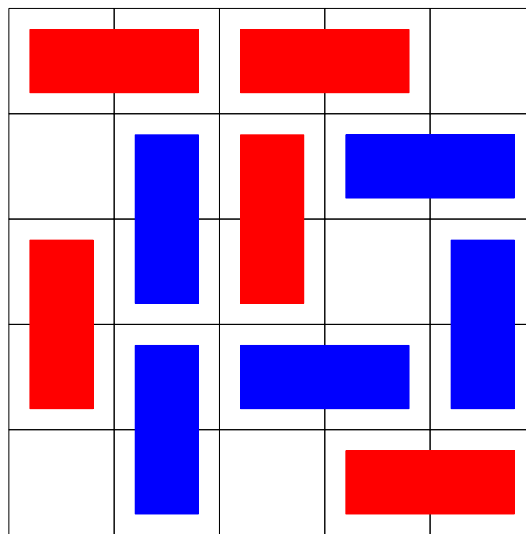


Bakkalaureatsarbeit

Domino Juvavum - Eine Spielanalyse



Martin Schneider

WS 2005/06

eingereicht bei Prof. Peter Gerl

Inhaltsverzeichnis

1	Einleitung	4
2	Mathematische Darstellung und Analyse eines Spiels	5
2.1	Einführende Definitionen	5
2.2	Spielegraph	5
2.3	Grundyfunktion und Grundywerte	6
3	Juvavum und Domino Juvavum	9
3.1	Spielbeschreibung	9
3.1.1	Regeln, Spielbrett, Spielsteine	9
3.1.2	Spielende, Sieger	10
3.2	Definitionen und Schreibweisen	10
4	Allgemeine Analyse von Domino Juvavum	11
4.1	Anzahl der möglichen Züge	11
4.2	Anzahl der Spielpositionen	12
4.3	Wer kann gewinnen?	12
5	Analyse von Domino Juvavum	13
5.1	$DJ(1,n)$	13
5.2	$DJ(2m,2n)$	13
5.3	$DJ(2m,2n+1)$	13
5.4	$DJ(2m+1,2n+1)$	14
5.4.1	$DJ(3,2n+1)$	14
6	Analyse von Domino Juvavum Misère	16
6.1	$DJM(1,n)$	16
6.2	$DJM(2,2k+1)$	16
6.3	$DJM(2,2k)$	17

7	Programmbeschreibung und Sourcecode	18
7.1	Darstellung von Spielbrettern	18
7.1.1	Board	18
7.1.2	Position	27
7.2	Finden aller Nachfolger einer Startposition	28
7.3	Grundywerte berechnen	31
7.4	Bemerkungen	33
7.4.1	Symmetrien	33
8	Literaturangaben	39

1 Einleitung

Diese Arbeit entstand im Rahmen des Projektpraktikums (WS 2004/05 und SS 2005) an der Universität Salzburg unter der Leitung von O.Univ.-Prof.Dr. Peter Gerl. Das Ziel war es, einige wichtige Methoden zur mathematischen Darstellung und Analyse eines (kombinatorischen) Spiels zu beschreiben und beispielhaft zur Untersuchung eines Spiels anzuwenden.

Die Arbeit teilt sich im Wesentlichen in vier Teile. Im ersten Teil werden einige mathematische Grundlagen zur Spielanalyse besprochen, im zweiten Teil werden die Spiele *Juvavum* und *Domino-Juvavum* vorgestellt und deren Regeln erklärt bevor schließlich im dritten Teil einige Ergebnisse der Analyse von *Domino-Juvavum* angegeben werden. Im Anschluss daran enthält das letzte Kapitel dieser Arbeit eine Beschreibung des beiliegenden Computerprogramms.

2 Mathematische Darstellung und Analyse eines Spiels

2.1 Einführende Definitionen

Definition 2.1 (Kombinatorisches Zweipersonenspiel) Ein Spiel heißt kombinatorisches Zweipersonenspiel, wenn gilt:

- Die beiden Spieler ziehen abwechselnd.
- Es gibt keinen Zufallseinfluss.
- Es ist vollständige Information gegeben (nichts ist verdeckt).

Definition 2.2 (Endliches Spiel) Ein Zweipersonenspiel heißt endlich, wenn gilt:

- Jede Partie endet nach einer endlichen Anzahl von Zügen.

Definition 2.3 (Stark endliches Spiel) Ein Zweipersonenspiel heißt stark endlich, wenn es endlich ist und überdies gilt:

- Von jeder Spielposition gibt es nur endlich viele Zugmöglichkeiten.

Definition 2.4 (Sieger im normalen Spiel) Sieger eines Spiels ist jener Spieler, der den letzten Zug macht.

Definition 2.5 (Misère-Form eines Spiels) Ein Spiel $M(G)$, das die selben Regeln und Zugmöglichkeiten wie ein Spiel G hat, heißt Misère-Form oder Misère-Spiel von G , wenn nicht der Spieler gewinnt, der den letzten Zug macht, sondern jener, der zuerst nicht mehr ziehen kann.

2.2 Spielegraph

Zur Analyse eines kombinatorischen Zweipersonenspiels empfiehlt sich die Darstellung mit Hilfe eines Spielegraphen. Dabei wird das Spiel durch einen Graphen repräsentiert, dessen Ecken die Spielpositionen und dessen Kanten die möglichen Spielzüge darstellen.

Definition 2.6 (Startecke, Zielecken) Als Startecke bezeichnen wir jene Ecke, welche die Startposition des Spiels darstellt. Als Zielecken bezeichnen wir all jene Positionen, die eine Zielposition repräsentieren, das heißt eine Position, von der aus kein Zug mehr möglich ist.

Das Spiel beginnt bei der Startecke. Der erste Spieler zieht entlang einer Kante zu einer weiteren Ecke (=Spielposition), danach der zweite Spieler. Die beiden Spieler ziehen nun so lange abwechselnd bis einer von ihnen eine Zielecke erreicht. Dieser Spieler ist Sieger des Spiels.

Bemerkung 2.7 (Spielegraph für Misère-Spiele) Für die Misère-Form eines stark endlichen Zweipersonenspiels kann im Wesentlichen der selbe Spielegraph wie für das gewöhnliche Spiel verwendet werden. Es werden lediglich eine zusätzliche uneigentliche Spielposition **A**, sowie Kanten von jeder Zielecke des normalen Spiels nach **A** hinzugefügt, was dazu führt, dass **A** zur einzigen Zielecke des Graphen wird. Damit wird berücksichtigt, dass in einem Misère-Spiel jener Spieler verliert, der zuerst eine Zielposition des normalen Spiels erreicht (da sein Gegner von dort noch eine Zugmöglichkeit zur Ecke **A** hat).

Folgerung 2.8 Es gilt:

- Der Spielegraph G eines stark endlichen Zweipersonenspiels hat endlich viele Ecken und Kanten.
- Er enthält keine Kreise.
- Von jeder Ecke $x \in G$ gibt es einen Weg zu einer Zielecke.

Definition 2.9 (Nachfolger einer Position) Sei x eine Ecke des Spielegraphen. Dann bezeichnen wir mit $N(x)$ die Menge aller direkten Nachfolger von x .

Die Menge $N(x)$ enthält alle Spielpositionen, die von x in einem Zug erreichbar sind.

2.3 Grundyfunktion und Grundywerte

Definition 2.10 Sei G ein Spielegraph und $x \in G$ eine Ecke. Dann bezeichnen wir mit $l(x)$ den längsten Weg von x zu einer Zielecke und mit N_i die Menge aller Ecken x , deren längster Weg zu einer Zielecke Länge i hat: $N_i := \{x \in G : l(x) = i\}$.

