UNIKASSEL
VERSITÄT

Fixpunktiteration für Paritätsspiele
Fixpoint iteration for parity games

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
an der Universität Kassel

vorgelegt von
Michael Falk

ausgegeben und betreut von
Prof. Dr. Martin Lange
Fachgebiet Formale Methoden und Software-Verifikation

Zweitgutachter
Prof. Dr. Friedrich Otto
Fachgebiet Theoretische Informatik

Kassel, den 28. Juni 2013

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig und ausschließlich unter Verwendung der angegebenen Quellen angefertigt habe.

_____

Inhaltsverzeichnis

1 Einführung

Mit dem Model Checking werden Eigenschaften eines Systems automatisch geprüft. Bei diesem Vorgang liegt die Systembeschreibung M in einer formalen Sprache (bspw. als Transitionssystem) und die zu prüfende Eigenschaft als logische Formel ϕ vor. Gesucht ist nun die Antwort auf die Frage, ob die Formel die Systembeschreibung erfüllt, formal also, ob $M \vDash \phi$ gilt.

Dieses Model Checking Problem lässt sich auf das Lösen eines Paritätsspiels reduzieren. Dazu erzeugt man aus M, ϕ das zwei Personen Paritätsspiel $G(M, \phi)$ bei dem Spieler 0 zeigen soll, dass $M \vDash \phi$ und der Gegner, Spieler 1, gerade das Gegenteil, also $M \nvDash \phi$ zeigen soll. Dann ist es möglich, das Paritätsspiel G derart zu erzeugen, dass folgende Äquivalenz gilt

$G \vDash \phi \Leftrightarrow$ Spieler 1 hat eine Gewinnstrategie im Paritätsspiel $G(M, \phi)$ [Sti95]

Paritätsspiele zu lösen ist ein wichtiges Problem im Bereich formaler Methoden, da sich nicht nur Model Checking Probleme, sondern viele andere Verifikationsprobleme, wie z. B. Erfüllbarkeitstests oder Synthese darauf reduzieren lassen.

Am Fachgebiet Formale Methoden und Software-Verifikation wird PGSolver, bei dem es sich um ein umfangreiches Tool zum Lösen von Paritätsspielen handelt, entwickelt. PGSolver enthält verschiedene Algorithmen zur Lösung dieses Problems.

Ziel dieser Arbeit ist, den Fixpunktiterationsalgorithmus zu optimieren und unter Verwendung der funktionalen Programmiersprache OCaml zu implementieren.

2 Paritätsspiele

2.1 Definition

Paritätspiele werden auf gerichteten, totalen Graphen von zwei Akteuren gespielt. Im Rahmen dieser Arbeit sind Spieler 0 und Spieler 1 namentlich diese Spieler. Abbildung 1 zeigt eine konkrete Ausprägung eines Paritätsspiels.

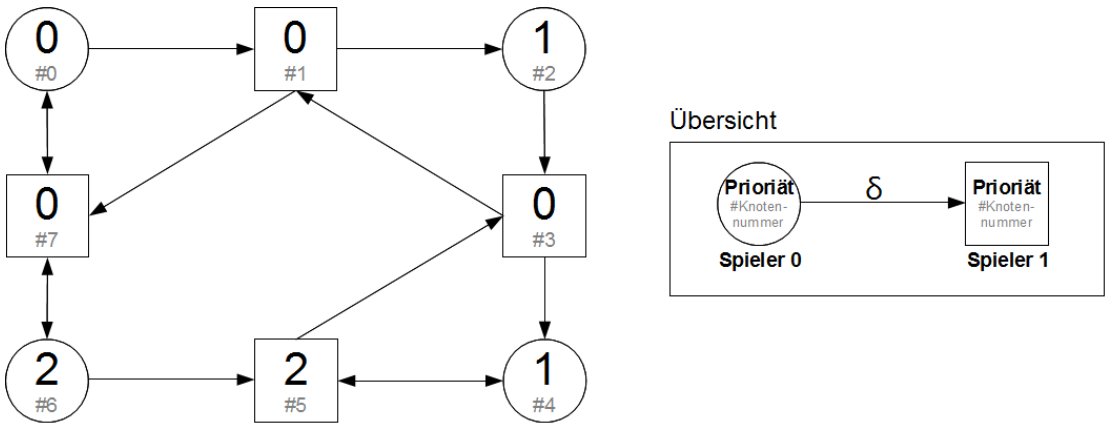

Abbildung 1: Paritätsspiel als gerichteter Graph

Im Graphen sind Positionen, die Spieler 0 gehören, als Kreise dargestellt, alle anderen Knoten im Graphen sind rechteckig und Spieler 1 zugeordnet. Innerhalb dieser Knoten ist eine groß dargestellte Ziffer, die seine Priorität angibt und eine kleiner geschriebene Ziffer, welche die Nummer des Knotens zu seiner eindeutigen Identifizierbarkeit angibt, eingetragen. Während des unendlich langen Spielablaufs ziehen die Spieler den Kanten entlang von Knoten zu Knoten, wobei der Besitzer des aktuell aufgesuchten Knotens am Zug ist. Wenn die größte Priorität, die dadurch unendlich oft durchschritten wird, gerade ist, gewinnt Spieler 0, andernfalls Spieler 1.

Formal dargestellt werden kann ein Paritätsspiel durch ein vier-Tupel $G = (V_0, V_1, \delta, \Omega)$ mit sämtlichen Knoten $V = V_0 \cup V_1$ für welche $V_0 \cap V_1 = \emptyset$ gilt. Die zweistellige Relation $\delta \subseteq \{(a, b) : a, b \in V\}$ beinhaltet alle gültigen Züge. Mit der Funktion $\Omega : V \to \mathbb{N}$ wird jeder Knoten auf eine Priorität abgebildet. Wir fordern $\forall x \in V. \exists y \in V. \delta(x, y)$, sodass jeder Knoten mindestens einen Nachfolger hat. Die Folge der Spielzüge beschreibt eine Partie $p = v_0, v_1, v_2, ...$ mit der sich auch der Gewinner ermitteln lässt. Dazu betrachtet man die Folge der Abbildungen aller Elemente von p über der Funktion Ω also $\Omega(v_0), \Omega(v_1), \Omega(v_2), ...$ Spieler 0 gewinnt

eine Partie genau dann, wenn die in dieser Folge größte, unendlich oft auftretende Zahl, also die Priorität, gerade ist.

