die meisten Spiele verloren werden. Das hat zur Folge, dass i.A. die Spielweise mit diesem Ansatz sehr vorsichtig ist.

13.3 Drücken

Mit den gelernten Klassifikatoren kann auch das Drücken realisiert werden.

- Konstruiere aus den 12 Karten des Alleinspielers alle 66 mögliche Kombinationen für den Skat.

- Wende jeweils auf die restlichen 10 Karten die Klassifikatoren an. Drücke die Karten, für die ein Klassifikator das höchste positive Ergebnis zurückgeliefert hat und spiele das entsprechende Spiel.
- Wenn kein Klassifikator ein positives Ergebnis geliefert hat, dann drücke die beiden Karten aus dem Skat und spiele das Spiel, auf das gereizt wurde.

Das ist ein sehr einfacher Algorithmus, der auf das Problem der Überreizung nicht eingeht. Betrachtet man die gedrückten Karten, dann fällt auf, das zum Teil offensichtliche Fehler gemacht werden. Es kann vorkommen, dass eine blanke 10 nicht gedrückt wird und stattdessen die 7 der gleichen Farbe. In manchen Fällen wurde sogar Trumpf gedrückt.
In den neun im Anhang aufgelisteten Skatspielen hat das Drücken zufällig funktioniert.
Die Entwicklung einer guten Lösung für diese Problem hätte den Rahmen dieser Arbeit gesprengt.
In den folgenden Tabellen ist zu sehen, welche Karten durch diesen Ansatz gedrückt werden und welches Spiel vorgeschlagen wird.

12 Karten												**Skat**		**Spiel**
♢9	♢J	♣J	♡7	♡8	♡9	♡K	♡A	♠7	♠8	♣7	♣8	♢9	♡K	♢

12 Karten												**Skat**		**Spiel**
♢7	♢9	♡7	♡8	♡Q	♡A	♠9	♠Q	♣9	♣K	♣10	♢J	♢7	♡Q	Grand

12 Karten												**Skat**		**Spiel**
♢7	♢8	♢K	♡7	♡Q	♠7	♠9	♠10	♢J	♡J	♠J	♣7	♢7	♢8	♢

12 Karten												**Skat**		**Spiel**
♢7	♢8	♢10	♢A	♢J	♡J	♠J	♠7	♠8	♠K	♣7	♣K	♢10	♣7	Grand

12 Karten												**Skat**		**Spiel**
♢7	♢8	♡7	♡9	♠7	♠9	♠10	♠A	♣A	♡J	♠J	♣J	♢7	♢8	♠

12 Karten												**Skat**		**Spiel**
♢Q	♡J	♠J	♣J	♡8	♡10	♡A	♠8	♣7	♣8	♣K	♣10	♢Q	♡8	♢

Zusammenfassung und Ausblick

Durch die in dieser Arbeit verwendeten und entwickelten Mechanismen ist der Double Dummy Solver in der Lage, in durchschnittlich 0,06 Sekunden zu entscheiden, ob bei einem offenes Spiel ein Sieg erzwungen werden kann oder nicht. Pro Spielbaum werden durchschnittlich nur noch 150.000 Knoten analysiert. Im Vergleich zu einem der ersten Ergebnisse mit dem $\alpha\beta$-Algorithmus mit Transpositionstabelle, der für die Berechnung eines Spiels durchschnittlich 25 Sekunden benötigt und dabei 85.000.000 Knoten evaluiert, ist das eine sehr hohe Leistungssteigerung.

Algorithmus	**Durchschnittswerte**	
	Knoten	**Laufzeiten**
$\alpha\beta\_tt$	85.000.000	25 s
mws_tt	2.500.000	2,4 s
mws_equiv	1.100.000	1,4 s
mws_ss	500.000	0,14 s
mws_mo	150.000	0,06 s

Die Zahl der analysierten Knoten pro Spiel ist vergleichbar mit der von Ginsbergs GIB [4] nach einem Jahr Entwicklungszeit. Dieses Programm benötigte damals im Schnitt 200.000 Knoten pro Spiel. Allerdings wird Bridge auch mit 52 Karten gespielt.