Definition 2.11 (Grundywert einer Spielposition) Unter dem Grundywert $g(x)$ einer Position x verstehen wir die kleinste Zahl $\in \mathbb{N}_0$, die bei keinem direkten Nachfolger von x vorkommt: $g(x) := \min\{k \in \mathbb{N}_0 \setminus \{g(y_1), \dots, g(y_n)\} \mid N(x) = \{y_1, \dots, y_n\}\}$. Die Funktion g heißt Grundyfunktion des Spielegraphen.

Folgerung 2.12 Aus der Definition folgt sofort, dass der Grundywert jeder Zielecke 0 ist.

Satz 2.13 Der Graph jedes stark endlichen Spiels hat eine eindeutige Grundyfunktion (ohne Beweis).

Definition 2.14 (Kern eines Spiels) Der Kern eines Spiels A ist die Menge aller Ecken (Spielpositionen) mit Grundywert 0:

$$\text{Ker}(A) = \{x : g(x) = 0\}$$

Satz 2.15 Jedes stark endliche Spiel hat einen Kern. Dieser ist eindeutig. (*ohne Beweis*)

Es gilt nun folgende für die Spielanalyse wesentliche

Folgerung 2.16 Sei G ein Spiel mit Kern $\text{Ker}(G)$. Dann gilt:

- Jede Zielecke liegt im Kern. $\forall x \in \mathbb{N}_0 : x \in \text{Ker}(G)$.
- Jeder Zug von einer Ecke im Kern führt zu einer Ecke, die nicht im Kern liegt. $\forall x \in \text{Ker}(G) : \forall y \in N(x) : y \notin \text{Ker}(G)$
- Von jeder Ecke, die nicht im Kern liegt, gibt es einen Zug zu einer Ecke im Kern. $\forall x \notin \text{Ker}(G) : \exists y \in N(x) : y \in \text{Ker}(G)$

Eine Gewinnstrategie besteht folglich darin nach Möglichkeit stets auf eine Spielposition mit Grundywert 0 zu spielen. Hat man einmal eine solche Position erreicht, hat man anschließend in jedem Zug die Möglichkeit wieder auf eine Position im Kern zu spielen (da der Gegenspieler auf eine Position nicht im Kern ziehen musste).

Definition 2.17 (Gewinnstrategie) Ein Spieler besitzt von einer Spielposition x aus eine Gewinnstrategie, genau dann wenn

- er bei optimalem Spiel stets gewinnen kann (egal wie sein Gegner zieht)

Definition 2.18 (Summe von Spielen) Werden n stark endliche Spiele G_1, \dots, G_n gleichzeitig gespielt (das heißt in jedem Zug wird in einem der Spiele gezogen), sprechen wir im Folgenden von der Summe dieser Spiele $\sum_{k=1}^n G_k$. Eine Spielposition eines Summenspiels ist ein Vektor $x = (x_1, \dots, x_n)$, wobei x_i ($1 \leq i \leq n$) die Spielpositionen der einzelnen Spiele sind.

Folgerung 2.19 Es gilt:

- Die Summe von stark endlichen Spielen ist wieder stark endlich.
- Die Summe von stark endlichen Spielen besitzt wieder eine Grundyfunktion und einen Kern.

Definition 2.20 (Binäre Summe von Zahlen in \mathbb{N}_0) Seien $a, b \in \mathbb{N}_0$ und seien a_1, \dots, a_k beziehungsweise b_1, \dots, b_m die Ziffern der Binärdarstellung von a und b :

$$a = \sum_{i=1}^k a_i \cdot 2^i$$
$$b = \sum_{i=1}^m b_i \cdot 2^i.$$

Sei o.B.d.A. $k \leq m$ und $b_i = 0 \forall i > k$. Sei schließlich $c_i = a_i + b_i \bmod 2$, $\forall 1 \leq i \leq l$ und $c = \sum_{i=1}^k c_i \cdot 2^i$, dann nennen wir c die binäre Summe von a und b (*Schreibweise*: $c = a \dot{+} b$).

Satz 2.21 (Hauptsatz über Summenspiele) Seien G_1, \dots, G_n Spiele (=Graphen) mit Grundyfunktionen g_1, \dots, g_n . Dann gilt: Das Summenspiel $\sum_{k=1}^n G_k$ hat die Grundyfunktion $g(x) = g(x_1, \dots, x_n) = g_1(x_1) \dot{+} \dots \dot{+} g_n(x_n)$.