2.2 Begriffe

Mit einer *Partie* ist ein Pfad im Graphen (V, δ) gemeint. Die Menge aller Partien ist durch die Menge $\{p \in V^\omega : p \text{ ist eine Partie auf } G\}$ gegeben. Dabei wird mit $V^\omega := \{\alpha : \mathbb{N} \to V\}$ die Menge aller unendlichen Wörter bzw. Partien über V bezeichnet.

Eine *Strategie* ist eine Abbildung $f : V^* \to V$, eine *positionale Strategie* ist durch die Abbildung $s : V \to V$ gegeben, weist also für jeden Knoten des Spielers den nachfolgenden Knoten gemäß dieser Strategie zu, wobei $(s, v(s)) \in \delta$. Der Definitionsbereich von f sind Pfade und es kann somit die Historie des Weges berücksichtigt werden. Im Gegensatz dazu sind es bei der Funktion s einzelne Positionen die abgebildet werden, weshalb es keine Rolle spielen kann, wie der Spieler dorthin gekommen ist. Eine Partie $p = v_0, v_1, v_2, ...$ ist mit der Strategie s übereinstimmend oder konform, falls $\forall v_{k:\, 0 \leq k \leq |p|-1} \in p : s(v_k) = v_{k+1}$ gilt. Gibt es eine Position v_k, von der Spieler i ausgehend durch die Strategie s sämtliche Spiele gewinnt, nennt man v_k *Gewinnposition* und die Funktion s *Gewinnstrategie*. Alle Gewinnpositionen von Spieler 0 werden in der Menge W_0 zusammengefasst. Für Spieler 1 werden die Gewinnpositionen in der Menge W_1 zusammengefasst. Es gilt das folgende Theorem:

> Es gibt keinen Knoten, für den es keine Gewinnstrategie gibt (positionale Determiniertheit) [Zie98]

Ein *Attraktor* für einen Spieler i einer Menge $U \subseteq V$ ist diejenige Menge $Attr_i(U)$, innerhalb derer Spieler i einen Besuch der Menge U erzwingen kann, weil der Gegner nicht vom Weg dorthin abkommen kann. Es gilt (wobei durch $\overline{i} = 1 - i$ der Gegenspieler von i benannt wird):

$$Attr_i(U) := \bigcup_{k \in \mathbb{N}} Attr_i^k(U)$$

mit

$$\begin{aligned} Attr_i^0(U) &:= U \\ Attr_i^{k+1}(U) &:= Attr_i^k(U) \\ &\quad \cup \left\{ p \in V_i \mid \exists v \in Attr_i^k(U)\, mit\, (p,\, v) \in \delta \right\} \\ &\quad \cup \left\{ p \in V_{\overline{i}} \mid \forall v \in V. (p,\, v) \in \delta \Rightarrow v \in Attr_i^k(U) \right\} \end{aligned}$$

Das Paritätsspiel in Abbildung 1 kann mit

$G = (V, V_0, V_1, \delta, \Omega)$
$max = max\{\Omega(v) :\ v \in V\}$
$min = min\{\Omega(v) :\ v \in V\}$
$P_i = \{v :\ \Omega(v) = i\}$

und den Mengen

$V = \{0, 1, 2, 3, 4, 5, 6, 7\}$
$V_0 = \{0, 2, 4, 6\}$
$V_1 = V \setminus V_0 = \{1, 3, 5, 7\}$
$P_0 = \{0, 1, 3, 7\}$
$P_1 = \{2, 4\}$
$P_2 = \{5, 6\}$

sowie den Werten

$max = 2$
$min = 0$

sowie δ wie in Abbildung 1 dargestellt, formal beschrieben werden.

Durch Hinzunahme der Symbole $\Box$ (Box) für den Notwendigkeitsoperator und $\Diamond$ (Diamond) für den Möglichkeitsoperator zur Aussagenlogik entsteht die Menge der modallogischen Formeln. Im Zusammenhang mit dem Fixpunktiterationsalgorithmus wird der Diamond-Operator als Menge von Knoten interpretiert, die mindestens einen Nachfolger innerhalb einer spezifische Menge haben. Der Box-

Operator fordert, dass alle Nachfolger in dieser Menge sein müssen. Nun gilt also $\Diamond T = \{s \mid \exists t \in T : \ s \to t \in \delta\}$ und $\Box T = \{s \mid \forall t : \ s \to t \in \delta \Rightarrow t \in T\}$. Durch $V_0 \cap \Diamond T$ wird also die Menge von Knoten beschrieben, auf denen Spieler 0 die Möglichkeit hat, einen Besuch der Menge T zu erzwingen. Innerhalb der Menge $V_0 \cap \Box T$ hätte Spieler 0 allerdings keine andere Wahl, als die Menge T zu betreten.

2.3 Fixpunktiteration

Der Fixpunktiterationsalgorithmus [Wal02], gelistet als Pseudocode in Algorithmus 1, findet zu einem gegebenen Paritätsspiel den Gewinnbereich für Spieler 0 und gibt diesen zurück. Die einzelnen X_i sind globale Variablen, der Parameter i entspricht der höchsten Priorität des Paritätsspiels.

