HTWK LEIPZIG
Fakultät Informatik, Mathematik und
Naturwissenschaften
Prof. Dr. rer. nat. Karsten Weicker

Masterprojekt

Automatisiertes Spielen des SameGames

– Erkennen, Lösen und Steuern des SameGames
mittles Automatisierungssoftware –

vorgelegt von:

Student:	Wolfgang Teuber, B. Sc.
Studiengang:	Medien-Informatik Master
Matrikelnummer:	52693
Geburtsdatum:	07.07.1985
E-Mail:	wolfgang.teuber@gmx.net

Leipzig, den 30.05.2012

Inhalt

1 Einleitung

Das SameGame ist ein Puzzlespiel. Es wird auf einem rechteckigen Spielfeld gespielt, welches mehrere Zeilen und Spalten hat. Zu Spielbeginn hat jedes Feld eine bestimmte Farbe. Benachbarte gleichfarbige Felder können vom Spieler in einem Zug entfernt werden. Ziel des Spiels ist es, möglichst viele Punkte zu erhalten. Im Rahmen dieser Arbeit bedeutet das, möglichst viele Runden zu spielen. Eine Runde ist beendet, wenn das Spielfeld völlig geleert wurde. Die nächste Runde beginnt unmittelbar nach dem Beenden der vorigen. Das Ziel dieser Arbeit ist es, eine Software für das automatisierte Spielen des SameGames zu entwickeln, welche alle SameGame-Implementierungen unterstützt, die lauffähig unter Windows XP oder höher sind, farbige Felder verwendet, keine Sonderfelder hat, deren Orientierung nach unten gerichtet ist und mit Mausaktionen zu steuern ist und sich höchstens in den folgenden Merkmalen unterscheiden: minimale Gruppengröße, Anzahl der Spalten und Zeilen, Anzahl der im Spiel vorkommenden Feldfarben und die konkret verwendeten Feldfarbwerte, Ausmaße eines Feldes in Pixel, die Anzahl der für einen Zug nötigen Mausklicks, die horizontale Orientierung, Spielanimationen und die für die Implementierung verwendeten Technologien. Die Herausforderung besteht im Entwurf und der Umsetzung einer Automatisierungssoftware, die eigenständig ein SameGame erkennt, löst und steuert. Die Kodierung der SamGame-spezifischen Daten wird an die Arbeit „Optimale Lösungen des SameGames“[1] angelehnt. In der genannten Arbeit wurde bereits ein Evolutionärer Lösungsalgorithmus entwickelt, der als Komponente in die Architektur der Automatisierungssoftware eingebunden wird. In dieser Arbeit werden alle weiteren Komponenten sowie ein Brute-Force-Algorithmus entwickelt, um verschiedene SameGame-Implementierung automatisiert zu spielen. Eine theoretische Betrachtung des Problem und die Beschreibung der Implementierung sind Bestandteil dieser Arbeit. Im Ergebnis dieser Arbeit steht eine lauffähige Automatisierungssoftware, die ihre Grenzen erst bei Spielen mit erhöhter Komplexität erreicht. Es wird eine modulare Architektur gewählt, die sich für diese Arbeit als zielführend erweist. Die Software wird mit geeigneten Programmiersprachen umge-

[1] [Teu11], Wolfgang Teuber, Optimale Lösungen des SameGames, HTWK Leipzig, 2011

setzt. Zum Einsatz kommen Ruby für die Programmablaufsteuerung, AutoIt für die Erkennung und Steuerung des SameGames sowie C für die Umsetzung des Brute-Force-Lösungsalgorithmus. Laut der Arbeit „The Complexity of Clickomania“[2] ist das Entscheidungsproblem „*Leeren des Spielfelds*“ bereits für kleine Spielfelder (5 Spalten, 3 Farben) NP-vollständig. Die Praxistauglichkeit der Automationssoftware wird deshalb ausführlich untersucht. Die Ergebnisse dieser Untersuchungen sind ebenfalls Bestandteil dieser Arbeit.

[2] [Bie02],Therese C. Bied et al. (2002): The Complexity of Clickomania, More Games of No Chance - MSRI Publications, http://library.msri.org/books/Book42/files/biedl.pdf [Stand 01.02.2011], S. 4

2 Automatisierung

„Automatisierung, im Allgemeinen, bedeutet unabhängiges Bedienen oder Agieren, oder Selbstregulieren, ohne menschlichen Eingriff“[3]. Automatisierung ist stets ein Prozess mit dem Ziel, eine Arbeitserleichterung für den Menschen zu erreichen. Die Realisierung einer Automatisierung nimmt verschiedene Formen an. Sie ist entweder eine alleinstehende Komponente (integriert) oder eine Installation verschiedener Komponenten (modular). Software ist ein wesentlicher Bestandteil der meisten Automatisierungen. Jedes elektrische Gerät, das zur Automatisierung dient, braucht eine Steuerung. Integrierte Lösungen sind insbesondere im Hardwarebereich erwünscht und üblich, da äußere Kommunikationskanäle im Vergleich zu inneren anfälliger gegenüber Störungen und Manipulationen sind. Für Software gelten ähnliche Einschränkungen, die durch Rechtemanagement und definierten öffentlichen Programmschnittstellen aufgehoben werden. Im Bezug auf diese Arbeit besteht die Automatisierung aus drei sich wiederholenden Schritten. Der erste Schritt ist das Erkennen des SameGame-Spielfelds. Im zweiten Schritt wird das erkannte Spielfeld gelöst. Der dritte Schritt steuert das Spiel mittels Mausaktionen entsprechend der Lösungen aus dem zweiten Schritt. Die Identifizierung dieser Schritte resultiert in einer modularen Architektur. In diesem Abschnitt werden Voraussetzungen und Anforderungen ausführlich beschrieben, die diese drei Schritte betreffen. Außerdem werden theoretische Grundlagen, Herausfordereungen und mögliche Lösungsansätze für die Realisierung des Automatisierungssoftware zum Spielen des SameGames vorgestellt und diskutiert.

2.1 SameGame

Ein SameGame hat ein rechteckiges Spielfeld mit einer festen Anzahl von Spalten und Zeilen. Zu Beginn des Spiels ist jedes Feld mit einer Farbe belegt. Benachbarte Felder mit der gleichen Farbe bilden eine Gruppe. Die Nachbarschaft ist definiert als Vierernachbarschaft. Das heißt, dass nur die Felder direk-

[3] [Shi09] Shimon Y. Nof, Springer Handbook of Automation, Springer, 2009, (Seite 14, Part A | 3.1.1), übersetzt aus dem Englischen

te Nachbarn sind, die eine gemeinsame Grenze oben, unten, links oder rechts haben. Diagonal anliegende gleichfarbige Felder sind keine direkten Nachbarn. Erreicht eine Gruppe die spielspezifische Mindestgröße (meist 2 oder 3), kann sie mit einen Zug vom Spieler entfernt werden. Felder die in den Spalten über der entfernten Gruppe liegen rutschen nach unten. Die entstehenden leeren Felder am oberen Spielrand bleiben unbesetzt. Wir eine Spalte komplett geleert, so wird die entstandene Lücke durch Zusammenrücken der gefüllten Spalten geschlossen. Auch hier werden die am Rand frei gewordenen Felder nicht neu befüllt. Ziel des Spiels ist es das Spielfeld vollständig zu leeren. Das ursprüngliche SameGame gehört zu der Klasse von Spielen mit vollständiger Information. Das heißt, dass bereits zu Beginn eines Spiels alle Informationen bekannt sind und dass der Spielverlauf nicht von zufälligen Ereignissen abhängt. Seit der ersten Implementierung des SameGames 1985, sind zahlreiche Varianten des populären Puzzlespiel entstanden. Der Großteil der Implementierungen unterscheidet sich in mehreren Merkmalen vom ursprünglichen SameGame. Zu diesem Merkmalen gehören:

- die minimale Gruppengröße,
- die Anzahl der Spalten und Zeilen,
- die Anzahl der im Spiel vorkommenden Feldfarben und
- die konkret verwendeten Farben (RGB-Werte),
- die Ausmaße eines Feldes in Pixel,
- die Aktion um einen Zug auszulösen,
- die horizontale Orientierung,
- Animationen und
- die für die Implementierung verwendeten Technologien.

Unabhängig von Veränderungen dieser Spieleigenschaften, soll die Automatisierungssoftware anwendbar sein. Das Ziel dieser Vorgabe ist es, alle rundenbasierten SameGame-Implementierungen zu unterstützen, die sich höchstens in den genannten Merkmalen unterscheiden. Unter Orientierung wird das Verhalten des Spiels beim Schließen von Lücken verstanden. Dabei unterteilt sich die Orientierung in eine horizontale und eine vertikale Komponente. Die horizontale Orientierung ist entweder links, rechts oder gemischt und bezieht sich auf das Zusammenrücken gefüllter Spalten beim Leeren einer oder mehrere Spalten. Die vertikale Orientierung ist entweder oben oder unten. Das ursprüngliche

SameGame hat die Orientierung **unten-links**. Das heißt zum einen, dass geleerte Felder stets mit darüber liegenden gefüllten Feldern belegt werden – die Felder fallen nach **unten**. Zum anderen bedeutet es, dass leere Spalten stets mit den rechts liegenden gefüllten Spalten belegt werden – Spalten rutschen nach **links**. Unter Animationen sind alle Verhalten eines SameGames zusammengefasst, die entweder zeitverzögert oder interaktionsabhängig auftreten. Dazu gehören unter anderem animierte Fallbewegungen, Mouse-Over-Effekt und die Dauer des Spielneuaufbaus nach einem Rundenwechsel. Die Automatisierungssoftware ist für das Spielen rundenbasierter Spiele ausgelegt. Alle variablen Eigenschaften verschiedener SameGame-Implementierungen werden entweder per Konfiguration vom Nutzer bereitgestellt, oder zur Laufzeit von der Software automatisch bestimmt.

2.2 Erkennen

Der erste Schritt des automatisierten Spielens ist das Erkennen des Spielfelds. Für die Erkennung werden Konfigurationsdaten und das Bildschirmbild verarbeitet. Die Aufgabe des Erkennungsalgorithmus ist es, das auf dem Bildschirmbild dargestellte SameGame einzulesen und in eine normalisierte Form zu übertragen. Das Ergebnis liefert der Algorithmus als Rückgabewert an den aufrufenden Prozess zurück. Folgende Eigenschaften des SameGames werden für die Erkennung benötigt:

- die Position auf dem Bildschirm,
- gleichgroße rechteckige farbige Felder mit bestimmter Höhe und Breite,
- die Anzahl von Spalten und Zeilen und
- ein Hintergrund, der sich farblich von den Feldern unterscheidet.

