Algorithmen und Hardwarearchitekturen zur optimierten Aufzählung von Automaten und deren Einsatz bei der Simulation künstlicher Kreaturen

Dissertation
zur Erlangung des akademischen Grades eines Doktor-Ingenieurs (Dr.-Ing.) im Fachbereich Informatik

TECHNISCHE UNIVERSITÄT DARMSTADT

Bewerber
Dipl.-Inform. Mathias Halbach, geboren in Berlin

Referenten
Prof. Dr.-Ing. Rolf Hoffmann
Prof. Dr.-Ing. Dietmar Fey

Einreichung
10. März 2008

Mündliche Prüfung
14. Mai 2008

Erschienen in Darmstadt im Jahre 2008
Darmstädter Dissertation D 17

Mathias Halbach:
Algorithmen und Hardwarearchitekturen zur optimierten Aufzählung von Automaten und deren Einsatz bei der Simulation künstlicher Kreaturen. Darmstädter Dissertation, Technische Universität Darmstadt, Fachbereich Informatik, D 17, 2008.

Zusammenfassung

Eine minimalistische Robotersteuerung zur vollständigen und autarken Überquerung eines Gebietes bildet die Basisidee. Prinzipiell wäre es möglich, auch andere Aufgaben mit dieser, auf einem Zustandsautomaten basierenden Steuerung zu erfüllen, sie ist nicht einmal an einen Roboter gebunden. So können die steuernden Automaten auch für unterschiedlichste Bereiche selbst in der Theoretischen Informatik Verwendung finden.

Die sich stellende Frage ist, wie ein minimalistischer Automat aussieht. Dazu werden alle möglichen Kombinationen aufgezählt. Da dabei aber zahlreiche Duplikate entstehen, die sich in ihrer Auswirkung nicht unterscheiden, gilt es, nur die relevanten Automaten aufzuzählen. Hierzu wurde ein neuartiges, allgemein anwendbares Schema entwickelt. Dazu müssen die Automaten mehrere, effizient auswertbare Kriterien erfüllen – andernfalls lassen sich aufgrund dessen eine Reihe von Automaten ungeprüft überspringen. Dies führt zu einer erheblichen Reduzierung der notwendigen Überprüfungen.

Statt real mit Robotern die Steuerung durch die gewonnenen Automaten auszutesten, ist es einfacher, ein Simulationssystem zu schaffen, basierend auf dem Prinzip des zellularen Automaten. Neben einer rein softwarebasierten Lösung gibt es auch verschiedene, hardwarebasierte Spezialarchitekturen unter Verwendung eines FPGA-Bausteins, um so zur Lösungsfindung beizutragen. Nach einem Vergleich der unterschiedlichen Konzepte erfolgt schließlich die Präsentation einiger Ergebnisse von erfolgreichen Automaten.

Abstract

The fundamental idea is a minimalist control that enables a robot to cross a defined space independently. This control is a finite state machine. In principle it would be possible to complete other tasks with this finite state machine, as it is a self-contained unit. Thus it might also be useful for different fields of the theoretic computer science.

The first question to answer is the construction of such a finite state machine or automaton. For that purpose all possible combinations are enumerated. As numerous combinations will be named double or triple the main task is now to develop a scheme which makes it possible to select only the automata of relevance. For that, relevance has to be defined. This is done by using general criteria, which can be calculated efficiently. For one missed criterion similar automatons can be skipped in the enumeration without being checked. This reduces the amount of automata being checked considerably.

Instead of using real robots to test the generated automatons, it is easier to construct a simulation system, based on the principles of cellular automata. In addition to a purely software based solution also various hardware based architectures are designed. Field programmable gate arrays (FPGA) are used to calculate the solutions. After comparing the different concepts the results will be presented.

Danksagung

Besonders möchte ich mich bei meinem Doktorvater Prof. Rolf Hoffmann bedanken, ohne dessen Unterstützung und Ideen bei gleichzeitig großzügig gewährtem Freiraum diese Arbeit nicht entstanden wäre. Auch bei Prof. Dietmar Fey für die spontane Bereitschaft, als Korreferent zur Verfügung zu stehen und diese Arbeit kurzfristig zu begutachten, möchte ich mich sehr bedanken. Mein Dank geht auch an die Mitglieder der Prüfungskommission, den Professoren Andreas Koch, Johannes Fürnkranz und insbesondere dem Vorsitzenden Michael Goesele, der für eine angenehme Atmosphäre bei der Prüfung gesorgt hat. Des Weiteren möchte ich mich bei Gudrun Jörs für ihre tatkräftige Unterstützung bei der Organisation bedanken; nicht zu vergessen ist auch Ulrike Hissen für ihren geduldigen Einsatz im Prüfungssekretariat. Des Weiteren haben Kollegen und Zimmernachbarn ihren Anteil zum Gelingen beigetragen, allen voran Dr. Jörg Baumgart, der jederzeit für Fragen ein offenes Ohr hatte.

Wesentlich waren auch die Vorarbeiten, angefangen bei Jan Tisje, der den ersten Hardwareentwurf für das *Creature's Exploration Problem* erstellt hat, Patrick Röder, der in seiner Studienarbeit eine optimierte Programmiermethodik beleuchtet hat, sowie Lars Both, der eine Simulationsumgebung für zahlreiche Experimente geschaffen und auch Anregung zu bedarfsorientiertem Zähleraufbau gegeben hat. Einen weiteren Hardwareentwurf hat Xuesong Yuan realisiert, der die Probleme in der Simulation bei mehreren Kreaturen aufgezeigt hat.

Daneben haben viele andere zu meinem Erfolg durch ihre Unterstützung beigetragen. Besonders genannt seien Jens Mandavid, der kurzfristig und ausführlich die Arbeit lektoriert hat, Veronika Wigand, die mir mit sprachlicher Unterstützung zur Verfügung stand, Felix Heidt für Diskussionen und Anregungen insbesondere für praxisnahen Hardwareentwurf der unterschiedlichen Architekturen, Manfred Kühnel, der mir zu seinen Lebzeiten durch sein Interesse verschiedene Perspektiven aufgezeigt hat, Dieter Schopohl für anregende Diskussionen, Hanna Maria Annas, die mir mit Tipps und Anregungen aus ihrer Erfahrung geholfen hat, sowie die großartige Unterstützung von Frau Krüger von der Fachbereichsbibliothek Mathematik und Prof. Adalbert Kerber von der Universität Bayreuth für seine selbstlose Hilfe, die mich entscheidend bei der Aufzählung voran gebracht hat. Darüber hinaus danke ich Prof. Peter Kammerer, der mir vorab aufgezeigt hat, mit welchen Schwierigkeiten eine Promotion verbunden ist, so dass ich mich rechtzeitig darauf einstellen konnte. Wichtig für mich waren und sind auch der Darmstädter Lauf-Treff und der Chor der TU Darmstadt, die für den notwendigen Ausgleich gesorgt haben.

Mein besonderer Dank geht an meine Eltern, die mich jederzeit unterstützt haben. Leider konnte mein Vater durch seinen viel zu frühen Tod die Schlussphase der Dissertation nicht mehr miterleben und mir auch nicht mehr mit Rat und Tat zur Seite stehen.

Mathias Halbach
Darmstadt, im Mai 2008

Inhaltsübersicht

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Was ist ein guter Plan für die Zukunft? Ein guter Plan für die Zukunft muss sorgfältig durchdacht sein. Ein guter Plan muss ein klares Ziel haben, und er muss von allen verstanden und gebilligt werden, die mitmachen sollen. Ein guter Plan muss begeistern können. Und ein guter Plan muss geändert werden können, denn ein Plan, den man nicht mehr ändern kann, ist ein schlechter Plan.

Joachim Feige

Kapitel 1

Einleitung

Um Problemstellungen wie das vollständige Abschreiten eines Gebietes systematisch zu lösen, muss eine Strategie entwickelt und durchgeführt werden. Zur Erstellung einer Strategie gibt es verschiedene Herangehensweisen. Das Verwenden von „künstlichen Kreaturen“ als aktive Elemente ist eine mögliche Variante, um unterschiedliche Problemstellungen zu bearbeiten. Aber auch dabei gibt es zahlreiche Abwandlungen.

1.1 Motivation

Für das Erlangen eines bestimmten Zieles gibt es verschiedene Wege. Eine Variante ist das Ausprobieren aller Möglichkeiten, um sich dann für die beste davon zu entscheiden. Dafür ist unter Umständen jedoch ein hoher Zeitbedarf notwendig – abhängig von der Komplexität. Um diese Zeit bis zur Erlangung eines Ergebnisses zu reduzieren, kann die Anzahl der auszuprobierenden Möglichkeiten eingeschränkt werden. Hierfür ist bei intelligenten Systemen ausreichende Erfahrung oder ein Orakel[1] hilfreich.

Da derzeit keine zuverlässigen Orakel verfügbar sind und für ausreichende Erfahrung viel Wissen bzw. Speicherplatz erforderlich ist, stellt sich für einfache Systeme die Frage, wie trotzdem ein Ziel schnell erreicht werden kann. Ein Ansatz ist, die Hilfe von künstlichen Kreaturen in Zellularen Automaten zu verwenden.

Vereinfacht dargestellt ist ein Zellularer Automat ein Verbund verschiedener Zellen, die untereinander Informationen austauschen können [Zus69]. Eine Zelle hat einen Zustand (Speicher), Informationsein- und -ausgänge sowie eine Übergangsfunktion zur Berechnung eines Folgezustands. Es ist nicht notwendig, dass die Zellen gleich arbeiten oder auf gleiche Art (im gleichen Muster) miteinander verbunden sind. Zur Vereinfachung haben alle Zellen einen gemeinsamen Takt, mit dem der Folgezustand der aktuelle Zustand wird, d. h. eine Generation weiterschaltet. Zu diesem Themengebiet sind zahlreiche Literaturstellen verfügbar, z. B. [TM87] und [Wol02].

Die Funktion der Zellen für die künstlichen Kreaturen besteht im Wesentlichen aus einer Zustandsübergangstabelle eines Automaten, einer Blickrichtung und einer Auswertungslogik für diese Blickrichtung. Der Inhalt der Zustandsübergangstabelle kann auch als Algorithmus bezeichnet werden, basierend auf der Begriffserklärung „nach einem be-

[1] In der Komplexitätstheorie bezeichnet der Begriff Orakel ein potentielles Unterprogramm, von dem nicht erwartet wird, durch ein effizientes Unterprogramm ersetzt werden zu können. [Weg93, Seite 64]

stimmten Schema ablaufender Rechenvorgang“, entnommen aus [SSO+00]. Die Zellen folgen dem Prinzip eines Mealy-Automaten[2].

Der Inhalt dieser Zustandsübergangstabelle besteht zum einen aus einer vollständigen Aufzählung aller möglichen Eingangswerte kombiniert mit allen erlaubten Zustandswerten, daraus folgt dann davon abhängend als variabler Teil jeweils ein Folgezustand und ein Ausgabewert. Das Verhalten einer künstlichen Kreatur ergibt sich dann aus dem Inhalt dieser Zustandsübergangstabelle. Eine Veränderung des Verhaltens ist gleichbedeutend mit der Änderung der Zustandsübergangstabelle.

1.2 Zielsetzung

Um das Optimum des Verhaltens von künstlichen Kreaturen zu erforschen, ist es neben einer Umgebung notwendig, Bewertungskriterien festzulegen. Diese können zum Beispiel das vollständige Abschreiten eines Gebietes sein, wie es auch beim Rasenmäherproblem beschrieben ist (lawn mower, [Koz92]; bereits 1891 von Hilbert in [Hil91] diskutiert).

Unabhängig davon ist die Beschreibung der Algorithmen bzw. der Zustandsübergangstabellen, die eine Grundlage für die Simulationsdurchführung und damit eine mögliche Bewertung bilden. Um ein Optimum des Verhaltens erhalten zu können, ist eine nahezu vollständige Aufzählung der Algorithmen notwendig. Dabei sollen jedoch keine Algorithmen mit gleichem Verhalten mehrfach aufgezählt werden. Eine Vorauswahl vor einer Simulation ist daher notwendig. Dies wird in Kapitel 3 behandelt; die Simulation selbst für das Problem der Künstlichen Kreaturen ist in Kapitel 4 beschrieben. Dabei werden jeweils der mathematische Aspekt sowie die Durchführung beleuchtet, um mit Mitteln des Hardwaredesigns und der Softwareentwicklung ein Laufzeitoptimum zu erhalten.

Alternative Vorgehensweisen und angrenzende Forschungsgebiete werden in Kapitel 2 analysiert. Dazu zählen insbesondere die Ergebnisse der Aufzählung der Algorithmen bzw. Automaten. Diese lassen sich im übrigen auch für andere Gebiete verwenden. So arbeitet die Theoretische Informatik mit Sprachen, die auf nichts anderem als erkennenden Automaten basieren. Die hier behandelte Aufzählung liefert auch für diesen Zweck deterministische Automaten. In Kapitel 5 erfolgt dann der Abgleich und die Darstellung der Ergebnisse. Eine Zusammenfassung und ein Ausblick ist in Kapitel 6 vorhanden.

Im Anhang A ist die zur Beschreibung verwendete Programmiersprache erläutert. Danach folgen im Anhang B einige Beispielautomaten der verschiedenen Einordnungen. Anhang C stellt die für die Simulationen verwendeten Felder und die daraus entstandenen Ergebnisse dar, sofern diese nicht bereits in Kapitel 5 Erwähnung gefunden haben.

Kapitel D enthält eine Auswahl von Programmen, um die gewählte Implementierung der Algorithmen aufzuzeigen. Abgerundet ist die Darstellung durch Kapitel E mit einer Übersicht der Automatenaufzählungs- und -überprüfungsalgorithmen des Kapitels 3.

Abgeschlossen ist dieses Dokument mit Quellenverzeichnis, Glossar und Stichwortverzeichnis.

[2] Der Ursprung der Einteilung in die verschiedenen Automaten resultiert aus [Mea55] und [Moo56]; hier wird die inzwischen allgemein gebräuchliche Form verwendet, siehe auch [Hof93, Seite 197ff.].

Kapitel 2

Hintergrund

Um an Problemstellungen heranzugehen, ist es erst einmal wichtig, einen Plan für das Gewinnen einer Lösung aufzustellen. Zum Finden eines globalen Optimums ist die Betrachtung aller Möglichkeiten notwendig, für ein lokales Optimum sind andere Herangehensweisen evtl. geeigneter. Um alle Kombinationen zu erhalten, ist eine feste Reihenfolge und das Ausschließen von Permutationen sinnvoll, die dann zu einer Aufzählung unterschiedlicher Elemente führen. Ein Anwendungsbeispiel hierfür sind Automaten.

Jeder Schritt dieser Aufzählung entspricht einem anderen Automaten. Mit den gefundenen Automaten werden Experimente durchgeführt, ob diese für das Absuchen eines Gebietes oder ähnliche Aufgaben geeignet sind. Sowohl zur Thematik der Aufzählung als auch zur Arealüberquerung sind zahlreiche Forschungen in Mathematik, Chemie, Physik und Informatik aus verschiedenen Themenbereichen vorhanden.

Für das Gesamtziel ist es wichtig, nicht nur die Theorie der Berechnung zu haben, z. B. für die Aufzählung von Automaten, sondern diese auch real zu testen – und das mit möglichst wenig Ressourcenverbrauch, wobei auch die Zeit eine Ressource darstellt. Abgerundet ist diese Darstellung des bisherigen Standes der verschiedenen Forschungsgebiete durch verschiedene Aspekte bereits existierender Problemlösungsstrategien.

2.1 Lösungsfindung

Das Erreichen einer Lösung lässt sich gezielt vorantreiben. Pólya hat dazu in seinem Buch „Schule des Denkens (*How to solve it*)“ eine schrittweise Anleitung aufgeführt, die auch in [DH96, Seite 298ff.] erschienen ist. Zusammenfassend besteht sie aus den Punkten

1. Aufgabe verstehen,
2. Zusammenhang zwischen den Daten und den Unbekannten suchen, evtl. Hilfsaufgaben betrachten, um dann einen Plan der Lösung zu erhalten,
3. Plan ausführen,
4. Lösung prüfen.

Bemerkenswert ist auch eine weitere, in [DH96, Seite 2] aufgestellte These, dass die Mathematik die Wissenschaft von Quantität und Raum ist. Von daher ist es unabdingbar, zuerst die für die Lösungsfindung erforderlichen mathematischen Grundlagen und Ausführungen zu verstehen, wie es sich auch für das Kapitel 3 als notwendig gezeigt hat.

2.2 Permutation

Um mit der Permutation im nachfolgenden algorithmisch umzugehen, ist für den Begriff der Permutation eine methodische Beschreibung erforderlich. Zu diesem Zweck wird im Folgenden als Beispiel eine Permutation π basierend auf der Menge $\mathbb{P}$ verwendet. Anders als in der Mathematik üblich beginnt hier die Nummerierung bei 0.

Eine Beschreibungsmöglichkeit ist die Schreibweise als Matrix.

$$\pi = \begin{pmatrix} 0 & 1 & \cdots & |\mathbb{P}|-1 \\ p_0 & p_1 & \cdots & p_{|\mathbb{P}|-1} \end{pmatrix}$$

In der ersten Zeile sind alle Werte von $\mathbb{P}$ sortiert aufgeführt, in der zweiten Zeile stehen deren Permutationswert $p_i \in \mathbb{P}$ für $i \in \mathbb{P}$, so dass $i \mapsto p_i$ gilt. Um die Permutation zu verwenden, ist das gleiche Symbol π als Funktion $\pi(i) = p_i$ zu schreiben. Analog dazu erfolgt eine einzelne Wertzuweisung mit $\pi_i := p_i$. Um in π eine korrekte Permutation zu erhalten, muss $\pi_i \neq \pi_j$ für alle $i \neq j$ mit $i, j \in \mathbb{P}$ gelten. Zur einfacheren Handhabung steht $|\mathbb{P}|$ für die Anzahl der Elemente der Permutation.

Algorithmus 2.1: Aufzählung aller Permutationen

```
1  function InitPermutation(n, π)
2      π := ( 0  1  ···  n−1 )
            ( 0  1  ···  n−1 )
3      return

4  function NextPermutation(n, π)
5      i := n − 2
6      j := n − 1
7      if i < 0 then
8          return false
9      while π_i > π_{i+1} do
10         if i = 0 then
11             foreach i ∈ {0, 1, ..., n/2 − 1} do
12                 π_i, π_{j−i} := π_{j−i}, π_i
13             return false
14         i := i − 1
15     while π_j ≤ π_i do
16         j := j − 1
17     π_i, π_j := π_j, π_i
18     i := i + 1
19     j := n − 1
20     while i < j do
21         π_i, π_j := π_j, π_i
22         i := i + 1
23         j := j − 1
24     return true
```

Um für π alle möglichen Kombinationsmöglichkeiten zu haben, ist eine Aufzählung erforderlich. In [ECS88, Seite 440] ist ein solcher Algorithmus angegeben, der eine geordnete Sequenz gewährleistet. Daran orientiert beginnt die funktionale Schreibweise des

Algorithmus 2.1 mit der identischen Abbildung, bei der kein Element vertauscht ist und somit $\pi_i = i$ gilt. Danach ist nach jedem Aufruf von „NextPermutation“ über π eine weitere Permutationen abrufbar, bis die Funktion den Wert „false“ zurückgibt. Danach ist π wieder in der identischen Form.

Ebenso ist es möglich, die Umkehrung der Permutation zu beschreiben. Statt für einen Wert i dessen permutierten Wert $j = \pi(i)$ zu erhalten, ist es manchmal erforderlich, aus dem permutierten Wert den ursprünglichen zu finden. Hierfür wird die Umkehrpermutation definiert, für die $\pi^{-1}(j) = i$ für alle $i, j \in \mathbb{P}$ mit $\pi(i) = j$ gilt.

Die Aufzählung nach Harary, siehe [HP67], basiert auf der *symmetrischen Gruppe*, die alle möglichen Permutationen enthält. Bezeichnet wird sie mit S_n, wobei n für die Anzahl der Elemente einer Permutation steht, z. B. $n = |\mathbb{P}|$. Die Gruppe enthält $n!$ Elemente, da es n-Fakultät verschiedene Permutationen einer Menge mit n Elementen gibt. Für S_2 gibt es zwei Permutationen, zum einen die identische Abbildung ($0 \mapsto 0$, $1 \mapsto 1$), zum anderen die Transposition ($0 \mapsto 1$, $1 \mapsto 0$), die exakt zwei Werte miteinander vertauscht. Näheres ist in [Art91, Seite 46f.] zu finden; eine Einführung in symmetrische Gruppen sowie deren mathematischen Anwendungen gibt [JK81].

Des weiteren ist die Zykelzerlegung einer Permutation notwendig, beschrieben z. B. in [Art91, Kapitel 6, Paragraph 6]. Zur Veranschaulichung sei das dortige Beispiel 6.1 mit $\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 4 & 6 & 8 & 3 & 5 & 2 & 1 & 7 \end{pmatrix}$ verwendet. Ein darin enthaltener Zykel lautet $1 \mapsto 4 \mapsto 3 \mapsto 8 \mapsto 7 \mapsto 1$, eine alternative Darstellung davon ist (14387). Ein solcher Zykel wird auch als Bahn bezeichnet. π hat somit die Bahnen $(14387)(26)(5)$.

Für eine Berechnung ist die Länge eines jeden Zykel notwendig. Hierfür gibt es die Bezeichnung *Typ*, im Beispiel ist dies $[5,2,1]$, es gibt also einen 5-Zykel, einen 2-Zykel und einen 1-Zykel. Zusammengefasst wird die Anzahl der Zykellängen im Wert j; im Index steht die gewünschte Länge, als Parameter in Klammern ist zur genaueren Identifizierung optional der Permutationsbuchstabe angegeben. Im Beispiel ist also $j_5 = 1$, $j_2 = 1$, $j_1 = 1$, alle anderen $j_k = 0$. Die Summe über alle j, multipliziert jeweils mit der Länge, ergibt die Anzahl der Elemente einer Permutation, $|\mathbb{P}| = \sum_{1 \le k \le |\mathbb{P}|} k j_k(\pi)$. Algorithmus 2.2 bestimmt diese Werte.

Algorithmus 2.2: Permutationszyklen

```
1  function PermutationCycles(n, π, j)
2      foreach i ∈ {0,1,...,n−1} do
3          p_i := true
4          j_{i+1} := 0
5      foreach i ∈ {0,1,...,n−1} do
6          if p_i then
7              l := 0
8              k := i
9              do
10                 l := l + 1
11                 p_k := false
12                 k := π(k)
13             while p_k
14             j_l := j_l + 1
```

Zusätzlich besteht die Möglichkeit, Zustände in einzelne Bereiche zu unterteilen, wobei die verwendeten Spalten als hochgestellte Bereichsangabe mit dem hierzu definierten Operator $\dashrightarrow$ angegeben sind. Ohne eine solche Bezeichnung sind alle Spalten verwendet, also $\pi = \pi^{0\dashrightarrow|\mathbb{P}|-1}$. Allgemein wird als Parameter bei Funktionsaufrufen die gleichbedeutende Matrix

$$\pi^{a\dashrightarrow b} = \begin{pmatrix} a & a+1 & \cdots & b-1 & b \\ \pi_a & \pi_{a+1} & \cdots & \pi_{b-1} & \pi_b \end{pmatrix}$$

übergeben. Die Spaltenauswahl für die Aufrufe der Funktion „NextPermutation“ kann bis zum Maximalwert $|\mathbb{P}|-1$ angegeben sein, da die Anzahl der Elemente durch den ersten Parameter des Funktionsaufrufes gegeben ist.

2.3 Automaten

Nach einer kurzen Einführung in die Thematik führt ein historischer Abriss über die Aufzählung von Automaten zu weiteren Ausführungen der Theoretischen Informatik. All dies bietet die Grundlage für die in Kapitel 3 behandelte Aufzählung von Automaten zur Problemlösungssuche.

Die entstehenden Automaten sind für den bekannten Turing-Test aus [Tur50] mit der Frage „Können Maschinen denken?“ nicht geeignet, da dieser das menschliche Antwortverhalten, aber nicht eine animalische Bewegungseigenschaft testet.