Sicher kann die durchschnittliche Berechnungszeit für ein Spiel noch weiter reduziert werden. Dazu könnte man den $\alpha\beta$-Algorithmus durch *Proof Number Search* ersetzen. Wie die in dieser Arbeit vorgestellten Erweiterungen in diesem Algorithmus einsetzbar sind, müßte untersucht werden.

Der Satz über Äquivalenzklassen wird nur auf Karten angewendet, die den gleichen Wert haben. Allerdings liefert dieser Satz auch Informationen über Karten, bei denen dies nicht gegeben ist. Es müßte untersucht werden, wie diese Informationen effizient ausgenutzt werden können. Dadurch ließe sich die Zahl der analysierten Knoten sicher noch weiter reduzieren.

Die Spielweise in verdeckten Skatspielen, also solchen mit unvollständiger Information, ist recht gut. Die Spiele, die dieses Programm als Alleinspieler gegen das Programm XSkat gespielt hat, wurden alle gewonnen (siehe Anhang).

Leider funktioniert das Reizen und Drücken nicht genauso gut. Hier werden oft Spiele vorgeschlagen, die nicht gewonnen werden können und zum Teil wird sogar Trumpf gedrückt. Vielleicht kann man die Vorschläge der Klassifikatoren durch einige zusätzliche Regeln wie z.B. "Drücke nie Trumpf" oder "Drücke blanke 10er" verbessern.
Ein weiteren Nachteil, der das Reizen und Drücken betrifft, ist die Tatsache, dass die Trainingsbeispiele offene Spiele sind, während im normalen Spiel nie offen gespielt wird. Ein Ausweg wäre vielleicht, dass Testbeispiele erstellt werden, bei denen das Ergebnis aus verdeckten Skatspielen besteht.

Anhang

3 Runden Skat

Die folgenden Tabellen zeigen 3 Runden Skat, die der Double Dummy Solver mit dem Monte Carlo Ansatz und den Klassifikatoren gegen das freie **XSkat-3.4** von Gunter Gerhardt gespielt hat. Unter `http://www.gulu.net/xskat/` steht dieses Programm zum Download bereit.
Die Tabellen zeigen die gespielten Stiche. Die unterstrichene Karte in jedem Stich gibt an, dass mit ihr dieser Stich gewonnen wurde. Spieler 0 ist immer der entwickelte Double Dummy Solver.

	Spieler		
	0	**1**	**2**
1.	♢J	♢8	$\underline{\clubsuit J}$
2.	♢9	♢Q	$\underline{\diamondsuit A}$
3.	♢7	♢K	$\underline{\diamondsuit 10}$
4.	♠Q	♠7	$\underline{\spadesuit A}$
5.	$\underline{\heartsuit 10}$	♡K	♡9
6.	$\underline{\spadesuit J}$	♠10	♡J
7.	$\underline{\heartsuit A}$	♡8	♡7
8.	$\underline{\spadesuit K}$	♣10	♠8
9.	♣7	$\underline{\clubsuit A}$	♠9
10.	$\underline{\clubsuit Q}$	♣9	♡Q

Gereizt wurde bis 18, ,
im Skat sind ♣8, ♣K,
im Skat waren ♢A, ♠8
Spieler 2 spielte Grand,
Spieler 2 kam heraus,
der Alleinspieler erzielte 50 Punkte.

	Spieler		
	0	**1**	**2**
1.	♠9	$\underline{\diamondsuit J}$	♠K
2.	$\underline{\spadesuit 10}$	♡A	♡8
3.	$\underline{\clubsuit A}$	♣8	♣9
4.	$\underline{\diamondsuit A}$	♢9	♢7
5.	♢10	$\underline{\spadesuit 8}$	♢K
6.	♣7	♡7	$\underline{\heartsuit K}$
7.	♠Q	$\underline{\heartsuit J}$	♢8
8.	♠J	$\underline{\clubsuit J}$	♠7
9.	$\underline{\spadesuit A}$	♡Q	♡9
10.	$\underline{\clubsuit 10}$	♣K	♣Q

Gereizt wurde bis 18,
im Skat sind ♢Q, ♡10,
im Skat waren ♢Q, ♢10,
Spieler 0 spielte ♠,
Spieler 0 kam heraus,
der Alleinspieler erzielte 87 Punkte.