Die Konfigurationsdaten werden vom Nutzer vor Beginn des automatisierten Spielens bereitgestellt. Das Ermitteln dieser Daten erfordert nicht, dass der Nutzer das Spiel gespielt hat oder Implementierungsdetails kennt. Alle Informationen lassen sich vor dem ersten Zug ermitteln. Die Anzahl der Farben wird vom Erkennungsalgorithmus bestimmt und ist daher nicht vom Nutzer anzugeben. Die Erkennung eines SameGames ist sowohl in der initialen Erkennungsphase, als auch während des Steuerns ein wichtiger Teil des automatisierten Spielens. Die Abbildung 1 zeigt eine SameGame-Implementierung, die sich an

einer beliebigen Stelle auf dem Bildschirm befindet. Ohne technologienspezifische Programmierschnittstellen der Implementierung zu verwenden, wird das Spielfeld vom Erkennungsalgorithmus in seine normalisierte Form übertragen. Das Ergebnis der Erkennung ist eine Zeichenkette, die beginnend mit der linken Spalte aufsteigend und weiter mit den rechts liegenden Spalten die Farben nach ihrem Vorkommen in Spiel durchnummeriert.

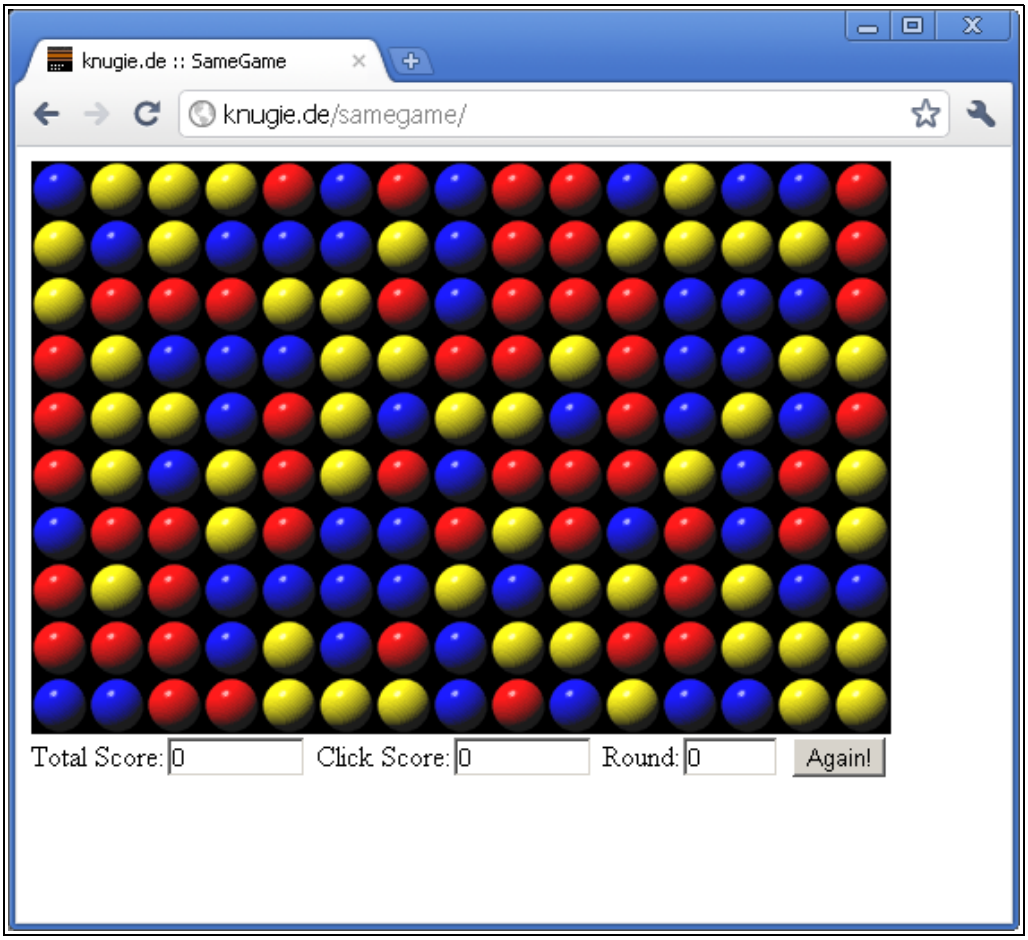

Abbildung 1: Erkennen eines SameGames vor Spielbeginn

Das Ergebnis der Erkennung des Beispiels aus der Abbildung 1 lautet:
„[[1,2,2,1,2,2,2,3,3,1],[1,2,3,2,3,3,3,2,1,3],[2,2,2,2,1,3,1,2,3,3],
[2,1,1,3,3,1,1,2,1,3],[3,3,1,2,2,2,1,3,1,2],[3,1,1,1,3,3,3,3,1,1],
[3,2,1,1,2,1,3,2,3,2],[1,1,3,2,1,3,2,1,1,1],[2,3,1,3,2,3,2,2,2,2],

[1,3,3,2,2,1,3,2,2,2],[3,2,3,1,2,2,2,2,3,1],[1,2,2,2,3,1,1,1,3,3],
[1,3,3,1,1,3,1,1,3,1],[3,3,1,2,2,1,3,1,3,1],[3,3,1,3,3,2,3,2,2,2]]“
Blau hat hier den Wert 1, Rot 2 und Gelb 3. Das entspricht der Reihenfolge des Vorkommens in dem beschriebenen Durchlauf. Im Abschnitt 3.3 werden Implementierungsdetails des Erkennungsalgorithmus beschrieben. Fehlerquellen bei der Erkennung liegen bei der manuellen Positionierung des Mauszeigers auf dem linke untere Feld des SameGames. Es kann nicht davon ausgegangen werden, dass der Nutzer stets den gleichen Pixel trifft. Eine Möglichkeit dem Nutzer beim Positionieren zu assistieren oder das Positionieren ebenfalls zu automatisieren muss bei der Umsetzung gegeben sein. Des Weiteren ist zu beachten, dass grafische Besonderheiten wie Antialiasing oder Zaunlatteneffekte nicht automatisch zu einer fehlerhaften Erkennung führen dürfen. Mögliche Lösungen dieses Problems bei der Umsetzungen sind Nutzerangaben zu möglichen Farbabweichungen, oder die Verwendung eines Klassifizierungsalgorithmus, der zuverlässig ähnliche Farbwerte als gleiche Feldfarben erkennt. Für das initiale Erkennen vor dem ersten Zug ist eine zuverlässige Erkennung nicht abhängig vom Hintergrund des Spielfelds und von der Position des linken unteren gefüllten Feldes. Sobald die Erkennung auch für bereits begonnene Spiele oder während der automatisierten Steuerung des SameGames verwendet wird, ist es wichtig, Felder vom Hintergrund zu unterscheiden. Aufgrund unterschiedlicher horizontaler Orientierungen müssen auch leere Spalten links der ersten gefüllten Spalte erkannt werden. Die Abbildung 2 zeigt ein bereits begonnenes Spiel, welches vom Erkennungsalgorithmus ebenfalls zuverlässig erkennt wird. Das erwartete Ergebnis des Erkennungsalgorithmus ist:
[[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],[1,2,1,1,2,2,1,0,0,0,0,0,0,0,0],
[1,3,2,2,0,0,0,0,0,0,0,0,0,0,0],[2,3,2,2,1,0,0,0,0,0,0,0,0,0,0],
[2,1,3,1,1,1,0,0,0,0,0,0,0,0,0],[1,3,3,3,1,0,0,0,0,0,0,0,0,0,0],
[2,3,0,0,0,0,0,0,0,0,0,0,0,0,0],[2,3,0,0,0,0,0,0,0,0,0,0,0,0,0],
[2,3,1,2,0,0,0,0,0,0,0,0,0,0,0],[3,2,2,2,3,2,1,3,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]

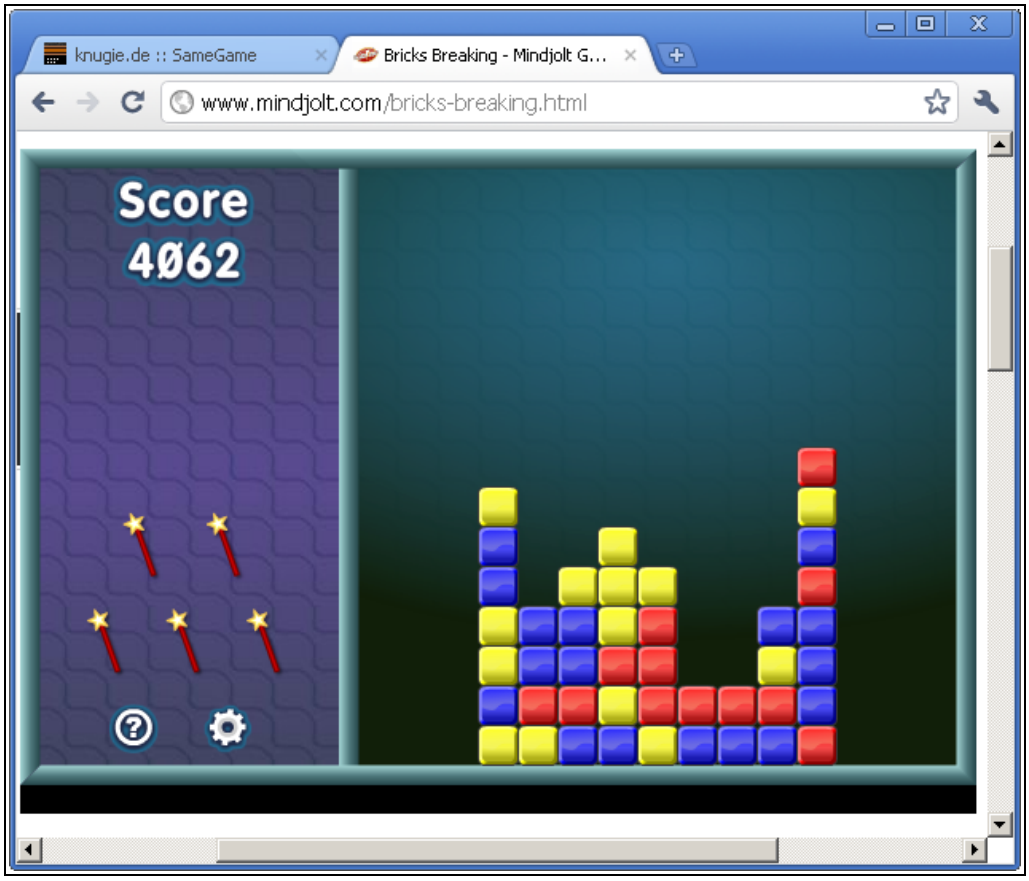

Abbildung 2: Erkennen eines SameGames während des Spiels

0 steht für den Hintergrund, die übrigen Zahlen stehen wie auch schon im oberen Beispiel für die Farben in der Reihenfolge ihres Vorkommens bei einem normalisierten Durchlauf. Ohne Einschränkungen für das im Abschnitt 1 erklärte Ziel, werden Spielfelder nicht unterstützt, deren Orientierung nach oben gerichtet ist. Eine Orientierung nach oben hätte direkten Einfluss auf die Darstellung des Spielfelds in seiner normalisierten Form, da Spalten dann nicht aufwärts, sondern von oben nach unten durchlaufen werden müssten. Insbesondere für die Erkennung von laufenden Spielen, führt diese Eigenschaft zu Fehlern. Die Umsetzung und Implementierungsdetails des Erkennungsalgorithmus werden im Abschnitt 3.3 beschrieben.