2.3.1 Grundlagen

Bereits mit der Beschreibung einer Turing-Maschine in [Tur37] gab es einen Ablaufmechanismus für Automaten. Aufgrund einer Eingabe, in diesem Fall von einem Band, und einer aktuellen „Konfiguration“ gab es eine Entscheidung über die Ausgabe und der folgenden „Konfiguration“, festgelegt in einer Tabelle. Heutzutage ist der Begriff „Konfiguration“ durch „Zustand“ ersetzt.

Gemäß [Glu63, Seite 29] behandelte zum ersten Mal Kleene in [Kle56] und später in abstrakter Form Medwedew in [Med56] die Darstellung von Ereignissen in Automaten. Die Hinzunahme von Ausgabesignalen erfolgte demnach zuerst in der Arbeit [Glu61].

Zusammenfassend besteht ein Automat aus einem Zustandsspeicher, einem Eingang und einer Ausgabemöglichkeit sowie einer Tabelle, die diese drei Komponenten miteinander verknüpft. Ein Zustand kann ein beliebiger Wert sein, der Einfachheit halber ist es ein Zahlenwert. Alle möglichen Zustände seien in der Menge $\mathbb{S}$ zusammengefasst. Der Wert, den der Zustandsspeicher enthält, hat die Beschreibung „aktueller Zustand“ und die Bezeichnung s mit $s \in \mathbb{S}$. Die Anzahl der Zustände entspricht der Kardinalzahl $|\mathbb{S}|$ der Menge $\mathbb{S}$, weitere mathematische Operatoren sind analog zu [BSMM00].

Für die Eingänge und Ausgaben seien ebenfalls Bezeichner eingeführt. Ein Eingang x sei ein Wert aus $\mathbb{X}$, eine Ausgabe y ein Wert aus $\mathbb{Y}$. Bei einer Turing-Maschine besteht $\mathbb{Y}$ aus den Werten $\mathbb{X} \times \{R, L, N\}$, also einer Kombination aus Eingangswerten und der Bewegung des Schreib-/Lesekopfes.

Die beschreibende Tabelle des Automaten besteht aus der Kombination von Zustand und Eingangswert $\mathbb{S} \times \mathbb{X}$ die zu einem Folgezustand s' und einer Ausgabe y führt. Statt eine Tabelle mit $\mathbb{S} \times \mathbb{X} \to \mathbb{S} \times \mathbb{Y}$ aufzustellen, ist die funktionale Schreibweise mit $s' = f(s,x)$ und $y = g(s,x)$ gebräuchlich. Ein Automat mit diesen Möglichkeiten für

die Funktionstabelle wird heutzutage, nach einem Vorschlag aus [SMM59], als Mealy-Automat bezeichnet.

Für Moore-Automaten gibt es die Einschränkung, dass die Eingangswerte nicht die Ausgabe beeinflussen, also die Ausgabe nur vom aktuellen Zustand abhängt und somit $y = g(s)$ gilt. Bei Medwedew-Automaten gibt es keine gesonderte Ausgabe, es gilt $y = s$ und somit $\mathbb{Y} = \mathbb{S}$. Eine gute Übersicht über die verschiedenen Automatenmodelle von Mealy, Moore und Medwedew enthält das Buch [Glu63].

Die Wandlung von Folgezustand s' zu aktuellem Zustand s erfolgt bei einem synchronen Automaten durch einen Takt, der am Zustandsregister anliegt. Alternativ dazu kommen asynchrone Automaten ohne einen Takt aus, es gibt kein konkretes Ereignis für einen Zustandsübergang. Im folgenden beziehen sich alle Automaten ausschließlich auf die synchrone Variante.

Die Menge der Zustände lässt sich noch in mögliche Startzustände $\mathbb{E}$ und Terminalzustände $\mathbb{F}$ einteilen, wobei ein einzelner Zustand auch in beiden Mengen enthalten sein kann. Es gilt $\mathbb{E} \subseteq \mathbb{S}$ und $\mathbb{F} \subseteq \mathbb{S}$. Startzustände sind potentielle Kandidaten für den ersten Zustand, den ein Automat jemals haben kann. Mindestens einen Startzustand muss es für einen Automaten geben. Terminalzustände werden für endliche Durchführungen benötigt, um ein gültiges Ende zu erkennen, wenn z. B. der Eingabedatenstrom beendet ist.

Ein Zustand entspricht in einem Graphen einem Punkt bzw. einem Knoten, die Verbindungen sind in Abbildung 2.1 als gerichtete Kanten dargestellt. An den Kanten ist sowohl der bestimmende Eingangswert als auch der zugehörige Ausgabewert angegeben, voneinander durch einen Schrägstrich getrennt. Zustand 2 enthält für $x = 0$ eine Schlinge, also Verbindungen von einem Knoten zu sich selbst. Nur Zustand 0 ist ein Startzustand und Zustand 3 ist einziger Terminalzustand.

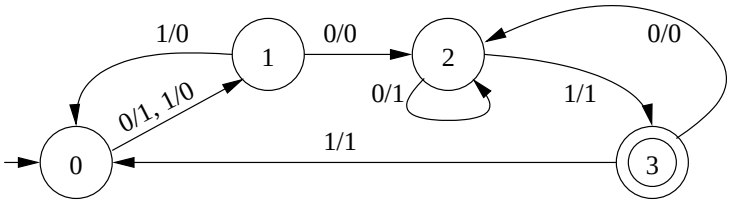


Abbildung 2.1: Graph mit Kanten und Knoten

Graphen mit nur zueinander unterschiedlichen Bezeichnern sind isomorph. Davon ist aber nur ein Graph als Vertreter interessant, bei einer Simulation hätte der andere, isomorphe Graph das gleiche Resultat zur Folge und ist somit für eine Diversifizierung uninteressant. Da Start- und Terminalzustände ein Unterscheidungsmerkmal darstellen, gibt es mehr zueinander isomorphe Graphen, wenn alle Zustände initial und terminal sind, also $\mathbb{E} = \mathbb{F} = \mathbb{S}$ gilt. Durch die endliche Anzahl an Zuständen und wiederholtes Erreichen von Zuständen, um eine unendliche Abfolge von Ausgaben zu ermöglichen, ist der entstehende Graph zyklisch. Eine weitere Eigenschaft des Graphen ist die Multiplizität der Kanten, da es je Knoten zwei abgehende Verbindungen gibt – je Eingangswert eine eigene Kante. Es handelt sich also insgesamt um einen gerichteten Multigraphen.

2.3.2 Aufzählung in der Algebra

In [BLW76, Kapitel 4] ist das Thema zur Aufzählung von Graphen historisch aufgegriffen. Eine Anwendung ist die Konstruktion von Bäumen für Kohlenstoffketten für die

chemische Forschung, beschrieben vom Mathematiker Arthur Cayley in [Cay75]. Allerdings wurde dies vom Chemiker Hugo Schiff in [Sch75] als nicht handhabbar, aber trotzdem als interessant abgetan. Als Alternative bietet er zur Anzahlbestimmung eine Formel mit Bruchrechnung statt einer Aufzählungsregel für Bäume an, die auf die chemischen Gegebenheiten eingeht, ohne dabei nach eigener Behauptung Glieder doppelt aufzuführen oder zu übersehen.

In Folge dessen wurden gemäß [BLW76, Seite 68] weitere rekursive Herleitungen aufgestellt, die allerdings nicht alle Möglichkeiten abdecken konnten. George Pólya hat dies verallgemeinert und mehr mathematisch betrachtet und mit [Pól37] eine Basis geschaffen, auf der viele weitere Arbeiten aufsetzen. Hier wird bereits die polynomiale Schreibweise eingeführt, die auch später Frank Harary für seine Arbeiten verwendet. [Deo74] bietet einen Kommentar zur Berechnungsformel von [Pól37]. Probleme werden demnach mit Koeffizienten beschrieben, der dazugehörige Faktor gibt die Anzahl der unterschiedlichen Möglichkeiten der jeweiligen Bedingung an.

Allerdings muss die Wichtungsfunktion spezielle Eigenschaften erfüllen. Ein Verfahren hierfür ist in [Leh76, Seite 72f.] dargestellt, das jedoch nicht alle Notwendigkeiten der in Kapitel 3 aufgestellten Eigenschaften beinhaltet, so dass die dort entstandene Formel hierfür nicht angewendet werden kann, eine Erweiterung ist zu komplex.

In [Gin62] werden die gleichen Verfahren wie in Kapitel 3.2.1 und 3.2.3 verwendet, um isomorphe, streng verbundene Graphen zu erkennen. Die Aufzählung selbst ist allerdings als noch nicht gelöstes Problem bezeichnet [Gin62, Seite 30].

In [Kap65a] ist eine Konstruktionsmethode für isomorphe, homogene Automaten mit $|\mathbb{X}| = 1$ und $|\mathbb{Y}| = 1$ beschrieben. Dies entspricht den normierten Automaten ohne Präfix. Als Rechenzeit bei $n = |\mathbb{S}| = 10$ benötigt der damals zur Verfügung stehende Kiëw-Computer für 1 000 Anweisungen 35 Minuten Rechenzeit. Für die Konstruktion der Automaten mit n Zuständen werden allerdings die Automaten mit weniger als n Zuständen benötigt, es ist also entweder ein großer Speicherbedarf notwendig oder die mehrfache Berechnung der bereits gefundenen Automaten. Das in Kapitel 3 vorgestellte Verfahren benötigt mit 200 Programmzeilen nicht einmal eine Millisekunde auf einem heutigen Computer, um $n^n = 10^{10}$ mögliche Automaten zu testen und aufzuzählen.

Im zweiten Teil der Arbeit, [Kap65b], wird eine Methode aufgeführt, um mit mehr Eingangsmöglichkeiten ebenfalls isomorphe, homogene Automaten aufzuzählen, ein Berechnungsergebnis ist allerdings nicht aufgeführt. Stattdessen wird eine Kleiner-Als-Relation definiert, wie sie auch implizit in der Definition 3.1 enthalten ist.

Der Forscher Frank Harary hat sich über mehrere Jahre mit der Aufzählung gerichteter Graphen beschäftigt, unter anderem ist ein Beispiel mit drei Knoten in [Har57] aufgeführt. Eine Liste von Graphenaufzählungsproblemen mit zahlreichen Beispielen enthält [Har60], die mit [Har69, Seite 194ff.] eine Aktualisierung erfahren hat. Demnach ist die Aufzählung gerichteter Digraphen durch die Aufsätze [Har65] und [HP67] gelöst, allerdings gilt dies nur die Isomorphie betreffend. Weiterhin ist mit *enumeration* nur eine Formel zur Anzahlbestimmung gemeint, nicht die konkrete Aufzählung der dahinter befindlichen Automaten. Beide Berichte verwenden symmetrische Subgruppen als Grundlage für eine Aufzählung. Mit dem *cycle index* als Basiszähler erklärt [HH68], was auch Harary als Prinzip verwendet hat, allerdings hat Harrison nur eine Formel ohne nähere Darstellung angegeben.

Aufbauend auf einem plastischem Beispiel ist in [Pól37] unter Einsatz der Kombinatorik und Anwendung der Permutation die Bestimmung der Anzahl für Anordnungen hergeleitet. [HP67] greift dies verallgemeinernd auf, um so für die entstehenden endli-

chen Automaten, die zueinander nicht-isomorph sind, die Gesamtanzahl in Abhängigkeit von Anzahl der Zustände, Eingabesymbole, Ausgabesymbole und akzeptierenden Zuständen zu bestimmen. Die dabei entstehende Formel verwendet eine Summation über alle Permutationen unter Verwendung von größten gemeinsamen Teilern und kleinsten gemeinsamen Vielfachen. Ein Wertebeispiel ist in [HP67, Tabelle 1] gegeben. In dem Buch [HP73] ist die Thematik ausführlich behandelt, für die Anzahl der Automaten mit $n = |\mathbb{S}|$ Zuständen, $k = |\mathbb{X}|$ Eingabe- und $m = |\mathbb{Y}|$ Ausgabesymbolen gelten die Formeln

$$a(n,k,m,t) = \frac{1}{|F|} \sum_{(((\alpha,\beta);\alpha^{-1}),((\alpha,\beta);\gamma))\in F} I(\alpha,\beta,\alpha)\, I(\alpha,\beta,\gamma) \tag{2.1}$$

$$I(\alpha,\beta,\gamma) = \prod_{p=1}^{n}\prod_{q=1}^{k}\left(\sum_{s|\operatorname{lcm}(p,q)} s j_s(\gamma)\right)^{j_p(\alpha) j_q(\beta)\gcd(p,q)} \tag{2.2}$$

$$F \subset (S_n^{S_n\times S_k}) \times (S_m^{S_n\times S_k}),\ |F| = n!k!m! \quad \text{für } t = 0 \tag{2.3}$$

$$\begin{gathered} F \subset \left((S_1 S_{n-t-1} S_t)^{(S_1 S_{n-t-1} S_t)\times S_k}\right) \times (S_m^{(S_1 S_{n-t-1} S_t)\times S_k}), \\ |F| = (n-t-1)!t!k!m! \quad \text{für } t > 0 \end{gathered} \tag{2.4}$$

mit „lcm“ für lowest common multiple (kleinstes gemeinsames Vielfaches) und „gcd“ für greatest common divisor (größter gemeinsamer Teiler), bestimmbar mit Algorithmus 2.3, der auf dem Euklidschen Algorithmus basiert und den Zusammenhang zwischen gcd und lcm ausnutzt, entnommen aus [BSMM00]. Die Summe in Formel 2.2 geht über alle Teiler des kleinsten gemeinsamen Vielfachen. Dies ist mit Algorithmus 2.4 unter Nutzung der Formel 2.1 als Ausgangspunkt umgesetzt.

Algorithmus 2.3: Größter gemeinsamer Teiler und kleinstes gemeinsames Vielfaches

```
1 function gcd(a, b)
2     i, j := max{a,b}, min{a,b}
3     while j ≠ 0 do
4         h := i mod j
5         i := j
6         j := h
7     return i

8 function lcm(a, b)
9     return a · b ÷ gcd(a,b)
```

Koršunov arbeitet in dem oft zitiertem Bericht [Kor78] mit Wahrscheinlichkeiten für die Aufzählung der Zustandsübergänge, wie es auch schon in [Obe70] der Fall ist. Allerdings sind die Berechnungen im wesentlichen theoretisch und für $|\mathbb{X}| + |\mathbb{S}| \to \infty$. Die Automaten sind zwar nach dem Konzept von Mealy gestaltet, allerdings sind die Ausgänge gemäß Lemma 1 nur allgemein ohne Einfluss auf die Automaten betrachtet, so dass deren Anzahl nur als Ergebnis von Anzahl der Ausgabewerte potenziert mit dem Produkt aus Eingangswerte- und Zustandsanzahl in das Gesamtergebnis einfließt. Dies ist den gesamten Bericht über beibehalten.

Algorithmus 2.4: Berechnung der Anzahl Automaten mit n Zuständen, davon t terminal, r initial und r_t sowohl terminal als auch initial, k Eingangswerten, m Ausgabewerten

```
1  function I(α, β, γ)
2      r := 1
3      foreach p ∈ {1,2,...,n} do
4          foreach q ∈ {1,2,...,k} do
5              if j_p(α) j_q(β) gcd(p,q) ≥ 1 then
6                  Σ := 0
7                  foreach s ∈ {1,2,...,lcm(p,q)} do
8                      if (lcm(p,q) mod s) = 0 then
9                          Σ := Σ + s j_s(γ)
10                 r := r · Σ^(j_p(α) j_q(β) gcd(p,q))
11     return r

12 function AutomataAmount(n, k, m, t, r, r_t)
13     InitPermutation(n, σ)
14     InitPermutation(k, χ)
15     InitPermutation(m, υ)
16     i := 0
17     do
18         do
19             do
20                 do
21                     do
22                         do
23                             PermutationCycles(n, σ, j(σ))
24                             PermutationCycles(k, χ, j(χ))
25                             PermutationCycles(m, υ, j(υ))
26                             i := i + I(σ,χ,σ) + I(σ,χ,υ)
27                         while NextPermutation(m, υ)
28                     while NextPermutation(k, χ)
29                 while NextPermutation(r − r_t, σ)
30             while NextPermutation(r_t, σ^(r−r_t ⇢ n−1))
31         while NextPermutation(n − t − r + r_t, σ^(r ⇢ n−1))
32     while NextPermutation(t − r_t, σ^(n−t−r+r_t ⇢ n−1))
33     return i ÷ ((r − r_t)! r_t! (n − r − t + r_t)! (t − r_t)! k! m!)
```

[BN06a] und [BN06b] greifen dies auf. Als Auswahl auf [Kor78] erfolgt eine Abschätzung für die Anzahl möglicher azyklischer Automaten, die nicht unbedingt frei von Isomorphie sind. Die Anzahl der isomorphen, deterministischen Automaten ist mit

$$|\mathscr{A}_n| \sim \frac{1+\sum_{r=1}^{\infty}\frac{1}{r}\binom{kr}{r-1}(e^{k-1}\beta_k)^{-r}}{1+\sum_{r=1}^{\infty}\binom{kr}{r}(e^{k-1}\beta_k)^{-r}} n 2^n \frac{1}{n!}\sum_{j=0}^{n}(-1)^j\binom{n}{j}(n-j)^{kn} \tag{2.5}$$

in [BN06b, Theorem 16] angegeben. Zur Automatengewinnung selbst dient ein sechsseitiger Spielwürfel als notwendiges Zufallselement. Weder die Formel 2.5 noch der dort angegebene Algorithmus zur Automatengenerierung sind für einen allgemeinen technischen Ansatz aufgrund der Anforderungen verwendbar.

Es gibt noch zahlreiche weitere Literaturstellen, die sich mit dieser Thematik der Automatenaufzählung bzw. -anzahlbestimmung befassen. Eine Zusammenstellung bietet [Bas05, Kapitel 4].

In [DKS02] ist eine obere Grenze für die Anzahl nichtisomorpher, minimaler deterministischer finiter Automaten (DFA) mit n Zuständen und k Buchstaben Eingabealphabet angegeben, die eine endliche Sprache akzeptierten. Dortiges Kapitel 6 enthält eine Tabelle mit Werten, wobei die Anzahl der Terminalzustände unter gewissen Bedingungen variabel ist (siehe dort Proposition 14). Diese variable Anzahl der Terminalzustände entspricht bei dem verwendeten Modell nach Algorithmus 2.4 der Addition aller Ergebnisse für jeden sich aus dem gegebenen Wertebereich für Terminalzustände ergebenden Resultat, also unter Nutzung der Formel 2.1 ist die Anzahl der Automaten gleich $\sum_{i=0}^{t} a(n,k,1,i)$ mit t als Obergrenze der möglichen Terminalzustände.

Eine theoretische Betrachtung für die Anzahl möglicher Automaten, um eine unendliche Eingabe eines Alphabets mit endlich vielen Zuständen zu akzeptieren, bietet [Dom03] mit einer Formel, die über eine Kombinatorik eine Ober- und Untergrenze ermittelt. Aber auch andere Ideen für eine Arbeitsweise gibt es. So bietet [KVBSV97] ein Verfahren an, um mit Hilfe der transitiven Hülle nichtdeterministische Automaten aufzuzählen.

Insgesamt lässt sich kein einheitliches Konzept oder eine übereinstimmende Herangehensweise erkennen, dafür aber bieten die mathematischen Arbeiten verschiedene Anregungen für ein eigenes Konzept und zeigen die notwendigen Parameter auf.

2.3.3 Automaten in der Theoretischen Informatik

Neben den bisher behandelten deterministischen endlichen Automat (DFSM, *deterministic finite state machine*, siehe z. B. [HMU02]) gibt es auch noch nichtdeterministische Automaten, bei denen sich ein Automat zu einer Zeit auch in mehreren Zuständen befinden kann.

Die Bedeutung des Nicht-Determinismus zeigt das Beispiel mit den möglichen Eingaben $\{a,b\}$ und einem erkennenden Automaten, dargestellt in Abbildung 2.2, mit Zustandswechsel von 0 nach 0 bei beliebiger Eingabe und von 0 zum abschließenden Endzustand 1, wenn die Eingabe b ist. Damit ist die akzeptierte Sprache $L = (a+b)^* + b$. Bei Zustand 0 und aufkommender Eingabe b ist nicht sofort entscheidbar, welcher Zustandsübergang zum tragen kommt, es liegt also ein Nicht-Determinismus vor.

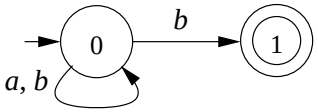


Abbildung 2.2: Beispiel für einen nicht-deterministischen Automaten

Ein in polynomialer Zeit nicht-handhabbares Problem ist in exponentieller Zeit oder mit einer nichtdeterministischen polynomialen (NP) Turing-Maschine lösbar. Bei großen Problemgrößen ist dies aber nicht effizient lösbar. Zahlreiche Beispiele hierfür sind in [GJ79] gegeben. Das folgende Zitat aus [HMU02, Seite 455] zeigt die Problematik, die bei der Lösung von NP-Problemen entstehen.

> „Wenn wir ein Problem als NP-vollständig erkennen, dann wissen wir, dass nur eine geringe Wahrscheinlichkeit besteht, einen effizienten Algorithmus

> zu entwickeln, der das Problem löst. Wir sollten also nach Heuristiken, partiellen Lösungen, Näherungswerten oder anderen Möglichkeiten suchen, um zu vermeiden, das Problem in voller Allgemeinheit direkt anzugehen. Wir können das guten Gewissens tun, weil eben eine praktisch brauchbare allgemeine Lösung fast sicher nicht existiert.“

Eines der bekanntesten NP-vollständigen Probleme ist das des Handlungsreisenden (*traveling salesman*). Verschiedene Städte, deren Abstände zueinander bekannt sind, sollen alle auf einer Route genau einmal erreicht werden, die Route darf dabei eine maximale Länge haben bzw. soll minimal sein. Es gibt verschiedene Näherungsverfahren, einen solchen Lösungsansatz stellen die Ameisenalgorithmen dar, z. B. beschrieben in [DS04].

2.4 Systeme

Um ein Problem zu lösen, ist eine systematische Herangehensweise notwendig. Ein System bietet die Möglichkeit, ein berechnetes Ergebnis z. B. durch eine Simulation herbeizuführen. Dabei gilt es aber einiges zu beachten, insbesondere eine geeignete Datenstruktur und ein effizientes Berechnungskonzept sind fundamental. Verschiedene Herangehensweisen sind im Folgenden erläutert, mit denen sich auch die bekannten Aufgabenstellungen wie das Density Problem oder das „Firing Squad Synchronization Problem“ aus [Bal67], beschrieben in [Goo98, Seiten 92ff.], lösen lassen.

Die hier vorgestellten Probleme basieren meist auf einem zellularen Automaten. Dieser besteht aus einzelnen Zellen und deren, in diesem Fall statischen Verbindungen. Die Zellen selbst sind in einem regelmäßigen Raster angeordnet, die Verbindung ist nur mit den direkten Nachbarzellen gegeben. Eine Zelle ist im wesentlichen ein Zustandsautomat, wie er in Abschnitt 2.3.1 beschrieben ist. Eine Einführung in die Thematik gibt z. B. [TM87].

2.4.1 Roboter

Einen eher theoretischen Problemlösungsansatz beschreibt [HAB+02] mit unendlich vielen Robotern, die über eine Tür in eine Umgebung eintreten, um dann vollständig das eingegrenzte, polygone Gebiet zu überqueren. Die Roboter, die sich an Leitpositionen orientieren, lösen die Aufgabe in der Simulation erfolgreich. Dabei bestehen sie aus einem einfachen finiten Automaten, haben lokale Kommunikation und Sensoren sowie einen Speicher; genauere Angaben sind leider nicht gemacht. Trotz allem erscheinen die mobilen Gebilde doch recht komplex, so dass sich evtl. die Aufgabe auch mit weniger Aufwand lösen lassen sollte.