Um zu zeigen, wie der Algorithmus arbeitet, übergeben wir zunächst als einzigen Parameter die Zahl i. Üblicherweise ist zu Beginn $i \geq min$ (min is die niedrigste im Spiel vorkommende Priorität), weshalb der else-Zweig in Zeile 3 betreten wird. Dort werden die globalen X_i alternierend mit der Menge aller Knoten und der leeren Menge initialisiert. Jeweils nach einer Initialisierung wird die Schleife in Zeile 5 betreten, welche nach einem Fixpunkt sucht, die nämlich solange durchlaufen wird, bis $X_i = X_i'$ gilt. Innerhalb eines jeden Durchlaufs dieser Schleife wird eine Kopie der Menge X_i in der Menge X_i' gesichert. X_i enthält dann das Ergebnis der aktuellen Berechnung, X_i' hingegen das der Vorherigen. Die unmittelbar nachfolgende Anweisung veranlasst, solve mit $i-1$ als Parameter aufzurufen. Durch diese rekursiven, wiederkehrenden Aufrufe wird der Parameter i den Wert von min irgendwann unterschreiten, wodurch der Weg in den if-then-Zweig in Zeile 1 frei wird. Diesen betreten, führt zu einer Berechnung, dessen Ergebnis als Rückgabewert in X_i abgelegt wird. Der Rückgabewert wird, wie bereits beschrieben, mit dem Vorherigen (durch X_i' repräsentiert) verglichen. Solange also $X_i \neq X_i'$ wird solve immer wieder rekursiv aufgerufen. Gilt aber endlich $X_i = X_i'$, wird der just gefundene Fixpunkt zurückgegeben und in der Menge X_{i+1} gespeichert. Es wird nun geprüft, ob auch ein Fixpunkt für X_{i+1} gefunden wurde, formal ob $X_{i+1} = X_{i+1}'$ gilt. Falls ja, wird der letzte Schritt analog wiederholt: der Fixpunkt X_{i+1} wird in X_{i+2} gespeichert und so weiter. Falls nein, wird die Fixpunktiteration für X_i erneut gestartet - mit dem Unterschied, dass sich der Inhalt in X_{i+1} verändert hat, dort wurde nämlich der eben gefundene Fixpunkt für X_i eingetragen. Bringt die erneute Fixpunktiteration für X_i den nächsten Fixpunkt hervor, geht es genauso weiter, wie nach der ersten erfolgreichen Fixpunktberechnung für X_i.

Bereits im vorgehenden Kapitel wurde eine Interpretationsmöglichkeit für die

Formel in Zeile 2, welche die Ergebnisse für die Variablen X_i liefert vorgestellt. Demnach kann $\left(V_0 \cap \Diamond \left(\bigcup_{j=min}^{max} P_j \cap X_j\right)\right) \cup \left(V_1 \cap \Box \left(\bigcup_{j=min}^{max} P_j \cap X_j\right)\right)$ also als Menge von Knoten verstanden werden, die, wenn sie zu Spieler 0 gehören, mindestens einen Nachfolger in $M = \bigcup_{j=min}^{max} P_j \cap X_j$ besitzen und falls sie zu Spieler 1 gehören, alle Nachfolger in dieser Menge M sehen. Es sind also Knoten, von denen aus Spieler 0 im laufenden Spiel die Menge M betreten *kann*, während Spieler 0 von diesen aus die Menge M betreten *muss*.

Algorithmus 1 Fixpunktiteration

```
0  solve (i){
1      if (i < min) then
2          return (V_0 ∩ ◇(⋃_{j=min}^{max} P_j ∩ X_j)) ∪ (V_1 ∩ □(⋃_{j=min}^{max} P_j ∩ X_j))
3      else
4          if (i gerade) then X_i = V else X_i = ∅
5          repeat
6              X'_i := X_i
7              X_i := solve (i − 1)
8          until (X_i = X'_i)
9          return X_i
10 }
```

Betrachten wir ein konkretes Beispiel, bei dem das Paritätsspiel in Abbildung 1 auf den Algorithmus 1 bis einschließlich zu dem Punkt, an dem der Algorithmus erstmalig die Berechnung in Zeile 2 durchführt, angesetzt wird. Der zu übergebende Parameter ist $i = max = 2$.

Zu Beginn der Laufzeit der Algorithmusabarbeitung wird der else-Zweig in Zeile 3 betreten, weil $i < min$ falsch ist. Dort werden die einzelnen X_i mit der Menge aller Knoten bzw. der leeren Menge initialisiert - je nachdem ob die Variable i gerade oder ungerade ist. Sobald i den Wert der kleinsten Priorität unterschreitet - in unserem Fall wenn $i < 0$ gilt, wird X_0 erstmals berechnet. Dazu muss zuerst die Menge $\bigcup_{j=0}^{max} P_j \cap X_j$ gebildet werden. Zum gegenwärtigen Zeitpunkt ist das die Menge G der Knoten(nummern), die einer gerade Prioritäten zugeordnet sind, mit $G = \{0, 1, 3, 5, 6, 7\}$. Unmittelbar danach wird auf dieser Menge der Diamond-

Operator, gefolgt von einem Schnitt mit der Menge V_0, ausgeführt. Es werden also Knoten des Spielers 0 gesucht, die einen Nachfolger in der Menge G, welche genau die Knoten mit gerader Priorität enthält, besitzen. Analog wird durch den Box-Operator die Menge von Knoten gefunden, die Spieler 1 gehören und von denen alle Nachfolger in G zu finden sind. Das Ergebnis dieser Formel, $\{0, 2, 4, 6, 7\}$, wird nun der globalen Variablen X_2 zugewiesen. Diese Menge $\{0, 2, 4, 6, 7\}$ lässt sich als Sammlung von Knoten oder als Ausschnitt des Spielfeldes sehen, von welchem aus Spieler 0 die Menge G besuchen kann während Spieler 1 diese Menge G zwingend besuchen muss. Es werden also insbesondere auch aus V diejenigen Knoten herausgenommen, welche G in einem Schritt (über eine Kante) verlassen *müssen* aus Sicht von Spieler 0 und *können* aus der Perspektive von Spieler 1. Die Menge $\{0, 2, 4, 6, 7\}$ enthält also ausschließlich Knoten, von denen aus Spieler 0 ein Besuch in die Menge G der geraden Prioritäten erzwingen kann. Dies erinnert stark an die Definition des Attraktors. Allerdings müsste dann per Definition $\{0, 2, 4, 6, 7\} \subseteq G$ gelten, weil $Attr_i^0(G) := G$ im 0. Schritt gesetzt wird. Definiert man den Attraktor für den 0. Schritt mit $Attr_i^0(U) := \emptyset$, passt das Konzept auch an diese Stelle. Nun kann man sagen, dass $\{0, 2, 4, 6, 7\} = Attr_0^1(G)$.