2.3 Lösen

Das Lösen eines SameGames bedeutet, eine beliebige Zugfolge zu ermitteln, die zum Leeren des Spielfelds führt. Das Lösen wird nach dem Erkennen ausgeführt. Die möglichen Zustände, die ein SameGame annehmen kann, lassen sich in einem baumähnlichen Graphen darstellen. Der Graph ist kein echter Baum, da zwischen Teilbäumen, auch verschiedener Ebenen, Kanten existieren. Es ist ein endlicher, zyklenfreier, gerichteter Graph. Die Zyklenfreiheit und die Endlichkeit des Graphen werden dadurch garantiert, dass bei jedem Zug Felder entfernt werden. Es ist demzufolge unmöglich im Lösungsgraphen über die gerichteten Kanten zum selben Knoten zurückzukehren. Da keine neuen Felder während des Spiels hinzukommen, ist der Lösungsgraph endlich. Die Abbildung 3 zeigt einen vollständigen Lösungsgraphen des Spiels [[1,2,1,1], [1,2,2,2],[1,2,1,1]].

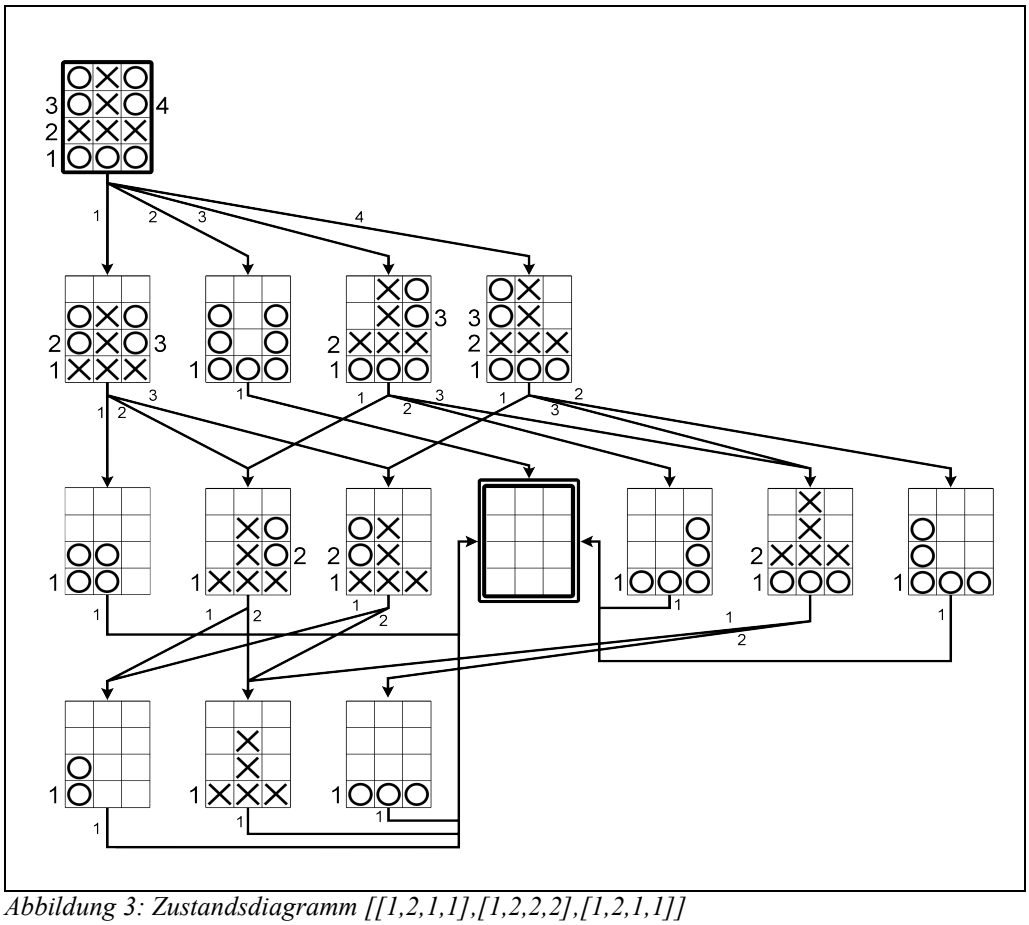

Abbildung 3: Zustandsdiagramm [[1,2,1,1],[1,2,2,2],[1,2,1,1]]

Der Lösungsgraph ist ein Zustandsdiagramm, in dem der Wurzelknoten das SameGame in seinem Startzustand zeigt. Alle möglichen Züge eines Spielzustandes werden mit Zahlen neben dem Spielfeld gekennzeichnet. Jede Kante repräsentiert einen Zug und ist gerichtet vom Ursprungszustand zum Folgezustand. In diesem Graphen führen verschiedene Pfade zu gleichen Teilgraphen, beispielsweise die Teilpfade 1-2 und 3-1. In diesem Beispiel führen alle Pfade zum erwünschten Endzustand eines geleerten Spielfelds. Das gilt nicht für den allgemeinen Fall. Es gibt in der Regel mehr Pfade zu Endzuständen, die ein nichtgeleertes Spielfeld beschreiben als solche, die das SameGame lösen. Um einen geeigneten Pfad zu finden, wird ein Suchalgorithmus benötigt, der Endzustände identifiziert, die ein geleertes Feld enthalten. Für die Umsetzung ist der Praxisbezug von großer Bedeutung. Für rundenbasierte Spiele genügt es, ein beliebige Lösung zu finden, anstelle von Lösungen, die besonders viele Punkte bringen. Im Rahmen dieser Arbeit ist nicht die höchste Punktzahl pro Runde, sondern das Leeren jedes Spielfelds das Ziel der Lösung. Der Lösungsalgorithmus ist demzufolge ein Suchalgorithmus. Verschiedene Umsetzung sind für diese Aufgabe geeignet. Aufgrund der hohen Komplexität der betrachteten Spiele, sind gierige Algorithmen zu bevorzugen. Als Erkenntnis aus der Arbeit „Optimale Lösungen des SameGames“, wird die Modellierung eines SameGames und der Zugfolgen übernommen. Es gilt also, dass zwei Spielfelder identisch sind, die die gleiche Anordnung von Felder haben, unabhängig von ihrer Farbe.

```
     [X O X]      [O X O]
L =  [X X X]  =   [O O O]  = R
     [X O X]      [O X O]
```

Sowohl `L` als auch `R` werden als [[1,1,1],[2,1,2],[1,1,1]] kodiert. Die Realisierung des Lösungsalgorithmus muss ein normalisiertes Spielfeld zusammen mit der minimalen Gruppengröße als Eingabe akzeptieren. Als Ausgabe wird entweder eine Lösung in Form einer normalisierten Zugfolge erwartet, oder die Angabe, dass das gegebene Spielfeld nicht lösbar ist. Der Evolutionäre Algorithmus der in Rahmen Arbeit „Optimale Lösungen des SameGames“ entwickelt wurde, ist für diese Aufgabe geeignet. Er kann allerdings nicht garantieren, dass eine Aussage über die Nicht-Lösbarkeit korrekt ist, da er nicht erschöpfend ist. Das heißt, dass nicht garantiert jeder Pfand des Lösungsgraphen

durchlaufen wird. Dennoch wird mit entsprechend großen Populationen und vielen Generationen in der Regel eine Lösung des betreffenden SameGames erwartet. Um falsch-negative Aussagen auszuschließen, wird im Rahmen dieser Arbeit ein Brute-Force-Algorithmus entwickelt, der den Lösungsgraphen vollständig durchläuft. Dabei werden keine Laufzeitoptimierungen vorgenommen. Mögliche Optimierungen sind beispielsweise die Überprüfung, ob ein Spiel im aktuellen Zustand bereits unlösbar ist, weil eine Anzahl der Felder einer Farbe – unabhängig von der Anordnung – kleiner ist als die minimale Gruppengröße. Ein weiter Optimierung ist es, zu überprüft, welche Pfade sich im Graphen überschneiden. Teilergebnisse werden zwischengespeichert und für das frühzeitige Abschneiden von Teilgraphen verwendet. Eine dritte Optimierung des Suchalgorithmus ist die Ausnutzung der Symmetrie des SameGames.

```
     [O O X]      [X O O]
L =  [O O X]  ~   [X O O]  = R
     [X O O]      [O O X]
```

Ist die Lösung für eines der Felder `L` oder `R` bekannt, so lässt sich die jeweils andere Lösung mit geringem Aufwand bestimmen. Der Lösungsalgorithmus wird als Komponente der Automatisierungssoftware eingesetzt und ist somit austauschbar. Eine mögliche Neu- oder Weiterentwicklung ist gewährleistet. Implementierungsdetails der Lösungsalgorithmen sind im Abschnitt 3.4 beschrieben. Der Nutzer kann über die Konfiguration den Lösungsalgorithmus für das automatisierte Spielen des SameGames festlegen.

2.4 Steuern

Das Steuern eines SameGames erfolgt nach dem Lösen. Es ist der dritte und letzte Schritt des automatisierten Spielens des SameGames. Die Aufgabe des Steuerungsalgorithmus ist es, mithilfe der Konfigurationsdaten und der Lösung des betreffenden SameGames automatisierte Mausaktionen auszulösen, die das Spielfeld leeren. Beim Steuern des Spiels sind zusätzlich zu den für die Erkennung relevanten Merkmalen folgende Spieleigenschaften zu beachten:

- minimale Gruppengröße,
- die Aktion, um einen Zug auszulösen,

- die horizontale Orientierung und
- Animationen.

Der zu entwickelnde Algorithmus muss unabhängig von Unterschieden diesen Spielmerkmale zuverlässig arbeiten. Wie im Abschnitt 2.1 beschrieben, wird das allgemeine SameGame nicht länger als Spiel mit vollständiger Information verstanden. Die Orientierung ist zu Beginn eines Spieles nicht bekannt. Aufgrund des Aufwands, wird vom Nutzer nicht verlangt, eine vollständige Beschreibung der Orientierung bereitzustellen. Eine Unabhängigkeit des Algorithmus gegenüber der horizontalen Orientierung wird durch die Verwendung der im Abschnitt 2.2 beschriebenen Erkennung in Kombination mit der Kodierung einer Zugfolge realisiert. Das Steuerungsalgorithmus muss im Gegensatz zum Erkennungsalgorithmus die Anordnung der Felder analysieren, um die indexbasierte Züge den entsprechenden Gruppen zuordnen zu können. Er muss anhand der Eingaben Positionen auf dem Bildschirm berechnen, an denen eine festgelegte Anzahl von Mausklicks (ein Zug) ausgeführt wird. Um die Erkennung nicht zu beeinflussen, müssen alle Animationen beendet sein, bevor eine erneute Erkennung durchgeführt werden kann. Das gilt für animierte Bewegungen beim Schließen von entfernten Gruppen, sowohl vertikal (Fallen) als auch horizontal (Zusammenrücken von gefüllten Spalten). Außerdem gilt das auch für Mouse-Over-Effekte wie Hervorhebung von Gruppen oder Einblenden von Informationen über dem Spielfeld, z.B. Punkteanzeige für den aktuellen Zug o.Ä. Es gibt SameGame-Implementierungen, die einen Mausklick pro Zug brauchen und welche, die zwei benötigen. Der Nutzer stellt Angaben über die Anzahl der benötigten Klicks pro Zug, die minimale Gruppengröße und Wartezeiten zur Verfügung. Es gibt zwei konfigurierbare Wartezeiten: zum einen die Zeit, die der Algorithmus mindestens warten muss, bis alle Bewegungsanimationen beendet sind; zum anderen die Zeit, die ein Mouse-Over-Effekt höchstens benötigt um zu verschwinden. Der Nutzer muss diese Informationen aus der Spieldokumentation oder durch manuelles Steuern des Spiels ermitteln. Der Steuerungsalgorithmus führt die gegebene Zugfolge auf dem betreffenden SameGame aus. Sollte eine gegebene Zugfolge keine Leeren des Spielfelds nach sich ziehen, so bricht der Steuerungsalgorithmus nach dem letzten ausführbaren Zug der gegebenen Zugfolge ab. Implementierungsdetails des Steuerungsalgorithmus werden im Abschnitt 3.3 genau beschrieben.