Ein Fundament für zahlreiche weitere, unterschiedliche Forschungsprojekte bildet [PR90], das ein System mit dem Namen Tileworld beschreibt. Dieses besteht aus einem rechteckig abgegrenztem Gebiet, auf dem sich Hindernisse, bewegliche Ziegel, Löcher und ein Agent befinden. Ziel des intelligenten Agenten ist es, jeweils einen Ziegel in ein mit Punkten versehenes Loch zu befördern. Sind alle zusammenhängenden Löcher mit Ziegeln abgedeckt, so erhält der Agent die angegebenen Punkte. Ein veränderlicher Schwierigkeitsgrad ist die Variabilität von Hindernissen, Ziegeln und Löchern, die während einer Simulation auftauchen und verschwinden können. Bei entsprechend hoher Veränderungsgeschwindigkeit kann ein Agent erfolgreicher sein als ein gleichzeitig dagegen antretender Mensch, der einen Agenten auf dem Bildschirm steuert.

Agenten mit künstlicher Intelligenz beschreibt ausführlich das Buch [HR99]. Die Durchführung erfolgt auf einem zwei Dimensionen umfassenden Feld ähnlich der Tileworld, es ist ein Sensor mit den Eingabewerten *clear* und *blocked* vorhanden sowie ein Effektor mit den Bewegungsanweisungen *move, clock, anti* (gegen den Uhrzeigersinn auf der Stelle drehen). Die Basis für die Aktionen bildet ein zehn Zustände umfassender Zustandsautomat mit fest vorgegebenen Zustandsübergängen und Ausgaben entsprechend den Eingangswerten. Der Agent löst erfolgreich die Aufgabe, an sich verändernden Hindernissen vorbei einen Zielpunkt zu erreichen. In [Dro93] ist aufgezeigt, dass mehrere Agenten ohne eigenes Ziel trotzdem ein globales Ziel erreichen können.

Aber nicht nur die Eigenschaften eines Agenten sind wichtig. Ein Projekt der TU Berlin zeigt dazu mit [Hom04], wie wichtig es ist, verschiedene Fachrichtungen zueinander finden zu lassen, damit ein System effizient zusammenspielt. In einem internationalen Wettbewerb in den USA hat das fliegende Erkundungssystem als Bestes die Aufgabe vollständig autonom durchgeführt.

2.4.2 Selbstkonfigurierende Systeme

Ein Aspekt der Ausfalltoleranz ist die Existenz von Ersatz. Das System ReCoNets (reconfigurable network) aus [HKT03] stellt bei Ausfall einer Verbindungsstruktur die Verbindung auf anderer Route wieder her. Damit der Programmablauf fortgesetzt werden kann, ist eine Rekonfiguration der verwendeten Hardware notwendig, da nicht alle Programme gleichzeitig auf einem FPGA verfügbar sind. Hierzu werden eventuell laufende Tasks angehalten und auf andere Systeme verlagert, die dazu in der Lage sind, und dann die Rekonfiguration des ganzen FPGAs vorgenommen. Der Test mit vier Experimentalplatinen verlief erfolgreich.

Einen anderen Ansatz der Rekonfiguration wählt [HEW05] mit SDAARC – Self-Distributing Associative Architecture. Die Aufteilung eines Algorithmus erfolgt in parallel ausführbare Programmteile (Threads), um dynamisch zur Laufzeit die berechnenden Komponenten (Prozessoren) optimal auszulasten. Darin ist das Projekt Self-Distributing Virtual Machine enthalten. Es verteilt die Programmausführung in einem heterogenen Rechnernetz und berücksichtigt dabei auch Rechnerausfälle, bekannt unter der Bezeichnung *Organic Computing*.

Ein rekonfigurierbares System ist in [US06] vorgestellt, das als Beispiel verschiedene Zellularautomatenprobleme erfolgreich implementiert. Dazu errechnet ein auf dem FPGA befindlicher Soft-Prozessor die Teil-Rekonfigurationsdaten und führt mit diesen einen Eignungstest durch. Mit dem bestqualifiziertesten Ergebnis erfolgt dann die eigentliche Problemlösung.

Eine Verbesserung der Problemberechnung durch Verlagerung von universellen Computern hin zu Datenflussprozessoren, mit teilweise frei konfigurierbaren, teilweise fest vorgegebener Berechnungslogik – gemäß dem Prinzip „die Berechnung zu den vielen Daten bringen statt umgekehrt“, nimmt [Har06] in Angriff. Dadurch verringert sich nicht nur die Notwendigkeit der Datenzwischenpufferung, sondern auch der Energieverbrauch und Platzbedarf für Rechenanlagen. Zudem läuft die Berechnung schneller ab, trotz geringerer Taktfrequenz der FPGAs. Bei DPU (Data Processing Unit) gibt es auch keinen Instruktionszähler, sondern die Arbeit beginnt, sobald Daten verfügbar sind. Der am Ende des Kapitels 3 entstandene Hardwareentwurf arbeitet nach dieser grundsätzlichen Idee, allerdings nur mit einer speziellen Berechnungseinheit und einer daran angeschlossenen Ausgabeeinheit.

2.4.3 Hardware für Simulation

Ein Beispiel aus dem Jahre 1994 verdeutlicht den Sinn einer speziellen Hardware trotz des damit verbundenen hohen Aufwands. In [MS94] erfolgt ein Vergleich zwischen einer pipelinebetriebenen Spezialhardware und dem damals sehr leistungsstarken universellen Computer Cray 2-XMP bezüglich der Rechenkapazität zur Simulation eines zellularen Automaten: beide haben 100 000 000 Updates pro Sekunde bei einer Feldgröße von 4 194 304 Zellen, also einem Takt von etwa 23,84 Hz pro Generation – mit erheblich unterschiedlichen Kosten und Einarbeitungskomplexitäten zugunsten der Spezialhardware.

Eine andere, allgemeine Zellularautomaten-Hardware ist CEPRA (Cellular Processing Architecture), entwickelt an der Technischen Hochschule Darmstadt mit Beginn der Veröffentlichungsreihe durch [HVS94], die ihren bisherigen Abschluss in [HUVW00] und [HUWV01] mit einer Architektur zur Verarbeitung von Datenströmen fand. Prinzipiell erfolgt die Speicherung des vollständigen Feldes in der Hardware, lediglich eines Teil des zellularen Feldes kommt zur Verarbeitung. Um nun nicht das vollständige Feld doppelt vorzuhalten, sind nur die noch benötigten Nachbarzellen mit ihren alten und neuen Werten gespeichert. Die Art der Datenverarbeitung, -speicherung und Möglichkeiten der Visualisierung ändert sich bei den verschiedenen Versionen der unterschiedlichen CEPRA-Architekturen.

Es gibt auch noch andere Systeme, zum Beispiel in [BK99] mit Spezialprozessoren, die direkt die Simulation künstlichen Lebens unterstützen, die einen speziellen, auf die Problemstellung abgestimmten Assembler anbieten.

2.4.4 Spezialhardware

Statt mit einem universell programmierbaren Spezialrechner eine Simulation durchzuführen, ist es möglich, eine Hardware speziell für ein Problem zu konstruieren. Hier sind nochmals Leistungssteigerungen möglich, wie die eigenen Forschungsergebnisse zeigen. Den ersten Entwurf für Simulationssysteme aus dieser Reihe für das Kreaturen-Problem sowohl mit Hardware als auch mit Software gab es in [Tis04], das auch in [HHHT04] dargestellt ist. Dort erfolgt neben der Einführung in die Thematik „Moving Creatures“ auch die Auflistung der initialen Felder und die damit verbundenen, erfolgreichen Zwei- und Vier-Zustandsautomaten, die mit der prinzipiell dargestellten Hardwarearchitektur entstanden.

Zuvor gab es Untersuchungen bezüglich der optimalen Umsetzung sowohl in Hardware als auch in Software, um so einen gerechten Vergleich durchführen zu können. Mit einer Moore-Nachbarschaft ist das Resultat in [HHR04] aufgeführt, gleiches mit einer von-Neumann-Nachbarschaft in [HH04]. Im ersten Fall gab es eine Geschwindigkeitssteigerung (*speed up*) zum Vorteil der Hardware um den Faktor 14 für eine komplexe Berechnungsregel bzw. 19 für eine nicht so komplexe Variante. Mit der kleineren Nachbarschaft ohne die diagonalen Zugriffe auf die Nachbarzellen entwickelten sich für andere, der Umgebung angepasste Problemstellungen die Faktoren 3 und 22. Zum Einsatz kamen dabei jeweils ein FPGA von Altera mit der Bezeichnung Flex EPF10K70RC240-4, betrieben mit 25,175 MHz und ein Computer mit einem Pentium 4 mit 2,4 GHz.

Mit der bestehenden Grundlage sind weitergehende Experimente möglich. So gab es in [HH05a] erstmals Ergebnisse mit Fünf-Zustandsautomaten. Die verwendete Hardware hat eine Aufteilung in Umgebung, Kreaturen und Bewertungslogik erfahren, so dass die Berechnung für fünf Felder aus Abbildung C.1 parallel erfolgte, deren Ergebnisse mittels USB-Übertragung zur Aufzeichnung und weiteren Auswertung geleitet wurden. Des

weiteren gab es erstmals den Begriff „Capacity of Intelligence (COI)“, um die Anzahl der Bits zur Zustandsspeicherung zu beschreiben. Mit vorangeschrittener Berechnungsdauer gab es dann auch erste Ergebnisse für Sechs-Zustandsautomaten in [HH05b].

Um größere Felder in der Hardwareberechnung bewerkstelligen zu können, bedurfte es eines neuen Konzeptes für die Aufteilung der einzelnen Berechnungsgruppen; [HH06a] hat dies untersucht. Die entstandenen Konzepte stellen die Abschnitte 4.3.3.1, 4.3.3.2 und 4.3.3.3 dar.

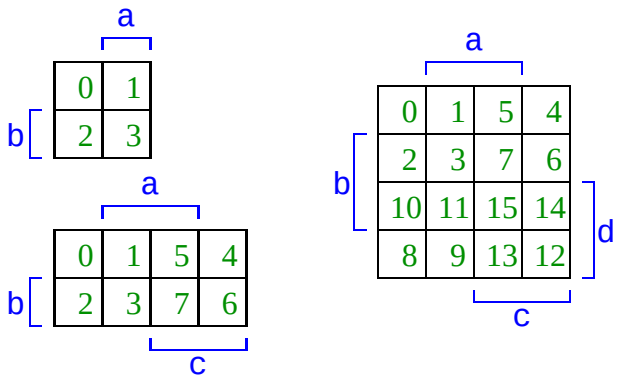
Eine Einteilung der Automaten, wie sie in Abschnitt 3.2 fortgeführt ist, nimmt in einem ersten Ansatz [HHB06] vor. Die Automaten, die am Besten die Simulation der Felder aus Abbildung C.1 hardwareermittelt bestanden haben, mussten sich dann auf weiteren, in Abschnitt C.1 aufgeführten Umgebungen bewähren. Um eine Vergleichbarkeit zu ermöglichen, bildet sich aus Robustheit und Geschwindigkeit eine Bewertungsformel. Mit der neuen Software ist es zusätzlich möglich, mehrere Kreaturen auf einem Feld zu haben, genauere Ergebnisse sind für ein quadratisches, 15 Zellen breites Feld mit unterschiedlicher Anzahl Kreaturen in [HH06b] aufgeführt. Dort ist auch die Idee der Effizienzsteigerung bei Kollision thematisiert.

Weitergeführt auf größere Felder bezogen betrachtet [HH07b], vertieft in [HH07c], die effektive Zusammenarbeit mit unterschiedlicher Anzahl Kreaturen für die in Abbildung C.7 aufgeführten Felder. Die unterstützende oder verhindernde Wirkung von Kollisionen stellt sich durch die statistischen Daten deutlich dar.

Eine neue Architektur, die auch größere Felder als bisher für die Hardwareberechnung ermöglicht, präsentiert der Aufsatz [HH07a], der sich auch in Abschnitt 4.3.3.4 wiederfindet. Die zur Simulation verwendeten Automaten stammen noch aus den früheren Versuchen, eine automatische Ermittlung mit der neuen Konstellation größerer Felder gab es bisher nicht.

2.4.5 Optimierungsmöglichkeiten

Eine Basis der Optimierung stellt die Minimierung logischer Bedingungen dar, anschaulich dafür ist der Karnaugh-Veitch-Plan nach [Kar53]. Er bedient sich der schematischen Darstellung von vorzugsweise zwei bis vier (Eingangs-)Variablen. Grundlage ist ein Feld wie in Abbildung 2.3. Dabei wird – in Abhängigkeit der Eingänge – der Ausgabewert (ein Bit mit dem Wert `0` oder `1`) in die Felder eingetragen.



$$Feld = (\texttt{a} \equiv 1) * 1 + (\texttt{b} \equiv 1) * 2 + (\texttt{c} \equiv 1) * 4 + (\texttt{d} \equiv 1) * 8$$

Abbildung 2.3: KV-Diagramme für zwei bis vier Variablen

Ursprünglich wird ein KV-Diagramm verwendet, um einen minimalen logischen Ausdruck herzuleiten. Dazu werden die `1`-Ergebnisse markiert und im Block gruppiert, wobei dies auch über den Rand hinaus möglich ist. Eine `1` darf auch mehrfach in verschiedenen Gruppen enthalten sein. Die Bedingung für den Block lässt sich anhand der Randbeschriftung ablesen. Die einzelnen Blöcke werden „oder“-verknüpft. Abschließend kann nach mathematischen Regeln die Formel vereinfacht werden (z. B. unter Anwendung des Distributiv-Gesetzes).

Compiler-Optimierer arbeiten nach einem anderen Berechnungsverfahren, das aber zu einem gleichen Ergebnis kommt. Von daher ist die Angabe von reduzierten Bedingungen ohne Einfluss auf die Durchführung.

Für die Programmierung in einer sequentiellen Programmiersprache wie z. B. C, beschrieben in [KR88], ist es einem selbst möglich, neben der Optimierung durch den Compiler wesentlich zu Erhöhung der Ausführungsgeschwindigkeit beizutragen, wie sich bei der Umsetzung von Zellularautomaten in [Röd03] und der Zusammenfassung in [HHR04] und [HH04] gezeigt hat.

Eine Option, die auch der Compiler bewerkstelligen kann, aber auch zur besseren Lesbarkeit beitragen, ist das Aussondern von nicht-erreichbaren Code. Zu Vermeiden sind z. B.

- eine Bedingung kann niemals wahr werden,
- eine Berechnung wird nicht weiter verwendet,
- eine Zuweisung auf die gleiche Variable erfolgt mehrfach, obwohl dazwischen keine Auswertung erfolgt.

Auch hilft es, die Prozessorarchitektur zu beachten. Zum einen gibt es Ausrichtungen der einzelnen Bytes zu den Wortadressen (Alignment), die bei Zugriffen auch auf Nachbarzellen beachtet werden sollen. Dafür kann es hilfreich sein, die Speicheranordnung der Zellen zu ändern, so dass ein optimaler Zugriff möglich ist. Zum anderen kann es sein, dass ein Prozessor spezielle Befehle für Mehrfachoperationen oder nur eingeschränkte Operationen anbietet, z. B. bei den MMX-Befehlen für $8 \cdot 8$-Bit-Operationen nur $>$ und $=$ statt auch $<$, $\leq$ etc.

Ein weiterer Punkt ist das Vermeiden von Sprüngen. Für bedingte Anweisungen (`if-then-else`) kann es günstiger sein, mit Masken zu arbeiten. So kann aus `if (a > 5) s = 3` die Berechnung $s = (a > 5) \cdot 3 \mid \overline{(a > 5)} \cdot s$ werden, wobei der Vergleich $(a > 5)$ eine 1 im Erfolgsfall liefert. Entspricht hingegen das Ergebnis der Bit-Breite von z. B. 8 Bit, dann kann die Multiplikation durch ein „und“ (&) ersetzt werden, was zu einer weiteren Beschleunigung führen kann.

Speziell bei der Simulation von zellularen Automaten, bei der pseudo-parallel ein zellulares Feld für die nächste Generation neu zu berechnen ist, gilt die Aufmerksamkeit der Speicherung eben diesen Feldes. Die Speicherung von nur einem vollständigen Feld ist möglich, wenn für die Nachbarzellen bei zeilenweiser Berechnung nur die vorhergehende und aktuelle Zeile (und zur Vereinfachung evtl. auch die nächste Zeile) mit Daten der alten Generation gespeichert wird. Für den Zugriff empfiehlt sich eine eigene, bei jedem Durchlauf inkrementierte Zeigervariable. Dann entfällt die Berechnung des Indexwertes.

2.5 Problemlösungsstrategien

Im folgenden sind spezielle Problembeschreibungen betrachtet, die von einem Automaten lösbar sind. Ein Automat kann auch Basis für eine nicht-natürliche Lebensform sein, z. B. eine künstliche Kreatur, ein Agent oder ein sonstiges mobiles Objekt. Ziel ist eine Algorithmenoptimierung, um allgemeinere Aufgaben nach einem gefundenen Schema lösen zu können.

2.5.1 Gebiet vollständig überqueren

Eine inzwischen bekannte technische Neuerung ist der autonome Rasenmäher. Dieser arbeitet nach einem einfachen Prinzip, z. B. beschrieben in [Fri03]. Zuerst ist das Gebiet abzugrenzen, so dass der Roboter nicht darüber hinausfährt und auch nicht innerhalb des Gebietes in verbotene Bereiche vordringt. Anschließend erkundet der Roboter die Randbegrenzung, um so danach von einer idealen Position aus mit einer vorgegebenen Winkeleinstellung alle Stellen des Rasens zu erreichen und dabei zu mähen. Bei einem zu großen Gebiet sind Zwischenwege zur Basisstation zum Aufladen der Akkumulatoren und Entleeren des Rasenauffangbehälters erforderlich.

Es gibt auch Staubsauger mit einer Automatik, die laut [Jon03] auch eine künstliche Wand ähnlich dem Rasenmäher erkennen können. Zusätzlich zu den Sensoren für Hindernisse gibt es auch solche für fehlenden Boden, z. B. durch nach unten führende Treppen, die der Roboter nicht bewältigen kann. Die Automatik kann auch zugunsten einer handgesteuerten Fernbedienung abgeschaltet werden. Andere Modelle können selbstständig automatisch zur Basisstation zurückkehren, um die Akkumulatoren aufzuladen und den Auffangbehälter zu entleeren. Der Routenverlauf basiert auf einem zufälligen Suchbetrieb, bei stark verschmutzten Stellen erweitert um eine sternförmige Bewegung.

2.5.2 Raum durchqueren ohne Karte

Um minimale Eigenschaften für das Abschreiten eines Gebietes festzulegen, bietet [Hie73] eine Hilfestellung. Ziel ist es, einen möglichst ununterbrochenen Linienzug durch eine Punktmenge zu führen, wobei die Punkte untereinander Verbindungen aufweisen und somit jeder Punkt eine gewisse Anzahl an Anknüpfpunkten aufweist. Der Bericht zeigt auf, dass eine gerade Anzahl an eben diesen Anknüpfpunkten notwendig ist, es sei denn, es handelt sich dabei um Anfangs- oder Endpunkt für einen offenen im Gegensatz zu einem geschlossenen Linienzug.

Weiterführend ist der Aufsatz [Pea90], ergänzt durch die Arbeit [Hil91]. Giuseppe Peano entwirft ein mathematisches Modell, um von einer reellen Zahl aus dem Bereich von 0 bis 1 auf eine Fläche mit gleichen Wertgrenzen der Seitenlängen stetig und eindeutig abzubilden. David Hilbert entwickelt dies zu einem Mechanismus weiter, der gleichzeitig das Abschreiten einer Geraden auf einen Punkt im Quadrat anschaulich visualisiert. Dabei entsteht ein Weg durch das Quadrat. Zum einen ist damit gezeigt, dass es auf einem Raster in einem Quadrat einen Weg gibt, zum anderen, dass es mit den einfachen Bewegungsformen rechts, links und vorwärts möglich ist, eine vollständig abdeckende Strecke durch ein quadratisches Raster zu finden, ohne dabei eine Rasterfläche doppelt überqueren zu müssen.

Eine Anwendung bietet [BZ06]. Für eine Matrixmultiplikation mit vielen Werten ist ein effizientes Durchschreiten des Speichers notwendig, um den Cache möglichst optimal zu nutzen. Ein trivialer Ansatz liefert angeblich meist nur schlechte Ergebnisse. Anders

bei Einsatz der eben erwähnten Wegfindung, die eine sichtbare Effizienzsteigerung erbringt.

Einen anderen Ansatz wählt [MSPPU02] mit der selbstlernenden Methode namens *State-Observation-State*, bei der für jeden Folgezustand eine Auswahlbewertung vorhanden ist. Der Folgezustand in Abhängigkeit von aktuellem Zustand und Eingabewert aus der Umgebung, der den meisten Erfolg verspricht, wird ausgewählt. Lernen bedeutet eine Veränderung der Auswahlbewertung, die in diesem Fall auf der Bewertung des Erfolgs von zwei Zustandsübergängen beruht. Mit einem solchen Automaten ist es dann z. B. möglich, einen Irrgarten zu durchlaufen. Das Lernverfahren hat einen hohen Rechenaufwand zur Folge, ist also für einen minimalistischen Ansatz nicht geeignet, wohl aber der dadurch entstehende Automat. Allerdings kann mit diesem Mechanismus nach eigener Aussage nicht immer eine Lösung garantiert werden und der Erfolg ist im Wesentlichen von den richtigen Einstellungen der Parameter der Bewertungsfunktion abhängig.

Auch die Komplexität von Algorithmen generell ist nicht außer Acht zu lassen, so dass ein effizienter Programmablauf auch bei steigender Problemgröße gewährleistet ist. Dabei gibt es eine Problemklasse, beschrieben in [Kar72], die möglicherweise nicht mit polynomialen Algorithmen zu lösen ist. Dazu zählt auch die Routenplanung, die für ein Abschreiten eines Gebietes notwendig ist.

2.5.3 Schwarm: Verhalten einer Gruppe

In Kooperation Aufgaben leichter lösen ist das Ziel eines Schwarms in der Natur. Der Artikel [Kla06] zieht eine Analogie eines Fischschwarms ohne sichtbaren Anführer mit dem Verkehrsfluss auf einer vollen Autobahn. Von außen betrachtet scheint es Verhaltensregeln zu geben, an die sich alle halten, die aber so nicht offensichtlich sind. Auch das Verhalten von Ameisenkolonien zur Nahrungssuche gehorcht Regeln, die sich aber bei z. B. doppelter Bevölkerung mehr als doppelt effizient auswirken. Für einen Schwarm sind die Regeln

- Kollision vermeiden,
- in der Gruppe bleiben,
- Bewegungsrichtung ist denen in der näheren Umgebung anzupassen

ausreichend. So zeigt sich für zehn Heuschrecken noch eine zufällige, zusammenhängende Verhaltensweise, aber bereits bei einer Gruppengröße von 30 agiert der Schwarm als ein Lebewesen.

Um einen Schwarm zu lenken, reichen etwa fünf Prozent als Experten aus, um den Schwarm zu beeinflussen, ohne das der Schwarm weiß oder mitbekommt, wer diese Experten sind, wie Versuche aus [CKFL05] gezeigt haben. Dort sind auch Formeln für den erforderlichen Abstand aufgeführt.

Einen anderen Herdenantrieb untersucht [HFV00] mit dem Fluchtverhalten eines Menschenschwarms. Hier kommt es eher auf die Geschwindigkeit eines Einzelnen und auf die Gestaltung der Hindernisse an, die teilweise ein Schwarmverhalten aufzeigen. Ein zellulares Automatenmodell hierfür mit den erforderlichen Regeln stellt [BA00] auf, die das tatsächliche Verhalten von Fußgängern mit unterschiedlichen maximalen Geschwindigkeiten in bestimmten Situationen abbilden.

Ist erst einmal das Grundverhalten von Lebewesen – bzw. künstlichen Kreaturen – erkannt bzw. festgelegt, so ist ein mehr oder weniger kooperativer Umgang unter Nutzung

von Regeln eines Schwarms für Experimente interessant, um so ein optimales Verhalten künstlicher Kreaturen in der Gruppe mit einfachen Mitteln erreichen zu können.