Würde man das Beispiel fortsetzen, ergäben sich die Zwischenergebnisse, welche die folgende Tabelle auflistet:

Zeilennummer	i	X_{min}	X_{min+1}	X_{min+2}
1	2			V
2	1		$\emptyset$	
3	0	V		
4	-1	$\{0, 2, 4, 6, 7\}$		
5	-1	$\{0, 4, 6, 7\}$		
6	-1	$\{0, 4, 6, 7\}$		
7	1		$\{0, 4, 6, 7\}$	
8	0	V		
9	-1	$\{0, 2, 3, 4, 5, 6, 7\}$		
10	-1	$\{0, 2, 4, 5, 6, 7\}$		
11	-1	$\{0, 4, 6, 7\}$		
12	-1	$\{0, 4, 6, 7\}$		
13	1		$\{0, 4, 6, 7\}$	
14	2			$\{0, 4, 6, 7\}$
15	1		$\emptyset$	
16	0	V		
17	-1	$\{0, 2, 6, 7\}$		
18	-1	$\{0, 6, 7\}$		
19	-1	$\{0, 6, 7\}$		
20	1		$\{0, 6, 7\}$	
21	0	V		
22	-1	$\{0, 2, 6, 7\}$		
23	-1	$\{0, 6, 7\}$		
24	-1	$\{0, 6, 7\}$		
25	1		$\{0, 6, 7\}$	
26	2			$\{0, 6, 7\}$
27	1		$\emptyset$	
28	0	V		
29	-1	$\{0, 2, 6, 7\}$		
30	-1	$\{0, 6, 7\}$		
31	-1	$\{0, 6, 7\}$		
32	1		$\{0, 6, 7\}$	
33	0	V		
34	-1	$\{0, 2, 6, 7\}$		
35	-1	$\{0, 6, 7\}$		
36	-1	$\{0, 6, 7\}$		
37	1		$\{0, 6, 7\}$	
38	2			$\{0, 6, 7\}$

Tabelle 1: Zwischenergebnisse Algorithmus 1

3 Optimierung des Fixpunktiterationsalgorithmus

In sechs Punkten konnte der Algorithmus durch gezielte Veränderungen seine Laufzeit verbessern, oder durch äquivalente Umformungen als bessere Schablone für die Implementierung dienen. Damit das nachfolgend Geschriebene einfacher über das konkrete Beispiel gelegt und so miteinander verglichen werden kann, muss für min der Wert 0 eingesetzt werden.

1. Ist in Spalte X_{min} in Tabelle 1 die berechnete Menge mit der zuvor berechneten identisch, wird diese Menge X_{min+1} zugewiesen. Falls diese Menge X_{min+1} wiederum identisch mit der letztmalig an X_{min+1} zugewiesenen Menge ist, würde diese sofort an X_{min+2} zugewiesen und so weiter. Man kann den Vorgang, dass unter den genannten Bedingungen (Mengengleichheit) die Mengen X_{j-1} an X_j zugewiesen werden, auch so beschreiben, dass man an Stelle von X_j die Menge X_{min} anlegt. Es wird also letztlich die Variable X_{min} so lange an die nachfolgenden X_j durchgereicht, bis die zu vergleichenden Variablen bzw. Mengen nicht mehr identisch sind. Im Algorithmus 1 wird dieser Vorgang durch die Zeilen 5 bis 8 erzwungen. Eine Folgerung daraus, auf die wir später noch zurückgreifen werden, ist, dass es mit dem Ergebnis gleichbedeutend ist, wenn in Zeile 9 statt X_i der Wert von X_{min} zurückgegeben wird. Denn der Algorithmus endet, wenn der Vergleich $X'_{max} = X_{max}$ wahr ist. Unmittelbar zuvor wurde aber X_{max} der Wert von X_{max-1}, der dem von X_{min} entsprechen muss, zugewiesen.

2. Die Tabelle 1 lässt durch die vermerkten Mengen in den Zeilen 17, 22, 29 und 34 erkennen, dass der Algorithmus augenscheinlich oft gleiche oder zumindest sehr ähnliche Berechnungen durchführt. Eine Analyse dieser Auffälligkeit hat die Erkenntnis eingebracht, mit der dieses Phänomen durch eine gleiche "Berechnungsumgebung" erklärbar wird. Damit ist gemeint, dass in den erwähnten Fällen die Variablen X_{min}, X_{min+1}, X_{min+2} zwar teilweise unterschiedlich sind, aber die für die weiteren Rechenschritte relevanten Zwischenergebnisse, also $X_{min} \cap P_{min}$, $X_{min+1} \cap P_{min+1}$, $X_{min+2} \cap P_{min+2}$ identisch mit den jeweiligen Vorgängern sind. Es wäre also folglich wesentlich effizienter, würde man nicht die einzelnen $X_{j>min}$ (mit der abkürzenden Notation $X_{j>min}$ sind alle X gemeint, deren Index j größer als min ist) in der bisherigen Form speichern, sondern sie zuvor mit P_j schneiden. Das ist auch deswegen möglich, weil die $X_{j>min}$ außer für den Schnitt mit dem zugehörigen P_j nie wieder ausgelesen werden (die Anweisung "return X_j" haben wir bereits durch "return X_{min}" gleichwertig er-

setzt). Die Einschränkung $j > min$ ist notwendig, denn die Menge X_{min} muss erhalten bleiben, weil innerhalb derer die eigentlichen Berechnungen stattfinden und diese Variable tatsächlich ausgelesen wird.
Diese Veränderung des Fixpunktiterationsalgorithmus wird seine Laufzeit deutlich minimieren, weil Berechnungen eingespart werden können und die Mengen $X_{j>0}$ durch den Schnitt mit dem zugehörigen P_j kleiner werden können, was sich wiederum auf weitere Berechnungen auswirkt.