3 Implementierung

In diesem Abschnitt wird auf Implementierungsdetails der Automatisierungssoftware eingegangen. Dazu wird zunächst die modulare Architektur im Abschnitt 3.1 erklärt, gefolgt von den Voraussetzungen für einen zuverlässigen Ablauf des automatisierten Spielens und den einzelnen Komponenten. Für die Umsetzung der Programme zur automatisierten Erkennung, Lösung und Steuerung des SameGames werden die Programmiersprachen Ruby[4], AutoIt[5] und C[6] verwendet. Die Quelltexte haben englische Bezeichner und Kommentare, die ausführlich erklärt werden.

3.1 Architektur

Wie im Abschnitt 2 beschrieben, wird eine modulare Architektur für die Umsetzung der Automatisierungssoftware verwendet. Jede Komponente ist unabhängig von den andern verwendbar. Aufgrund der gemeinsamen Funktionen und ähnlicher Voraussetzungen des Erkennens und des Steuerns des SameGames, werden der Erkennungsalgorithmus und der Steuerungsalgorithmus zusammengefasst in einem Programm, dem Player. Je nach Aufruf führt der Player die entsprechende Aufgabe aus. Wie bereits im Abschnitt 2.2 beschrieben, gibt es Fehlerquellen bei der Erkennung, die die manuelle Positionierung des Mauszeigers auf dem linken unteren Feld des SameGames betreffen. Es kann nicht davon ausgegangen werden, das der Nutzer stets den gleichen Pixel mit dem Mauszeiger trifft, wes für eine zuverlässige Erkennung entscheidend ist. Um diese Fehlerquelle zu beseitigen, wird das Positionieren ebenfalls automatisiert. Dazu wird ein Parameter in der Konfiguration bereitgestellt, der den Pfad zu einem Programm enthält, welches den Mauszeiger pixelgenau positioniert. Das Programm wird von Nutzer erstellt und vom Player vor jeder Erkennung ausgeführt. Für die in dieser Arbeit betrachteten SameGame-Implementierungen werden diese Programme mitgeliefert. Der Lösungsalgorithmus ist eine eigenständige Komponente. In dieser Arbeit stehen ein Evolutionärer Algorith-

[4] http://www.ruby-lang.org/
[5] http://www.autoitscript.com/site/autoit/
[6] http://gcc.gnu.org/c99status.html

mus und ein Brute-Force-Algorithmus zur Verfügung. Die Konfiguration wird in eine Datei geschrieben, die von den einzelnen Komponenten ausgelesen wird. Die Ablaufsteuerung sorgt für den Datenaustausch zwischen den Komponenten. Alle Programme werden als Kommandozeilenanwendungen umgesetzt. Die Ausgaben auf `STDOUT` werden für die Funktion der Automatisierungssoftware benötigt. Protokoll- bzw. Log-Ausgaben werden auf `STDERR` geschrieben. Eingaben für die Programme werden als Parameter beim Aufruf angegeben. Eine detaillierte Beschreibung der Verwendung der Komponenten steht im Abschnitt 4. Der rundenbasierte Ablauf der Automatisierungssoftware wird im Abschnitt 3.5 genau beschrieben. Mit diesem Architekturentwurf wird sichergestellt, dass die Automatisierungssoftware im Hintergrund und somit ohne Einfluss auf das Bildschirmbild, läuft. So kann der Player ungestört mit dem Bildschirmbild arbeiten. Außerdem sorgt seine Modularität dafür, dass die Komponenten unabhängig benutzbar sind und gegebenenfalls mit geringem Aufwand austauschbar sind. Die verwendeten Programmiersprachen sind aufgrund ihres Einsatzgebietes gewählt worden. Ruby ist eine interpretierte Scriptsprache, die gute Lesbarkeit und Wartbarkeit auf Kosten der Performance realisiert. AutoIt ist auch eine Scriptsprache. Sie ist an BASIC angelehnt und wird kompiliert. AutoIt bietet umfangreiche Schnittstellen zum Betriebssystem Windows, von denen insbesondere die integrierten Maus- und Bildschirmfunktionen genutzt werden. Der Brute-Force-Algorithmus soll ressourcenschonend, gierig und erschöpfend nach Lösungen auf dem Lösungsgraphen suchen. In C geschrieben und kompiliert erreicht er eine effiziente Ressourcennutzung und geringer Laufzeiten als Realisierungen in Ruby oder AutoIt.

3.2 Voraussetzungen

Damit die Automatisierungssoftware entsprechend der Zielsetzung arbeitet, werden zunächst Voraussetzungen an die Laufzeitumgebung gestellt. Das Betriebssystem und die für das SameGame und die Automatisierungssoftware benötigten Applikationen müssen installiert sein. Die entwickelte Software in dieser Arbeit erfordert:

- Windows XP, oder höher,
- cygwin mit Entwicklungstools wie make und gcc,

- Ruby 1.9.3 und
- AutoIt, v3.3.8.0.

Neben den Anforderungen an die Laufzeitumgebung, werden für das automatisierte Spielen des SameGames Konfigurationsdaten benötigt, die der Nutzer bereitstellt. Wie im Abschnitt 2 beschrieben, erkennt, löst und steuert die Automatisierungssoftware konkrete Implementierungen des SameGames. Gemeinsamkeiten verschiedener Implementierungen werden in der Automatisierungssoftware als invariante Spieleigenschaften behandelt. Zu den Gemeinsamkeiten aller betrachteter SameGame-Implementierungen gehören:

- die Form des Spielfelds (rechteckig),
- die vertikale Orientierung (unten),
- die Vorschrift für die Neusortierung nach Entfernen einer Gruppe,
- farbige Felder,
- verhältnismäßig geringe Komplexität und
- sie sind rundenbasiert.

Die Unterschiede werden in zwei Kategorien unterteilt. Die erste Kategorie beinhalten diejenigen Unterschiede der SameGame-Implementierungen, die durch den Nutzer mit geringem Aufwand ermittelt werden können. Zu diesen gehören:

- Anzahl der Spalten,
- Anzahl der Zeilen,
- Breite eines Felds in Pixel,
- Höhe eines Feld in Pixel,
- Farbe(n) des Hintergrunds,
- die minimale Gruppengröße,
- benötigte Mausklicks zum entfernen einer Gruppe
- die Dauer, die das SameGame maximal benötigt, um nach einem Klick wieder interaktionsbereit zu sein und
- die Geschwindigkeit des MouseOver-Effekts.

Der Nutzer ermittelt diese Daten und schreibt sie mittels des vorgegebenen Formats in eine Konfigurationsdatei. Die zweite Kategorie beinhaltet diejenigen Unterschiede, die nur mit erheblichen Arbeits- und Zeitaufwand vom Nutzer ermittelt werden können. Zu diesen gehört:

- die horizontale Orientierung,

- die Punkteberechnung,
- die Anzahl der Feldfarben,
- die Farbwerte der Felder und
- die verwendete Technologien.

Bezüglich dieser Unterschiede ist die Automatisierungssoftware tolerant. Das heißt, dass jede Variation einer oder mehrerer dieser Eigenschaften unterstützt wird. Die im Abschnitt 3.1 beschriebe Architektur stellt diese Voraussetzung sicher. Sind alle Voraussetzungen erfüllt, so ist ein Automatisierungssoftware einsatzbereit.

3.3 Player

Der Player integriert zwei Komponenten der Automatisierung: das Erkennen und das Steuern eines SameGames. Die Aufgabe des Players in der Erkennungsphase ist es, mittels der Eingabedaten ein SameGame-Spielfeld vom Bildschirm einzulesen und in seiner normalisierten Form auszugeben. Ist die Lösung eines SameGames bereits bekannt, steuert der Player entsprechend der Eingabedaten die Maus, um das Spielfeld zu leeren. Der Player wird in AutoIt3 implementiert. AutoIt3 bietet umfangreiche Möglichkeiten softwaregesteuerte Prozesse in einer Windows-Umgebung zu automatisieren. Die Kommandozeilenanwendung liest die Konfiguration des betreffenden SameGames, erkennt den Modus (Erkennen oder Steuern) und verwendet anschließend die Bibliotheksmethoden um die aktuelle Aufgabe auszuführen. Die Implementierung besteht aus zwei Dateien. Zum einen die Bibliothek, in ihr sind alle Methoden zusammengefasst, die der Player benötigt. Zum anderen das Hauptprogramm. Es bindet die Bibliothek ein und verwendet deren Methoden um die aktuelle Aufgabe auszuführen. Der Quelltext 1 zeigt Auszüge aus dem Hauptprogramm des Players.

```
...
018 #include <..\lib\samegame.au3>
019
020 ;~ read configuration
021 Local $input = readConsole()
022 Local $config = readConfig($input[0])
...
```

```
034 Local $matrix[$cols][$rows]
...
052 Local $orig = MouseGetPos() ;save initial mouse cursor
position
...
056 Local $colors = setColors($orig, $matrix, $tile_width,
$tile_height, $color_tolerance, $background_threshold,
$background_compared) ;find and save all colors of the game
field
058 If $action == 'play' Then
059   logger('Play     | start, samegame (' & $name & ')...')
060   MouseMove($orig[0], $orig[1], 0)
061   playSolution($orig, $colors, $matrix, $tile_width,
$tile_height, $area, $solution, $clicks, $wait_after_click,
$wait_after_move, $color_tolerance, $background_threshold,
$background_compared) ;execute the hits according to the given
solution
064   MouseMove($orig[0], $orig[1], 0)
065   logger('Play     | done, samegame (' & $name & ')')
066 ElseIf $action == 'parse' Then
067   logger('Parse    | start, samegame (' & $name & ')...')
068   $matrix = parseScreenIntoMatrix($orig, $colors, $matrix,
$tile_width, $tile_height, $color_tolerance,
$background_threshold, $background_compared) ;parse same game
field
070   MouseMove($orig[0], $orig[1], 0)
071   ConsoleWrite(matrixToString($matrix) & @CRLF) ;print
same game field to stdout
072   logger('Parse    | result: ' & @CRLF &
matrixToPrettyString($matrix) & @CRLF &
matrixToString($matrix))
073   logger('Parse    | done, samegame (' & $name & ')')
074 EndIf
075
```

Quelltext 1: Player (Hauptprogramm)

In der Zeile 18 wird die Bibliothek eingebunden, die alle nötigen Methoden zur Realisierung des Players zur Verfügung stellt. Im oberen Teil des Programms werden die verwendeten Variablen deklariert und initialisiert. So wird beispielsweise die Variable `$matrix`, welche das Spielfeld repräsentiert, in der Zeile 34 mit der Spalten- und Zeilenanzahl als zweidimensionales Array initialisiert. In den Zeilen 58 und 66 beginnt jeweils der Programmblock für die aktuelle Aufgabe des Players – entweder Steuern (`$action == 'play'`) oder Erkennen (`$action == 'parse'`). Neben den Protokollausgaben, werden die beiden Verhalten jeweils in einer Bibliotheksmethode beschrieben. `playSolution()` ist der Aufruf für das Steuern, `parseScreenIntoMatrix()`

für das Erkennen eines SameGames. Die Methoden unterscheiden sich in ihrer Parameterliste. Beide erwarten folgende Parameter:

- `$orig` – initiale Mausposition,
- `$colors` – verwendete Farben im aktuellen Spiel,
- `$matrix` – leere Spielfeldmatrix mit definierter Spalten- und Zeilenanzahl,
- `$tile_width` – Breite eines Feldes in Pixel,
- `$tile_height` – Höhe eines Feldes in Pixel,
- `$color_tolerance` – Toleranz für Farbähnlichkeit,
- `$background_threshold` – Schwellwert für Hintergrundfarbe,
- `$background_compared` – Angabe, ob der Hintergrund heller oder dunkler als die Felder sind.