2.5.4 Optimierung durch naturnahe Berechnung

Eine Variante für die Suche nach dem optimalen Weg bieten Ameisenalgorithmen, vorgestellt und angewendet in [Foc06]. Der Algorithmus arbeitet nach dem Prinzip, dass Ameisen Pheromone ausschütten, die zur Wegorientierung aller Ameisen dienen. Je öfter ein Weg benutzt wird, um so mehr Pheromone befinden sich darauf. Je kürzer ein Weg ist, um so mehr Ameisenüberquerungen finden darauf statt. Von daher ist die Suche nach einem günstigen Weg gegeben.

Ein Problem dabei ist die Generierung eines Wegenetzes bzw. eines Graphen, auf dem die Ameisen agieren. Hier sind die verschiedenen Einflussfaktoren zu berücksichtigen, die in der Natur Hindernissen und Weglängen entsprechen. In [Foc06, Seiten 186ff.] sind die vier Parameter der Problemstellung auf 19 Arten unterschiedlich gewichtet aufgeführt, die allesamt Eingang in die Lösung gefunden haben. Allein mit der Auswahl eines Algorithmus ist demnach ein Problem noch nicht gelöst, aber der Lösungsweg ist aufgezeichnet.

Der Ameisenalgorithmus ist bei der Suche nach Entscheidungen hilfreich, es ist aber eine feste Vorgabe der Wege oder eine entsprechende Vorauswahl z. B. mit Hilfe eines anderen heuristischen Verfahrens erforderlich.

Einen ganz anderen Weg geht die organische Methode der *Marching Pixels*. Für eine schnelle optische Erkennung konzipiert, z. B. zur Erkennung des Schwerpunktes eines Werkstückes, beschreiten Pixel nach einem konfigurierbaren Verfahren eine Ebene, das in diesem Fall aus einem Schwarz-Weiß-Bild einer CMOS-Kamera stammt.

Gemäß [FS05b] bewegen sich die Pixel nach einer Initialisierungsphase, um die Kanten des Bildes zu finden, z. B. erst in der horizontalen Richtung. Treffen verschiedene Pixel aufeinander, ändert sich ab einen gewissen Schwellwert das Bewegungsmuster in die Vertikale, wobei sich die Anzahl der sich bewegenden Pixel verändern kann. Kommen danach an anderer Stelle weitere Pixel zusammen, so ist der Schwerpunkt gefunden. Weiterentwicklungen sind in [FS05a] und [KF07] enthalten, unter anderen ist ein industrieller Einsatz mit einer Zeitbeschränkung von nur 10 ms pro Bild, also 32 ns pro Pixel für eine eigens entwickelte Hardware beschrieben.

Die Regeln zur Steuerung der Pixel sind manuell oder durch ein genetisches Auswahlverfahren gestaltet und der Problemstellung angepasst. Zwar kamen verschiedene Algorithmen zum Einsatz, es ist jedoch nicht sichergestellt, dass der gewählte Algorithmus ein Optimum darstellt. Eine Abhilfe schafft der systematische Durchlauf, wie im Kapitel 5 dieser Arbeit beschrieben.

2.5.5 Künstliche Intelligenz

Auch ein Agent, der aufgrund von Eingaben entsprechende Ausgaben erzeugt, könnte z. B. die Aufgabe der Wegsuche erledigen. Dieser lässt sich auch mit Hilfe der künstlichen Intelligenz konstruieren. Die Möglichkeiten, die sich durch dieses Forschungsgebiet ergeben, zeigt die nachfolgende Übersicht auf, die sich an [RN03] orientiert.

Bei der Erstellung eines solchen Agenten ist eine Bewertung über den Erfolg erforderlich. Hierzu sind genaue Eingrenzungen erforderlich, wie ein Beispiel aus [RN03, Kapitel 2] verdeutlicht: Ein Agent in Form einer Person sieht, als er in Paris die Champs

Élysées entlanggeht, einen alten Freund auf der anderen Straßenseite. Da kein Verkehr in der Nähe ist und er Zeit hat, beschließt er, die Straße zu überqueren. Währenddessen, in 33 000 Fuß Höhe, verliert ein Flugzeug eine Frachtraumtür, wie es nach einem Bericht der Washington Post am 24. August 1989 von N. Henderson mit dem Titel „New door latches urged for Boing 747 jumbo jets“ bereits vorgekommen ist. Bevor der Agent die Straße überqueren kann, wird er von dieser Tür getroffen. War es nun irrational, die Straße zu überqueren? Wenn also eine Bewertung einen unerwünschten Eventualfall nicht enthält, die jedoch von einem Agenten „ausgenutzt“ wird, so ist auch die Frage zu stellen, ob es im Rahmen der Bewertung nicht doch ein gültiges Verhalten darstellt.

Bei der Entwicklung eines Agenten stellt sich die Frage, in welcher Form dies geschehen soll. Ein tabellengesteuerter Agent, der auf alle Vorkommnisse eine Antwort gespeichert hat, verbraucht neben viel Speicherplatz auch Arbeitszeit, diese Tabelle zu erstellen. Aber allein schon für einen Schachcomputer sind 10^{150} Einträge erforderlich, die Anzahl der Atome im bekannten Universum beträgt aber nur etwa 10^{78} gemäß [BBW99]. Von daher sind algorithmische Beschreibungen vorzuziehen. Aber auch hier gibt es verschiedene Beschreibungsformen. Eine einfache Variante ist die Angabe von Verhaltensregeln (*simple reflex agents*), die unter Zuhilfenahme von Zuständen und damit Zustandsübergängen sowie Ausgabeverhalten erweitert werden kann (*model-based reflex agents*). Eine weitere Zielorientierung ist die Einbindung von Planung, in der ein Agent mehrere zukünftige Schritte vorausberechnet und bewertet, bevor es zur Ausführung kommt (*goal-based agents*). Um den Agenten erfolgreicher zu gestalten, ist es erforderlich, dass der Agent unmittelbar über den Erfolg seiner Handlung erfährt, um entsprechend selbst in sein zukünftiges Verhalten eingreifen zu können. Dies könnte mit dem Streben nach „Glücklichsein“ beschrieben werden (oder mehr wissenschaftlich mit funktionell, *utility* bezeichnet). Erreichen lässt sich dies durch Einbindung einer zusätzlichen Stufe zwischen Planung und Ausführung (*model-based, utility-based agent*).

Bei einem zu Anfang nicht statisch festgelegten Verhalten kann der Agent hinzulernen. Dies erlaubt dem Agenten, sich in einer anfangs unbekannten Umgebung zurechtzufinden und Kompetenz zu seinem ursprünglichen Wissen aufzubauen. Dazu muss der Agent Kritik relativ zu festgelegten Ausführungsstandards von außen entgegennehmen können und diese als Feedback in einem Lernelement verwerten. Dieses ist mit einem Ausführungselement gekoppelt, das die Auswertung der Sensoren (Umgebungserfassung) zu einer Ausführung (auf die Umgebung) übernimmt; die Anbindung erfolgt über Änderung des Verhaltens und Einbindung von Wissen. Des weiteren ist ein *Problem Generator* notwendig, damit kurzfristige Misserfolge, die aber zu einem langfristigen Erfolg führen, aufgrund des Lernziels trotzdem insgesamt als positiv bewertet werden können. Demnach ist auch eine Rückmeldung zusammenfassend für einen längeren Zeitraum unabdingbar.

Je nach Problemstellung gibt es verschiedene Möglichkeiten, zu einer Lösung zu gelangen, die eine Handlungsreihenfolge von Aktionen darstellt. Eine Variante ist die Suche nach einer optimalen Lösung durch sequentielles Ausprobieren aller Möglichkeiten in einer so genannten Ausführungsphase. Dies führt offensichtlich zum Ziel, jedoch ist der Untersuchungszeitbedarf nicht optimal.

Eine Einschränkung kann man in der Herangehensweise machen, indem zuerst Teilprobleme in Angriff genommen werden, die zu einer optimalen Lösung führen, und dann darauf aufbauend weiter arbeitet. Dazu muss allerdings auch die Problemstellung geeignet sein.

Bekannte Probleme, die mit einem solchen Verfahren bearbeitet werden können, sind:

- Schiebe-Puzzle ($n_x \cdot n_y$-Feld mit $n_x \cdot n_y - 1$ durchnummerierten Steinen, die in einer beliebigen Anfangskonfiguration sortiert werden sollen, wobei nur ein dem freien Feld benachbarter Stein auf dieses freie Feld verschoben werden darf);
- 8-Damen-Problem (auf einem Schachbrett sind 8 Damen derart zu platzieren, so dass diese sich nicht schlagen können, d. h. keine andere Damenfigur in einer waagerechten, senkrechten oder diagonalen steht);
- Traveling Salesman Problem (ein Handlungsreisender soll bestimmte Städte auf möglichst kurzem Wege besuchen);
- VLSI-Design (Platzierung von Bauelementen auf einem Chip und Verlegung von Verbindungsleitungen, wobei einige Eigenschaften zu berücksichtigen sind, z. B. kurze Laufzeiten und minimal entstehende Kapazitäten);
- Internet-Suche durch so genannte Roboter für Suchmaschinen;
- Roboter- oder Automobilnavigation.

Es wird deutlich, dass eine gute Suchstrategie erforderlich ist, um schnell zu einem Erfolg zu kommen, insbesondere bei Realzeitproblemen. Die einzelnen möglichen Lösungsschritte können in einer Baumstruktur dargestellt werden. Mit den bekannten Verfahren lässt sich dann dieser Baum abschreiten, um zu einer Lösung zu gelangen. Die Komplexität für Zeit und Platzbedarf variieren je nach Verfahren, die Vor- und Nachteile und damit die Wahl des Suchverfahrens hängen von den Parametern des Problems ab.

2.5.6 Zufallszahlenvorhersage für Roulette

Bei Roulette gibt es verschiedene Möglichkeiten, zu einem Gewinn zu gelangen. Dieses Glücksspiel ermöglicht, konkrete Zahlen wie bei einer Lotterie vorherzusagen, als auch bestimmte Bereiche zu erraten, z. B. ob die gezogene Zahl gerade oder ungerade wird. Die Ziehung dieser Zahl erfolgt mit einer Kugel, die in einem sich rotierenden Kessel in eines von 37 Fächern fällt. Diese Fächer sind alle gleich beschaffen, so dass keine der Zahlen von 0 bis 36 gegenüber einer anderen bevorzugt wird. Viele Roulette-Spieler wünschen eine verlässliche Vorhersage, um so zu einer Gewinnmaximierung zu gelangen.

Für die Ziehung einer Zahl selbst gibt es verschiedene Einflussfaktoren. In [Bas87] sind unter anderem Kesselschieflagen, Ballistik der Kugel sowie schwankender Wurfweitenschwerpunkt aufgeführt und untersucht. Es lassen sich die verschiedenen Einflussfaktoren Drehgeschwindigkeit des Kessels und Einwurfpunkt der Kugel in Relation zum Kessel, aber auch die daraus resultierenden Laufweiten und Kollisionsparameter der Kugel optisch beobachten oder akustisch wahrnehmen. Aus den verschiedenen Messungen eines Wurfs ergeben sich somit verschiedene Eingangsparameter, die Einfluss in eine Vorhersage finden müssen.

Von daher ist es nahezu unmöglich, alle Faktoren zu bestimmen, um sie für eine – vielleicht dann einfache – Vorhersage zu verwenden. Ähnlich ist es auch bei einer Wettervorhersage, bei der zahlreiche Einflussfaktoren vorliegen. Je mehr Messungen berücksichtigt werden, um so präziser ist die Vorhersage. Bei der Wettervorhersage hat das vorher Gewesene jedoch Einfluss auf die Zukunft.

Bei Roulette startet jeder Wurf nicht nur durch Austausch von Kugel oder Croupier erneut ohne vorherige Einflussfaktoren. Es gibt Ausnahmen, z. B. durch immer gleichen

Einwurf der Kugel an gleicher Kesselposition mit konstanten Geschwindigkeiten. Mit Ausnahme dieser Sonderfälle ist eine gezielte Vorhersage mittels eines Automaten nicht möglich. Genauso gut könnte hier ein auf einer Rauschvorrichtung basierender Zufallszahlengenerator Verwendung finden. Hierzu sei [Bas87, Seite 241] zitiert:

> „Jede Strategie im Roulette, ob sie mehr oder weniger spitzfindig ist, ob sie mehr oder weniger Geschicklichkeit erfordert, ob sie wissenschaftlich tatsächlich funktionieren könnte oder nicht, kann daher stets nur eine Spielanregung ohne Erfolgsgarantie sein."

Ausgehend von in Spielbanken üblichen Roulette-Tischen mit homogener Wahrscheinlichkeit für alle Zahlen stehen als alternative Methode verschiedene Spielstrategien zur Verfügung. Hierbei geht es weniger um konkrete Vorhersagen der Gewinnzahlen, sondern um das Setzen mit variablen Einsatzhöhen, die lediglich durch Limits der Spielbank begrenzt sein können. Die Durchführung der Gewinnstrategien erfordert lediglich ausreichend Startkapital, Zeit und die Durchführungsrichtlinien. Details und der damit verbundene Ertrag ist dem Buch [Kok93] zu entnehmen; die dort beschrieben Resultate der Verfahren basieren auf Computersimulationen. Gemäß der Tabelle 26, [Kok93, Seite 112], ergeben sich für die unterschiedlichen Spielprogressionsarten lediglich unterschiedliche Verlustraten, jedoch kein Gewinn. Von daher gibt es keine Aussicht, einen beliebigen Automaten als Ersatz für eine Durchführungsregel konstruieren zu können, um so eine erfolgreiche Gewinnstrategie zu erhalten.

Kapitel 3

Der Weg zum perfekten Automaten

Um den garantiert besten Automaten zu erhalten, ist es notwendig, alle interessanten Automaten aufzuzählen und zu testen, z. B. indem sie eine Bewertung durchlaufen. Da allerdings die Anzahl der Möglichkeiten gegenüber der Anzahl der Zustände exponentiell ansteigt, ist eine eindämmende Regel erforderlich. Nur so ist es möglich, für ein ratendes Verfahren dessen Qualität objektiv beurteilen zu können.

Neben der Einführung in die Erfordernisse eines Automaten werden im Folgenden Kriterien für die Auswahl bestimmt und eine praktische Umsetzung analysiert. Um effizient ein Ergebnis zu erhalten, erfolgt für ein Verfahren die parallele Umsetzung der einzelnen notwendigen Rechenschritte, resultierend in einer Hardwarebeschreibung für einen FPGA.

3.1 Problemlösungssuche

Ein mit einzelnen Schritten lösbares Problem ist eine Anwendung für Zustandsautomaten. Eingebunden in eine Umgebung benötigt ein solcher Automat neben den Eingängen, die das Ergebnis von Sensoren sein können, noch eine Reaktion auf Ausgaben, d. h. Aktoren, wie in Abbildung 3.1 dargestellt.

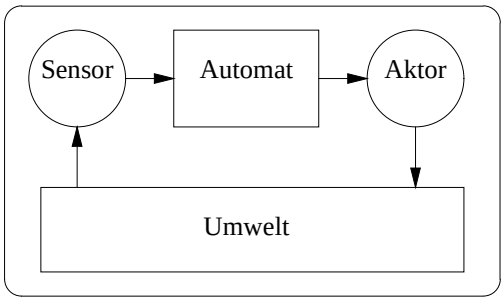


Abbildung 3.1: Ein in eine Umwelt eingebetteter Automat

Der Automat selbst kann durch einen Zustandsautomaten realisiert werden. Hierfür gibt es zwei allgemein übliche Varianten, vergleiche auch [Hof93, Seite 197ff.]:

- ein Moore-Automat mit der Übergangsfunktion $s' \leftarrow f(s,x)$ für den Zustand und $y \leftarrow g(s)$ für die Ausgabe [Moo56];

- ein Mealy-Automat mit der Übergangsfunktion $s' \leftarrow f(s,x)$ für den Zustand und $y \leftarrow g(s,x)$ für die Ausgabe [Mea55].

Der einzige Unterschied ist der zusätzliche Parameter x in der Ausgabefunktion g, wie sich auch aus der Abbildung 3.2 erkennen lässt. Der Zustandsspeicher s ist in beiden Fällen als Register (Flip-Flop) realisiert, der Wert s' gibt den Folgewert an. Gesteuert wird dies von einem externen Takt, der jedoch nicht eingezeichnet ist – genauso wie die Bitbreiten. Im folgenden wird die allgemeinere Form des Mealy-Automaten verwendet; um dann einen Moore-Automaten zu realisieren, muss lediglich die Funktion g den Funktionseingang x unberücksichtigt lassen.

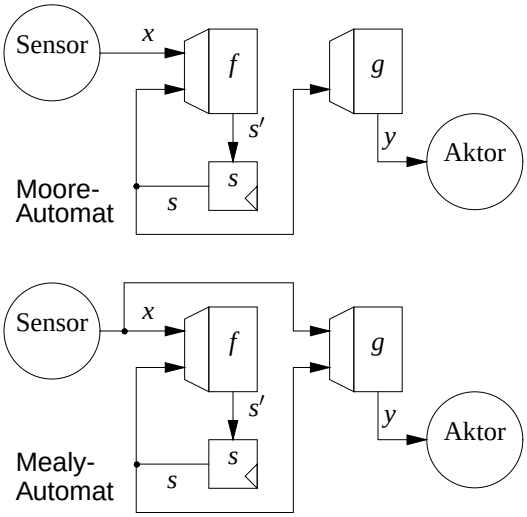


Abbildung 3.2: Hardware-Realisierungsmöglichkeiten für einen Moore- und einen Mealy-Automaten

Ziel ist es nun, mit Hilfe des Automaten die Umwelt so zu verändern, dass man das gewünschte Ergebnis erhält. Dafür müssen die richtigen Aktionen im Aktor ausgeführt werden. Diese hängen vom Sensor und einem Ablaufplan (Folgezustandsspeicher) ab.

Die Frage ist nun, wie dieser Ablaufplan gestaltet sein soll. Eine Variante ist, ähnlich wie bei einem Puzzle, die Aktionen unterschiedlich anzuordnen (auszuprobieren) und das Zwischenergebnis zu verifizieren, um zu einem gewünschten Resultat zu kommen. Das Resultat kann auch von einem externem Schiedsrichter festgestellt werden. Dies entlastet den Zustandsautomaten, so dass dieser einfacher gestaltet werden kann.

3.1.1 Zielwertsuche

Ein Beispiel soll das mögliche Vorgehen verdeutlichen: Es soll ein Zahlenwert z ermittelt werden. Es kann das Ergebnis einer Quadratwurzelberechnung für den Ursprungswert z^2 sein oder auch nur ein Wert, z. B. zum Durchführen einer Temperaturmessung, wie in Abbildung 3.3 dargestellt. Dabei erfasst der NTC-Widerstand R1 den gesuchten Temperaturwert z. Zur Durchführung eines Vergleichs wird ein Operationsverstärker verwendet, an dem sowohl die Vergleichsspannung Vref, z. B. als Ergebnis einer Digital-Analog-Wandlung, als auch R1 über einen Spannungsteiler angeschlossen ist. Teile der Schaltung sind aus [EM92, Seite 312] entnommen.

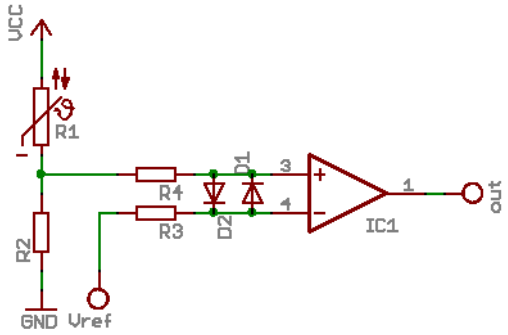


Abbildung 3.3: Schaltplan für eine Temperaturmessung

Ein Zustandsautomat muss von der Bedeutung des Wertes z nichts wissen. Es reicht, neben der Endbedingung zum Abbruch der Suche nur die Aussagen *ist kleiner* und *ist größer* zu haben. Dies entspricht einem 1-Bit-Eingang für den Automaten.

In der gebräuchlichen Technik zur Analog-Digital-Wandlung wird ein kostengünstigerer Digital-Analog-Wandler und ein daran angeschlossener Automat zur Approximation eingesetzt. Ein Vergleicher (Operationsverstärker) liefert nur *größer* bzw. *nicht-größer* als Entscheidungshilfe, so dass das Ende der Wandlung am einfachsten durch den Automaten selbst angezeigt wird, der durch die Position des zuletzt veränderten Bits die Ende-Information ohne weiteres Zutun generieren kann. Da diese Art der Implementierung als Spezialfall angesehen werden kann, ist der Schiedsrichter einfacher implementierbar und daher insgesamt kein Widerspruch zur obigen Annahme.

3.1.2 Einfache Umsetzung

Als Ausgabe kann die Erhöhung bzw. Verringerung eines Vergleichswertes w definiert werden. Um schneller zu einem Ergebnis zu kommen, sollen als mögliche Wertänderung die Zweierpotenzen von 1 bis $2^{\lceil \log_2 z_{\max} \rceil}$ zur Verfügung stehen, wobei $z_{\max}$ der maximal mögliche gesuchte Wert z ist ($0 \leq z \leq z_{\max}$). Um einheitlich die Qualität des Automaten bestimmen zu können, also wie viele Schritte benötigt werden, um das Ergebnis $w = z$ zu erreichen, wird als Startwert $w = 0$ festgelegt. Die Zeitindizes sind der Übersichtlichkeit halber weggelassen.

Eine mögliche Umsetzung ist mit Algorithmus 3.1 dargestellt, wobei die Funktion „agent“ den Zustandsspeicher beinhaltet und nur als Eingang x den Wert des Sensors erhält und als Ausgang y die Werte v und d für Vorzeichen und Änderungswert (Delta) für den Aktor des Automaten liefert.

Um eine ideale Funktion zu erhalten, also eine Belegung der Zustandstabelle des Automaten, müssen alle Konstellationen getestet werden. Eine solche Aufzählung könnte systematisch in Abhängigkeit von Zustand und Eingangswert erfolgen.

3.1.3 Aufzählung der Automatenfunktionen

Für das Beispiel der Temperaturmessung mit einer Genauigkeit von vier Bits bedeutet dies, dass v die Werte $+1$ und -1 sowie d die Werte von 1 bis 3 annehmen kann, für y also 8 verschiedene Werte möglich sind. Der Eingang x besteht nur aus *ist größer* bzw. *ist kleiner*, so dass es nur zwei Möglichkeiten gibt. Für die Berechnung reichen vier Zustände, die als Menge $\mathbb{S} = \{0,1,2,3\}$ mit $s \in \mathbb{S}$ dargestellt werden kön-

Algorithmus 3.1: Zielwertsuche

```
1 schritte := 0
2 w := 0
3 while z ≠ w do
4     d, v := agent(sensor)
5     if v then
6         w := max{w − 2^d, 0}
7     else
8         w := min{w + 2^d, z_max}
9     schritte := schritte + 1
```

nen. Analog kann eine Menge für die Eingangswerte ($\mathbb{X} = \{<, >\}$) und Ausgabewerte ($\mathbb{Y} = \{+0, +1, +2, +3, -0, -1, -2, -3\}$) definiert werden. Der erste Zustand soll der Einfachheit halber immer den Wert 0 haben, danach wird in Einer-Schritten inkrementiert, so dass $\mathbb{S} = \{0, 1, ..., |\mathbb{S}| - 1\}$ gilt.