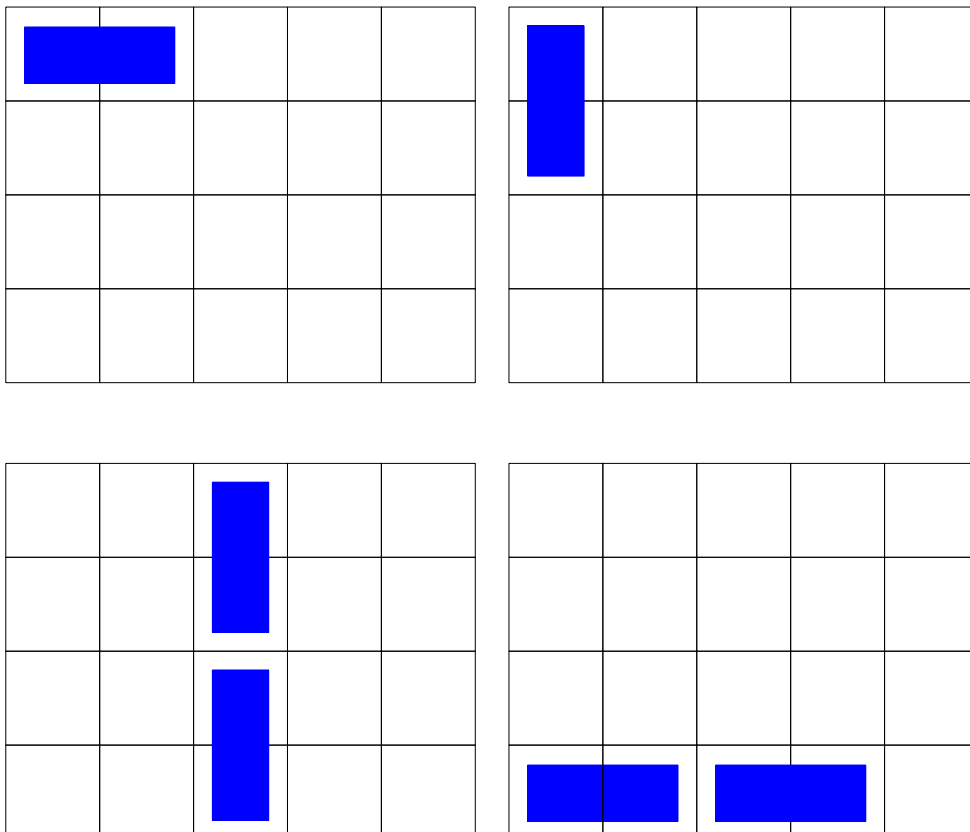
3 Juvavum und Domino Juvavum

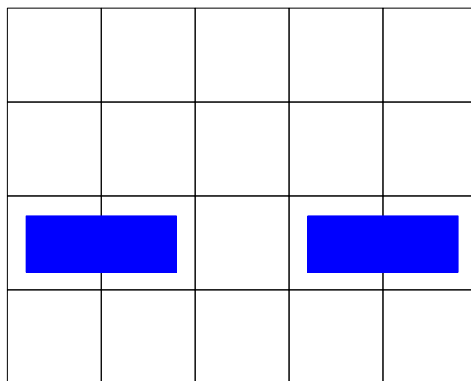
3.1 Spielbeschreibung

3.1.1 Regeln, Spielbrett, Spielsteine

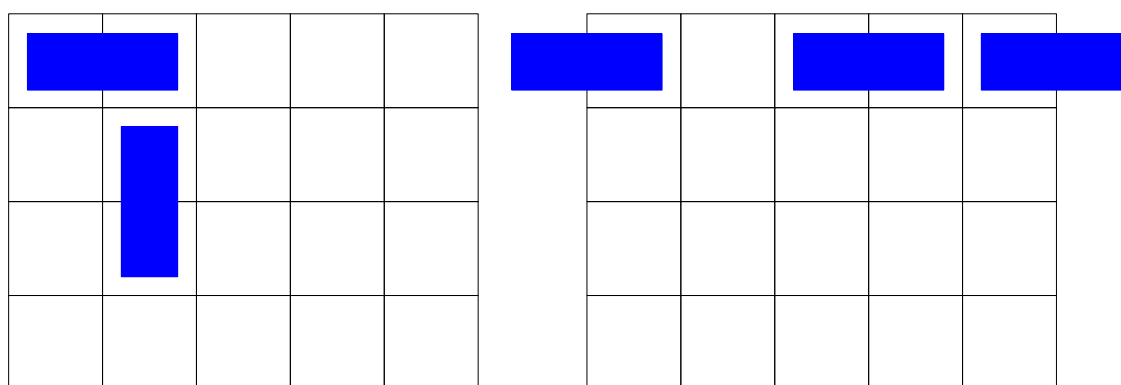
Sowohl Juvavum als auch Domino-Juvavum sind stark endliche Zweipersonenspiele, die auf einem $h \times w$ -Spielbrett gespielt werden ($h, w \in \mathbb{N}$). Beide Spieler verfügen über die gleichen Spielsteine. Dies sind bei Juvavum Münzen, bei Domino-Juvavum (wie der Name bereits vermuten lässt) Dominosteine. Jeder Spieler darf in jedem Zug beliebig viele Steine in eine beliebige Zeile oder Spalte setzen. Bei Domino-Juvavum müssen die Dominos stets so gelegt werden, dass sie zwei horizontal oder vertikal benachbarte Felder bedecken.

Beispiel: Im Folgenden sehen wir einige regelkonforme Züge von der Startposition (leeres Spielfeld) eines Domino Juvavum-Spiels auf einem 4×5 -Spielbrett.





Beispiel: Folgende Züge sind nicht erlaubt.



3.1.2 Spielende, Sieger

Sieger ist, wer den letzten Zug macht. Bei Juvavum bedeutet dies, dass kein Feld auf dem Spielbrett mehr frei ist, bei Domino-Juvavum ist das Spielende erreicht, sobald keine zwei benachbarten Felder mehr frei sind. Sieger bei Juvavum beziehungsweise Domino Juvavum in der Misère-Form ist jener Spieler, der als erster nicht mehr ziehen kann.

3.2 Definitionen und Schreibweisen

Definition 3.1 Der Einfachheit halber bezeichnen wir im Folgenden ein Domino Juvavum-Spiel auf einem $h \times w$ -Spielfeld mit $DJ(h, w)$. Mit $DJM(h, w)$ bezeichnen wir ein Domino Juvavum-Spiel auf einem $h \times w$ -Feld in der Misère-Form. h (height) bezeichne die Höhe, w (width) die Breite eines Spielfelds.

Definition 3.2 Weiters bezeichne A den 1. und B den 2. Spieler.

Da sich durch Drehung eines Spielbrettes um 90 Grad nichts Wesentliches an der Analyse des Spiels ändert, betrachten wir im Folgenden nur Spiele mit $h \leq w$.

4 Allgemeine Analyse von Domino Juvavum

Wir beziehen uns im Folgenden stets auf die Analyse von **Domino** Juvavum. Sollten sich einzelne Ergebnisse auch auf Juvavum anwenden lassen, wird darauf im Text hingewiesen werden.

4.1 Anzahl der möglichen Züge

Um eine grobe Idee zur Komplexität der Analyse von DJ zu bekommen ist es interessant sich die Anzahl der Nachfolger einer beliebigen Spielposition zu überlegen. Wir wollen dazu im Folgenden eine Rekursionsformel zu deren Berechnung angeben und werden einen Zusammenhang zur Fibonacci-Folge herstellen.

Definition 4.1 (Loch) Unter einem Loch der Länge l verstehen wir l nebeneinanderliegende freie Felder (in einer Zeile oder Spalte), die auf beiden Seiten durch ein belegtes Feld begrenzt werden.

Sei $f(l)$ die Anzahl der möglichen Spielzüge in einem Loch der Länge l . Dann gilt:

$$\begin{aligned} f(l) &= 1 + f(l-2) + f(l-1). \\ f(1) &= 0 \quad f(2) = 1 \quad f(3) = 2 \end{aligned}$$

Diese Gleichung ergibt sich aus folgender Überlegung:

Entweder das erste Domino wird an die erste mögliche Position gesetzt (siehe Grafik). Dann bleiben $l-2$ Felder für die restlichen Dominosteine und alle weiteren Zugmöglichkeiten frei. Dies entspricht dem ersten Teil der Rekursionsbeziehung ($1 + f(l-2)$). Die zweite Möglichkeit besteht darin das erste Feld frei zu lassen, wodurch noch $l-1$ besetzt werden können.



Addition mit 1 ergibt:

$$f(l) + 1 = (f(l-2) + 1) + (f(l-1) + 1).$$

Wir setzen nun $F(l) = f(l) + 1$

$$F(1) = 1 \quad F(2) = 2 \quad F(3) = 3$$

und erhalten die Fibonacci-Folge

$$F(l) = F(l-1) + F(l-2).$$

Es gilt folglich: $f(l) = F(l) - 1 \quad \forall l \geq 1$.

In einer Zeile beziehungsweise Spalte mit k Löchern mit Längen $l_1 \dots l_k$ gibt es $f(l_1, \dots, l_k) = \prod_{i=1}^k (1 + f(l_i)) - 1$ mögliche Züge.

Beispiel: Von der Startposition von $DJ(3, 5)$ beziehungsweise $DJM(3, 5)$ gibt es $3 \cdot f(5) + 5 \cdot f(3) = 3 \cdot 7 + 5 \cdot 2 = 31$ mögliche erste Züge.

4.2 Anzahl der Spielpositionen

Mit Hilfe des in (7) beschriebenen Computerprogramms ist es möglich, die Anzahl aller möglichen Spielpositionen eines beliebigen DJ-Spiels zu berechnen. In der folgenden Tabelle sind einige dieser Ergebnisse zusammengefasst.

Spielfeld	Anzahl der Spielpositionen (ohne Symmetrien)
1×2	2
1×3	3
1×4	5
1×5	8
1×6	13
1×7	21
2×2	6
2×3	18
2×4	54
2×5	162
2×6	486
2×7	1458
3×3	98
3×4	550
3×5	3054
3×6	17014
3×7	94682
4×4	5700

Anzahl der Spielpositionen

4.3 Wer kann gewinnen?

Die interessanteste Frage der Spielanalyse von DJ und DJM ist natürlich, welcher Spieler von einer gegebenen Position aus eine Gewinnstrategie hat, das heißt bei optimalem Spiel stets gewinnen kann, und wie diese aussieht.

Wir werden diese Fragen in den nächsten beiden Kapiteln für die Startpositionen einiger Spielfeldgrößen beantworten, sowie einige Ergebnisse des beiliegenden Computerprogramms angeben.

5 Analyse von Domino Juvavum

5.1 DJ(1,n)

Satz 5.1 $DJ(1, n)$ kann $\forall n$ stets A gewinnen.

Beweis: A belegt bei geradem n alle Felder, bei ungeradem alle bis auf eines. Damit hat er gewonnen, da B keine Zugmöglichkeit mehr hat.