Zusätzlich zu diesen, benötigt die Steuerungsmethode noch folgende Parameter:

- `$area` – Mindestgröße einer Gruppe,
- `$solution` – die normalisierte Lösung des aktuellen SameGames,
- `$clicks` – Anzahl der benötigten Klicks pro Zug,
- `$wait_after_click` – Dauer, die der Player nach jedem Zug wartet,
- `$wait_after_move` – Dauer, die der Player nach jeder Mausbewegung aus dem Spielfeld heraus wartet.

Die Parameter `$orig` und `$colors` werden zur Laufzeit bestimmt, wobei die initiale Mausposition (`$orig`) in der Zeile 52 gespeichert wird. Die vorkommenden Farben eines SameGames werden von der Bibliotheksmethode `setColors()` in der Zeile 56 bestimmt. Der Quelltext 2 zeigt die Implementierung dieser Methode.

```
...
301 Func setColors($orig, $matrix, $tile_width, $tile_height,
$color_tolerance, $background_threshold, $background_compared =
'lt')
...
309   For $col = 0 to ($col_count - 1)
310     For $row = 0 to ($row_count - 1)
311       $pxl = PixelGetColor(($x_offset + $col *
$tile_width), ($y_offset - $row * $tile_height))
312       $rgb = rgb($pxl)
313       If ($background_compared == "gt" Or
$background_compared == "bright" Or $background_compared ==
```

```
"brighter" Or $background_compared == "light" Or
$background_compared == "lighter") Then
314           $is_background = (($rgb[0] >=
$background_threshold) And ($rgb[1] >= $background_threshold)
And ($rgb[2] >= $background_threshold))
315         Else
316           $is_background = (($rgb[0] <=
$background_threshold) And ($rgb[1] <= $background_threshold)
And ($rgb[2] <= $background_threshold))
317         EndIf
...
319         If ($is_background) Then
320           $allColors[$col * $row_count + $row] = 0
321         Else
322           $allColors[$col * $row_count + $row] = $pxl
323         EndIf
324       Next   ;$row
325     Next   ;$col
327     $allColors = _ArrayUnique($allColors)
...
330     $found = 0
331     For $color = 1 To $allColors[0]
332       If ($allColors[$color - $found] == 0) Then
333         _ArrayDelete($allColors, ($color - $found))
334         $found = $found + 1
335       EndIf
336     Next   ;$color
337     $allColors[0] = $allColors[0] - $found
338
339     ;~ Combine similar colors to a single color
340     For $color1 = 1 To $allColors[0]
341       For $color2 = 1 To $allColors[0]
342         $rgb1 = rgb($allColors[$color1])
343         $rgb2 = rgb($allColors[$color2])
344         If (Abs($rgb1[0] - $rgb2[0]) <= $color_tolerance) And
(Abs($rgb1[1] - $rgb2[1]) <= $color_tolerance) And
(Abs($rgb1[2] - $rgb2[2]) <= $color_tolerance) Then
347           $color_res = Int(Round((($allColors[$color1] +
$allColors[$color2]) / 2), 0))
348           $allColors[$color1] = $color_res
349           $allColors[$color2] = $color_res
350         EndIf
351       Next   ;$color2
352     Next   ;$color1
...
357     $colors = _ArrayUnique($allColors)
358
359     Return $colors
360
361 EndFunc   ;==>setColors
...
```

Quelltext 2: Player (Bibliothek) - `setColors()`

Um die vorkommenden Farben in einem SameGame zu bestimmen, wird ein Raster, das je ein Pixel des Bildschirmbilds pro Feld einliest, durchlaufen. In den Zeilen 309 bis 325 werden alle Farben des durchlaufenen Rasters in der Variable `$allColors` gespeichert. Einträge mit gleichen Farbwerten werden zunächst in der Zeile 327 zu einem zusammengeführt. Danach werden in den Zeilen 330 bis 337 alle Einträge entfernt, die die Hintergrundfarbe enthalten. Im Anschluss werden in den Zeilen 339 bis 357 die Einträge mit ähnlichen Farben zu einem Eintrag zusammengefasst. Farben, deren Ähnlichkeit den vom Nutzer festgelegten Schwellwert unterschreiten, werden ebenfalls zu einem Eintrag zusammengefasst. Die Ähnlichkeit wird durch die Differenzen der rot-, grün- und blau-Anteile zweier Farben definiert. Überschreiten diese Differenzen den Schwellwert `$color_tolerance` nicht, so werden sie als gleiche Farben behandelt, ansonsten als unterschiedliche Farben. Der Rückgabewert der Methode `setColors()` ist ein Array, das an der Position 0 die Anzahl der im Array folgenden Elemente enthält. Die folgenden Elemente repräsentieren je eine im betreffenden Spielfeld vorkommende Farbe. Damit steht an Position 0 gleichzeitig die Anzahl der Farben, die deswegen nicht konfiguriert wird. Der Quelltext 3 zeigt die Implementierung der Methode `parseScreenIntoMatrix()` Bibliothek des Players.

```
...
251 Func parseScreenIntoMatrix($orig, $colors, $matrix,
$tile_width, $tile_height, $color_tolerance,
$background_threshold, $background_compared = 'lt')
252   Local $x_offset = $orig[0]
253   Local $y_offset = $orig[1]
254   Local $col_count = UBound($matrix)
255   Local $row_count = UBound($matrix, 2)
256   Local $col, $row, $pxl, $rgb1, $rgb2, $color
257
258   For $col = 0 To ($col_count - 1)
259     For $row = 0 To ($row_count - 1)
260       $matrix[$col][$row] = 0
...
268       If (Not $is_background) Then
269         For $color = 1 To $colors[0]
270           $rgb2 = rgb($colors[$color])
271           If (Abs($rgb1[0] - $rgb2[0]) <= $color_tolerance)
And (Abs($rgb1[1] - $rgb2[1]) <= $color_tolerance) And
(Abs($rgb1[2] - $rgb2[2]) <= $color_tolerance) Then
274             $matrix[$col][$row] = $color
275             ExitLoop
```

```
276             EndIf
277           Next    ;$color
278         EndIf
279       Next    ;$row
280     Next    ;$col
281
282     Return $matrix
283
284 EndFunc    ;==>parseScreenIntoMatrix
...
```

Quelltext 3: Player (Bibliothek) - `parseScreenIntoMatrix()`

Ähnlich der Implementierung der Methode `setColors()` wird auch hier das gleiche Raster für die Berechnung des Rückgabewertes verwendet. Die Farben sind bereits bekannt und werden mit dem Parameter `$colors` übergeben. Ziel ist es, in der Methode `parseScreenIntoMatrix()` die Anordnung der Farben zu bestimmen. Dazu wird jeweils die Farbe eines Pixels pro Feld abgefragt (Zeile 261) und anschließend klassifiziert, entweder als Hintergrundfarbe oder als Feldfarbe. Jedes Feld des zweidimensionalen Arrays `$matrix` wird in der Zeile 260 mit 0 initialisiert. Nur falls es sich um eine Feldfarbe handelt (Zeile 268), wird der Index der Farbe in das Array zurückgeschrieben (Zeile 274). Damit der Schleifendurchlauf in der Methode `setColors()` der normalisierten Form entspricht, werden die Werte von `$matrix` in der Methode `parseScreenIntoMatrix()` für die Weiterverarbeitung entsprechend gesetzt. Der Rückgabewert der Methode ist ein zweidimensionales Array, das das Spielfeld des betreffenden SameGames in ihrer normalisierten Form repräsentiert. Der Player ist somit in der Lage die Erkennung durchzuführen.

Die Implementierung der Steuerung eines SameGames wird im Quelltext 4 gezeigt. Die Methode `playSolution()` braucht Informationen bezüglich der der betreffenden SameGame-Implementierung und die normalisierte Lösung. Die Parameter `$area`, `$solution`, `$clicks`, `$wait_after_click` und `$wait_after_move` beinhalten diese Informationen.

```
...
503 Func playSolution($orig, $colors, $matrix, $tile_width,
$tile_height, $area, $solution, $clicks, $wait_after_click,
$wait_after_move, $color_tolerance, $background_threshold,
$background_compared)
```

```
504    Local $idx, $solve_hit, $hit_idx, $col, $row
505    Local $col_count = UBound($matrix)
506    Local $row_count = UBound($matrix, 2)
507    Global $checked[$col_count][$row_count] ;represents the
current state of the field analysis
508
509    For $idx = 1 To $solution[0] Step 1
510      $solve_hit = $solution[$idx]
511      $hit_idx = 0
512      For $col = 0 To ($col_count - 1) Step 1
513        For $row = 0 To ($row_count - 1) Step 1
514          $checked[$col][$row] = False
515        Next   ;$row
516      Next   ;$col
517
518      ;~ set $matrix according to screen
519      $matrix = parseScreenIntoMatrix($orig, $colors,
$matrix, $tile_width, $tile_height, $color_tolerance,
$background_threshold, $background_compared)
520      For $col = 0 To ($col_count - 1) Step 1
521        For $row = 0 To ($row_count - 1) Step 1
522          If ($checked[$col][$row] == False) And
isHittable($area, $matrix, $col, $row, $col_count, $row_count,
True) Then
523            $hit_idx = $hit_idx + 1
524            If $solve_hit == $hit_idx Then
525              MouseClick("primary", ($orig[0] + $col *
$tile_width), ($orig[1] - $row * $tile_height), $clicks)
526              Sleep($wait_after_click)
527              ;~ Move mouse out of game field to prevent
mouse over effects causing parsing errors.
528              If ($wait_after_move <> 0) Then
529                MouseMove(($orig[0] - $tile_width), ($orig[1]
+ $tile_height), 10)
530                Sleep($wait_after_move)
531              EndIf
532            Else
533              markForOneHit($matrix, $col, $row, $col_count,
$row_count, $hit_idx)
534            EndIf
535          EndIf
536        Next   ;$row
537      Next   ;$col
538    Next   ;$idx
539  EndFunc   ;==>playSolution
```