	Spieler		
	0	**1**	**2**
1.	$\underline{\spadesuit Q}$	♠7	♠8
2.	♡A	♡Q	$\underline{\diamondsuit K}$
3.	$\underline{\diamondsuit 7}$	♠10	♠A
4.	♡9	♡K	$\underline{\diamondsuit Q}$
5.	$\underline{\diamondsuit 9}$	♠9	$\underline{\spadesuit K}$
6.	♣9	♣10	$\underline{\clubsuit A}$
7.	♢J	♠J	$\underline{\clubsuit J}$
8.	$\underline{\clubsuit K}$	♡10	♣7
9.	$\underline{\heartsuit J}$	♢10	♢8
10.	♣Q	$\underline{\diamondsuit A}$	♣8

Gereizt wurde bis 18,
im Skat sind ♢K, ♢10,
im Skat waren ♠7, ♠Q,
Spieler 2 kam heraus,
der Alleinspieler erzielte 52 Punkte.

	Spieler		
	0	**1**	**2**
1.	$\underline{\heartsuit A}$	♡7	♡8
2.	♠8	$\underline{\heartsuit J}$	♠K
3.	♢8	$\underline{\diamondsuit A}$	♢Q
4.	♣9	$\underline{\diamondsuit 7}$	♡Q
5.	$\underline{\spadesuit 10}$	♢9	♡K
6.	♠7	$\underline{\spadesuit J}$	♠9
7.	$\underline{\clubsuit A}$	♣7	♣8
8.	$\underline{\clubsuit J}$	♡9	♢J
9.	$\underline{\spadesuit Q}$	♣K	♡10
10.	$\underline{\spadesuit A}$	♣Q	♣10

Gereizt wurde bis 18,
im Skat sind ♢K, ♢10,
im Skat waren ♠7, ♠Q,
Spieler 0 spielte ♠,
Spieler 2 kam heraus,
der Alleinspieler erzielte 95 Punkte.

	Spieler		
	0	**1**	**2**
1.	♡A	♠A	♡7
2.	♠7	♡J	♠J
3.	♡9	♠10	♡8
4.	♣7	♣A	♣8
5.	♣Q	♣10	♣9
6.	♢9	♢8	♢10
7.	♢A	♢Q	♢7
8.	♣K	♠9	♠Q
9.	♣J	♠K	♠8
10.	♡Q	♢K	♢J

Gereizt wurde bis 18,
im Skat sind ♡K, ♡10,
im Skat waren ♢Q, ♠A,
Spieler 1 spielte ♠,
Spieler 0 kam heraus,
der Alleinspieler erzielte 70 Punkte.

	Spieler		
	0	**1**	**2**
1.	♢7	♢A	♢8
2.	♣9	♣A	♣7
3.	♠10	♣10	♣8
4.	♡A	♡K	♡Q
5.	♡9	♡10	♢K
6.	♠Q	♣Q	♣K
7.	♡8	♠K	♠A
8.	♠8	♠9	♣J
9.	♠7	♢J	♢9
10.	♡7	♠J	♡J

Gereizt wurde bis 27,
im Skat sind ♢Q, ♢10,
im Skat waren ♢7, ♡9,
Spieler 0 spielte ♠,
Spieler 1 kam heraus,
der Alleinspieler erzielte 61 Punkte.

	Spieler		
	0	**1**	**2**
1.	♠9	♠7	♠A
2.	♡A	♡8	♡7
3.	♣A	♢K	♣7
4.	♢Q	♠J	♣J
5.	♣8	♡9	♣Q
6.	♣10	♢10	♣K
7.	♢8	♠K	♠8
8.	♡J	♢7	♢A
9.	♡10	♢9	♢J
10.	♡K	♠10	♠Q

Gereizt wurde bis 18,
im Skat sind ♡Q, ♣9,
im Skat waren ♠K, ♣9,
Spieler 1 spielte ♢,
Spieler 2 kam heraus,
der Alleinspieler erzielte 59 Punkte.