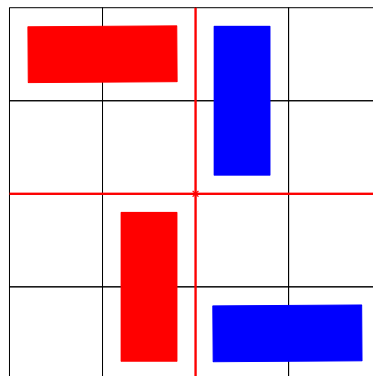
Satz 5.2 Die Startposition bei $DJ(1, n)$ hat Grundywert $\lfloor \frac{n}{2} \rfloor$.

Beweis: Bei $DJ(1, n)$ gilt: Jede Spielposition ist stets durch einen einzigen Zug von der Startposition erreichbar. Weiters sind maximal $m := \lfloor \frac{n}{2} \rfloor$ Züge möglich, das heißt es gibt Positionen mit $1, 2, \dots, m$ Dominos. Alle Positionen mit m Dominos haben Grundywert 0 (da es sich um Zielecken handelt). Positionen mit $m - 1$ Dominos haben Grundywert 1 (da alle Nachfolger Grundywert 0 haben). Positionen mit $m - k$ Dominos ($k \leq m$) haben Nachfolger mit Grundywerten $0, 1, \dots, m - 1$ und daher Grundywert k . Für $k = m$ erhält man den Grundywert der Startposition: $g(\text{Startposition}) = m = \lfloor \frac{n}{2} \rfloor$. Aus $g(\text{Startposition}) > 0$ folgt auch sofort die Aussage aus Satz (5.1).

5.2 DJ(2m,2n)

Satz 5.3 $DJ(2m, 2n)$ kann $\forall m, n \in \mathbb{N}$ stets B gewinnen.

Beweis: Bei $DJ(2m, 2n)$ liegt das Symmetriezentrum zwischen den vier Feldern in der Mitte des Spielbretts. B kann stets jeden Zug von A an diesem Symmetriezentrum spiegeln und mit dieser Spielweise immer den letzten Zug machen.

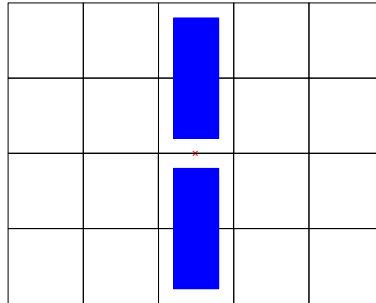


5.3 DJ(2m,2n+1)

Satz 5.4 $DJ(2m, 2n + 1)$ kann $\forall m, n \in \mathbb{N}$ stets A gewinnen.

Beweis: Bei $DJ(2m, 2n + 1)$ liegt das Symmetriezentrum auf einer Kante (siehe Grafik). Belegt A jene Spalte komplett, in der das Symmetriezentrum liegt (das ist stets möglich,

da die Anzahl der Zeilen gerade ist), kann er im Folgenden alle Züge von B am Symmetriezentrum spiegeln und so stets gewinnen. Die einzigen Zugmöglichkeiten, die sich nicht spiegeln lassen, sind durch den ersten Zug von A bereits eliminiert worden.



5.4 DJ(2m+1, 2n+1)

Ist sowohl die Spielfeldhöhe als auch die Spielfeldbreite ungerade, lassen sich noch keine allgemeinen Ergebnisse darüber angeben, welcher Spieler eine Gewinnstrategie besitzt. Wir werden daher im Folgenden einige spezielle Fälle diskutieren.

5.4.1 DJ(3, 2n+1)

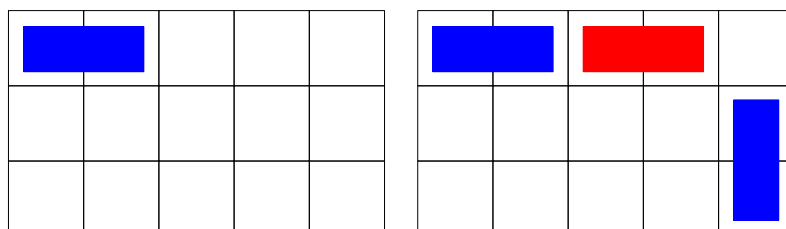
Satz 5.5 $DJ(3, 3)$ kann B stets gewinnen.

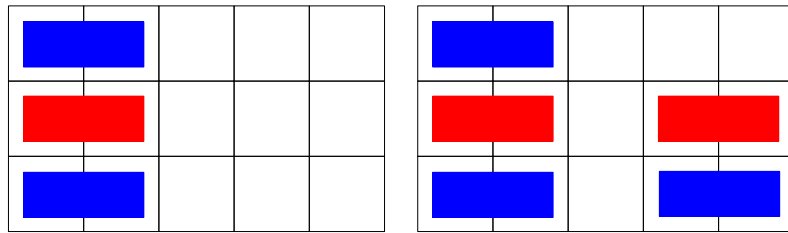
Beweis: diskutieren aller Zugmöglichkeiten.

Satz 5.6 $DJ(3, 2n + 1)$ kann für $n = 2, 3, 4$ A stets gewinnen.

Beweis: diskutieren aller Zugmöglichkeiten.

Vermutung 5.7 $DJ(3, 2n+1)$ kann $\forall n \geq 2$ stets A gewinnen. Ein guter erster Zug besteht darin, die ersten $2n - 2$ Felder der ersten Zeile zu belegen. Zieht B nun horizontal in der ersten Zeile kann A stets gewinnen, indem er auf eine zu $DJ(2, 2n)$ äquivalente Position zieht, von der er nach Satz () gewinnen kann. Zieht B horizontal in der zweiten/dritten Zeile, kopiert A diesen Zug in die jeweils andere Zeile. Noch zu diskutieren bleiben die vertikalen Zugmöglichkeiten von B . Es bleibt wipers zu zeigen, dass diese Spielweise stets zum Sieg führt.





Obige Behauptung wurde durch das Diskutieren aller möglichen Züge von B für $n \leq 4$ bestätigt. Für $n = 4$ wurde dazu das in (7) beschriebene Computerprogramm zur Hilfe genommen.

6 Analyse von Domino Juvavum Misère

Im Gegensatz zum gewöhnlichen Juvavum-Spiel lassen sich für die Misère-Form nicht so leicht allgemeine Ergebnisse angeben. Das Prinzip des Spiegelns der Züge des Gegners zum Beispiel, das ein wesentliches Hilfsmittel zur Analyse von DJ darstellt, lässt sich (zumindest in dieser Form) nicht auf das Misère-Spiel anwenden, da diese voraussetzt, dass es Ziel ist, nicht den letzten Zug zu machen.

Wir werden im Folgenden dennoch einige allgemeine Ergebnisse für DJM beweisen.

6.1 $DJM(1,n)$

Satz 6.1 $DJ(1,n)$ kann stets der 1. Spieler gewinnen.

Beweis: Bei geradem n zieht A so, dass 2 benachbarte Felder frei bleiben, bei ungeradem n lässt er 3 (davon 2 benachbart) frei. So muss B stets den letzten Zug machen.

Satz 6.2 Die Startposition bei $DJM(1,n)$ hat Grundywert $\lfloor \frac{n}{2} \rfloor$.

Beweis: Der Beweis lässt sich analog zum Beweis derselben Aussage für $DJ(1,n)$ führen. Durch das Hinzufügen der Endecke \mathbf{A} für die Misère-Form haben alle Positionen mit $m := \lfloor \frac{n}{2} \rfloor$ Dominos Grundywert 1 (da sie \mathbf{A} als einzigen Nachfolger haben). Die Positionen mit $m - 1$ Dominos haben Grundywert 0. Alle anderen Positionen haben die gleichen Grundywerte wie bei $DJ(1,n)$.