Quelltext 4: Player (Bibliothek) - `playSolution()`

In der Zeile 509 beginnt die Schleife, die die Zugfolge durchläuft. Pro Zug wir zunächst die globale Variable `$checked` auf ihren Ursprungswert zurückgesetzt. Sie ist ein zweidimensionales Array mit der gleichen Anzahl an Spalten

und Zeilen wie das Spielfeld. Das Spielfeld wir in der Zeile 519 mithilfe der Methode `parseScreenIntoMatrix()` bestimmt. Dass die Erkennung mit jedem Zug erneut ausgeführt wird, ist nötig, damit eine völlige Toleranz gegenüber der horizontalen Orientierung des betreffenden SameGames gewährleistet ist. Mit der normalisierten Richtung wird das Spielfeld durchlaufen und für jedes Feld in der Zeile 522 geprüft, ob es nicht bereits durchlaufen wurde (`$checked[$col][$row] == False`) und ein möglicher Zug ist (`isHittable()`). Das trifft immer dann zu, wenn eine Gruppe gefunden wurde, die noch nicht durchlaufen wurde. Sind diese Bedingungen erfüllt, so wird der aktuelle Zugindex `$hit_idx` mit dem Lösungszugindex verglichen. Stimmen sie überein, so wird der Zug in den Zeilen 525 bis 530 ausgeführt. Unterscheiden sie sich, so wird die aktuelle Gruppe als „bereits durchlaufen“ markiert (siehe Zeile 533). Nach dem Ausführen des letzten Zuges der gegebenen Lösung, ist das Spielfeld geleert.

3.4 Solver

Wie bereits im Abschnitt 2.3 beschrieben, werden in dieser Arbeit zwei Lösungsalgorithmen verwendet. Beide werden als Komponente in die modulare Architektur der Automatisierungssoftware eingebunden. Sie werden als Solver bezeichnet. Der Nutzer konfiguriert den einzusetzenden Solver.

3.4.1 Evolutionärer Algorithmus

Diese Arbeit orientiert sich an einer vorigen Arbeit mit dem Titel „Optimale Lösungen des SameGames“. Eine Kopie der Arbeit liegt dieser Arbeit als Datei auf der mitgelieferten CD bei. In der genannten Arbeit wird ein evolutionärer Algorithmus entwickelt. Er ist für das automatisierte Spielen des SameGames geeignet, da er als gieriger Algorithmus konfiguriert werden kann. Die Implementierungsdetails werden dort ausführlich beschrieben. Aus diesem Grund wird an dieser Stelle auf eine nähere Beschreibung verzichtet. Der Evolutionäre Algorithmus wird mit den Parametern „--finished“, „--first“, „--pop=100“, „--gen=40000“ und „-h“ aufgerufen. So konfiguriert verwendet der evolutionäre Lösungsalgorithmus einen Cache um Spielfeldstrukturen effizient zu berechnen

(„-h“ für *history*)´. Es wird eine Populationsgröße von 100 („--pop=100“) und eine maximale Generationenanzahl von 40000 („--gen=40000“) eingestellt. Außerdem wird der Algorithmus so konfiguriert, dass er unmittelbar nach dem ersten („--first“) Leeren des Spielfelds („--finished“) abbricht. Der Evolutionäre Lösungsalgorithmus wird unverändert als Komponente in die Architektur der Automatisierungssoftware dieser Arbeit eingebunden.

3.4.2 Brute-Force-Algorithmus

Der Brute-Force-Algorithmus zum Lösen eines SameGames ist ein erschöpfender Algorithmus. Falls mindestens eine Lösung für ein gegebenes SameGame existiert, findet er diese Lösung garantiert, da er den gesamten Lösungsraum (siehe Abschnitt 2.3) systematisch durchläuft. Außerdem gehört er zu den gierigen Algorithmen, da er nach dem ersten Finden einer Lösung abbricht, ohne möglicherweise bessere Lösungen zu betrachten. Der Brute-Force-Algorithmus ist in C geschrieben. Das Hauptziel dieses Algorithmus ist eine ressourcenschonende Berechnung einer Lösung eines SameGames, wobei auf Laufzeitoptimierungen im Rahmen dieser Arbeit nicht eingegangen wird. Obwohl der Brute-Force-Algorithmus Lösungskandidaten um ein Vielfaches schneller berechnet, ist sein Einsatz für die betrachteten SameGame-Implementierungen nicht notwendigerweise empfehlenswert, da die Gesamtlaufzeit für Spiele mit geringer Komplexität länger ist als die des evolutionären Algorithmus. Der Quelltext 5 zeigt die Implementierung der Methode `solve()`. Diese rekursive Methode durchläuft der unter 2.3 beschriebenen Lösungsgraphen vollständig, ohne mögliche Optimierungen zu berücksichtigen. `solve()` wird mit einer normalisierten Spielfeldstruktur `game_field` aufgerufen. Das ist der einzige Parameter für diese Methode. Nach der Initialisierung der Variablen in den Zeilen 4 bis 7 werden die möglichen Züge auf dem aktuellen Spielfeld in der Zeile 12 berechnet. Ist kein Zug mehr möglich, aber das Spielfeld nicht leer, so wird der Pfad im Graphen verworfen und `solve()` gibt in der Zeile 85 `NULL` zurück. Andererseits werden alle möglichen Züge `possible_hits->hitlistlist` des aktuellen Spielfeldes `game_field` durchlaufen. In der Zeile 38 beginnt der Schleifendurchlauf für jeden dieser Züge. Ein Zug wird ausge-

führt und das Spielfeld mit dem ausgeführten Zug wird anschließend gelöst. Der rekursive Aufruf von `solve()` steht in der Zeile 43.

```
001 #include "samegame.h"
002
003 struct numberListNode* solve(unsigned short int*
game_field) {
004   struct numberListNode* solution = NULL, *tmp_nl = NULL;
005   struct pointerListWithSize* possible_hits = NULL;
006   struct poiterListNode* tmp_hll = NULL;
007   unsigned short int* game_field_after_hit = NULL;
...
012   possible_hits = possibleHits(game_field);
...
017   if (possible_hits->length == 0 && game_field[0] != 0) {
...
034   } else {
035     /*************************************/
036     /* recursion tree node, not solved yet */
037     /*************************************/
038     while (possible_hits->hitlistlist != NULL) {
039       tmp_hll = possible_hits->hitlistlist->next;
040
041       game_field_after_hit = hit(game_field, possible_hits-
>hitlistlist->numberlist);
042
043       solution = solve(game_field_after_hit);
044
045       if (solution == NULL && game_field_after_hit[0] != 0)
{
046         /**************/
047         /* not solved */
048         /**************/
...
054         free(possible_hits->hitlistlist);
055         possible_hits->hitlistlist = tmp_hll;
056
057         free(game_field_after_hit);
058         game_field_after_hit = NULL;
059       } else {
060         /**********/
061         /* solved */
062         /**********/
063         struct numberListNode *tmp = solution;
064         solution = malloc(sizeof (struct numberListNode));
065         solution->next = tmp;
066         solution->value = possible_hits->hitlistlist-
>numberlist->value;
...
073         free(possible_hits->hitlistlist);
074         possible_hits->hitlistlist = tmp_hll;
075
076         free(game_field_after_hit);
077         game_field_after_hit = NULL;
```

```
078
079             return solution;
080           }
081         }
082         free(possible_hits);
083         possible_hits = NULL;
084       }
085       return NULL;
086   }
087
```

Quelltext 5: Brute-Force-Algorithmus - `solve()`

Die maximale Rekursionstiefe entspricht der Anzahl der Knoten des längsten Pfades im gerichteten Lösungsgraphen. Wird ein Blatt (Knoten ohne Kindelemente) erreicht und das Spielfeld ist nicht geleert, wird der nächste Geschwisterknoten durchlaufen. Der Geschwisterknoten ist dabei nicht notwendigerweise ein Blatt. Dazu wird `possible_hits->hitlistlist` mit dem zwischengespeicherten nächsten Knoten ersetzt (Zeile 55). Die Rekursion endet, wenn entweder sobald eine Lösung gefunden oder wenn im gesamten Lösungsgraphen keine Lösung gefunden wird. Wird keine Lösung gefunden, so ist die Bedingung der Zeile 17 für jedes Blatt des Lösungsgraphen erfüllt. Die Methode gibt, nachdem sie alle Knoten durchlaufen ist, in diesem Fall `NULL` zurück. Sobald das Spielfeld erstmals geleert wurde, steigt die Methode aus der Rekursion auf und speichert die Züge, die zum Erfolg geführt haben (`solution`) in den Zeile 63 bis 66. Während des Aufsteigens aus der Rekursion, wird die Zugfolge mit der Länge (maximale Länge des aktuellen Wegs im Lösungsgraphen) - (aktuelle Rekursionstiefe) zurückgegeben. Nach dem Auflösen der Rekursion hat die Lösungszugfolge somit die Länge des entsprechenden Wegs im Lösungsgraphen. Das Ergebnis des Aufrufs von `solve()` wird nach einem Normalisierungsschritt vom Hauptprogramms auf `SDTOUT` geschrieben. Wird keine Lösung gefunden bricht das Programm ohne Ausgabe auf `SDTOUT` ab.