3. Für die effiziente Implementierung des Fixpunktiterationsalgorithmus könnte es gewinnbringend sein, die Konstruktion in Zeile 2, nämlich $\bigcup_{j=min}^{max} P_j \cap X_j$ genauer zu prüfen. In der Tat wäre es unnötig aufwendig, sich an diese Vorschrift zu halten. Die Zwischenergebnisse in Tabelle 1 lassen erkennen, dass, solange $X_{min} \neq X'_{min}$ gilt, es keine Notwendigkeit gibt, die $\bigcup_{j=min+1}^{max} P_j \cap X_j$ sondern einzig $P_{min} \cap X_{min}$ in jedem Schritt neu zu berechnen. Der Grund ist, dass die Variablen $X_{j>min}$ unter der genannten Bedingung ihren Wert beibehalten und die einzelnen P_j allesamt konstant sind. In Punkt 5 wird diese Idee weiterentwickelt.

4. Nutzt man darüber hinaus noch die Eigenschaft $\Diamond(S \cup T) = \Diamond S \cup \Diamond T$ aus, ergibt sich noch eine weitere Möglichkeit der Optimierung. Anstelle von $\Diamond\left(\bigcup_{j=min}^{max} P_j \cap X_j\right)$ orientieren wir uns für die Implementierung an $\Diamond(P_{min} \cap X_{min}) \cup \Diamond(P_{min+1} \cap X_{min+1}) \cup ... \cup \Diamond(P_{max} \cap X_{max})$. Mit der selben Argumentation, weswegen wir im vorigen Abschnitt festgestellt hatten, dass einzig $P_0 \cap X_0$ in jedem Schritt neu zu berechnen ist, lässt sich auch der Diamond-Operator auf diese Weise sparsamer einsetzen. Es muss natürlich nur $\Diamond(P_0 \cap X_0)$ neu berechnet werden - die Ergebnisse für $\Diamond(P_j \cap X_j)$ mit $j > 0$ sollten einmal ermittelt, dann abgespeichert und erst dann wieder neu berechnet werden, wenn sich auch die relevanten X_j geändert haben.

5. Untersucht man die Optimierung aus Punkt 4, wird man für ein großes max viele Vereinigungen in Zeile 2 des Algorithmus vornehmen müssen. Statt $\Diamond(P_{min} \cap X_{min}) \cup \Diamond(P_{min+1} \cap X_{min+1}) \cup ... \cup \Diamond(P_{max} \cap X_{max})$ immer neu zu berechnen, wäre es günstiger, dies einzig durch $\Diamond(P_{min} \cap X_{min}) \cup D_{min+1}$ wobei

$$D_j = \begin{cases} j = max: & \Diamond(P_{max} \cap X_{max}) \\ j < max: & \Diamond(P_j \cap X_j) \cup D_{j+1} \end{cases}$$

zu ersetzen. Somit gilt $D_{min+1} \supseteq D_{min+2} ... \supseteq D_{max}$ und natürlich $D_{min+1} =$

$\Diamond(P_{min+1} \cap X_{min+1}) \cup ... \cup \Diamond(P_{max} \cap X_{max})$. Der Vorteil ist nun, dass im Vergleich zu Punkt 4 Vereinigungen eingespart werden können. Nun würde demnach in Zeile 2 nur noch die Menge aus Variable D_{min+1} ausgelesen werden; die Vereinigungen müssen nicht immer erneut berechnet werden. Ein weiterer Vorzug dieser Optimierung ist, dass D_{min+1} selbst aber nur immer soweit neu berechnet wird, wie es notwendig ist. Ändert sich beispielsweise D_k würde man lediglich D_{min+1}, ..., D_k neu berechnen müssen, denn $D_k \cup D_{k+1} = D_{min+1}$ (Die Werte für $D_{l>k}$ sind schließlich unverändert geblieben). Wichtig zu erkennen ist, dass sich die $X_{j<k}$ nicht geändert haben, sondern einzig X_k. Dennoch müssen die $D_{j<k}$ neu berechnet werden, weil sie auch die (gerade neu berechnete) Menge D_k beinhalten. Es muss folglich zuerst D_k berechnet werden und danach können D_{k-1}, D_{k-2}, ..., D_{min+1} in dieser Reihenfolge aktualisiert werden. Diese Optimierung ist ein wichtiger Unterschied zu der Variante, bei der die Vereinigungen $\Diamond(P_{min+1} \cap X_{min+1}) \cup ... \cup \Diamond(P_{max} \cap X_{max})$ einfach in *einer* Variablen C gespeichert wären. Stellt der Algorithmus beispielsweise fest, dass $X_0 = X_0'$ gilt, wird $X_1 := X_0$ gesetzt. Nehmen wir weiter $X_1 \neq X_1'$ an, sodass der Algorithmus sich jetzt wieder der Berechnung von X_0 widmet. Dann müsste C natürlich neu bestimmt werden. Und zwar indem die Vereinigungen $\Diamond(P_{min+1} \cap X_{min+1}) \cup ... \cup \Diamond(P_{max} \cap X_{max})$ berechnet werden. Anders sieht es mit unserer Optimierung aus. Hier muss einzig $\Diamond(P_{min+1} \cap X_{min+1}) \cup D_{min+2}$ ermittelt werden. Der Wert von D_{min+2} ist nocht gesichert und kann einfach ausgelesen werden.