Die Anzahl der Elemente im Speicher ist dann $|\mathbb{S}| \cdot |\mathbb{X}|$ mit $|\mathbb{S}| \cdot |\mathbb{Y}|$ unterschiedlichen Werten, die mit jeweils $\lceil \log_2 |\mathbb{S}| \rceil + \lceil \log_2 |\mathbb{Y}| \rceil$ Bits gespeichert werden können. Daraus ergibt sich die Anzahl der Möglichkeiten von

$$|\mathbb{S}|^{|\mathbb{S}| \cdot |\mathbb{X}|} \cdot |\mathbb{Y}|^{|\mathbb{S}| \cdot |\mathbb{X}|} = (|\mathbb{S}| \cdot |\mathbb{Y}|)^{|\mathbb{S}| \cdot |\mathbb{X}|} \tag{3.1}$$

im allgemeinen Fall für eine vollständige Aufzählung. Für das Beispiel ergeben sich $(4 \cdot 8)^{4 \cdot 2} = 32^8 = 1\,099\,511\,627\,776$ Wertkonstellationen zum Testen. Um diese aufzuführen, benötigt man einen variablen Automaten, bei dem Folgezustand und Ausgabe für jeden Eingang verändert werden können. Hierfür bieten sich zwei unterschiedliche Matrizen (Arrays) für s' und y an, mit Indizes für aktuellen Zustand s und Eingang x. Diese werden im folgenden mit $s'_{s,x}$ und $y_{s,x}$ bezeichnet, wobei die Indizes bei 0 beginnen. Zusammengefasst wird hierfür der Begriff „Automat“ verwendet, wobei auch die Bezeichnung „Algorithmus“ möglich wäre, da eine Ablaufsteuerung dargestellt wird.

Die initiale Belegung soll mit 0 als Kennzeichen für das erste Element der jeweiligen Menge für alle Werte der Matrizen beginnen, wie im Algorithmus 3.2 in der Funktion „InitiateAutomaton“ dargestellt. Die Erhöhung um einen Wert erfolgt analog zur Addition mit Überlauf, siehe auch [PH98, Kapitel 4.3].

Algorithmus 3.2: Initialisierung für eine Aufzählung von Automaten

```
1 function InitiateAutomaton
2     foreach i ∈ S do
3         foreach j ∈ X do
4             s'_{i,j} := 0
5             y_{i,j} := 0
```

Für jeden Schritt wird dazu die niedrigste Stelle so lange erhöht, bis der Maximalwert erreicht wird. Danach kommt es zu einem Überlauf, d. h. die niedrigste Stelle wird auf 0 zurückgesetzt und im gleichen Schritt wird die nächste Stelle auf gleiche Weise erhöht. Dies setzt sich zyklisch fort. Dabei wird deutlich, dass eine lineare Reihenfolge der einzelnen Stellen notwendig ist. Kommt es an der höchsten Stelle zu einem Überlauf, so wurden alle Möglichkeiten aufgeführt und die Aufzählung würde von vorne beginnen. Damit dies nicht endlos geschieht, ist dieser letzte Überlauf mit einem Endkriterium

in Zeile 16 des Algorithmus 3.3 versehen. Die Funktion „NextAutomaton“ ist für jeden einzelnen Schritt aufzurufen.

Algorithmus 3.3: Aufzählung von Automaten

```
1  function NextAutomaton
2      for i := |𝕊| − 1 to 0 do
3          for j := |𝕏| − 1 to 0 do
4              if y_{i,j} < |𝕐| − 1 then
5                  y_{i,j} := y_{i,j} + 1
6                  return true
7              else
8                  y_{i,j} := 0
9      for i := |𝕊| − 1 to 0 do
10         for j := |𝕏| − 1 to 0 do
11             if s'_{i,j} < |𝕊| − 1 then
12                 s'_{i,j} := s'_{i,j} + 1
13                 return true
14             else
15                 s'_{i,j} := 0
16     return false
```

Die Reihenfolge der Werte wurde im Algorithmus 3.3 so gewählt, dass zuerst alle Ausgaben aufgeführt und danach erst die einzelnen Zustände variiert werden. Dies hat den Vorteil, dass bei ungünstiger Zustandskonstellation diese einfacher übersprungen und ein anderer Ablauf der Zustände gewählt werden kann, wie es sich in [Bot06] gezeigt hat.

Um eine vollständige Liste zu erstellen, wird viel Zeit benötigt, wie sich schon an der hohen Zahl der Konstellationen erkennen lässt. Selbst wenn je möglicher Wert nur eine Mikrosekunde benötigt werden würde, dauert der gesamte Test über 116 Tage. Von daher liegt es nahe, zu untersuchen, ob eine Eingrenzung der zu betrachtenden Fälle möglich ist, da nicht alle Konstellationen sinnvoll sind. Die Analyse erfolgt im folgenden Abschnitt 3.2, mögliche Verfahren werden in 3.3 und 3.4 dargestellt. Als Vorausblick: Die Zielwertsuche mit vier Zuständen und acht Ausgaben war nach 23 Stunden durchgeführt, ein einzelner Schritt hat dabei etwa $1,64\,\mu$s benötigt. Die Details sind in Tabelle 3.2 auf Seite 45 aufgeführt.

3.1.4 Ordinalzahl

Um zwei Automaten $\dot{s}',\dot{y}$ und $\ddot{s}',\ddot{y}$ miteinander in der Reihenfolge ihrer Aufzählung zu vergleichen, gibt es neben dem Vergleich der Einzelwerte, z. B. mit Algorithmus 3.4, die Variante, von jedem Automaten die Ordinalzahl zu bestimmen und nur diese zu vergleichen. Dadurch verringert sich die Anzahl der notwendigen Vergleiche auf eins, allerdings benötigt die Bestimmung der Ordinalzahl Rechenaufwand, so dass sich insgesamt betrachtet nicht unbedingt ein Vorteil ergibt, abhängig von den vorhandenen Ressourcen. Ermittelt durch Algorithmus 3.5 bestimmt die Ordinalzahl die Position in der Aufzählungsreihenfolge durch Algorithmus 3.3. Damit ist es dann möglich, die Eigenschaften eines Automaten, z. B. Zustandsbezeichner, umzusortieren und das Ergebnis mit dem

Ursprung anhand des Kriteriums Aufzählungsreihenfolge zu vergleichen, um die zuerst aufgezählte Variante des Automaten zu erhalten.

Algorithmus 3.4: Vergleich zweier Automaten in deren Aufzählungsreihenfolge zur Bestimmung des mutmaßlichen Repräsentanten von zwei zueinander äquivalent verhaltenden Automaten

```
1  function PreviouslyEnumerated
2      for i := 0 to |𝕊| − 1 do
3          for j := 0 to |𝕏| − 1 do
4              if ṡ'_{i,j} ≠ s̈'_{i,j} then
5                  return ṡ'_{i,j} > s̈'_{i,j}
6      for i := 0 to |𝕊| − 1 do
7          for j := 0 to |𝕏| − 1 do
8              if ẏ_{i,j} ≠ ÿ_{i,j} then
9                  return ẏ_{i,j} > ÿ_{i,j}
10     return false
```

Algorithmus 3.5: Bestimmung der Ordinalzahl

```
1  function OrdinalNumber
2      o := 1
3      foreach i ∈ 𝕊 do
4          foreach j ∈ 𝕏 do
5              d := (|𝕊| − 1 − i) · |𝕏| + (|𝕏| − 1 − j)
6              o := o + s'_{i,j} · |𝕊|^d · |𝕐|^(|𝕊|·|𝕏|) + y_{i,j} · |𝕐|^d
7      return o
```

3.1.5 Terminalzustände

Neben den bisher betrachteten technischen Anwendungen gibt es in der Automatentheorie noch die Möglichkeit, Zustände als Terminalzustände auszuweisen. In der Erkennung von Sprachen in der Theoretischen Informatik bedeutet dies, dass der Automat die Eingabe als ein gültiges Wort erkannt hat, wenn der letzte, aktuell erreichte Zustand ein Terminalzustand ist.

In [HP67] verwendet Harary zur Bestimmung der Anzahl unterschiedlicher Automaten bezüglich der Isomorphie von Zuständen, Eingabe- und Ausgabewerten den Parameter t, um den Umfang der Terminalzustände vorzugeben. Um einen einfachen Vergleich der Ergebnisse erzielen zu können, sei hier das gleiche Prinzip verwendet.

Des weiteren gibt es in [HP73] auch die Möglichkeit, festzulegen, ob ein Startzustand gleichzeitig ein Terminalzustand sein kann oder nicht. Hierfür werden zwei Teilmengen von $\mathbb{S}$ eingeführt. Zum Einen gibt es die Menge $\mathbb{E}$ der Startzustände und zum anderen die Menge $\mathbb{F}$ der Terminalzustände. Ist ein Zustand sowohl Start- als auch Terminalzustand, so ist dieser in beiden Mengen enthalten. Insgesamt ergeben sich vier mögliche Bedingungen, die sich zur einfacheren Handhabung in zusammenhängende Bereiche einteilen lassen:

1. Startzustände, die nicht Terminalzustände sind ($\mathbb{E}\backslash\mathbb{F}$),
2. Startzustände, die gleichzeitig auch Terminalzustände sind ($\mathbb{E}\cap\mathbb{F}$),
3. Zustände, die weder Startzustände noch Terminalzustände sind ($\mathbb{S}\backslash(\mathbb{E}\cup\mathbb{F})$),
4. mögliche Terminalzustände, die nicht Startzustände sind ($\mathbb{F}\backslash\mathbb{E}$).

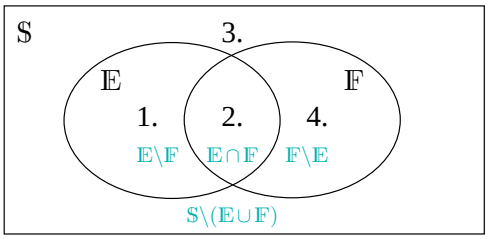


Abbildung 3.4: Mengendiagramm der einzelnen Zustandszuordnungen

Die einzelnen Bereiche, zu sehen in Abbildung 3.4, können auch leer sein, also keine Zustände enthalten. Es muss aber immer einen Startzustand geben ($|\mathbb{E}| \geq 1$), die aber auch alle terminal sein können. Für Terminalzustände gibt es keine solche Einschränkung, es können alle, keine oder nur ein Teil der Zustände in $\mathbb{F}$ enthalten sein ($0 \leq |\mathbb{F}| \leq |\mathbb{S}|$, $\mathbb{F} \subseteq \mathbb{S}$). Zur einfacheren Handhabung sind die einzelnen Blöcke in obiger Reihenfolge auch bei der Umsetzung und Überprüfung beibehalten. Somit ist Zustand 0 immer ein Startzustand.

Bezüglich der Aufzählung ändert sich nichts, lediglich der Automat kann initial unterschiedliche Werte aus $\mathbb{E}$ erhalten. Traditionell hat ein Automat nur einen Startzustand ($|\mathbb{E}| = 1$), für einige Problemstellungen ist es auch interessant, alle Zustände als potentielle Startzustände zu haben ($\mathbb{E} = \mathbb{S}$).

3.1.6 Darstellung von Automaten

Das Verfahren des Algorithmus 3.3 zur Aufzählung aller Automaten funktioniert in gleicher Weise wie das Erhöhen einzelner Ziffern einer Zahl. Dies ist in Abbildung 3.5 dargestellt. Im Unterschied dazu haben die einzelnen „Ziffern“ allerdings unterschiedliche Wertebereiche: $s'_{i,j} \in \mathbb{S}$ sowie $y_{i,j} \in \mathbb{Y}$, die Indizes umfassen dabei die Werte $i \in \mathbb{S}$ und $j \in \mathbb{X}$, die zugehörige Übergangstabelle zeigt Tabelle 3.1. Für eine vergleichende Visualisierung der Charakteristik unterschiedlicher Automaten ist diese Darstellung allerdings wenig geeignet.

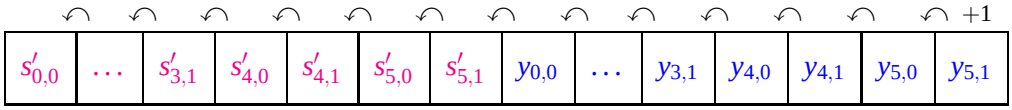


Abbildung 3.5: Anordnung der Zählerstellen eines Automaten nach deren Priorität

Um diesen Vergleich von Automaten durchführen zu können, ist eine andere Darstellung notwendig. Hierfür gibt es verschiedene Möglichkeiten, die in Abbildung 3.6 an Hand eines Beispiels aufgeführt sind. Hierbei gibt es zwei Ausgabemöglichkeiten, $\mathbb{Y} = \{0,1\} = \{\mathrm{R},\mathrm{L}\}$, und zwei Eingänge, $|\mathbb{X}| = 2$.

Die einfachste ist eine rein textbasierte Angabe, in der je Eingangsbedingung und aktuellem Zustand der Folgezustand und die Ausgabe notiert sind (Abbildung 3.6a). Wenn

a) 2/0 1/1 4/0 3/1 5/0 0/0 – 1/0 0/1 3/0 2/1 5/1 4/1

b) 2R1L4R3L5R0R–1R0L3R2L5L4L

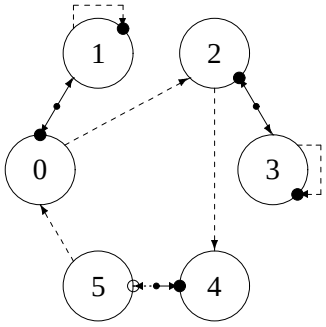
c) `4387A0 2165B9`

d)
$$\begin{pmatrix} (2,\mathrm{R}) & (1,\mathrm{L}) & (4,\mathrm{R}) & (3,\mathrm{L}) & (5,\mathrm{R}) & (0,\mathrm{R}) \\ (1,\mathrm{R}) & (0,\mathrm{L}) & (3,\mathrm{R}) & (2,\mathrm{L}) & (5,\mathrm{L}) & (4,\mathrm{L}) \end{pmatrix}$$

e)
$$\begin{pmatrix} 0 & 2 & 1 & 0 & 0 & 0 \\ 6 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 6 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 \\ 1 & 0 & 0 & 0 & 6 & 0 \end{pmatrix}$$

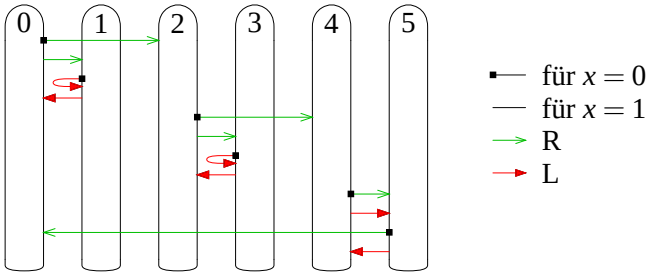
Adjazenzwert	0	1	2	3	4	5	6	7	8
für $x=0$	—	R	—	R	L	L	—	R	L
für $x=1$	—	—	R	R	—	R	L	L	L

f)



Pfeil	für $x=0$	für $x=1$
⋯○	R	L
⋯□	L	R
⋯●	L	L
⋯→	R	R
- -●	L	—
- -→	R	—
——●	—	L
——→	—	R

g)



h)

s	0	0	1	1	2	2	3	3	4	4	5	5
x	0	1	0	1	0	1	0	1	0	1	0	1
$s'_{s,x}$	2	1	1	0	4	3	3	2	5	5	0	4
$y_{s,x}$	0	0	1	1	0	0	1	1	0	1	0	1

i)

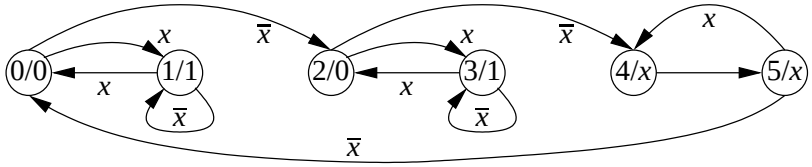


Abbildung 3.6: Verschiedene Darstellungen eines Automaten: a) textbasiert, b) Kompaktcodierung, c) Hexadezimalwert, d) Matrix, e) Adjazenzmatrix, f) Graph mit komplexen Zustandsübergängen, g) Graph mit langgezogener Zustandsdarstellung, h) Übergangstabelle, i) gerichteter Graph mit Ausgabewertangabe im Zustand

Tabelle 3.1: Übergangstabelle der Zählerstellen aus Abbildung 3.5

$s \in \mathbb{S}$	0	…	3	4	4	5	5
$x \in \mathbb{X}$	0	…	1	0	1	0	1
$s' \in \mathbb{S}$	$s'_{0,0}$	$\cdots$	$s'_{3,1}$	$s'_{4,0}$	$s'_{4,1}$	$s'_{5,0}$	$s'_{5,1}$
$y \in \mathbb{Y}$	$y_{0,0}$	$\cdots$	$y_{3,1}$	$y_{4,0}$	$y_{4,1}$	$y_{5,0}$	$y_{5,1}$

für die Ausgabe statt einer dezimalen Angabe Buchstaben verwendet werden, lässt sich das Ganze kompakter darstellen (Abbildung 3.6b). In einzelne Bits umgesetzt, ergibt sich eine hexadezimale Repräsentation, die sich jedoch bei unterschiedlichen Ausgabebitbreiten als unpraktisch erwiesen hat und daher im Folgenden nicht weiter verwendet wird (Abbildung 3.6c).

Eine Visualisierung als Matrix ist ebenfalls möglich, dazu werden in den Spalten der Zustand und in den Zeilen die Eingabemöglichkeiten aufgetragen (Abbildung 3.6d). Das gleiche lässt sich auch mit einer Adjazenzmatrix repräsentieren, wobei die einzelnen Werte in der Matrix von Eingang und Ausgang abhängen (Abbildung 3.6e), die Bedeutungen der Werte sind unterhalb der Matrix aufgeführt. Dabei sind die Werte derart sortiert, dass eine Drehrichtung alleine aufgeführt ist und sich erst dann die neuen Kombinationen der nächsten Drehrichtung anschließen. So ist gewährleistet, dass weitere Drehrichtungen hinzugefügt werden können, ohne die vorherigen Werte ändern zu müssen.

Die Werte der Adjazenzmatrix besagen eigentlich, wie viele Kanten zwischen zwei Knoten vorhanden sind. Da in der einfachen Variante $|\mathbb{X}| = 2$ bereits bis zu acht Kanten möglich sind, ist eine übersichtliche Darstellung als Graph nicht mehr gewährleistet. Daher können die Kanten in der Graphendarstellung durch unterschiedliche Pfeile repräsentiert werden, in Abhängigkeit des Adjazenzwertes. Eine Variante ist in Abbildung 3.6f zu sehen, bei der die Pfeile in der Mitte geteilt sein können – in Abhängigkeit von Eingabe und Ausgabe für beide möglichen Zustandsübergangsrichtungen.

Da jedoch die Semantik der Pfeile des Zustandsdiagramms in der Abbildung 3.6f nicht offensichtlich gestaltbar ist, gibt es eine andere, kompakte Repräsentation, die ein einfaches, automatisches Generieren ermöglicht (Abbildung 3.6g). Die Zustände selbst werden in Ovalen statt Kreisen dargestellt und ein Pfeil je Eingabemöglichkeit gezeichnet. Die Gestaltung der Pfeilspitze ist allein abhängig von der Ausgabe, das andere Ende des Pfeiles markiert die Eingabe. Die Pfeile selbst sind in einer einheitlichen Farbe je Ausgabewert gestaltet, um sich schnell einen Überblick verschaffen zu können. Zusätzlich weist jede Pfeilspitzenart unterschiedliche Formen auf, um sich im Schwarz-Weiß-Druck einfacher voneinander zu differenzieren.

Des weiteren sind die Pfeile in der Reihenfolge der Aufzählung durch Algorithmus 3.3 angeordnet, beginnend mit den Ausgabewerten, also Form und Farbe, des untersten Pfeils $y_{5,1}$ bis hin zum obersten Pfeil mit $y_{0,0}$. Danach folgen die Zustandsübergänge und damit die Ziele der Pfeile von $s'_{5,1}$ bis $s'_{0,0}$ in gleicher Weise. In der Übergangstabelle in Abbildung 3.6h ist diese Reihenfolge durch die Werte in der unteren Zeile für $y_{s,x}$, danach in der darüberliegenden Zeile für $s'_{s,x}$ jeweils von rechts nach links gegeben.

Angelehnt an die in der Mathematik gebräuchliche Darstellung von Digraphen zeigt die manuell erstellte Abbildung 3.6i eine weitere, abschließende Variation mit den Ausgabewerten direkt den Zuständen statt den Zustandsübergängen zugeordnet.

Die generierte Darstellung des Graphen aus Abbildung 3.6g wird im Folgenden hauptsächlich verwendet, da das Wesentliche auch ohne Legende einfach zu verstehen ist und verschiedene Eigenschaften des Graphen bildlich leicht zu erfassen sind.

3.2 Klassifizierungskriterien zur Automatenauswahl

Bei der Durchführung des Algorithmus 3.3 zur Aufzählung von Automaten fällt auf, dass es als Ergebnis verschiedene Automaten mit gleichem Verhalten gibt. Ein einfaches Beispiel ist schnell gefunden: Wenn die Zustände mit Ausnahme des Startzustandes permutiert werden, ändert sich lediglich die Beschreibung, nicht jedoch das Ausgabeverhalten.

Ebenso leicht auffindbar ist die Eigenschaft, wenn Zustände vorhanden sind, die nie erreicht werden können. Ein solcher Zustandsautomat kann durch einen mit weniger Zuständen ersetzt werden.

Eine Sonderrolle spielen Automaten, die quasi einen Anfangsablauf haben und danach einen Zyklus erreichen, der nicht mehr zum Anfangszustand 0 zurückkehrt. Ein Automat kann auch mehrere dieser „Endzyklen“ aufweisen. Die Zustände eines Anfangsablaufs werden im Folgenden als Präfix bezeichnet. Bevor dieser Endzyklus erreicht wird, lässt sich die Kombination aus Startposition und Abfolge des Präfix auch durch Voranschreiten in einer Eingabeabfolge erreichen, also z. B. eine Änderung der Startposition für ein Fahrzeug, gesteuert durch einen solchen Automaten.

Um einen prinzipiell agil funktionierenden Automaten zu erhalten, sind solche mit Präfix uninteressant, da einige der Zustände für eben diesen Präfix benötigt werden und somit weniger Zustände für den eigentlichen Hauptzyklus übrig bleiben. Des weiteren sollen die gefundenen Automaten zu Beginn beliebig in der Umwelt, analog zur Abbildung 3.1 auf Seite 23, positioniert werden können, was ebenso gegen präfixbehaftete Automaten spricht.

Diese drei Bedingungen beziehen sich lediglich auf die Abfolge der Zustände, ohne dass eine Ausgabe mit betrachtet wird. Nimmt man dies jedoch noch hinzu, ergibt sich eine weitere Einschränkung: Verschiedene Zustände können zusammengefasst werden, ohne dass es einen Einfluss auf ein Endresultat hat. Am offensichtlichen ist es bei einem Automaten, der lediglich einen Wert für die Ausgabe hat. Dieser Automat könnte ohne jegliche Zustände auskommen und nur diese eine Ausgabe liefern, die unabhängig von Eingang und Zustand ist.

Zur Ermittlung dieser Eigenschaften gibt es verschiedene Varianten für Software- und Hardwarelösungen, die aufgrund der Parallelisierbarkeit vollständig unterschiedlich ausfallen können. Zudem ist es nicht immer einfach zu erkennen, welche Zustände unnötig sind oder ob eine Permutation vorliegt. Dies wird in den folgenden Abschnitten behandelt.

3.2.1 Normierung der Abfolge von Zustandsübergängen

Ziel ist es, von gleich verhaltenden Automaten, die sich nur durch eine Permutation der Zustände voneinander unterscheiden, einen Repräsentanten auszuwählen. Hierfür kann ein Verfahren zur Sortierung der Zustände erstellt werden. Wenn dann das Ergebnis der Sortierung mit dem ursprünglichen Automaten übereinstimmt, so war dieser repräsentativ. Sortieren bedeutet hierbei, die in einer Zustandsübergangsfunktionen verwendeten Zustandsbezeichnungen eines Automaten in eine zu definierende Reihenfolge zu bringen.

Die aus der Graphentheorie bekannte topologische Sortierung für zyklenfreie Graphen kann nicht angewendet werden, da es bei den Graphen bzw. Automaten gerade auf einen Zyklus ankommt. Somit kann die Bedingung $(v,w) \in E \Rightarrow ord(v) < ord(w)$ eines Graphen $G = (V,E)$ nicht eingehalten werden.