3.5 Ablaufsteuerung

Der modulare Aufbau der Automatisierungssoftware erfordert eine Ablaufsteuerung, die Ausführung der verschiedenen Komponenten abstimmt. Dazu werden die Komponenten mit ihrer jeweiligen Programmierschnittstelle aufge-

rufen und Kommunikation zwischen ihnen verwaltet. Die Abbildung 4 zeigt das Sequenzdiagramm, welches die modulare Automatisierungssoftware beschreibt. Es zeigt schematisch den zeitlichen Ablauf des automatisierten Spielens eines SameGames. Dazu werden der Akteur (**Nutzer**), die Konfigurationsdatei (**config.cfg**) und die involvierten Prozesse nebeneinander dargestellt. Von ihnen geht jeweils eine gestrichelte Linie senkrecht nach unten – die Lebenslinie. Die Aktionen werden in der Reihenfolge von oben nach unten ausgeführt. Aktionen werden mittels durchgezogener Pfeile zwischen den Linien der Komponenten dargestellt. Die Pfeile zeigen vom Aktionssender zum Aktionsempfänger. Es gibt synchrone Aktionen, gekennzeichnet durch eine gefüllte Pfeilspitze (z.B. Punkt 2.1), und asynchrone Aktionen, gekennzeichnet durch eine leere Pfeilspitze (z.B. Punkt 1). Solange eine Komponente aktiv ist, wird ihre Lebenslinie durch einen Balken überlagert. Synchrone Aktionen erfordern eine Antwort vom Aktionsempfänger. Die Antwort wird durch einen waagerechten gestrichelten Pfeil dargestellt. Das Sequenzdiagramm enthält folgende Prozesse: **Autoplay**, **MausTool**, **Maus**, **Player**, **Solver.** Die Konfigurationsdatei **config.cfg** ist bereits vorhanden, wird demzufolge im Diagramm nicht vom **Nutzer** angelegt. Das automatisierte Spielen des SameGames erfolgt durch zwei Aktionen seinerseits. Zunächst *positioniert* der **Nutzer** im Punkt 1 die **Maus**. Im Punkt 2 *startet* der **Nutzer** unter Angabe der Rundenanzahl (*rounds*) und des zu automatisierenden SameGames (*samegame*) den **Autoplay**-Prozess. Da die Automatisierungssoftware nach dem Start selbständig arbeitet, sind weiter Aktionen des **Nutzers** nicht nötig. Seine Lebenslinie endet nach dem Punkt 2. **Autoplay** *liest* danach Daten (*config*) aus der Konfigurationsdatei **config.cfg**. Im Punkt 2.2 wird das Erkennen ausgelöst. Der **Player** *liest* dazu ebenfalls die **config.cfg**, *startet* das konfigurierte **MausTool** um anschließend die Mausposition abzufragen. Mithilfe dieser Daten ermittelt der Erkennungsalgorithmus die normalisierte Form des gegebenen SameGames (*samegameString*) und gibt sie an **Autoplay** zurück. Anschließend veranlasst **Autoplay** den konfigurierten **Solver** das erkannte SameGame zu *lösen*. Danach wird das *Steuern* des SameGames gestartet. Dazu sendet **Autoplay** die Antwort des **Solvers** (*solutionString*) an den **Player**, der nach dem *Lesen* der **config.cfg**, dem *Starten* des **MausTools** und *Lesen* der Mausposition die **Maus** wiederholt bewegt und *klickt* um die Gruppen zu entfernen.

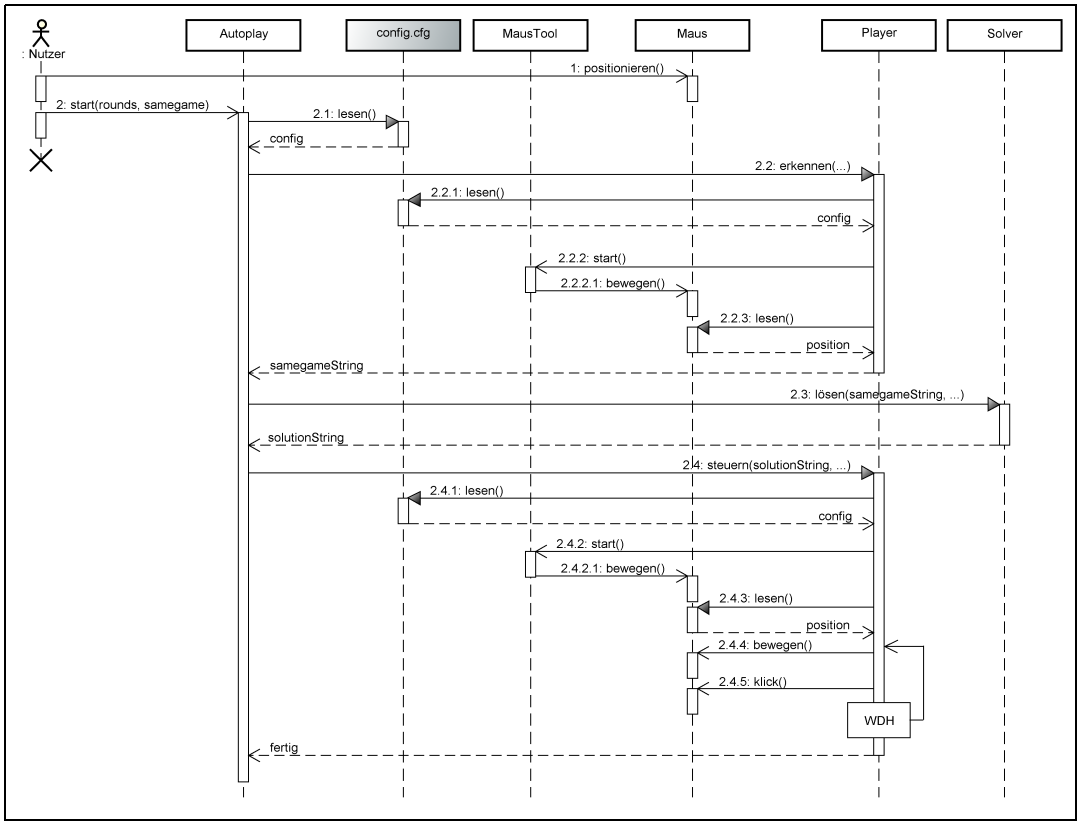

Abbildung 4: Sequenzdiagramm - Ablaufsteuerung

Das Sequenzdiagramm zeigt den Ablauf einer Runde automatisierten Spielens des SameGames. Der beschriebene Vorgang wird entsprechend der Nutzereingaben *rounds* mal wiederholt. **Autoplay** ist in Ruby implementiert. Der Quelltext 6 zeigt einen Ausschnitt der implementierten Ablaufsteuerung.

```
...
061   rounds.times do |i|
062     logger("Autoplay | Round: #{(i+1)} of #{rounds}")
063     game_field = (%x"#{samegame_bin}
--config=#{game[:cfg]}").strip
064     if game[:solver_type] == 'brute-force'
065       logger('Autoplay | use brute-force solver')
066       solution = %x"#{bt_solver} #{game[:area]}
#{game_field}".split('SOLUTION:').last.gsub(/\n|\r/, '')
067     else
068       logger('Solve    | start, use evolutionary
solver...')
069       solution = %x"#{evo_solver} --finished --first
--pop=100 --gen=40000 -h --area=#{game[:area]} #{game_field}"
070     end
071
072     if solution == ''
073       logger('Solve    | No Solution found. Done')
074       exit
075     else
076       logger('Solve    | done, solution: ' + "\n" +
solution)
077       %x"#{samegame_bin} --config=#{game[:cfg]}
--solution=#{solution.dump}"
078       $stderr.puts
079       sleep 3
080     end
081   end
...
```

Quelltext 6: Autoplay - autoplay.rb

In der Zeile 61 beginnt die Schleife für die Runden. Die Zeile 63 entspricht dem Punkt 2.2 des Sequenzdiagramms. Die Zeilen 66 bzw. 69 sind die Realisierungen des Punkts 2.3. Der Punkt 2.4 wird in der Zeile 77 ausgeführt. Die Wartezeit von 3 Sekunden nach dem Beenden einer Runde (Zeile 79) wird von den betrachteten SameGame-Implementierungen höchstens gebraucht, um die nächste Runde zu starten. Die `logger`-Methode schreibt log-Nachrichten auf `STDERR` in einer vereinheitlichten Form, die einen auf Millisekunden genauen Zeitstempel enthält.

4 Verwendung

Wie bereits im Abschnitt 3.1 beschrieben, arbeitet jede Komponente eigenständig. Es wird beschrieben, wie die Komponenten der Automatisierungssoftware zu verwenden sind und wie das automatisierte Spielen des SameGames auf weitere Implementierungen angewendet wird. Vor der Verwendung muss die Software installiert werden. Vor der Installation müssen alle Voraussetzungen an die Laufzeitumgebung wie im Abschnitt 3.2 beschrieben, erfüllt sein. Um die Automatisierungssoftware zu installieren, wird im Ordner `software` folgender Befehl in der Eingabeaufforderung des cygwin-bash ausgeführt:

```
$ bash install.sh
```

Das Script enthält die Befehlsfolge:

```
#!/usr/bin/bash
make -C ./player/
make -C ./solver/brute-force/
chmod -R 644 *
chmod 755 ./*.rb ./*.sh
chmod 755 ./player/bin/*
chmod 755 ./solver/brute-force/bin/*.*
chmod 755 ./solver/evolutionary/*.*
```

Die `make`-Befehle kompilieren den Player, die MausTools, und das Brute-Force-Lösungsprogramm. Die `chmod`-Befehle stellen sicher, dass alle nötigen Rechte für die Benutzung der Software gegeben sind. Nach der Installation befinden sich die MausTools und der Player im Ordner `software/player/bin/` . Der Brute-Force-Solver hat den Dateipfad

```
software/solver/brute-force/bin/sg_bt_solver.exe
```

und der Evolutionäre Solver

```
software/solver/evolutionary/samegame.rb
```

. Die Solver können ohne manuelle Konfiguration verwendet werden. Dazu wird im Ordner `software` folgender Befehl aufgerufen:

```
$ ./solver/evolutionary/samegame.rb --area=2 --finished
--first [[1,3,2,2],[3,1,1,2],[1,3,2,3],[1,2,3,2],[1,2,3,
2]]
```

Der Evolutionäre Solver liefert das Ergebnis

```
[3,6,2,2,1,1,1]
```

zurück. Dieses Ergebnis ist ein Beispiel und weicht bei erneutem Aufruf unter Umständen ab.

Der Brute-Force-Solver wird wie folgt aufgerufen:

```
$ ./solver/brute-force/bin/sg_bt_solver.exe 2 [[1,3,2,2],
[3,1,1,2],[1,3,2,3],[1,2,3,2],[1,2,3,2]]
```

Er schreibt das Ergebnis

```
[6,5,4,3,2,1,2,2,1]
```

zurück. Dieses Ergebnis ist die erste Lösung beim rekursiven links-gerichteten Durchlauf des Lösungsgraphen. Das Algorithmus ist deterministisch und nicht randomisiert. Ein Aufruf erzeugt jedes mal das gleiche Ergebnis. Genau wie die Solver, sind die MausTools auch unabhängig von Konfigurationsdaten des Nutzers. Die mitgelieferten Maustools lassen sich wie folgt aufrufen:

```
$ ./player/bin/bezumie_same_init_cursor.exe
$ ./player/bin/bricks_breaking_init_cursor.exe
$ ./player/bin/cube_crash_init_cursor.exe
$ ./player/bin/haynes_same_init_cursor.exe
$ ./player/bin/knugie_same_init_cursor.exe
```

Sie führen ihre Routine zum automatischen Positionieren des Mauszeigers auf dem linken unteren Feld des entsprechenden SameGames aus. Liegt der Mauzeiger unmittelbar vor der Ausführung über dem entsprechenden Spielfeld, positioniert das MausTool den Mauszeiger zuverlässig, ansonsten wird der Mauszeiger u.U. An eine andere Stelle verschoben. Die MausTools beenden sich selbst nach der Ausführung und haben keinen Rückgabewert.