6. Unter der Bedingung, dass es genau zwei Spieler gibt, gilt $\Box\left(\bigcup_{j=min}^{max} P_j \cap X_j\right) = \bigcap_{j=min}^{max} \Box(\overline{P_j} \cup X_j)$. Analog zu Punkt 5 definieren wir mit

$$B_j = \begin{cases} j = max: & \Box(\overline{P_{max}} \cup X_{max}) \\ j < max: & \Box(\overline{P_j} \cup X_j) \cap B_{j+1} \end{cases}$$

eine auf Implementierungsseite effizientere Berechnungsmethode. Weil in der eigentlichen Berechnung, das heißt wenn im Algorithmus zur Laufzeit die Variable i gleich dem Wert min ist, einzig X_{min} verändert wird, die anderen $X_{j>min}$ aber nur ausgelesen werden, kann diese Optimierung als $(\overline{P_{min}} \cup X_{min}) \cap B_{min+1}$ an Stelle von $\Box\left(\bigcup_{j=min}^{max} P_j \cap X_j\right)$ gesetzt werden. Es ergeben sich somit die gleichen Vorteile, wie sie bereits in Punkt 5 aufgezählt wurden.

7. Diese Optimierungsmaßnahme ist nicht auf den Algorithmus direkt zielend, sondern beeinflusst dessen Implementierung. Intuitiv könnte man zur Bestim-

mung der Menge $\Diamond M$ nach Knoten in V mit Nachfolgern in M suchen. Dreht man den Graphen aber um, setzt also für alle $(a, b) \in \delta$ an deren Stelle (b, a) ein, kann man zur Erlangung des selben Ergebnisses eine andere Frage stellen: Welche Knoten in M haben einen Nachfolger außerhalb von M? Der Vorteil des Ganzen ist, dass nicht $|V|$ Knoten, sondern lediglich $|M|$ Knoten untersucht werden müssen.

Alle diese besprochenen Optimierungsmaßnahmen wurden nun gebündelt in Algorithmus 2 eingebracht.

Algorithmus 2 optimierte Fixpunktiteration

```
0  solve (i){
1      if (i < min) then
2      return(V_0 ∩ (◇(P_min ∩ X_min) ∪ D_{min+1})) ∪ (V_1 ∩ □(¬P_min ∪ X_min) ∩ B_{min+1})
3      else
4          if (i gerade) then X_i = P_i else X_i = ∅
5          repeat
6              X'_i := X_i
7              X_i := solve (i − 1)
8              if (i > min) then
9                  X_i := X_min ∩ P_i
10                 if (X_i ≠ X'_i) then
11                     if (i < max) then
12                         B_i := □(¬P_i ∪ X_i) ∩ B_{i+1}
13                         D_i := ◇X_i ∪ D_{i+1}
14                     else
15                         B_i := □(¬P_max ∪ X_max)
16                         D_i := ◇X_i
17         until (X_i = X'_i)
18         return X_min
19 }
```

wobei

$$D_j = \begin{cases} j = max: & \Diamond(P_{max} \cap X_{max}) \\ j < max: & \Diamond(P_j \cap X_j) \cup D_{j+1} \end{cases}$$

$$B_j = \begin{cases} j = max: & \Box(\overline{P_{max}} \cup X_{max}) \\ j < max: & \Box(\overline{P_j} \cup X_j) \cap B_{j+1} \end{cases}$$

Zeile 2 in Algorithmus 2 wurde gemäß den Punkten 2, 4, 5 und 6 des vorherigen Abschnittes angepasst. In Zeile 4 wird ebenfalls Punkt 2 konsequent umgesetzt. Statt $X_i = V$ wird demnach $X_i = V \cap P_i = P_i$ geschrieben. Zeile 8 schützt die Menge X_0. Zeile 9 resultiert aus Punkt 1 und Punkt 2. Etwas mehr Erklärungsbedarf wird durch Zeile 10 geschuldet. Die neu eingeführten Variablen D_i und B_i mit $0 < i \leq max$ müssen unbedingt bereits vor Aufruf des Algorithmus mit $D_i = \Diamond P_i$ bzw. $B_i = \Box P_i$, falls i gerade und mit $D_i = \Diamond\emptyset = \emptyset$ bzw. $B_i = \Box\emptyset = \emptyset$, falls i ungerade, initialisiert worden sein. Durch Prüfung von $X_i \neq X_i'$ wird die eventuelle Notwendigkeit, $\Diamond D_i$ neu berechnen zu müssen, ermittelt. Wenn X_i immer noch den selben Wert wie sein Vorgänger X_i' hat, ist klar, dass auch $\Diamond X_i = \Diamond X_i'$ gelten würde, was eine erneute Berechnung von $\Diamond X_i$ unnötig machen würde. Die Änderung von "return X_j" zu "return X_0" in Zeile 12 geht nicht nur aus Punkt 1 hervor, sondern ist durch die Umbaumaßnahmen des Algorithmus notwendig geworden.

Interessant dürften nun die in Tabelle 2 protokollierten Zwischenergebnisse des optimierten Algorithmus sein. Die Maßnahmen führten demnach zu Einsparmöglichkeiten von Berechnungen und Rechenwege konnten darüber hinaus kürzer gehalten werden, da die Mengen X_j erheblich weniger Elemente enthalten, was insbesondere in den beiden Spalten X_1 und X_2 im direkten Vergleich mit Tabelle 1 deutlicher sichtbar wird.