6.2 $DJM(2,2k+1)$

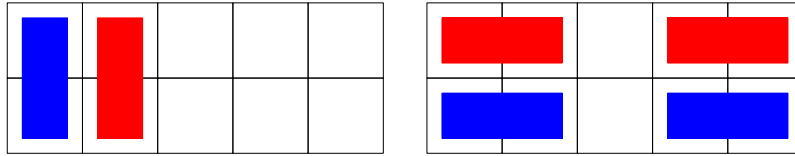
Satz 6.3 $DJM(2,2k+1)$ kann $\forall k \geq 0$ stets B gewinnen.

Beweis: Wir zeigen eine leichte Verallgemeinerung. Gegeben sei eine Spielposition von $DJM(2,n)$, $n \in \mathbb{N}$, in der $m := 2k + 1$ ($0 \leq k \leq \frac{n-1}{2}$) Spalten frei und die anderen belegt sind (d.h. es gibt keine Spalte, in der nur eines der beiden Felder durch ein Domino bedeckt wird). Dann gilt: Von einer solchen Position kann B stets gewinnen.

Für $k=0$ ist die Behauptung trivial, da auf einem solchen Spielfeld nur ein einziger Zug möglich ist und B stets gewinnen wird. Angenommen die Behauptung stimmt für alle ungeraden $m \leq 2k - 1$.

Wir betrachten nun eine Spielposition von $DJM(2,n)$ (n beliebig, $n \geq 2k + 1$) mit $2k + 1$ freien Spalten. A ist am Zug. Egal wie er zieht, bietet sich B stets die Möglichkeit, eine Position mit $\leq 2k - 1$ freien Spalten zu erreichen. Setzt A vertikal, d.h. er belegt eine weitere Spalte, besetzt B ebenfalls eine weitere Spalte, wodurch $2k - 1$ Spalten frei bleiben. Zieht A horizontal, kopiert B dessen Zug in der jeweils anderen Zeile, wodurch ebenfalls eine Position mit $\leq 2k - 1$ freien Spalten bleibt. Folglich kann B stets gewinnen.

Die Startposition von $DJM(2,2k+1)$ ist eine Spielposition von $DJM(2,n)$ mit $2k + 1$ freien Spalten ($n = 2k + 1$). Damit ist der Satz bewiesen.



6.3 DJM(2,2k)

Satz 6.4 $DJM(2, 2k)$ kann $\forall k \geq 0$ stets A gewinnen.

Beweis: A besetzt im ersten Zug eine beliebige Spalte und erreicht auf diese Weise eine Position, von der er nach Satz () stets gewinnen kann.

7 Programmbeschreibung und Sourcecode

Ziel war es, ein Programm zu entwickeln, das in der Lage ist, zu einer gegebenen Spielposition (insbesondere der Startposition) eines beliebigen *DJ* bzw. *DJM*-Spiels den Grundywert zu berechnen.

Die Entwicklung gliederte sich im Wesentlichen in 3 Teilbereiche:

1. Repräsentierung von Spielbrettern/Spielpositionen
2. Finden und Verwalten aller Nachfolger einer Spielposition
3. Berechnen der Grundywerte

Zur Realisierung wurde die Programmiersprache JAVA gewählt. Die im folgenden angeführten Ideen und Algorithmen lassen sich jedoch ohne größeren Aufwand auch auf andere Programmiersprachen übertragen.

7.1 Darstellung von Spielbrettern

7.1.1 Board

Ein Spielbrett wird durch das Objekt *Board* dargestellt.

Dieses speichert die Belegung einer beliebigen *DJ(h, w)*-Spielposition (für beliebige *h, w*). Intern wird dies durch ein 2-dimensionales *boolean*-Array gelöst. Das Objekt *Board* enthält weiters zwei *int*-Werte (Höhe, Breite), die über entsprechende *get*-Methoden abgefragt werden können.

Im Folgenden werden einige wichtige Methoden der Klasse *Board* angegeben.

```
import java.util.HashSet;

/**
 * Repräsentierung eines (Domino-)Juvavum Spielfelds.
 *
 * @author Martin Schneider
 */
public class BoardImpl {

    private boolean[][] board;

    private int h; // Höhe

    private int w; // Breite
```

```
/**
 * @param h
 *         Höhe des Spielfelds
 * @param w
 *         Breite des Spielfelds
 */
public BoardImpl(int h, int w) {
    board = new boolean[w][h];
    this.w = w;
    this.h = h;
    fill();
}

/**
 * @return Breite des Spielfelds
 */
public int getWidth() {
    return w;
}

/**
 * @return Höhe des Spielfelds
 */
public int getHeight() {
    return h;
}

/**
 * @param x
 *         horizontale Koordinate
 * @param y
 *         vertikale Koordinate
 * @return true, wenn Feld(x,y) belegt ist; false, wenn frei; true, wenn
 *         Feld außerhalb des Spielfelds
 */
public boolean isSet(int x, int y) {
    boolean ret = false;
    try {
        ret = board[x - 1][y - 1];
    } catch (ArrayIndexOutOfBoundsException e) {
        return true;
    }
}
```

```
        return ret;
    }

/**
 * @param x
 *         horizontale Koordinate
 * @param y
 *         vertikale Koordinate
 * @return true, wenn Feld(x,y) frei ist; false, wenn belegt oder außerhalb
 *         des Spielfelds
 */
public boolean isFree(int x, int y) {
    boolean ret = false;
    try {
        ret = !board[x - 1][y - 1];
    } catch (ArrayIndexOutOfBoundsException e) {
        return false;
    }
    return ret;
}

/**
 * Setzt das Feld an der Position (x,y) des Spielbretts auf true (belegt).
 *
 * @param x
 *         horizontale Koordinate
 * @param y
 *         vertikale Koordinate
 */
public void set(int x, int y) {
    board[x - 1][y - 1] = true;
}

/**
 * Setzt das Feld an der Position (x,y) des Spielbretts auf false (nicht
 * belegt).
 *
 * @param x
 *         horizontale Koordinate
 * @param y
 *         vertikale Koordinate
 */
public void clear(int x, int y) {
```

```
        board[x - 1][y - 1] = false;
    }

/**
 * Erstellt ein Spielfeld-Objekt aus einer Binärkodierung, wie sie z.B.
 * durch flatten() erzeugt wird. Dabei wird der übergebene integer-Wert in
 * eine Binärfolge umgewandelt und die Felder des Spielbretts entsprechend
 * der Werte dieser Folge zeilenweise gesetzt.
 *
 * z.B.: 39 --> 1*2^0 + 1*2^1 + 1*2^2 + 1*2^5--> 1 1 1 0 0 1 0 0 0.<br>
 *
 * Für h=3 und w=3 folgt daraus folgendes Spielbrett: <br>
 *
 *      1 1 1
 *      0 0 1
 *      0 0 0
 *
 * @param binary
 *          Binärkodierung des Spielbretts (als Dezimalzahl)
 * @param w
 *          Breite des Spielbretts
 * @param h
 *          Höhe des Spielbretts
 */
public BoardImpl(long binary, int h, int w) {
    this.h = h;
    this.w = w;

    int x = 0;
    int y = 0;
    this.board = new boolean[w][h];
    String z = decimalToBinary(binary);
    int j = 0;
    for (int i = z.length() - 1; i >= 0; i--) {
        x = j / w;
        y = j % w;
        if (z.substring(i, i + 1).equals("1"))
            board[y][x] = true;
        else
            board[y][x] = false;
        j++;
    }
}
```

```
    }
}

/**
 * Erstellt ein Spielfeld-Objekt aus einer zeilenweise Binärkodierung, wie
 * sie flatten2() erstellt.
 *
 * @param binary
 *         zeilenweise Binärkodierung
 * @param h
 *         Höhe des Spielbretts
 * @param w
 *         Breite des Spielbretts
 */
public BoardImpl(int[] binary, int h, int w) {
    int x = 0;
    int y = 0;
    int l = 0;
    this.board = new boolean[w][h];
    for (int i = 1; i <= h; i++) {
        String z = decimalToBinary(binary[i - 1]);
        l = z.length();
        for (int j = l - 1; j >= 0; j--) {
            if (z.substring(j, j + 1).equals("1"))
                set(l - j, i);
            else
                clear(l - j, i);
        }
    }
}

/**
 * Kopiert die Belegungen eines Spielbretts an eine bestimmte Stelle.
 *
 * @param b
 *         Spielbrett, von dem eingefügt werden soll
 * @param x
 *         x-Koordinate des Punkts, an dem eingefügt werden soll
 * @param y
 *         y-Koordinate des Einfügepunkts
 * @param row
 *         true, wenn in Zeile; false, wenn in Spalte eingefügt werden
 *         soll