Eine Möglichkeit für eine Sortierregel ist, eine Priorisierung der Eingänge und Zustände vorzunehmen und die Definition 3.1 anzuwenden, ausgehend vom Startzustand 0. Ein Eingangswert ist höher priorisiert, wenn der verwendete Wert kleiner ist. Gleiches gilt für die Zustandswerte.

Definition 3.1 (Sortierregel) *Die Zustände erhalten ihre Nummer $\in \mathbb{S}$ in aufsteigender Reihenfolge beginnend beim Startzustand $s = 0$ und dem Eingangswert $x = 0$, solange dies für die Zustandsübergänge des höchstpriorisierten Eingangs möglich ist. Findet sich kein weiterer Zustandsübergang zu einer noch nicht vergebenen Zustandsnummer, so erfolgt für niedriger priorisierte Eingänge die Suche nach Zustandsübergängen, ausgehend bei der niedrigsten, bereits vergebenen Zustandsnummer.*

Daraus folgt, dass der Zustand 0 für seinen höchstpriorisierten Eingang nur sich selbst oder den Zustand 1 erreichen kann; Zustand 2 und höher sind nicht direkt erreichbar. Der in Abbildung 3.6 auf Seite 30 dargestellte Automat ist demnach nicht normiert.

Um die Zustände zu sortieren, ist ein Änderungsvektor hilfreich, der zwischen alter und neuer Zustandsnummer verweist. Von alter zu neuer Nummer soll der Bezeichner *o* verwendet werden, in umgekehrter Richtung *n*. Als Index wird die jeweilige Zustandsnummer verwendet. Als weiteres ist ein Zähler *c* notwendig, der je Eingangsebene die

Algorithmus 3.6: Abbildung eines Automaten auf einen Repräsentanten bezüglich Zustandspermutation durch Sortieren von Zuständen, ausgehend vom einzigen Startzustand 0

```
1  function MapToNormalized
2      ∀i ∈ 𝕊: (o[i] = ε ∧ n[i] = ε) mit ε ∉ 𝕊
3      o[0], n[0] := 0, 0
4      ∀j ∈ 𝕏: c[j] = 0
5      for i := 1 to |𝕊| − 1 do
6          f := false
7          for j := 0 to |𝕏| − 1 do
8              while c[j] < i do
9                  if o[s'_{c[j],j}] ∉ 𝕊 then
10                     o[s'_{c[j],j}] := i
11                     n[i] := s'_{c[j],j}
12                     f := true
13                     c[j] := c[j] + 1
14                     endloop j
15                 else
16                     c[j] := c[j] + 1
17         if ¬f then
18             endloop   // Ende der Vorsortierung, nicht alle Zustände erreicht
19     foreach i ∈ 𝕊 do // unerreichte Zustände auf feste Werte setzen
20         foreach j ∈ 𝕏 do
21             if n[i] ∉ 𝕊 then
22                 s'_{i,j}, y_{i,j} := 0, 0
23             else
24                 s'_{i,j}, y_{i,j} := o[s'_{n[i],j}], y_{n[i],j}
```

zuletzt verwendete Zustandsnummer speichert. Die Sortierung, die der Algorithmus 3.6 vornimmt, erfolgt von s', y nach $\mathbf{s}', \mathbf{y}$ für die jeweiligen Indizes.

Daraus ergibt sich, dass zwei Automaten $\dot{s}', \dot{y}$ und $\ddot{s}', \ddot{y}$ zueinander isomorph sind, wenn deren Sortierung $\dot{\mathbf{s}}', \dot{\mathbf{y}}$ und $\ddot{\mathbf{s}}', \ddot{\mathbf{y}}$ für alle Eingabemöglichkeiten und Zustandsübergänge gleich sind, also den gleichen Repräsentanten haben. Zwei zueinander isomorphe Automaten sind zueinander äquivalent, da lediglich die Bezeichnung der Zustände, aber nicht das daraus resultierende Verhalten unterschiedlich ist. Die Isomorphie gerichteter Graphen ist z. B. in [BS68, Seite 45] definiert. Die Äquivalenz sequentieller Maschinen führt z. B. [Gin62, Definition 1.4] auf.

Ein weiterer Aspekt ist, ob ein Graph zusammenhängend ist. Leonhard Euler hat im 18. Jahrhundert gemäß [BLW76, Seite 10] einen Graphen als zusammenhängenden bezeichnet, wenn es stets zwischen zwei beliebigen Punkten eines Graphen einen Pfad mit diesen als Anfangs- und Endpunkt gibt. Für gerichtete Graphen, auf denen die Automaten basieren, geht aus [Kön36, Satz VII 1][1] hervor, dass die Zustände eines separierten Teils eines Graphen nicht erreicht werden können; der Graph ist dann nicht mehr zusammenhängend.

Von einem Startzustand aus betrachtet sind dann die separierten Zustände nicht erreichbar. Da eine Änderung bei diesen Zustandsübergängen keinen Einfluss auf das Verhalten des Automaten hat, sollte ein einheitlicher, fixer Wert festgelegt werden, mit dem dann ein Automat als normiert gilt. Hierfür bieten sich die ersten Werte aus den Mengen $\mathbb{S}$ und $\mathbb{Y}$ an, da diese garantiert immer vorhanden sind. In der folgenden Definition 3.2 sind die Kriterien zusammengetragen.

Definition 3.2 (Normiert) *Ein Automat wird als **normiert** bezeichnet, wenn die Definition 3.1 für jeden Zustandsübergang eingehalten ist, also die Zustände **sortiert** vorliegen, **und ungenutzte Zustände** auf Zustand* 0 *verweisen und dabei den Ausgabewert* 0 *haben.*

Da mit diesen Definitionen eine einheitliche Regel unabhängig von den ursprünglichen Zustandsnummern festgelegt ist, sind Permutationen der Automatenzustände ausgeschlossen. Im Umkehrschluss gilt somit auch, dass es für einen nicht-normierten Automaten immer einen anderen, normierten Automaten gleichen Verhaltens gibt, der dann den Kriterien entspricht.

Die dabei entstehende Auswahl von Automaten entspricht allerdings nicht der Ordnung, die durch die Verfahren aus Abschnitt 3.1.4 beschrieben ist. Abbildung 3.7 zeigt hierzu ein Beispiel, bei dem sich die Ordinalzahl des Repräsentanten gegenüber den ursprünglichen Automaten erhöht. Die zusätzlichen Zustände 4 und 5 dienen lediglich dazu, den über die Sortierung hinausgehende Bedeutung des Begriffs „normiert“ darzustellen.

3.2.2 Vereinfachung – Alle Zustände erreichbar

Wie bereits im vorigen Abschnitt 3.2.1 erwähnt und mit Algorithmus 3.6 durchgeführt, ist mit der Sortierung der Zustände bereits eine Auswertung vorhanden, ob alle Zustände erreicht und dadurch genutzt werden. Um dies jedoch unabhängig davon festzustellen, muss für jeden Zustand überprüft werden, ob dieser von anderen Zuständen erreicht werden kann; dies gilt jedoch nicht für den Zustand 0, da dieser ein Startzustand ist und damit von sich aus bereits erreichbar ist.

[1] *Führt eine Bahn von A nach B und eine Bahn von B nach $C \neq A$, so führt auch eine Bahn von A nach C.*

0. Automat mit der oktalen Ordinalzahl 1231021355447022_8. **1.** Überprüfte Zustandsübergänge bis $i = 3$, stets mit $j = 0$ und somit schließlich $c[0] = 3$, $o = (01\varepsilon 2\varepsilon\varepsilon)$ und $n = (013\varepsilon\varepsilon\varepsilon)$. **2.** Erstmals Inkrementieren des Zählers $c[1]$ bei $i = 3$. **3.** Letzte Stufe des fertig vorsortierten Automaten, dessen Zustand $i = 4$ nicht erreichbar ist. **4.** Normierter Automat 1321120300007101_8.

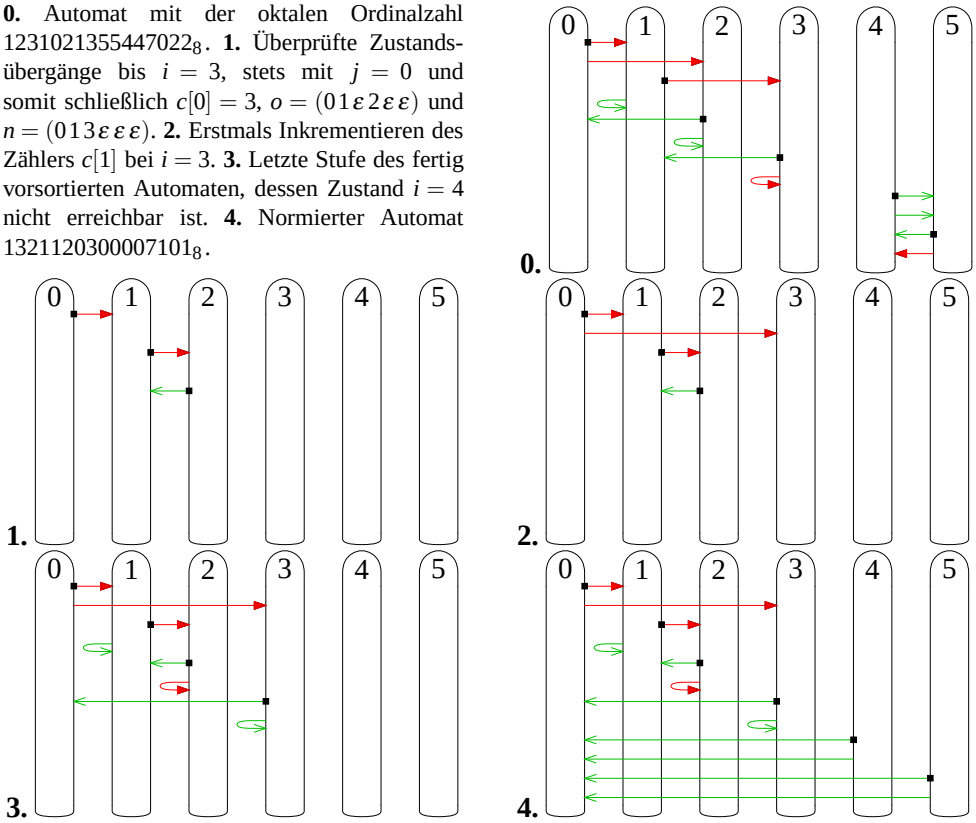


Abbildung 3.7: Schrittweise Darstellung der Normierung durch Algorithmus 3.6

Allerdings erfüllen zwei Zustände, die gegenseitig aufeinander zeigen, nicht die Bedingung, wenn diese selbst nicht erreicht werden können. In einem solchen Fall ist zuerst die reflexive transitive Hülle zu ermitteln, wie es im Algorithmus 3.8 nach dem Verfahren aus [War62] geschieht. Demnach sind alle Zustände von einem Startzustand aus erreichbar, wenn die Bedingung $\forall i \in \mathbb{S} : A[0,i]$ erfüllt ist.

Gibt es mehrere Startzustände, ist eine andere Bedingung erforderlich. Alle Zustände eines Automaten sind erreichbar, wenn $\forall i \in \mathbb{S} \backslash \mathbb{E} : \exists j \in \mathbb{E} : A[j,i]$ gilt. Ob die Startzustände untereinander erreichbar sind und sich somit ein vollständig genutzter Automat ergibt, ist hiermit nicht mehr überprüft. Dafür ist die einfachere Bedingung wie mit einem Startzustand zu verwenden.

Eine alternative Herleitung für „vereinfacht“ ist z. B. in [Ste88, Seite 120f.] gegeben. Ein Automat, der mit und ohne einem unerreichbaren Zustand $z' \in \mathbb{S}$ für alle möglichen Eingabeworte $w \in \mathbb{X}^+$ die jeweils gleichen Ausgabeworte generiert, ist nicht vereinfacht. Der Beweis ist dort zu finden.

3.2.3 Reduzierung der Zustandsanzahl

Bereits mit [Huf54] ist das notwendige Zusammenfassen von Zuständen eines Automaten erkannt. Allerdings ist das Verfahren dort nur unspezifisch und unter Berücksichtigung von Relais-Verzögerungszeiten beschrieben. Ein Algorithmus zum Reduzieren von Zustandsautomaten von Huffman und Mealy ist in [Phi60, Seite 150ff.] erschienen, von Hopcroft wurde in [Hop71] ein alternatives Verfahren entwickelt.

Das Verfahren nach Huffman und Mealy, das auch Hopcroft und Ullmann in [HU79, Seite 68ff.] aufgreifen und in [ASU88] erweitert ist, gruppiert in einem ersten Schritt die

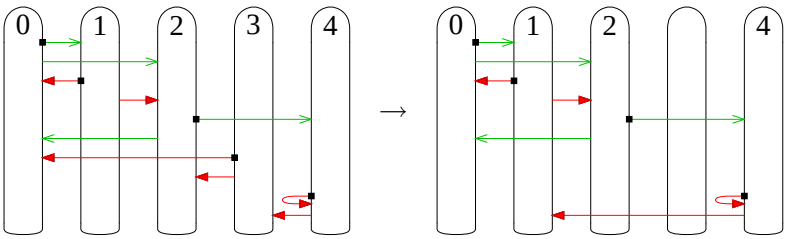
Zustände anhand der Ausgabe. Konkret bedeutet dies bei den Literaturstellen eine Aufteilung in akzeptierte und nicht-akzeptierte Zustände. Eine weitere Unterteilung erfolgt für die so entstandenen Mengen, wenn für mindestens einen der möglichen Eingänge ein Zustandsübergang zu Zuständen in unterschiedlichen Mengen führt. Ist keine weitere Unterteilung mehr möglich, sind alle Zustände innerhalb einer Menge identisch. Dieses Verfahren hat die Komplexität $O(|\mathbb{S}|^2)$.

Bei Hopcroft in [Hop71] erfolgt die Aufteilung ebenfalls auf gleiche Weise, die Suche ist jedoch optimiert: Statt immer alle Kombinationen zu testen, wird eine Liste erstellt, die die kleinere Hälfte der noch nicht untersuchten Mengenindizes enthält. Auf diese Liste beschränkt sich der Vergleich. Dadurch ist eine Komplexität von $O(|\mathbb{S}|\log|\mathbb{S}|)$ erzielt.

Es gibt noch zahlreiche andere Verfahren, unter anderem optimiert für große Datenmengen, die nicht gleichzeitig im flüchtigen Speicher vorgehalten werden können. Eine andere Optimierung ist mit dem Verfahren nach Gilio und Liebig aus [GL73], aufgegriffen in [Hof74, Kapitel 8.1] gegeben. Es arbeitet auf gut parallelisierbaren Matrizenoperationen. Allerdings ist die Zahl der Operationen zu umfangreich, so dass es im Endeffekt zu einem komplexeren Verfahren führt, als bereits nach Huffman und Mealy möglich.

Gemäß [Zha96, Seite 33f.] gab es zwischenzeitlich auf der Basis von [Phi60] und [Hop71] keine wesentliche Weiterentwicklung auf diesem Gebiet. Von daher wird das Verfahren in erweiterter Form analog zu [Wal04] verwendet, das auch in [Web77, Kapitel 6.3] beschrieben ist.

Prinzip
1. gleiche Ausgaben erfassen $y_i = y_j \Leftrightarrow e[i,j]$
2. Zustandsübergänge kontrollieren $e[i,j] \stackrel{?}{\Rightarrow} e[s'_{i,k}, s'_{j,k}]\,\forall k$
3. unnötige Zustände entfernen $(e[i,j] \wedge i < j) \Rightarrow \lightning j$



Durchführung
1. $e[0,2], e[2,0], e[1,3], e[3,1], e[1,4], e[4,1], e[3,4], e[4,3], e[i,i]$
2. $e[0,2], e[2,0], e[1,3], e[3,1], \sout{e[1,4]}, \sout{e[4,1]}, \sout{e[3,4]}, \sout{e[4,3]}, e[i,i]$
   $\sout{e[0,2]}, \sout{e[2,0]}, e[1,3], e[3,1], \sout{e[1,4]}, \sout{e[4,1]}, \sout{e[3,4]}, \sout{e[4,3]}, e[i,i]$
   $\sout{e[0,2]}, \sout{e[2,0]}, e[1,3], e[3,1], \sout{e[1,4]}, \sout{e[4,1]}, \sout{e[3,4]}, \sout{e[4,3]}, e[i,i]$
3. $\Rightarrow \lightning 3$

Abbildung 3.8: Beispielhafte Durchführung der Zustandsreduktion

Die Reduktion ist in drei Phasen eingeteilt, illustriert mit Abbildung 3.8. Die erste dient der Initialisierung, in der Zustände mit gleichen Ausgaben für alle möglichen Eingänge gruppiert werden; dies wird bei dem Algorithmus 3.7 im Vektor e gespeichert.

Anschließend werden in der zweiten Phase die Gruppierungen aufgelöst, wenn ein Folgezustand unterschiedliche Ausgabegruppen erreicht. Die zweite Phase wird so oft ausgeführt, bis sich keine Änderungen mehr ergeben – dies ist maximal $|\mathbb{S}|$-mal möglich.

Danach folgt in der dritten Phase die Auswertung. Wenn für unterschiedliche Zustände noch eine Gruppierung in e eingetragen ist, so lassen sich diese zu einem Zustand zusammenfassen. In den Zeilen 24 bis 30 ist dies dargestellt, für eine reine Überprüfung wird dies jedoch nicht benötigt.

Algorithmus 3.7: Zustandsreduktion

```
1  function Reducable
2      // Phase 1: Initialisierung
3      foreach i ∈ 𝕊\{|𝕊| − 1} do
4          foreach j ∈ 𝕊\{0,1,...,i} do
5              e[i, j] := ⋀_{k∈𝕏}(y_{i,k} = y_{j,k}) ∧ ((i ∈ 𝔽) = (j ∈ 𝔽))
6              e[j,i] := e[i, j]
7          e[i,i] := true
8      e[|𝕊| − 1, |𝕊| − 1] := true
9      // Phase 2: Äquivalenzpaare bilden
10     do
11         c := false
12         foreach i ∈ 𝕊\{|𝕊| − 1} do
13             foreach j ∈ 𝕊\{0,1,...,i} do
14                 if e[i, j] ∧ ¬∀k ∈ 𝕏: e[s'_{i,k}, s'_{j,k}] then
15                     e[i, j] := false
16                     e[j,i] := false
17                     c := true
18     while c
19     // Phase 3: Zusammenfassen
20     c := false
21     foreach i ∈ 𝕊\{|𝕊| − 1} do
22         foreach j ∈ 𝕊\{0,1,...,i} do
23             if e[i, j] then
24                 foreach k ∈ 𝕊 do
25                     foreach l ∈ 𝕏 do
26                         if s'_{k,l} = j then
27                             s'_{k,l} := i
28                 foreach l ∈ 𝕏 do
29                     s'_{j,l} := 0
30                     y_{j,l} := 0
31                 c := true
32     return c
```

Nach [HU79, Theorem 3.11] liefert dieses Verfahren einen minimalen Zustandsautomaten. Ein gleiches Ausgabeverhalten bei identischen Eingaben ist mit weniger Zuständen nicht möglich.

3.2.4 Präfix

Als Präfix wird hier der Teil einer Automaten-Zustandsübergangsfunktion bezeichnet, der nur zu Beginn erreicht wird, dessen Zustände aber im weiteren Verlauf keine Verwendung mehr finden. Dies kann geschehen, wenn ein Zyklus vorhanden ist, der unter keinen Umständen zum Zustand 0 zurückkehrt. Für einen Automaten mit einzigem Startzustand 0 ist es hinreichend, nur diesen Zustand 0 als Ziel zu überprüfen.

Es kann aber vorkommen, dass ein Zyklus innerhalb einer Automatenbeschreibung vorhanden ist, wie in Abbildung 3.9 dargestellt. Von daher ist zu überprüfen, ob ein Zy-

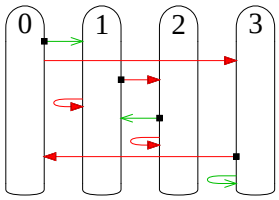


Abbildung 3.9: Ein Automat mit Präfix

klus existiert, der nicht mehr zum Zustand 0 zurückkehren kann. Hierfür bietet sich der Algorithmus zum Ermitteln der reflexiven transitiven Hülle aus [OW90, Seite 551] an, basierend auf [War62]. Inklusive Initialisierung und Endergebnisermittlung ist das Verfahren im Algorithmus 3.8 dargestellt.

Dabei ist zu beachten, dass für nicht erreichbare Zustände keine Präfixermittlung erfolgen darf, da es für diese Zustände unerheblich ist, ob der Zustand 0 in irgendeiner Form erreicht werden kann oder nicht.

Algorithmus 3.8: Reflexive transitive Hülle zur Präfixermittlung bezüglich Zustand 0

```
1 function PrefixFree
2     ∀i, j ∈ 𝕊: (Verbindung von i nach j ∨ i = j) ⇔ A[i, j] = true
3     foreach j ∈ 𝕊 do
4         foreach i ∈ 𝕊\{j} do
5             if A[i, j] then
6                 foreach k ∈ 𝕊 do
7                     if A[j,k] then
8                         A[i,k] := true
9     return ∀i ∈ 𝕊: ¬A[0,i] ∨ A[i,0]
```

Die transitive Hülle bestimmt in einem gerichteten Graphen die jeweils untereinander erreichbaren Knoten, die über gerichtete Kanten miteinander verbunden sind. Die Zustandsübergänge eines Automaten entsprechen den gerichteten Kanten, die Knoten repräsentieren die Zustände. Existiert also ein Weg zwischen zwei Knoten, bestehend aus einer Folge von gerichteten Kanten, so gibt es bei passender Eingabe eine Möglichkeit, diese Knoten bzw. Zustände zu erreichen.

Um eine vollständige Aussage auf Bezug des Präfix des gesamten Automaten zu erhalten, sind alle Startzustände aus $\mathbb{E}$ zu überprüfen, so dass sich die Bedingung $\forall i \in \mathbb{S}, j \in \mathbb{E}: A[j,i] \Rightarrow A[i,j]$ ergibt. Kommt allerdings noch die Überprüfung hinzu, ob alle Zustände genutzt werden („vereinfacht“, Abschnitt 3.2.2, ab Seite 34), so ist die zusätzliche Überprüfung nicht notwendig, da sich mit der zusätzlichen Bedingung $\forall i \in \mathbb{S}$: $A[0,i]$ die Gesamtbedingung $\forall i \in \mathbb{S}: A[0,i] \wedge A[i,0]$ ergibt. Die Graphen bzw. Automaten, die diese Bedingung erfüllen, sind stark zusammenhängend, besitzen also keinen nichtunterbrechbaren Teilzyklus.

3.2.5 Permutationsvarianten

Eine Permutation kann nicht nur für Zustände, sondern auch für die Eingangs- und Ausgabewerte existieren, wenn z. B. diesen Werte keine Semantik zugeordnet ist. Die entsprechenden Werte haben dann keine spezielle, semantische Bedeutung, da sie ja von den Werten her austauschbar sind. Es kommt nur auf die Wertunterschiede an, also dass

sich z. B. ein Automat bei zyklisch gleichen Eingangswerten anders verhält als bei konstant gleichen.

Kommen mehrere Permutationen in Frage, z. B. sowohl für Zustände als auch für die Eingänge, ist ein Test aller Kombinationsmöglichkeiten erforderlich. Die Permutationen je Wert nacheinander einzeln zu testen hätte zur Folge, dass sich bei nur gleichzeitig auftretender Permutation von Zustand und Eingang eine Identität mit einem anderem Automaten nicht zeigt. Abbildung 3.10 stellt ein Beispiel dafür dar.