Zeilennummer	i	X_0	X_1	X_2
1	2			$\{5,6\}$
2	1		$\emptyset$	
3	0	$\{0,1,3,7\}$		
4	-1	$\{0,2,4,6,7\}$		
5	-1	$\{0,4,6,7\}$		
6	-1	$\{0,4,6,7\}$		
7	1		$\{4\}$	
8	0	$\{0,1,3,7\}$		
9	-1	$\{0,2,3,4,5,6,7\}$		
10	-1	$\{0,2,4,5,6,7\}$		
11	-1	$\{0,4,6,7\}$		
12	-1	$\{0,4,6,7\}$		
13	1		$\{4\}$	
14	2			$\{6\}$
15	1		$\emptyset$	
16	0	$\{0,1,3,7\}$		
17	-1	$\{0,2,6,7\}$		
18	-1	$\{0,6,7\}$		
19	-1	$\{0,6,7\}$		
20	1		$\emptyset$	
21	2			$\{6\}$

Tabelle 2: Zwischenergebnisse Algorithmus 2

Zurückgegeben wird die Menge $\{0,6,7\}$. Diese Menge ist der Gewinnbereich von Spieler 0. Wegen der positionalen Determiniertheit folgt, dass die Menge $V \setminus \{0,6,7\} = \{1,2,3,4,5\}$ der Gewinnbereich des Gegners sein muss.

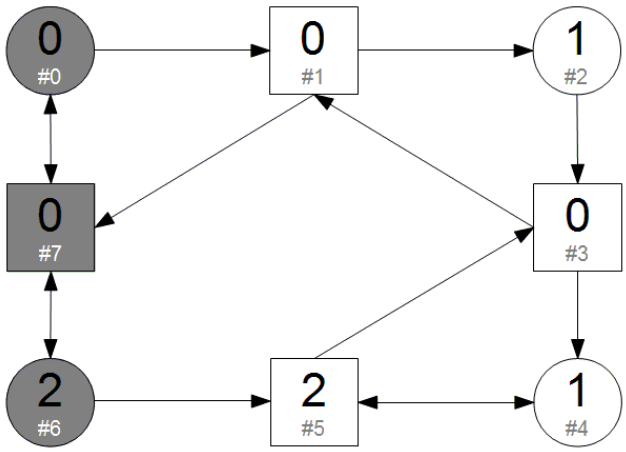

Abbildung 2: Paritätsspiel mit hervorgehobenem Gewinnbereich

Abbildung 2 zeigt das Paritätsspiel innerhalb dessen die grau eingefärbten Knoten den Gewinnbereich von Spieler 0 abbilden. Klar zu erkennen ist, dass Spieler 0 sich stets innerhalb dieses Gewinnbereichs bewegen kann, während der Gegner niemals eine Möglichkeit zur Flucht hat. Es wird also darauf hinauslaufen, dass die größte Priorität, die in der Partie unendlich oft auftritt, entweder der Zahl Null oder Zwei entspricht. In jedem Fall ist die größte davon gerade, weswegen ein Sieg auf Seite von Spieler 0 sicher ist.

4 Implementierung und Laufzeitanalyse

Der Fixpunktiterationsalgorithmus soll nun eine Sammlung von vorhandenen Lösungsalgorithmen ergänzen. In PGSolver [FL09], einem Software-Tool das von den Universitäten München und Kassel entwickelt und verwaltet wird, sind verschiedene Lösungsstrategien für Paritätsspiele zusammengefasst. PGSolver ist in der funktionalen Programmiersprache OCaml [Hic08] geschrieben.

Zunächst wurde der Fixpunktiterationsalgorithmus nach Vorgabe des Algorithmus 1 implementiert. Im zweiten Schritt sind alle bisher besprochenen Optimierungsmaßnahmen auf den Quelltext übertragen worden. Diese zwei Versionen lassen sich bezüglich ihrer Laufzeit vergleichen, wodurch ein möglicher Erfolg der Verbesserungen messbar wird. Dafür unterstüzt das PGSolver Tool mit Spielegeneratoren und Benchmarktools die Laufzeitbestimmung. Wir lassen den Generator für unseren Zweck Zufallsspiele erzeugen. Das Zufallsspiel $R_{n,p,a,l}$ besteht aus n Knoten, Prioritäten aus dem Bereich von 0 bis p zufällig ausgewählt, sowie für jeden Knoten aus mindestens a und höchstens l Nachfolgerknoten. Um zu beobachten, wie die Anzahl der Knoten auf die Laufzeit wirkt, werden immer zehn Zufallsspiele für eine feste Anzahl von Knoten generiert. Daraus werden dann Best-, Worst- und Average-Zeiten gewonnen. Alle Messung werden dreimal wiederholt, wodurch möglichst wenig Einfluss des Prozess-Schedulers auf unsere Ergebnisse hinzunehmen ist. Tabelle 3 präsentiert die so ermittelten Werte.

	nicht optimiert			optimiert		
Zufallsspiel	Best	Average	Worst	Best	Average	Worst
$R_{10,10,2,5}$	0,00	0,31	1,33	0,00	0,02	0,03
$R_{20,10,2,5}$	0,28	3,17	7,39	0,05	0,19	0,44
$R_{30,10,2,5}$	1,95	13,99	32,24	0,28	1,28	2,30
$R_{40,10,2,5}$	9,73	39,54	135,92	0,41	3,42	9,20
$R_{50,10,2,5}$	24,41	63,23	155,78	0,92	7,77	20,88
$R_{60,10,2,5}$	61,30	155,87	321,16	3,23	11,76	29,05
$R_{70,10,2,5}$	132,11	300,40	566,58	6,94	24,36	48,72
$R_{80,10,2,5}$	100,06	340,75	582,80	11,97	33,44	94,77
$R_{90,10,2,5}$	350,27	840,48	1344,45	26,99	52,26	95,55
$R_{100,10,2,5}$	174,11	917,47	1826,33	11,20	72,39	181,59

Tabelle 3: Laufzeiten von zufällig erzeugten Paritätsspielen mit variabler Knotenanzahl

Offensichtlich ist die optimierte Variante schneller. Setzt man die Average-Werte der beiden Algorithmen ins Verhältnis, für die erste Zeile also $\frac{0{,}31}{0{,}02}$, erhält man $15,5$. Setzt man die Berechnung der Verhältniszahlen Zeile für Zeile fort, ergeben sich als Resultate die Zahlen $16,7$; $10,9$; $11,6$; $8,1$; $13,3$; $12,3$; $10,2$; $16,1$; $12,7$ (wir runden die Verhältniszahlen auf eine Stelle nach dem Komma). Summiert man diese auf und dividiert die Summe durch 10, ermittelt also den Durchschnittswert, ergibt sich $12,7$. Das heißt also, dass die optimierte Variante hinsichtlich der betrachteten Zufallsspielen durchschnittlich $12,7$ mal schneller ist, als die nicht optimierte Variante.