```

```
*/
public void pasteFrom(BoardImpl b, int x, int y, boolean row) {
    for (int i = 1; i <= b.getWidth(); i++) {
        if (row) {
            if (b.isSet(i, 1))
                set(x + i - 1, y);
        } else {
            if (b.isSet(i, 1))
                set(x, y + i - 1);
        }
    }
}

/**
 * Gibt eine Binärdarstellung des Spielbretts (als Dezimalzahl) zurück.
 *
 * z.B.:
 *
 * 1 1 1
 * 0 0 0
 * 0 0 1
 *
 * 1 1 1 0 0 0 0 0 1 --> 1*2^0 + 1*2^1 + 1*2^2 + 1*2^8 = 263.
 *
 * @return Binärcodierung des Spielbretts
 */
public int flatten() {
    int length = w * h;
    int value = 0;
    int x, y = 0;
    for (int i = 0; i < length; i++) {
        x = i / w;
        y = i % w;
        if (board[y][x])
            value += Math.pow(2, i);
    }
    return value;
}

/**
 * Gibt eine zeilenweise Binärdarstellung des Spielbretts (als Dezimalzahl)
```

```

* zurück.
*
* z.B.:
*
*   1 1 1
*   0 0 0
*   0 0 1
*
*
* 1 1 1 0 0 0 0 0 1 --> [1*2^0 + 1*2^1 + 1*2^2, 0, 1*2^8] = [7, 0, 256].
*
* @return zeilenweise Binärkodierung des Spielbretts
*/
public int[] flatten2() {
    int[] ret = new int[h];
    int row = 0;
    for (int i = 1; i <= h; i++) {
        row = 0;
        for (int j = 1; j <= w; j++) {
            if (isSet(j, i))
                row += Math.pow(2, j - 1);
        }
        ret[i - 1] = row;
    }
    return ret;
}

/**
 * Füllt das 2-elementige Array, welches das Spielbrett darstellt mit false
 * (nicht belegt) auf.
 */
private void fill() {
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[i].length; j++) {
            board[i][j] = false;
        }
    }
}

/**
 * Gibt das Spielfeld auf der Konsole aus.
 */

```

```
* X ... Feld belegt <br>- ... Feld frei
*/
public void output() {
    for (int i = 0; i < board[0].length; i++) {
        for (int j = 0; j < board.length; j++) {
            if (board[j][i] == true)
                System.out.print("X ");
            else
                System.out.print("- ");
        }
        System.out.println();
    }
    System.out.println();
}

/**
 * @return Liste (HashSet) aller Löcher des Spielbretts
 */
public HashSet getHoles() {
    HashSet v = getHolesRow();
    v.addAll(getHolesColumn());
    return v;
}

/**
 * @return alle Löcher für Zeilen
 */
public HashSet getHolesRow() {
    HashSet holes = new HashSet();
    for (int row = 1; row <= h; row++) {
        int i = 1;
        int start = 1;
        int length = 0;
        while (i <= w) {
            length = 0;
            while (isFree(i, row)) {
                length++;
                i++;
            }
            if (length > 1)
                holes.add(new Hole(start, row, length, true));
            i++;
            start = i;
        }
    }
}
```

```
        }
    }
    return holes;
}

/**
 * @return alle Löcher für Spalten
 */
public HashSet getHolesColumn() {
    HashSet holes = new HashSet();
    for (int column = 1; column <= w; column++) {
        int i = 1;
        int start = 1;
        int length = 0;
        while (i <= h) {
            length = 0;
            while (isFree(column, i)) {
                length++;
                i++;
            }
            if (length > 1)
                holes.add(new Hole(column, start, length, false));
            i++;
            start = i;
        }
    }
    return holes;
}

/**
 * Spiegelt das Spielbrett vertikal
 */
public void fliplr() {
}

/**
 * Spiegelt das Spielbrett horizontal
 */
public void flipud() {
}

/**
 * Dreht das Spielbrett um 180°
 */
```

```
    */
    public void rotate180() {
    }

    /**
     * Dreht das Spielbrett um 270°
     */
    public void rotate270() {
    }

    /**
     * Dreht das Spielbrett um 90° (im Uhrzeigersinn)
     */
    public void rotate90() {
    }

    /**
     * Matrix an der Hauptdiagonale spiegeln
     */
    public void flipd1() {
    }

    /**
     * Matrix an der Nebendiagonale spiegeln
     */
    public void flipd2() {
    }
}
```

7.1.2 Position

Das zweite wichtige Objekt zur Verwaltung der Spielpositionen ist ***Position*** und beinhaltet neben der Spielfeldbelegung (*Board*) die Informationen über alle Nachfolger (in Form einer Liste).

Wichtig ist in diesem Zusammenhang, dass die Liste der Nachfolger nur Verweise auf die entsprechenden *Position*-Objekte (und nicht die Daten selbst) enthält, da es sonst bereits bei kleinen Spielfeldern zu erheblichen Speicherproblemen kommt. Weiters wird darauf geachtet, dass jede Spielposition nur einmal abgespeichert wird.

7.2 Finden aller Nachfolger einer Startposition

Wie bereits erwähnt, bietet die Repräsentierung einer Spielposition die Möglichkeit, Nachfolger zu speichern. Im Folgenden sollen nun 2 Algorithmen beschrieben werden, diese zu finden.

Eine Möglichkeit besteht darin, mit Hilfe einer Schleife das nächste Loch der Länge ≥ 2 zu finden (zuerst Zeilen-, dann Spaltenzüge), es zu füllen und die neue Position als Nachfolger zu speichern, d.h. einen Verweis auf die gefundene Position zur Liste der Nachfolger der ursprünglichen Position hinzuzufügen. Sollte diese Position zu einem früheren Zeitpunkt bereits erstellt worden sein, wird kein neues *Position*-Objekt erzeugt. Von dieser Position kann nun **rekursiv** weitergesucht werden. Dabei muss allerdings berücksichtigt werden, dass eventuell noch weitere Züge in derselben Zeile beziehungsweise Spalte möglich sind, die dann ebenfalls als Nachfolger der ursprünglichen Position vermerkt werden müssen (dies geschieht in den Methoden *moveCurrentRow* sowie *moveCurrentColumn*).