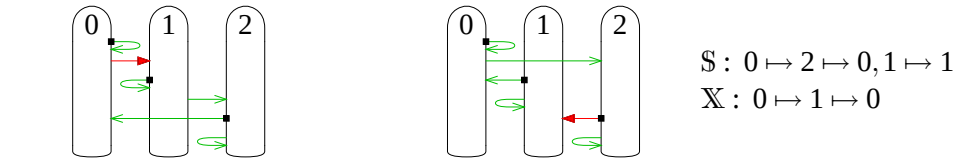


Abbildung 3.10: Gleichzeitige Zustands- und Eingangswertpermutation

Für die Aufzählung aller Automaten bleibt noch festzulegen, welcher Automat bei einer Permutation als Vertreter ausgewählt wird. Mit dem Algorithmus 3.3 ist eine Reihenfolge festgelegt. Der zuerst aufgelistete soll der Vertreter sein. Dies hat den Nachteil, dass die Überprüfung durch Algorithmus 3.6 auf Seite 33, ob die Zustände sortiert vorliegen, nicht mehr verwendet werden kann, da hier auch Automaten als normiert gelten, die eine in der Liste vorhergehende Zustandspermutation ausgeschlossen haben. Insofern erfolgt bei gleichzeitiger Normierungs- und Permutationsüberprüfungen der Ausschluss zu vieler Automaten.

Algorithmus 3.9 zeigt ein an die Anzahlbestimmung aus [HP73] bzw. der Funktion „AutomataAmount“ des Algorithmus 2.4 auf Seite 10 angelehntes Verfahren, um eine Permutation als Vertreter zu bestimmen. Dabei ist die Nomenklatur aus Abschnitt 2.2 verwendet, die Auswahl der Automaten erfolgt gemäß des Algorithmus 3.4 von Seite 28. Die Zustände sind in einzelne Gruppen von Start- und Terminalzuständen gemäß Abschnitt 3.1.5 ab Seite 28 mit hochgestellter Bereichsangabe für die verwendeten Spalten unterteilt.

Soll eine bestimmte Permutation, z. B. der Eingangswerte, nicht auftreten, so kann die entsprechende „**while**“-Bedingung ersatzlos entfallen; z. B. für die Beibehaltung der Ausgabewerte ohne Permutation entfallen die Zeilen 10 und 32. Die Initialisierung in Zeile 4 wird benötigt, damit in der Überprüfung unverändert υ stehen kann.

Der erste Test ab Zeile 12 kann übersprungen werden, da alle Permutationen initial vorliegen und eigentlich kein Wert permutiert wird. Daher kann nach Zeile 4 die Permutation der innersten Schleife durch einen Aufruf der entsprechenden „NextPermutation“-Funktion weitergeschaltet werden, im vorliegenden Beispiel mit allen drei Permutationsarten kann also zwischen den Zeilen 4 und 5 der Aufruf „NextPermutation($|\mathbb{Y}|$, υ)“ eingefügt werden.

Bei mehreren Startzuständen erfolgt ebenso eine Ermittlung des günstigsten Vertreters an erster Position. Die Startzustände sind in der Menge $\mathbb{E} \subseteq \mathbb{S}$ enthalten, die Terminalzustände in der Menge $\mathbb{F} \subseteq \mathbb{S}$. Die Verknüpfung $\mathbb{E} \cap \mathbb{F}$ besteht aus Zuständen, die sowohl Start- als auch Terminalzustand sind. Die Mengen $\mathbb{F}$, $\mathbb{E} \cap \mathbb{F}$, $\mathbb{E} \backslash \mathbb{F}$ sowie $\mathbb{S} \backslash (\mathbb{E} \cup \mathbb{F})$ können jeweils auch leer sein, nicht jedoch $\mathbb{S}$ und $\mathbb{E}$.

Anstatt einen Vertreter mit $\mathbf{s}'_{i,j}$ und $\mathbf{y}_{i,j}$ zu konstruieren, ist ein Vergleich mit $s'_{i,j} > \sigma(s'_{\sigma^{-1}(i),\chi^{-1}(j)})$ möglich. Dafür ist dann allerdings die Umkehrung der Permutation notwendig, z. B. für $\sigma(i) = j$ ist dies $\sigma^{-1}(j) = i$. Für den durchzuführenden Vergleich der

Algorithmus 3.9: Isomorphieüberprüfung aller Permutationen

```
1  function IsomorphismTest
2      InitPermutation(|𝕊|, σ)
3      InitPermutation(|𝕏|, χ)
4      InitPermutation(|𝕐|, υ)
5      do
6          do
7              do
8                  do
9                      do
10                         do
11                             // permutierten Automaten erstellen
12                             foreach i ∈ 𝕊 do
13                                 foreach j ∈ 𝕏 do
14                                     s'_{σ(i),χ(j)} := σ(s'_{i,j})
15                                     y_{σ(i),χ(j)} := υ(y_{i,j})
16                             // Test des permutierten Automaten bezüglich der
                                   Aufzählungsreihenfolge
17                             e := true
18                             for i := 0 to |𝕊| − 1 do
19                                 for j := 0 to |𝕏| − 1 do
20                                     if s'_{i,j} > s'_{i,j} then
21                                         return false
22                                     else
23                                         e := e ∧ (s'_{i,j} = s'_{i,j})
24                                         endloop i
25                             if e then
26                                 for i := 0 to |𝕊| − 1 do
27                                     for j := 0 to |𝕏| − 1 do
28                                         if y_{i,j} > y_{i,j} then
29                                             return false
30                                         if y_{i,j} < y_{i,j} then
31                                             endloop i
32                         while NextPermutation(|𝕐|, υ)
33                     while NextPermutation(|𝕏|, χ)
34                 while NextPermutation(|𝔼\𝔽|, σ)
35             while NextPermutation(|𝔼∩𝔽|, σ^{|𝔼\𝔽|⇢|𝕊|−1})
36         while NextPermutation(|𝕊\(𝔼∪𝔽|), σ^{|𝔼|⇢|𝕊|−1})
37     while NextPermutation(|𝔽\𝔼|, σ^{|𝔼∪(𝕊\𝔽)|⇢|𝕊|−1})
38     return true // kein permutierter Automat eher aufgezählt
```

Ausgabewertpermutation gilt dies analog, ist aber auch mit einer Sortierregel gemäß Algorithmus 3.29 möglich. Eine Kontrolle der Menge $\mathbb{F}$ von Terminalzuständen ist nicht notwendig, da kein Zustand die Mengenzugehörigkeit zu $\mathbb{E}$ oder $\mathbb{F}$ ändert.

3.2.6 Vollständigkeit der Klassifizierung

Die Überprüfungen auf Normiert bzw. Isomorphie, Vereinfacht und Präfix vermeiden lediglich Duplikate. Eine Änderung am Automaten wird dabei nicht vorgenommen – im Gegensatz zur Reduzierung, die bei den Zustandsübergängen eingreift.

Die Reduzierung der Anzahl von Zuständen, behandelt im Abschnitt 3.2.3, stellt somit die einzige Möglichkeit dar, die Anzahl der erreichbaren Zustände zu verringern und dabei gleichzeitig das Ausgabeverhalten beizubehalten. Allerdings ist die Reduktion nur bei unterschiedlichen Ausgaben anwendbar ($|\mathbb{Y}| > 1$) oder wenn einige der Zustände Terminalzustände sind ($0 < |\mathbb{F}| < |\mathbb{S}|$).

Für eine Aufzählung stellt sich die Frage, ob noch weitere Überprüfungen möglich sind, um die Anzahl der möglichen Automaten zu verringern. Aus obigen Erkenntnissen ergibt sich, dass dies allgemein für Automaten nicht machbar ist. Auch die mit [Mil80, § 7] und [Par81, Abschnitt 7] eingeführte Bisimulationsäquivalenz bietet keine weitere Unterstützung. Durch die Reduktion sind sogar nichtdeterministische Automaten notwendig, um die Äquivalenzbedingung herzustellen.

Bisher lag jedoch die Konzentration auf den Zuständen. Ein weiterer Aspekt ist die Relation von Ein- und Ausgaben. Wenn in die Betrachtung auch die Umgebung mit einfließt, kann man nicht nur unterschiedliches Verhalten, sondern auch für unterschiedlichste Einsatzzwecke brauchbare Automaten erhalten.

So kann überprüft werden, ob alle Eingänge und Ausgaben genutzt werden bzw. ob die Kombination aus Eingängen und Ausgaben ein sinnvolles Verhalten des Automaten darstellt. Damit ist die Analyse von der Semantik abhängig. Zwei mögliche Überprüfungen sind in den Abschnitten 3.4.7 und 3.4.8 gezeigt.

Bei der Auswahl der Überprüfungsbedingungen, also z. B. in einem bestehenden Programm die Einstellung von Parametern zur Auswahl der Kriterien zum Erhalt der gewünschten Automaten, ist das eigene Ziel und Umfeld zu beachten. Wenn mit einer maximalen Anzahl von Zuständen untersucht wird, ist es unter Umständen wenig sinnvoll, nur nach Automaten zu suchen, die alle Zustände nutzen, da sich auch mit weniger Zuständen relevante Automaten ergeben können. Andererseits ist eine Untersuchung von Automaten mit weniger als allen genutzten Zuständen nicht notwendig, wenn bereits alle Automaten mit weniger Zuständen in der Untersuchungsreihe enthalten sind.

Gibt es mehrere Startzustände, so ist die Untersuchung auf Isomorphie mittels Algorithmus 3.9 auf Seite 40 erforderlich, die Analyse der Zustände mittels Normierung gemäß Abschnitt 3.2.1 ab Seite 32 ist aufgrund der Charakteristik der Untersuchungsmethoden nicht mehr möglich.

3.3 Aufzählen von Automaten

Eine Auflistung aller interessanten Automaten ist mit dem Algorithmus 3.10 und der Funktion „Validate“, die alle notwendigen Klassifizierungskriterien auswertet, möglich. Allerdings werden auch viele uninteressante Automaten generiert und überprüft. Eine Optimierung liegt nahe.

Algorithmus 3.10: Testen der Automaten

```
1 InitiateAutomaton
2 do
3     if Validate then
4         IssueAutomaton
5 while NextAutomaton
```

In diesem Abschnitt sind verschiedene Methoden vorgestellt, die Untersuchung konzentriert sich dabei auf die in der Praxis meist verwendeten Automaten mit geordneten Zustandsübergängen (normiert), die alle Zustände nutzen (vereinfacht), ohne Präfix sind und keine reduzierbaren Zustände enthalten. Ziel ist es, eine Heuristik zu erstellen, aus der sich Automaten mit mehr Zuständen ergeben.

3.3.1 Gezieltes Testen

Eine Idee ist, eine Logik zu erstellen, die einen beliebigen Automaten als Eingabe erhält und einen gültigen Automaten ausgibt, der als nächstes in der Aufzählung erscheint. So ist es vielleicht möglich, durch Kombination von Zwei-Zustandsautomaten Mehr-Zustandsautomaten zu erhalten.

Leider hat sich weder für Zwei- noch für Drei-Zustandsautomaten eine Minimierbarkeit gezeigt. Das Ergebnis eines solchen Verfahrens entspricht der Logik, die sich mit den Verfahren aus Abschnitt 3.2 erstellen lässt, und bietet keine Vorteile gegenüber der simplen Aufzählung durch Algorithmus 3.10. Zum Einsatz kam dabei das mit [BHMSV84] vorgestellte Verfahren ESPRESSO II aus [BHM85], das in den meisten Fällen die beste Minimierungslösung zeigt, wie in [GB89] beschrieben ist.

3.3.2 Perfektes Orakel – Aufzählung durch Logik-Minimierung

Anstatt die Automaten mit der in Abschnitt 3.1.3 beschriebenen Methode durchzugehen, könnte auch eine Logik erstellt werden, die als Eingabe einen Zählerwert, die Ordinalzahl bezogen auf die Ergebnisse, erhält und den zugehörigen Automaten ausgibt. Aber auch hier hat sich mit den gleichen Minimierungsverfahren gezeigt, dass keine Reduzierung bei der Erstellung möglich ist. Es wird lediglich eine Art Speicherabbild erstellt und die gültigen Automaten in einer Liste angegeben. Zudem bleibt auch hier das Problem, dass zuerst eine Liste aller gültigen Automaten zu erstellen ist.

3.3.3 DCP: Dekomposition

Die Dekomposition einer binären Funktion basiert auf dem Verfahren, dass in [Ash59] beschrieben ist und in [Ive62] eine Umsetzung gefunden hat. Eine Funktion $f(\vec{x}) = y \in \{0,1\}$ wird dabei in Unterfunktionen zerlegt und der Eingangswert $\vec{x} \in \{0,1\}^n$ mittels $\vec{u}$ in zwei Eingangswerte aufgeteilt. Der Vektor[2] $\vec{u}$ hat je Eingangsbit ein Bit, um die Zuteilung anzugeben. Die Teile von x_i mit $u_i = 0$ durchlaufen eine Funktion $\vec{a}$, deren ein Bit breites Ergebnis zusammen mit den anderen Werten aus $\vec{x}$ in eine weitere Funktion gelangen, die oder-verknüpft aus drei Teilen besteht. Zum einen der Teil $\vec{b}$ für den Fall,

[2] Für Vektoren soll allgemein gelten, dass deren einzelne Werte per Index angesprochen werden können, also z. B. $\vec{u} = \begin{pmatrix} u_1 & \cdots & u_n \end{pmatrix}$.

dass das Ergebnis von $\vec{a}$ gleich 0 ist, des weiteren ein Teil $\vec{c}$ für den Fall, dass $\vec{a}$ gleich 1 ist, sowie einen von $\vec{a}$ unabhängigen Teil $\vec{d}$, dessen Werte ausschließlich von x_i mit $u_i = 1$ stammen. Zusammengetragen ist dies in Abbildung 3.11.

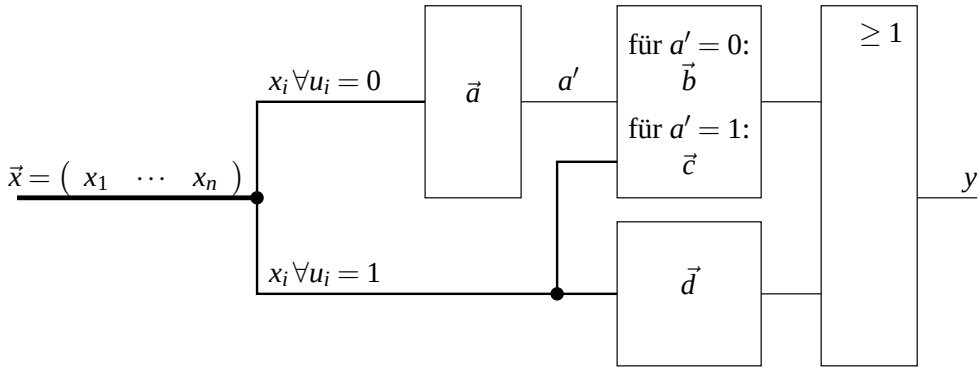


Abbildung 3.11: Prinzip der Dekomposition

Eine rekursive Ermittlung, mit der zusätzlich $\vec{a}$ eine Dekomposition erfährt, ist in [Hof74, Seite 59ff.] dargestellt; das ausführende APL/360-Programm ist dort auf Seite 139 zu finden. Mit Algorithmus 3.11 ist das Verfahren nochmals veranschaulicht, die dort aufgerufene Funktion „Ausgabe“ stellt die abstrahierte Ausgabe des Berechnungsergebnisses dar und ist hier nicht näher erläutert. Als Parameter des Funktionsaufrufes erhält „dcp“ die Funktion f in Form eines Vektors $\vec{y}$, dessen Elemente den jeweiligen Ausgabewert für die einzelnen Eingangswerte angeben, z. B. bei $x = 0$ ergibt sich der Index 0 und somit der zugehörige Ausgabewert y_0.

Auch hier hat das Verfahren weder für einen Index noch den Vorgängeralgorithmus als Eingabe ein Ergebnis hervorgebracht. So ist für die Indexierung der interessanten Zwei-Zustandsautomaten für die Bits von $s'_{1,1}, y_{1,0}$ sowie $y_{1,1}$ keine Dekomposition möglich. Für Drei-Zustandsautomaten hat sich für kein einziges Bit eine Zerlegung gezeigt. Von daher ist dieses Verfahren ungeeignet, da sich keine geeigneten Automaten mit mehr Zuständen aus bereits bekannten Automaten kombinieren lassen.

3.3.4 Aufzählen und Überspringen

Wie sich bisher gezeigt hat, gibt es keine allgemeine Schlussfolgerung, die eine Erweiterung auf mehr Zustände zulässt. Außer für die offensichtlichen Fälle, bei denen alle Zustände die gleichen Ausgaben aufweisen, ist insbesondere bei der Reduzierung kein Schema vorhanden.

Bleibt als Alternative, nur eine ungefähre Lösung zu erstellen, um in etwa den nächsten interessanten Automaten aufzuzählen, ohne dabei einen geeigneten Automaten zu überspringen. Dies entspricht einem Abschätzen der Kriterien, wobei die Kriterien einzeln zu betrachten sind. Der dabei berechnete Sprung, der die meisten Automaten überspringt, wird verwendet.

Neben dieser Sprungtechnik ist auch noch eine Abänderung zur reinen Überprüfung interessant. Zum Beispiel für die Normierung erfolgt mit Algorithmus 3.6 eine Neusortierung der Zustände und dann ein Vergleich. Stattdessen reicht eine Kontrolle aus. Dies geschieht mit Algorithmus 3.15.

Da sich die einzelnen Kriterien und Ergebnisse nicht trivial beschreiben lassen, ist deren Anwendung ausführlich in Abschnitt 3.4 ab Seite 46 beschrieben. Die Tabelle 3.2

Algorithmus 3.11: Dekomposition

1 **function** dcp($\vec{y} = \begin{pmatrix} y_1 & \cdots & y_n \end{pmatrix}$)
2 $n := \lceil \log_2 |\vec{y}| \rceil$
3 $u, \breve{u}, \acute{u} := 1, n-1, 1$
4 **do**
5 **if** $2 \leq \breve{u}$ **then**
6 $\vec{u} := \begin{pmatrix} u_1 & \cdots & u_n \end{pmatrix}$ mit $u_i \in \{0,1\}$ und $u = \sum_{i \in \mathbb{N}_0} u_i 2^i$
7 $\vec{f} := \begin{pmatrix} f_{1,1} & \cdots & f_{1,2^{\breve{u}}} \\ \vdots & \ddots & \vdots \\ f_{2^{\acute{u}},1} & \cdots & f_{2^{\acute{u}},2^{\breve{u}}} \end{pmatrix}$ mit (bei $j > n$ gilt $y_j = 0$)
$\forall 0 \leq j < 2^{\lceil \log_2 n \rceil}$: $f_{\sum_{k \in \{k \mid h_k = 0\}} u_k 2^{o_k}, \sum_{k \in \{k \mid h_k = 1\}} u_k 2^{l_k}} = y_j$ wobei
$(o_i + 1 = o_j \Leftrightarrow h_i = h_j = 0 \wedge i < k < j \wedge (\nexists k\colon h_k = 0))$ und
$(l_i + 1 = l_j \Leftrightarrow h_i = h_j = 1 \wedge i < k < j \wedge (\nexists k\colon h_k = 1))$ sowie
$\vec{h} = \begin{pmatrix} h_1 & \cdots & h_n \end{pmatrix}$ mit $h_k \in \{0,1\}$ und $j = \sum_{i \in \mathbb{N}_0} h_i 2^i$
8 $\vec{d} := \begin{pmatrix} d_1 & \cdots & d_{2^{\acute{u}}} \end{pmatrix}$ mit $\forall i\colon d_i = \begin{cases} 1 & \text{wenn } \forall j\colon f_{j,i} = 1 \\ 0 & \text{sonst} \end{cases}$
9 $\vec{e} := \begin{pmatrix} e_1 & \cdots & e_{2^{\acute{u}}} \end{pmatrix}$ mit $\forall i\colon e_i = \begin{cases} 1 & \text{wenn } \forall j\colon f_{j,i} = 0 \\ 0 & \text{sonst} \end{cases}$
10 $\vec{g} := \begin{pmatrix} g_1 & \cdots & g_{2^{\acute{u}}} \end{pmatrix}$ mit $\forall i\colon g_i = (d_i \vee e_i)$
11 **if** $\forall i\colon g_i = 1$ **then**
12 $\vec{a} := \begin{pmatrix} a_1 \\ \vdots \\ a_{2^{\breve{u}}} \end{pmatrix}$ mit $\forall i\colon a_i = 0$
13 $\vec{b} := \begin{pmatrix} b_1 & \cdots & b_{2^{\acute{u}}} \end{pmatrix}$ mit $\forall i\colon b_i = 0$
14 $\vec{c} := \begin{pmatrix} c_1 & \cdots & c_{2^{\acute{u}}} \end{pmatrix}$ mit $\forall i\colon c_i = 0$
15 Ausgabe($\vec{y}$, $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{d}$, $\vec{u}$)
16 **else**
17 $\vec{a} := \begin{pmatrix} a_1 \\ \vdots \\ a_{2^{\breve{u}}} \end{pmatrix}$ mit $\forall i\colon a_i = \begin{cases} f_{i,j} & \text{wenn } \exists j \forall k < j\colon g_k = 1 \wedge g_j = 0 \\ 0 & \text{sonst} \end{cases}$
18 $\vec{b} := \begin{pmatrix} b_1 & \cdots & b_{2^{\acute{u}}} \end{pmatrix}$ mit $\forall i\colon b_i = \begin{cases} 1 & \text{wenn } \forall j\colon \neg a_j = f_{j,i} \\ 0 & \text{sonst} \end{cases}$
19 $\vec{c} := \begin{pmatrix} c_1 & \cdots & c_{2^{\acute{u}}} \end{pmatrix}$ mit $\forall i\colon c_i = \begin{cases} 1 & \text{wenn } \forall j\colon a_j = f_{j,i} \\ 0 & \text{sonst} \end{cases}$
20 **if** $\forall i\colon g_i \vee b_i \vee c_i$ **then**
21 Ausgabe($\vec{y}$, $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{d}$, $\vec{u}$)
22 **if** $\exists i\colon a_i \neq 0$ **then**
23 dcp($\vec{a}$)
24 $u := u + 1$
25 $\acute{u} := |\{\alpha_i \mid \alpha_i = 1 \text{ bei } u = \sum_{i \in \mathbb{N}_0} \alpha_i 2^i\}|$
26 $\breve{u} := n - \acute{u}$
27 **while** $u < 2^n$

zeigt die Anzahl der möglichen Automaten für den Algorithmus 3.1 von Seite 26, um die Zielwertsuche des Problems aus Abschnitt 3.1.1 zu bearbeiten. Die Zahlen sprechen für sich. Sie zeigen für $|\mathbb{S}| = 4$, $|\mathbb{X}| = 2$ und $|\mathbb{Y}| = 8$ mit $z_{\max} = 15$, dass das Überspringen ein erfolgreiches Verfahren ist, um die aufgezählte Anzahl möglicher, aber uninteressanter Automaten zu reduzieren. Gefunden wurden 512 Automaten, die in $z_{\max} + 1$ Experimenten mit insgesamt 49 Schritten jeweils jeden Wert z von 0 bis $z_{\max}$ erreicht haben, ausgehend vom Startwert 0 und Zustand 0; der erste gefundene Automat ist in Abbildung 3.12 dargestellt.

Tabelle 3.2: Ergebnis der reduzierten Suche für Zielwertsuche

Menge	**Anzahl**
Wertkonstellationen	1 099 511 627 776
Aufgezählt	2 147 483 663
Relevant	1 425 746 368
⇒ Sprünge	721 737 295
⇒ ungeprüft übersprungen	1 097 364 144 113
Gefunden	512

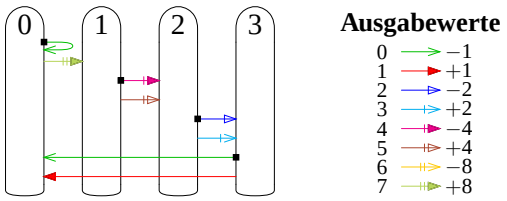


Abbildung 3.12: Ein Automat zur Durchführung der Zielwertsuche

3.3.5 Vorausschauende Berechnung

Das Verfahren des vorherigen Abschnitts 3.3.4 greift erst für die Berechnung bei einem uninteressanten Automaten, wie der nächste gültige Automat zu finden ist. Stattdessen ist denkbar, gleich eine Vorausberechnung vorzunehmen.