	Spieler		
	0	**1**	**2**
1.	$\underline{\diamondsuit A}$	$\diamondsuit 9$	$\diamondsuit 8$
2.	$\underline{\diamondsuit J}$	$\heartsuit J$	$\underline{\clubsuit J}$
3.	$\heartsuit A$	$\underline{\spadesuit J}$	$\heartsuit 7$
4.	$\underline{\spadesuit A}$	$\spadesuit 7$	$\spadesuit 9$
5.	$\underline{\diamondsuit 10}$	$\diamondsuit Q$	$\diamondsuit K$
6.	$\underline{\diamondsuit 7}$	$\clubsuit 7$	$\heartsuit 8$
7.	$\underline{\heartsuit Q}$	$\spadesuit 8$	$\heartsuit 9$
8.	$\underline{\heartsuit K}$	$\clubsuit 9$	$\clubsuit Q$
9.	$\underline{\heartsuit 10}$	$\spadesuit Q$	$\spadesuit 10$
10.	$\underline{\clubsuit A}$	$\clubsuit K$	$\clubsuit 10$

Gereizt wurde bis 18,
im Skat sind $\spadesuit K$, $\clubsuit 8$,
im Skat waren $\spadesuit K$, $\clubsuit 8$,
Spieler 0 spielte Grand,
Spieler 0 kam heraus,
der Alleinspieler erzielte 101 Punkte.

	Spieler		
	0	**1**	**2**
1.	$\heartsuit 9$	$\underline{\heartsuit A}$	$\heartsuit 7$
2.	$\underline{\spadesuit 10}$	$\heartsuit 10$	$\spadesuit K$
3.	$\underline{\spadesuit J}$	$\spadesuit Q$	$\spadesuit A$
4.	$\underline{\clubsuit J}$	$\heartsuit J$	$\diamondsuit 8$
5.	$\clubsuit 7$	$\underline{\clubsuit K}$	$\clubsuit 9$
6.	$\clubsuit 10$	$\underline{\heartsuit 8}$	$\diamondsuit 9$
7.	$\underline{\spadesuit 9}$	$\heartsuit K$	$\diamondsuit K$
8.	$\underline{\spadesuit 8}$	$\clubsuit 8$	$\diamondsuit 10$
9.	$\underline{\spadesuit 7}$	$\diamondsuit Q$	$\diamondsuit A$
10.	$\underline{\diamondsuit J}$	$\clubsuit Q$	$\clubsuit A$

Gereizt wurde bis 18,
im Skat sind $\diamondsuit 7$, $\heartsuit Q$,
im Skat waren $\diamondsuit 7$, $\heartsuit Q$,
Spieler 0 spielte $\spadesuit$,
Spieler 1 kam heraus,
der Alleinspieler erzielte 95 Punkte.

Literaturverzeichnis

[1] L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. Proof-number search. *Artificial Intelligence*, 66:91–124, 1994.

[2] Ian Frank and David A. Basin. Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, 100(1-2):87–123, 1998.

[3] Matthew L. Ginsberg. Partition search. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference, Vol. 2*, pages 228–233, Menlo Park, California, 1996. AAAI Press.

[4] Matthew. L. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 584–589, 1999.

[5] Internationale Skatordnung. Deutscher Skatverband e.V. (DSkV) und International Skatplayers Association e.V. (ISPA-World), Vom *27. Deutschen Skatkongress,* `www.skat.com/dskv/skatgericht/isko.pdf`, (23. Juli 2003). 1998.

[6] Tom M. Mitchell. *Machine Learning.* McGraw-Hill International Editions, 1997.

[7] Martin J. Osborne and Ariel Rubinstein. *A Course in Game Theory.* MIT Press, 1994.

[8] Judea Pearl. *Heuristics - Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing Co., Reading, MA, 1984.

[9] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Exploiting graph properties of game trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 234–239, Portland, OR, 1986.

[10] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first fixed-depth game-tree search in practice. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 273–279, Montreal, Canada, 1995.

[11] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(11):1203–1212, 1989.