Im Folgenden finden sich nun die wesentlichen Teile dieses Algorithmus im JAVA-Quellcode:

```
// finde Nachfolger
private static void move(Board b, int caller) {
    moveRow(b, caller);    // suche Zeilenzüge
    moveColumn(b, caller); // suche Spaltenzüge
}

// finde Nachfolger mit Zug in Zeile
private static void moveRow(Board b, int caller) {
    int posNr = 0;
    Position tmp = null;
    int i = 0;
    int j = 0;
    int foundPos = -1;

    while (i < b.getHeight()) {
        j = 0;
        while (j + 1 < b.getWidth()) {

            // es wurde eine freie Position
            // (Loch der Länge >= 2 gefunden)

            if (b.isFree(i, j) && b.isFree(i, j + 1)) {
                // in dieses Loch wird ein Domino gesetzt

                b.set(i, j);
                b.set(i, j + 1);
            }
        }
    }
}
```

```
// diese neue Position wird nun in der Liste
// der bereits erzeugten Positionen gesucht

... symmetrische Positionen in Liste suchen ...

foundPos = pos.search(b.flatten());
tmp = new Position(b);

// wird die Position
// (oder eine symmetrische) nicht gefunden
if (foundPos == -1) {
    if (!pos.get(caller).hasChildren(tmp)) {

        // neue Position (tmp) der Liste
        // aller Positionen hinzufügen

        posNr = pos.add(tmp);

        // der Position, von der aus
        // gesucht wurde (caller)
        // die neue Position als
        // Nachfolger hinzufügen (addChild)

        pos.get(caller).addChild(pos.get(posNr));

        // in dieser Zeile weitersuchen
        // (sind Züge mit mehr als einem
        // Domino möglich?)

        moveCurrentRow(b, caller, i);

        // weitere Züge suchen (rekursiver Aufruf)

        move(b, posNr);
    }
}
```

```
    }

    } else {
    // die gefundene Position wurde bereits erzeugt

        // wenn sich die neue Position noch nicht in der
        // Nachfolgerliste der ursprünglichen Position
        // befindet, wird sie dort hinzugefügt

        if (!pos.get(caller).hasChildren(tmp))
            pos.get(caller).addChild(pos.get(foundPos));

        // in dieser Zeile weitersuchen (sind Züge mit
        // mehr als einem Domino möglich)

        moveCurrentRow(b, caller, i);
    }

    // Domino wieder entfernen

    b.clear(i, j);
    b.clear(i, j + 1);

    }
    j++;
}
i++;
}
}

// finde Nachfolger mit Zug in Spalte
private static void moveColumn(Board b, int caller) {

    ... analog zu Zeilenzug ...

}
```

```
// finde alle Nachfolger mit Zug in selber Zeile
private static void moveCurrentRow(Board b,
    int caller, int rowNr) {
    ...
}
```

```
// finde alle Nachfolger mit Zug in selber Spalte
private static void moveCurrentColumn(Board b,
    int caller, int columnNr) {
    ...
}
```

Wendet man diesen Algorithmus rekursiv auf eine Ausgangsposition an, erhält man schließlich einen kompletten Spielegraphen. Es handelt sich dabei um keinen Baum, da jede Spielposition nur einmal vorkommt und folglich Ebenen übersprungen werden können. Zum Beispiel kann eine Position mit zwei Dominos in einer Zeile beziehungsweise Spalte, von der Startecke sowohl in zwei als auch in einem Zug erreicht werden.

Ein anderer Ansatz wäre, alle möglichen Domino-Belegungen für Löcher der Länge l , $2 \leq l \leq \max(h, w)$ zu erzeugen und jede dieser Belegungen dann in die Löcher einer Spielposition zu kopieren, um auf diese Art Nachfolger zu erzeugen. Auch hier muss die Möglichkeit betrachtet werden, dass ein Zug auch in mehreren Löchern gleichzeitig möglich sein kann (ab $w \geq 5$ oder $h \geq 5$).

7.3 Grundywerte berechnen

Haben wir nun ausgehend von einer Startposition einen kompletten Spielegraphen erzeugt (d.h. alle Blätter sind Zielecken), können wir uns folgenden Algorithmus zur Berechnung der Grundywerte überlegen.

Dazu rufen wir uns erneut die Definition des Grundywertes einer Ecke in Erinnerung: Der Grundywert einer Spielposition **POS** ist die kleinste Zahl aus \mathbb{N}_0 , die bei keinem direkten Nachfolger von **POS** vorkommt.

Sie lässt sich mittels folgendem Algorithmus finden.

```
/**
 * Berechnet den Grundywert der Spielposition für das
```

```
* gewöhnliche Spiel (rekursiv)
* @return Grundywert der aktuellen Spielposition
*/
public int getGrundy() {
    if (special != -1)
        // Grundywert oder Abschätzung (>0)
        // durch Vorinformation bekannt
        return special;

    if (grundy != -1)
        // Grundywert bereits bekannt (gerechnet)
        return grundy;

    if (children.isEmpty()) { // Zielecke
        this.grundy = 0;
        return 0;

    } else {
        int i = 0;
        int grundy = -1;
        while (grundy == -1) {
            boolean found = false;
            for (Enumeration e1 = children.elements(); e1.hasMoreElements();) {
                if (((Position) e1.nextElement()).getGrundy() == i) {
                    found = true; // Grundywert i kommt bei einem Nachfolger
                                // bereits vor
                    break;
                }
            }
            if (!found) // i ist die kleinste Zahl, die bei keinem Nachfolger
                       // vorkommt
                grundy = i;
            i++;
        }
        this.grundy = grundy;
        return grundy;
    }
}
```

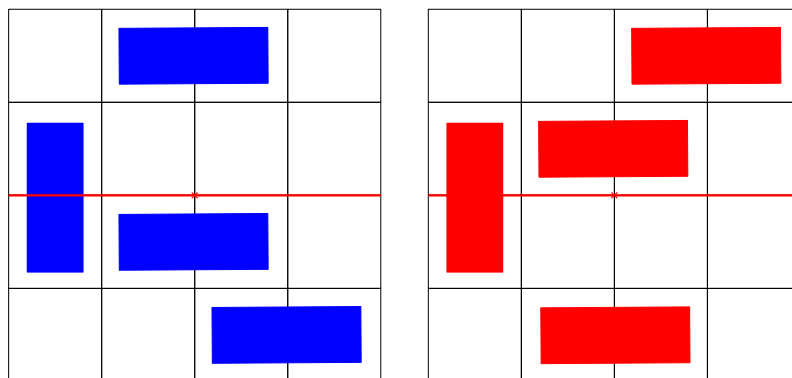
7.4 Bemerkungen

7.4.1 Symmetrien

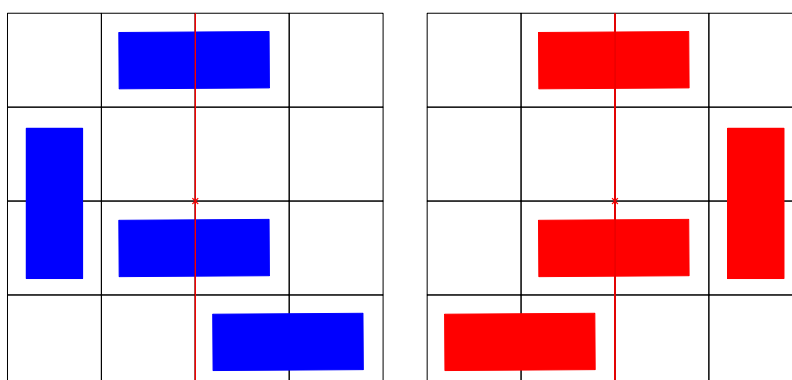
Um die Anzahl der zu untersuchenden Spielpositionen (drastisch) zu reduzieren (siehe Tabelle), wird zwischen einer Position **POS** und einer Position, die aus **POS** durch Drehungen und/oder Spiegelungen entsteht nicht unterschieden. Dabei werden folgende Symmetrien berücksichtigt:

1. horizontales Spiegeln um eine Kante oder Zeile
2. vertikales Spiegeln um eine Kante oder Spalte
3. Drehung um 180 Grad

Ist die Anzahl der Zeilen/Spalten gerade, wird um die Kante zwischen den beiden mittleren Zeilen/Spalten gespiegelt. Ist sie ungerade wird um die mittlere Zeile/Spalte gespiegelt, wobei diese unverändert bleibt.