Variiert man den Bereich der Prioritäten wie in Tabelle 4, wird der Unterschied zwischen den beiden Algorithmen noch deutlicher. Die Verhältniszahlen sind hier der Reihe nach 2; $16,7$; $151,1$; $3293,5$.

	nicht optimiert			optimiert		
Zufallsspiel	Best	Average	Worst	Best	Average	Worst
$R_{20,5,2,5}$	0,00	0,08	0,27	0,00	0,04	0,08
$R_{20,10,2,5}$	0,28	3,17	7,39	0,05	0,19	0,44
$R_{20,15,2,5}$	0,11	63,47	247,28	0,05	0,42	1,61
$R_{20,20,2,5}$	105,16	2733,61	5624,94	0,05	0,83	2,17

Tabelle 4: Laufzeiten von zufällig erzeugten Paritätsspielen mit variablem Prioritätenbereich

Betrachten wir nun noch eine andere Klasse von Paritätsspielen, die nach Marcin Jurdziński [Jur00]. Das Jurdzińskispiel $J_{d,w}$ der Tiefe d und der Breite w erzeugt ein Rechteck der Dimension $(2d+1) \times (2w)$. Für eine Laufzeitanalyse bezüglich dieser Spiele variieren wir die Parameter d und w. Die Resultate dessen sind in Tabelle 5 und Tabelle 6 dokumentiert.

	nicht optimiert			optimiert		
Jurdzińskispiel	Best	Average	Worst	Best	Average	Worst
$J_{1,5}$	0,11	0,12	0,13	0,03	0,04	0,05
$J_{2,5}$	5,64	5,65	5,66	1,08	1,08	1,08
$J_{3,5}$	208,94	209,01	209,09	23,38	23,39	23,41
$J_{4,5}$	6817,95	6826,46	6835,89	463,34	466,36	471,27

Tabelle 5: Laufzeiten von Jurdzińskispielen $J_{d,w}$ mit festem w

	nicht optimiert			optimiert		
Jurdzińskispiel	Best	Average	Worst	Best	Average	Worst
$J_{5,1}$	26,16	26,17	26,17	0,45	0,46	0,47
$J_{5,2}$	938,33	943,25	949,08	32,33	32,35	32,36
$J_{5,3}$	9274,98	9288,99	9306,25	386,69	388,86	390,86

Tabelle 6: Laufzeiten von Jurdzińskispielen $J_{d,w}$ mit festem d

Bildet man für die vier Zeilen in Tabelle 5 wieder der Reihe nach die Verhältniszahlen, kommt man auf die Ergebnisse 3; $5,2$; $8,9$; $14,6$. Hier beobachtet man ein Wachstum dieser Verhältniszahlen. Je größer das Jurdzińskispiel, desto schneller ist die optimierte Variante im Verhältnis zur nicht optimierten Variante. Anders sieht es für die Spiele in Tabelle 6 aus. Dort kommt man auf die Verhältniszahlen $56,9$; $29,2$; $23,9$.

5 Zusammenfassung

Der Fixpunktiterationsalgorithmus konnte derart optimiert werden, dass sich seine Laufzeit sehr deutlich verringert hat. Möglich wurde das durch einzelne Neuformulierungen und äquivalente Umformungen, deren neue Formen dann weniger Berechnungen und geringere Berechnungszeiten nach sich zogen.

Besonders Optimierungsmaßnahme 2, in Kapitel 2.3 genauer beschrieben, nach der die einzelnen X_j zuvor mit den zugehörigen P_j geschnitten werden, senkt die Anzahl der Berechnungen und verkleinert die als Operanden auftretenden Mengen. Allein dieser Maßnahme kann die Kürzung der Tabelle 1 zu Tabelle 2 zugeschrieben werden.

Innerhalb des OCaml-Quelltextes wurde darauf geachtet, dass beispielsweise die Box- und Diamondfunktionen oder die Zugriffe auf Datenstrukturen möglichst effizient programmiert sind.

Literatur

[FL09] Friedmann, Oliver ; Lange, Martin:
Solving Parity Games in Practice. 2009

[FL13] Friedmann, Oliver ; Lange, Martin:
The PGSolver Collection of Parity Game Solvers.
Report, Ludwig-Maximilians-Universität München, 2013

[Hic08] Hickey, Jason:
Introduction to Objective Caml.
Cambridge University Press, 2008

[HL11] Hofmann, Martin ; Lange, Martin:
Automatentheorie und Logik.
Springer, 2011

[Jur00] Jurdziński, Marcin ; Reichel, Horst (Hrsg.) ; Tison, Sophie (Hrsg.):
Small progress measures for solving parity games.
Springer-Verlag, 2000

[Kre11] Kreutzer, Stephan:
Logik, Spiele und Automaten.
Skript, Humboldt-Universität zu Berlin, 2011

[Loh10] Lohrey, Markus:
Spieltheoretische Methoden in der Logik.
Skript, Universität Leipzig, 2010

[Sti95] Stirling, Colin ; Lee, Insup (Hrsg.) ; Smolka, Scott A. (Hrsg.):
Local Model Checking Games (Extended Abstract).
Springer-Verlag, 1995

[Wal02] Walukiewicz:
Monadic Second-order Logic on Tree-like Structures.
TCS: Theoretical Computer Science, 2002

[Zie98] Zielonka:
Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees.
TCS: Theoretical Computer Science, 1998