Der Player kann in verschiedenen Modi verwendet werden, entweder Erkennen oder Steuern. Dabei entscheidet die Parameterliste über den Modus. Der Player hat den Dateipfad `software/player/bin/samegame.exe`. Vor der Ausführung muss eine Konfigurationsdatei angelegt werden. Konfigurationsdateien sind im ini-Format geschrieben und enthalten die Abschnitte [general] (Allgemeines), [game settings] (Spieleinstellungen), [color settings] (Farbeinstellungen) und [solver]. Alle Konfigurationsdaten werden in diese Datei geschrieben. Für weitere Informationen steht eine ausführliche Dokumentation der Datei `README` im Ordner `software` zur Verfügung. Um den Player mit einer bestimmten Konfiguration auszuführen, wir er aus dem Ordner `software` mit

```
$ ./player/bin/samegame.exe --config=<config> [--solution
=<solution>]
```

aufgerufen. Der Parameter `<config>` ist der Pfad zur Konfigurationsdatei und muss zwingen angegeben werden. Der Parameter `<solution>` ist optional und

entscheidet darüber, welcher Modus für die Ausführung verwendet wird. Ist er vorhanden, so führt der Player das Steuern aus, ansonsten das Erkennen. Ein Beispielaufruf für das Erkennen des SameGames `cube_crash` ist:

```
$ ./player/bin/samegame.exe --config=./player/config/cube
_crash.cfg
```

. Neben Protokollausgaben auf `STDERR` , liefert das Programm das erkannte Spielfeld in der normalisierten Form auf `SDTOUT` zurück. Diese Ausgabe wird von der Ablaufsteuerung an den Solver übergeben. Ist eine Zugfolge für ein SameGame gefunden, so wird das Steuern durch einen den Aufruf:

```
$ ./player/bin/samegame.exe --config=./player/config/cube
_crash.cfg --solution=[22,36,2,16,13,23,2,28,7,12,10,7,6,
5,2,2,2,1]
```

veranlasst. Um die beschriebenen Komponenten zu einer Software zusammenzufassen, fehlt nur noch die Ablaufsteuerung, die die Reihenfolge der Aufrufe und die Kommunikation zwischen den Prozessen regelt. Diese Komponente heißt AutoPlay. Sie wird mit dem Aufruf:

```
$ ./autoplay.rb <rounds> <samegame>
```

gestartet und ist nach erfolgreicher Installation und Konfiguration der Automatisierungssoftware der einzige Aufruf, der vom Nutzer verwendet wird. Sollen beispielsweise 7 Runden (<rounds>) des SameGames knugie_same automatisiert gespielt werden, so ruft der Nutzer

```
$ ./autoplay.rb 7 knugie_same
```

im Ordner `software` in der Kommandozeile auf.

Damit die Automatisierungssoftware neue SameGame-Implementierungen im Erkennungsschritt zuverlässig unterstützt, ist der aufwendigste Schritt für den Nutzer, das MausTool zu Schreiben. Der Quelltext 7 zeigt die vollständige Implementierung in AutoIt des MausTools für das SameGame cube_crash.

```
001 ;~ cube_crash_init_cursor.au3
002 ;~ makes sure the mouse cursor is positioned correctly
003 ;~ http://www.mindjolt.com/games/cube-crash
004
005 Local $orig = MouseGetPos()
006 ;~ Move left until white pixel is found
007 While PixelGetColor($orig[0], $orig[1]) <> 16777215
008    $orig[0] = $orig[0] - 1
009 WEnd
010 ;~ Move down to the corner of the game field
011 While PixelGetColor($orig[0] + 1, $orig[1]) <> 16777215
012    $orig[1] = $orig[1] + 1
013 WEnd
014 ;~ Set cursor to the center of the most bottom left tile
015 MouseMove($orig[0] + 30, $orig[1] - 30, 0) ;
```

Quelltext 7: MausTool für cube_crash

Das Script fragt die aktuelle Mausposition in der Zeile 5 ab, durchläuft alle nicht-weißen Pixel nach links und anschließend nach unten bis zur Ecke des Spielfeldes. Danach wird der Mauszeiger mit eine Korrektur von je 30 Pixeln nach recht und oben an die berechnete Stelle gesetzt. Damit neue MausTools mit dem `install.sh`-Script automatisch kompiliert werden, muss das Makefile im Ordner `software/player` aktualisiert werden.

5 Untersuchungen

Im Rahmen dieser Arbeit wird die Automatisierungssoftware ausführlich auf ihre Funktionssicherheit untersucht. Zu den untersuchten Spielen gehören:

- bezumie_same[7],
- bricks_breaking[8],
- cube_crash[9],
- haynes_same[10] und
- knuie_same[11].

Verwendete Technologien sind Javascript, Java und Flash. Für diese Spiele wurden das Erkennen, das Lösen und das Steuern durch die Automatisierungssoftware erfolgreich durchgeführt. Das Lösen des SameGames stellt sich als kritische Aktion heraus, denn das Lösen der Spiele bezumie_same und haynes_same dauert aufgrund höherer Komplexität länger als das Lösen anderer Spiele. Die Komplexität des SameGames cube_crash steigt zudem ab der Runde 6, indem eine Feldfarbe hinzukommt und kann somit auch nicht mehr praktikabel. Als praktikabel werden Lösungszeiten bis zu 5 Minuten betrachtet. Bis zu 30 Minuten ist eine akzeptable Lösungszeit, jeder längere Lösungszeit hat keinen Relevanz für das Automatisierte Spielen des SameGames. Beide Solver haben keine Lösungszeiten des SameGames haynes_same unter 12 Stunden erreicht. haynes_same hat 20 Spalten, 10 Zeilen, 5 Farben und die Mindestgruppengröße 2. bezumie_same wird in weniger als 5% aller Fälle in einer nicht akzeptablen Zeit gelöst, ist aber genau wie haynes_same nicht rundenbasiert und damit auch nicht geeignet für das automatisierte Spielen. Es bleiben die SameGame-Implementierungen bricks_breaking und knuie_same als Untersuchungsgegenstand für eine große Anzahl an Runden. Beide Spiele wurden 1000 Runden lang ohne Unterbrechung automatisiert gespielt. Die Log-Dateien und das Analyse-Script sind auf der CD zu dieser Arbeit im Order `research` zu finden.

[7] http://bezumie.com/same/index.php
[8] http://www.mindjolt.com/games/bricks-breaking
[9] http://www.mindjolt.com/games/cube-crash
[10] http://www.cse.nd.edu/java/SameGame/
[11] http://knugie.de/samegame/

bricks_breaking läuft als Flash-Applikation im Browser. Es hat 15 Spalten und 15 Zeilen, die Mindestgruppengröße 2 und 3 Feldfarben. Es wurden 1000 Runden automatisiert gespielt mit der Verwendung des Evolutionären Lösungsalgorithmus. Die Protokollausgaben wurden dazu in eine Datei umgeleitet. Die Untersuchung des Protokolls hat folgende Werte (in Sekunden) ergeben:

- Gesamtlaufzeit: 93676,960
- kürzeste Laufzeiten:
  - Lösen: 1,576
  - Erkennen: 0,031
  - Steuern: 12,406
  - Gesamte Runde: 15,588
- durchschnittliche Laufzeiten
  - Lösen: 3,811
  - Erkennen: 0,034
  - Steuern: 85,065
  - Gesamte Runde: 90,462
- längste Laufzeiten:
  - Lösen: 28,542
  - Erkennen: 0,063
  - Steuern: 116,375
  - Gesamte Runde: 131,379

Die Zeit für das Lösen des SameGames bricks_breaking dauert im Durchschnitt weniger als 3,9 Sekunden und stets unter 30 Sekunden durchgeführt. Das heißt, dass bricks_breaking gut für das automatisierte Spielen geeignet ist.

knugie_same ist in Javascript implementiert und hat ebenfalls 15 Spalten, aber nur 10 Zeilen. Das sind weniger Zeilen als bricks_breaking. knugie_same hat ebenfalls 3 Feldfarben und die Mindestgruppengröße 2. Zum Lösen wurde auch der Evolutionären Lösungsalgorithmus eingesetzt. Für die Untersuchung wurden 1000 Runden knugie_same automatisiert gespielt und die Protokolldaten analysiert. Die Ergebnisse der Untersuchung (in Sekunden) lauten:

- Gesamtlaufzeit: 32728.432
- kürzeste Laufzeiten:
  - Lösen: 0,015

    - Erkennen: 1,608
    - Steuern: 13,391
    - Gesamte Runde: 16,914
- durchschnittliche Laufzeiten:
    - Lösen: 2,605
    - Erkennen: 0,022
    - Steuern: 25,432
    - Gesamte Runde: 29,611
- längste Laufzeiten:
    - Lösen: 16,303
    - Erkennen: 0,032
    - Steuern: 35,578
    - Gesamte Runde: 45,121

Die Zeit für das Lösen des SameGames knugie_same dauert im Durchschnitt weniger als 2,7 Sekunden und nie länger als 17 Sekunden. Das heißt, dass knugie_same besonders gut für das automatisierte Spielen geeignet ist.

6 Ausblick

Im Laufe der Untersuchungen ist kein Fall aufgetreten, in dem das Erkennen oder das Steuern zu nicht praktikablen Laufzeiten des Gesamtalgorithmus geführt haben. Das Optimierungspotenzial liegt eindeutig im Lösungsschritt der Automatisierungssoftware. Zusammenfassend ist festzustellen, dass komplexere Spiele mit der in dieser Arbeit entwickelten Software für das automatisierte Spielen des SameGames nicht in jeden Fall praktikabel ist. Dazu sind die vorgestellten Lösungsalgorithmen nicht effizient genug. Für Spiele mit geringerer Komplexität, lässt sich die Automatisierungssoftware nach entsprechender Konfiguration erfolgreich einsetzen. Das erklärte Ziel dieser Arbeit wurde im Bezug aus die Funktion und Funktionssicherheit erreicht, ist aber im in der Praxis nicht in jedem Fall praktikabel. Verbesserungen sind besonders im Bereich der Lösungsalgorithmen wünschenswert. Mögliche Optimierungen sind:

- vorzeitiges Verlassen eines Teilgraphen, aufgrund erkannter Nichtlösbarkeit (Pruning des Lösungsgraphen)
- Ausnutzen der überschneidende Pfade (verbessertes Caching) und der Symmetrie eines SameGames.
- Anwenden von Strategien wie
  - spaltenorientiertes Lösen (Entfernen von Gruppen, sodass Spalte nach Spalte entfernt wird um die Komplexität frühzeitig zu verringern)
  - farborientiertes Lösunen (Entfernen von Gruppen der gleichen Farbe, sodass erst eine Farbe vollständig entfernt wird, bevor die anderen Felder geleert werden, um die Komplexität frühzeitig zu verringern)

Auch die anderen Komponenten haben Verbesserungspotential, welches in erster Linie mit einer vereinfachten Bedienung einhergeht. Ein Klassifizierungsalgorithmus für Feldfarben erspart dem Nutzer beispielsweise Angaben über Farbwertabweichungen und Hintergrundeigenschaften zu machen. Trotzdem hat diese Arbeit gezeigt, dass das automatisierte Spielen verschiedener SameGame-Implementierungen praxistauglich realisierbar ist.

7 Verzeichnisse

Literaturverzeichnis

[Teu11]: Wolfgang Teuber, Optimale Lösungen des SameGames, HTWK Leipzig, 2011

[Bie02]: Therese C. Bied et al., The Complexity of Clickomania, More Games of No Chance, http://library.msri.org/books/Book42/files/biedl.pdf, 2002

[Shi09]: Shimon Y. Nof, Springer Handbook of Automation, Springer, 2009

Abbildungsverzeichnis

Quelltextverzeichnis

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Leipzig, den 30.05.2012

_____

Wolfgang Teuber, B. Sc.