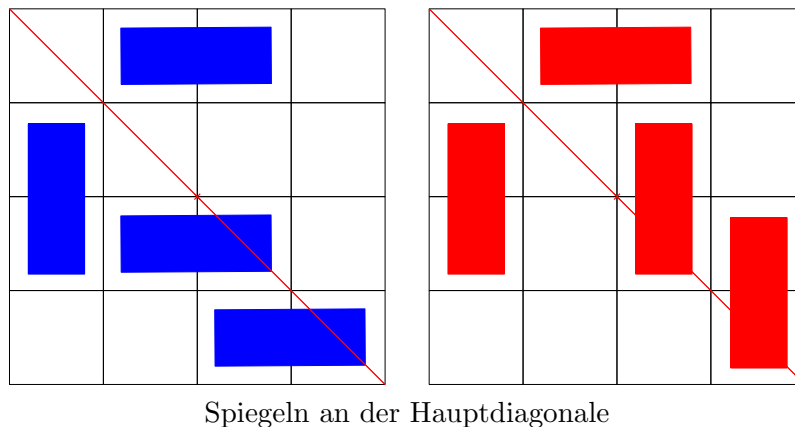


horizontales Spiegeln



vertikales Spiegeln

Bei quadratischen Spielfeldern werden zusätzlich Drehungen um 90 und 270 Grad sowie das Spiegeln an den beiden Diagonalen berücksichtigt, sodass es zu jeder Spielposition 7 symmetrische Positionen gibt. Diese müssen im Einzelfall nicht alle verschieden sein.



Spiegeln an der Hauptdiagonale

Bemerkung: Man kann die zueinander symmetrischen Spielpositionen eines quadratischen Spielfelds auch als Deckabbildungen eines Quadrats in der Ebene mit Mittelpunkt $M = (0, 0)$ betrachten. Bezeichne a die Drehung um 90 Grad, b die Spiegelung des Quadrats um die x -Achse und e die Identität. Dann ist

$$G = \{e, a, a^2, a^3, b, a \circ b, a^2 \circ b, a^3 \circ b\}$$

die Menge der Deckabbildungen dieses Quadrats. G bildet eine nichtabelsche Gruppe und wird als Diedergruppe (D_4) bezeichnet.

Um den Gewinn zu verdeutlichen, der durch das Betrachten von Symmetrien erzielt wird, betrachten wir abschließend folgende Tabelle, in der die Anzahl der zu untersuchenden Spielpositionen (für ausgewählte Spielfeldgrößen) mit und ohne Berücksichtigung von Symmetrien sowie das Verhältnis dieser beiden Werte (auf zwei Dezimalstellen gerundet) angegeben sind:

Spielfeld	Anzahl Spielpositionen ohne Symmetrien	Anzahl Spielpositionen Symmetrien berücksichtigt	Verhältnis
1 × 2	2	2	1.00
1 × 3	3	2	0.67
1 × 4	5	4	0.80
1 × 5	8	5	0.63
1 × 6	13	9	0.69
1 × 7	21	12	0.57
2 × 2	6	3	0.50
2 × 3	18	9	0.50
2 × 4	54	22	0.41
2 × 5	162	56	0.35
2 × 6	486	151	0.31
2 × 7	1458	419	0.29
3 × 3	98	18	0.18
3 × 4	550	164	0.30
3 × 5	3054	805	0.26
3 × 6	17014	4414	0.26
3 × 7	94682	23937	0.25
4 × 4	5700	778	0.14

Anzahl der Spielpositionen

Wie man dieser Tabelle entnehmen kann, wird die Anzahl der zu untersuchenden Positionen in manchen Fällen auf bis zu ein Viertel reduziert. Bei quadratischen Spielfeldern ist die Ersparnis (wie erwartet) noch größer, da auch Drehungen um 90 bzw. 270 Grad betrachtet werden können.

Die Berücksichtigung der Symmetrien wird realisiert, indem beim Erzeugen des Spielegraphen nicht nur geprüft wird ob, eine als Nachfolger gefundene Spielposition, sondern auch ob eine zu dieser Position symmetrische Position bereits in der Liste der Spielpositionen vorhanden ist. Die Methoden, um ein Spielbrett zu drehen beziehungsweise zu spiegeln, werden im Folgenden angegeben und finden sich in der Klasse *Board*.

```
/**
 * Spiegelt das Spielbrett vertikal
 */
public void fliplr() {
    int dy = board.length;
    int dx = board[0].length;
    boolean help;
    for (int xi = 0; xi < dx; xi++) {
        for (int yi = 0; yi < dy / 2; yi++) {
            help = board[yi][xi];
```

```
        board[yi][xi] = board[dy - 1 - yi][xi];
        board[dy - 1 - yi][xi] = help;
    }
}

/**
 * Spiegelt das Spielbrett horizontal
 */
public void flipud() {
    int dy = board.length;
    int dx = board[0].length;
    boolean help;
    for (int xi = 0; xi < dx / 2; xi++) {
        for (int yi = 0; yi < dy; yi++) {
            help = board[yi][xi];
            board[yi][xi] = board[yi][dx - 1 - xi];
            board[yi][dx - 1 - xi] = help;
        }
    }
}

/**
 * Dreht das Spielbrett um 180°
 */
public void rotate180() {
    flipud();
    fliplr();
}

/**
 * Dreht das Spielbrett um 270°
 */
public void rotate270() {
    boolean[][] help = new boolean[w][h];
    for (int i = 1; i <= h; i++) {
        for (int j = 1; j <= w; j++) {
            help[j - 1][w - i] = isSet(i, j);
        }
    }
    board = help;
}
```

```
/**
 * Dreht das Spielbrett um 90° (im Uhrzeigersinn)
 */
public void rotate90() {
    boolean[][] help = new boolean[w][h];
    for (int i = 1; i <= h; i++) {
        for (int j = 1; j <= w; j++) {
            help[h - j][i - 1] = isSet(i, j);
        }
    }
    board = help;
}

/**
 * Spielfeld an der Hauptdiagonale spiegeln
 */
public void flipd1() {
    boolean help;
    for (int i = 1; i < board.length; i++){
        for (int j = 0; j < i; j++) {
            help = board[i][j];
            board[i][j] = board[j][i];
            board[j][i] = help;
        }
    }
}

/**
 * Spielfeld an der Nebendiagonale spiegeln
 */
public void flipd2() {
    boolean help;
    for (int i = 0; i < board.length; i++){
        for (int j = 0; j < w-1-i; j++) {
            help = board[i][j];
            board[i][j] = board[h-1-j][w-1-i];
            board[h-1-j][w-1-i] = help;
        }
    }
}
```

Durch das Berücksichtigen von Symmetrien lässt sich die Anzahl der zu untersuchenden

Positionen zwar deutlich verringern (siehe Tabelle), will man den Spielegraphen jedoch dazu verwenden, einen guten nächsten Zug (d.h. eine Position mit Grundywert 0) anzugeben, kann folgendes Problem auftreten: Es kann passieren, dass der Nachfolger einer Spielposition keine gültige Zugmöglichkeit darstellt. In diesem Fall könnte man zum Beispiel in den Positionsobjekten zusätzliche Informationen abspeichern, die angeben, wie die Nachfolgeposition zu drehen beziehungsweise zu spiegeln ist.

Da das Hauptziel des vorliegenden Programms jedoch das Berechnen von Grundywerten ist, wurde dieser Zusatz vernachlässigt. Er kann jedoch ohne größeren Aufwand in das bestehende Programm implementiert werden.

8 Literaturangaben

- Vorlesungsunterlagen Projektpraktikum Prof. Gerl (WS 2004/05, SS 2005)
- Jörg Bewersdorff: Glück, Logik und Bluff: Mathematik im Spiel - Methoden, Ergebnisse und Grenzen. Wiesbaden, 2003
- Kombinatorische Spieltheorie. http://de.wikipedia.org/wiki/Kombinatorische_Spieltheorie
- Peter Hellekalek: Algebraische Strukturen. Skriptum Mai 2005