Hierzu wird, statt zu addieren und auf einen Fehlerfall zu prüfen, h im Voraus berechnet. Da jedoch mehr erfolgreiche Automaten gefunden als übersprungen werden, ist eine vorausschauende Alternative zu aufwendig, wie anhand der Zahlen in Tabelle 3.5 zu sehen ist. Des weiteren ist eine komplexere Berechnung notwendig, da allein schon für die Reduktion ein konkreter Automat benötigt wird, der dann temporär zu generieren ist. Dies bringt im Endeffekt keinen Unterschied zur vorherigen Lösung.

3.3.6 Genetik

Ein genetisch gewonnener Automat basiert auf einem beliebigen, zufällig ausgewählten Automaten. Dieser erfährt solange Modifikationen, bis ein Optimum erreicht ist. Dieses Verfahren, zum Beispiel beschrieben in [Poh00], kann schneller zu einem erfolgreichen Ergebnis führen. Es kann aber genauso passieren, dass die Veränderungen an einem lokalen Optimum hängen bleiben. Zudem ist ein Optimum als Kriterium festzulegen, um ein Ende für die Suche zu haben.

Der Unterschied zu dem Verfahren in Abschnitt 3.3.4 stellt die Rückkopplung des Simulationsergebnisses dar. Die Aufstellung von Kriterien ist, wenn überhaupt, lediglich für den ersten Schritt eines zufällig ausgewählten Automaten notwendig.

Damit ist ein simulationslastiges Verfahren gegeben, bestehend aus den Schritten Bestimmung der Fitness, Selektion, Rekombination, Mutation und Reinsertion. Um nun allgemein gültige Ergebnisse zu erhalten, sind zahlreiche Simulationen durchzuführen, wenn jeweils eine Minimalanforderung erfüllt ist. Ansonsten kann das Simulationsverfahren vorzeitig abgebrochen werden, um einen alternativen, entsprechend geänderten Algorithmus zu testen.

Allerdings ist ein allgemein gültiges Verfahren zu entwerfen, so dass nicht von einer effektiv durchführbaren Simulationsbasis ausgegangen werden kann. Von daher ist die genetische Automatengewinnung hierfür nicht geeignet, im konkreten Einzelfall ist aber eine Betrachtung durchaus angebracht.

3.4 Algorithmen zur Überprüfung

Die in Abschnitt 3.2 vorgestellten Algorithmen sind darauf ausgelegt, einen äquivalenten Automaten zu finden, der mit weniger Zuständen auskommt, die dann gemäß Definition 3.2 sortiert sind bzw. aufgrund einer Isomorphie keinen anderen Repräsentanten aufweisen. Wenn jedoch alle möglichen Automaten gemäß Abschnitt 3.3.4 aufgelistet werden sollen, reicht es, lediglich ein Entscheidungskriterium zu haben. Eine Äquivalenz braucht nicht ermittelt zu werden.

Interessant ist auch, welche Automaten bei der Suche übersprungen werden können, falls in einer Aufzählung mehrere aufeinander folgende Automaten das gleiche Kriterium nicht erfüllen.

3.4.1 Eingriff in die Aufzählung durch Überspringen

Wenn der höchstwertige Zustand zu einem nicht erfüllten Kriterium führt, so kann dieser Zustandsübergang im nächsten Aufzählungsschritt geändert werden, um rasch einen interessanten Automaten zu erhalten. Für die Überprüfung der Ausgaben y ist ein solch großer Ausschluss nicht gegeben, im Gegensatz zu den Zustandsübergängen. Von daher bleibt die mit den Algorithmen 3.2 und 3.3 gegebene Reihenfolge mit Lücken erhalten.

Bei diesem Verfahren hat der Zustand $|\mathbb{S}|-1$ die niedrigste Wertung. Ist ein Automat gefunden, der die Kriterien unabhängig von den Ausgabewerten y nicht erfüllt, stellt sich die Frage, welche Zustandsübergangsänderung sich auf die weitere Untersuchung positiv auswirkt.

Die Änderung an s'_{h_1,h_2} ist durch die Indizes h_1 und h_2 sowie durch den neuen Wert $h_3 \in \mathbb{S}$ gekennzeichnet. Zusammengefasst ergibt sich das Tupel $h = (h_1, h_2, h_3) \in \mathbb{S} \times \mathbb{X} \times (\mathbb{S} \cup \{|\mathbb{S}|\})$, wobei h für „hop“ steht. Der Wert $|\mathbb{S}|$ bei h_3 erzeugt ohne weitere Überprüfung einen Überlauf. Dies ist bei Analyseergebnissen nützlich, wenn kein Folgezustand für Zustand h_1 bei Eingangswert h_2 erreicht werden kann und somit sich kein gültiger Automat mehr bei dieser Konstellation ergibt.

Als Nebeneffekt dieses Verfahrens erfolgt automatisch ein Überlauf von s'_{h_1,h_2}, wenn h_3 den maximalen Wert von $|\mathbb{S}|-1$ erreicht hat und eine Änderung für den Index h_1, h_2 nicht mehr möglich ist, da dies bereits mit Algorithmus 3.3 implementiert ist. Damit dieser Algorithmus erhalten bleibt, soll eine Wertanpassung mit h so erfolgen, dass direkt

im Anschluss die Inkrementierung erfolgt. Dafür ist es notwendig, einen Zählerstand direkt vor dem gewünschten Wert zu setzen.

Algorithmus 3.12: Aufzählen von geeigneten Automaten unter Anwendung von Sprüngen

```
1  function NextValidAutomaton
2      while NextAutomaton do
3          h := (|𝕊|, |𝕏| − 1, 0) // h_1 ∉ 𝕊
4          if Validate then
5              return true
6          if h ∈ 𝕊 × 𝕏 × (𝕊 ∪ |𝕊|) then // h von Validate verändert?
7              foreach j ∈ 𝕏 do
8                  foreach i ∈ 𝕊 do
9                      y_{i,j} := |𝕐| − 1
10                 foreach i ∈ {h_1 + 1, h_1 + 2, ..., |𝕊| − 1} do
11                     s'_{i,j} := |𝕊| − 1
12             foreach j ∈ 𝕏\{h_2, h_2 + 1, ..., |𝕏| − 1} do
13                 s'_{h_1,j} := |𝕊| − 1
14             if s'_{h_1,h_2} < h_3 then // nur Vorwärtssprünge erlauben
15                 s'_{h_1,h_2} := h_3 − 1
16             if |𝕐| > 1 then // Automat aufgrund gleicher Ausgabewerte
                   reduzierbar
17                 if ¬NextAutomaton then
18                     endloop
19     return false
```

Die Ausgaben y erhalten ihren Maximalwert $|\mathbb{Y}|-1$, die niedrigwertigeren Zustandsübergänge h_1+1 bis $|\mathbb{S}|-1$ ihren maximalen Wert $|\mathbb{S}|-1$ und s'_{h_1,h_2} den Wert h_3-1. Dargestellt ist dies mit dem Algorithmus 3.12. Da nicht sichergestellt ist, dass der neu gefundene Automat die Kriterien erfüllt, muss eine erneute Überprüfung erfolgen, bevor ein neuer Automat erfolgreich gefunden ist.

Im Rahmen der Anwendung stellt sich noch die Frage, was passieren soll, wenn der aktuelle Wert von s'_{h_1,h_2} bereits größer als h_3 ist. Da der durch h_3 angegebene Zustand nur der minimale Wert des möglichen Zielbereiches ist, den es zu erreichen gilt, ist weiterhin ein Überspringen der Zustandsübergänge angebracht. Daher bleibt s'_{h_1,h_2} unverändert, um über die maximalen Ausgabewerte einen Überlauf und damit eine Erhöhung des Wertes s'_{h_1,h_2} zu generieren. In der Durchführung ist aber dieser Fall nie aufgetreten.

Aus diesem Berechnungsverfahren ergibt sich, dass ein kleiner Wert für die Indizes aus h mehr unnötige Überprüfungen auslässt. Von daher ist der Minimalwert zu finden, der sich aus allen unzutreffenden Kriterien ergibt. Existiert kein solcher, so erhält h_1 z. B. den ungültigen Wert $|\mathbb{S}| \notin \mathbb{S}$. Dies ermittelt die näher zu spezifizierende Funktion „Validate“, die in Zeile 4 des Algorithmus 3.12 aufgerufen wird und exemplarisch mit Algorithmus 3.13 dargestellt ist.

Verfehlt ein Automaten mehrere Kriterien, die allesamt aufgrund der Konstellation der Zustandsübergänge erfolglos sind, so gilt dies auch für die nachfolgenden Automaten mit den gleichen Zustandsübergängen, die demnach nicht untersucht werden müssen. Um möglichst viele dieser irrelevante Automaten zu überspringen, ist dasjenige Kriteri-

Algorithmus 3.13: Beispiel für die Überprüfung aller Kriterien eines Automaten

```
1  function Validate
2      v := true
3      v := v ∧ Normalized
4      if ¬(PrefixFree ∧ ∀i ∈ 𝕊\𝔼: ∃j ∈ 𝔼: A[j,i]) then // oder PartlyConnected
5          h := (|𝕊| − 1, |𝕏| − 1, s'_{|𝕊|−1,|𝕏|−1} + 1)
6          v := false
7      v := v ∧ SortedStates(true) // oder SortedStates1, PermutationRepresentative
8      v := v ∧ ¬Reducable
9      v := v ∧ OutputSemantic0 ∧ OutputSemantic1
10     v := v ∧ InputSemantic
11     return v
```

um zu finden, das einen maximalen Sprung über die nicht den Kriterien entsprechenden Automaten ermöglicht. Abbildung 3.13 zeigt das Prinzip in der Anwendung. Jedes Kriterium bietet hierfür einen eigenen Wert für h an, mit dem der ursächliche Zustandsübergang markiert und dafür ein das Problem lösender Wert angegeben ist, so dass dann – nach Anwendung des Sprungs – dieses eine Kriterium aufgrund der einen Ursache nicht mehr verletzt ist. Um jetzt alle Kriterien betreffend einen maximalen Sprung zu erzielen, ist der Minimalwert für h zu finden.

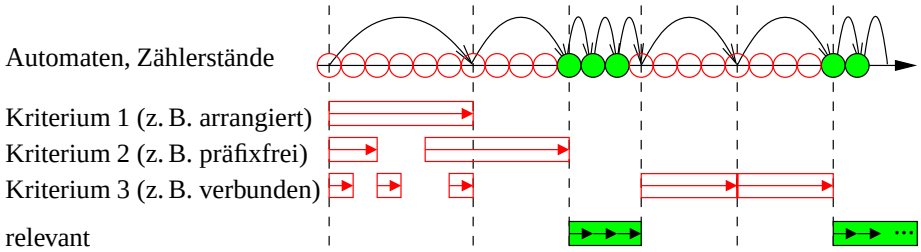


Abbildung 3.13: Prinzip für übersprungene Bereiche bei verletzten Kriterien

Zur Bestimmung eines Minimums für h müssen die einzelnen Teile des Tupels h getrennt betrachtet werden. Die beiden ersten Werte h_1 und h_2 sind voneinander abhängig und sollten minimal sein, der Wert h_3 hingegen erfordert ein Maximum, um möglichst viele Zustände zu überspringen. Das Ergebnis ist in der Gleichung 3.2 zu sehen.

$$\min\{a,b\} = \begin{cases} (b_1,b_2,b_3) & \text{wenn } b_1 < a_1 \\ (a_1,b_2,b_3) & \text{wenn } b_1 = a_1 \wedge b_2 < a_2 \\ (a_1,a_2,b_3) & \text{wenn } b_1 = a_1 \wedge b_2 = a_2 \wedge b_3 > a_3 \\ (a_1,a_2,a_3) & \text{sonst} \end{cases} \tag{3.2}$$

Alternativ zur Herbeiführung eines Überlaufs kann auch eine zusammengefasste Funktion die Zustandsübergänge von h_1 bis 0 inkrementieren und die anderen Werte zurücksetzen. Die Prozedur wird dadurch allerdings unübersichtlicher und nur geringfügig effizienter: Die Suche erfolgt nicht mehr für die y-Werte. Des weiteren überspringt die Suche den Fall der konstanten Ausgabe, bei der ein Automat ohne Zustände auskommt. Dieses Manko können aber die Zeilen 16 bis 18 des Algorithmus 3.12 beheben, indem sie eine Addition ohne Überprüfung vornehmen, die auf diese Weise diesen vollständig reduzierbaren Automaten übergeht.

Damit der Algorithmus 3.12 zur Anwendung kommt, ist eine Anpassung des Algorithmus 3.10 erforderlich. Da die Funktion „NextValidAutomaton“ bereits den nächsten gültigen Automaten in s' und y abgelegt hat, ist keine weitere Überprüfung mehr notwendig. Allerdings ist zu Beginn ein Test erforderlich, ob der initialisierte Automat die geforderten Kriterien erfüllt, bevor dieser übersprungen wird. Algorithmus 3.14 zeigt das Resultat.

Algorithmus 3.14: Testen der Automaten, alternativ zu Algorithmus 3.10

```
1 InitiateAutomaton
2 if Validate then
3     IssueAutomaton
4 while NextValidAutomaton do
5     IssueAutomaton
```

Den Erfolg der Sprungtechnik zeigt Abbildung 3.14 für Automaten mit vier Zuständen. Dabei sind die nachfolgend näher erläuterten Kriterien arrangiert, präfixfrei, verbunden und reduziert zu erfüllenden. Zwischen zwei relevanten Automaten befinden sich im Beispiel 2 503 427 Automaten, die allesamt übersprungen werden. Der zweite dargestellte Automat ist auf einen Zustand reduzierbar, der nachfolgende Automat weist einen Präfix auf, der erst mit Automat 1 835 009 nicht mehr besteht. Allerdings ist dieser nicht mehr arrangiert, so dass dies – mit einem Zwischenschritt über den ebenfalls nicht arrangierten Automaten 2 097 153 – insgesamt zu einer Änderung des Zustandsübergangs $s'_{0,0}$ von 0 nach 1 führt. Dabei werden alle, in der Aufzählung nachfolgende Automaten mit gleichen Zustandsübergängen und unterschiedlichen Ausgaben sowie alle Variationen der Zustandsübergängen $s'_{1,0}$ bis $s'_{3,1}$ und den Werten von 1 bis 3 für $s'_{0,1}$ übersprungen. Da der daraus resultierende Automat, viertletzter in der Abbildung, jedoch nur zwei Zustände verwendet, erfolgt noch eine Anpassung der übrigen Zustandsübergänge, so dass dann der zuletzt dargestellte, nicht-reduzierbare Automat entsteht. Mit fünf aufeinander folgenden Sprüngen sind somit über 14,9 % aller Vier-Zustands-Automaten ohne weitere Überprüfung aussortiert.

3.4.2 Normiert

Eine der Möglichkeiten, h zu setzen, ist die Überprüfung, ob der Automat normiert ist. Der auf Seite 33 vorgestellte Algorithmus 3.6 bereitet eine Neusortierung der Zustände vor, um dann mit einem Vergleich $s' \stackrel{?}{=} \mathbf{s}'$ ein Ergebnis zu erhalten. Für eine Überprüfung ist dieses Verfahren jedoch nicht notwendig. Vielmehr reicht es aus, die Reihenfolge der Zustände zu kontrollieren. Sobald es eine Unstimmigkeit gibt, ist der Automat als nicht-normiert erkannt.

Statt eine geplante Zustandsnummer in einem Array anzulegen, kann die tatsächlich existierende Nummer kontrolliert werden. Aber auch hierfür müssen die Programmschleifen in gleicher Weise durchlaufen werden. Es wird also die gleiche Komplexität $O(|\mathbb{S}|^2 \cdot |\mathbb{X}|)$ benötigt, aber der Speicherverbrauch von $2 \cdot |\mathbb{S}| + |\mathbb{X}|$ für die Algorithmusdurchführung und weitere $2 \cdot |\mathbb{S}| \cdot |\mathbb{Y}|$ Speicherplätze für das Ergebnis entfällt.

Im Prinzip vergibt der Algorithmus 3.15 für die höherwertigen Eingänge eine fortlaufende Nummer für den nächsten zu erreichenden Zustand, ohne dabei eine Zustandsnummer zu überspringen. Ein Test kann dann für einen Zielzustand t überprüfen, ob dieser

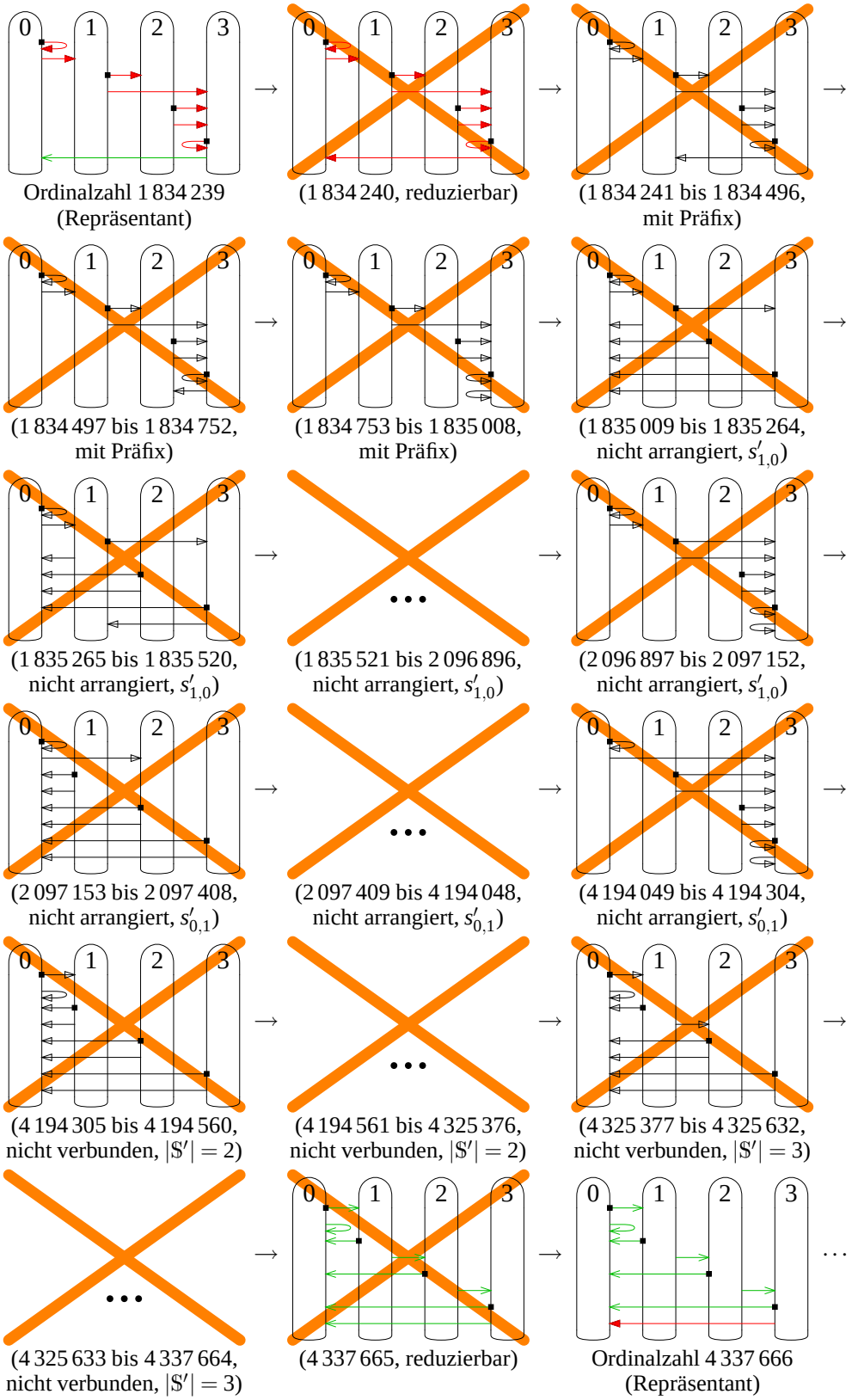


Abbildung 3.14: Bei der Aufzählung zwischen den Automaten mit den Ordinalzahlen 1 834 239 und 4 337 666 übersprungene Automaten – von der Ausgabe unabhängige Übergänge symbolisieren schwarze Pfeile mit einem unausgefüllten Dreieck als Spitze

Zustand auf adäquate Weise erreicht wird. Dies kann mit der im Satz 3.3 dargestellten Bedingung erfolgen.

Satz 3.3 *Für alle genutzten Folgezustände $t \in \mathbb{S}\backslash\{0\}$ gilt*

$$\bigvee_{j=0}^{|\mathbb{X}|-1} \bigvee_{i=0}^{t-1} \left((s'_{i,j} = t) \wedge \bigwedge_{k=0}^{t-1} \left((s'_{k,j} \leq t) \wedge \bigwedge_{l=0}^{j-1} (s'_{k,l} \leq t) \right) \right)$$

für Automaten mit den Eigenschaften normiert und vereinfacht.

Der Algorithmus 3.15 setzt dies um und sucht in Anlehnung an den Algorithmus 3.6 jeweils den nächsten zu erreichenden Zustand t, ausgehend von den Zuständen $< t$. Der höchstwertige, also kleinste Eingangswert ist dabei signifikant. Gibt es keinen Eingangswert, der kleiner ist und einen Zustand größer t erreicht, so ist der Zustand t korrekt angeordnet. Sind kleinere Eingangswerte als der gefundene möglich, so weisen diese auf Zustände $< t$ hin. Damit ist die Definition 3.1 hinreichend erfüllt.

Für den Fall, dass kein Wert gefunden werden konnte, also alle Zustandsverweise $< t$ sind, ist der Zustand t nicht erreichbar. Für eine reine Überprüfung aller Eigenschaften kann die Untersuchung damit abgebrochen werden, da der Automat die Kriterien nicht erfüllt.

Ist eine eingehende Diagnose erforderlich, so muss noch das Kriterium der Definition 3.2 untersucht werden. Dies erfolgt in den Zeilen 12 bis 14. Die Analyse vergleicht Folgezustände und Ausgaben aller verbliebenen Zustände auf den Wert 0, der durch die Definition vorgegeben ist. Sollen nur die relevanten Automaten aufgezählt werden, können diese Zeilen sowie Zeile 17 entfallen, da die Überprüfung in Zeile 11 hinreichend für eine Aussage ist.

Algorithmus 3.15: Überprüfen und Sprungziel setzen für die Eigenschaft „normiert“

```
1  function Normalized
2      for t := 1 to |𝕊| − 1 do
3          m, b := false, true
4          for j := 0 to |𝕏| − 1 do
5              for i := 0 to t − 1 do
6                  m := m ∨ (s'_{i,j} = t ∧ b)
7                  b := b ∧ (s'_{i,j} ≤ t)
8              if ¬m ∧ ¬b then
9                  h := min{h, (t − 1, j, t)}
10                 return false
11         if ¬m then
12             foreach i ∈ 𝕊\{0, 1, ..., t − 1} do
13                 foreach j ∈ 𝕏 do
14                     if s'_{i,j} ≠ 0 ∨ y_{i,j} ≠ 0 then
15                         h := min{h, (t − 1, |𝕏| − 1, t)}
16                         return false
17             return true // normiert, aber nicht alle Zustände genutzt
18     return true
```

Falls die Überprüfung negativ verlaufen ist, können Automaten in der Aufzählung übersprungen werden. Sollen nicht-vereinfachte Automaten mit in der Aufzählung enthalten sein, kommt die Bedingung in Zeile 14 zum Tragen. Der nicht erreichte Zustand t steht fest, die niedrigwertigste Änderung ist ein Verweis von Zustand $t-1$ nach t für den Fall der Eingabe $|\mathbb{X}|-1$. Dadurch ergibt sich die Bedingung in Zeile 15 zum Setzen von h.

Wird hingegen der Zustand t übersprungen, so muss auch hier h gesetzt werden. Ohne weitere Überprüfung gilt hier die gleiche Bedingung wie vorhin. Es lässt sich aber auch konkreter ermitteln, bei welchem Eingangswert der Sprung verursacht wird: In Zeile 4 erfolgt die Aufzählung der Eingangswerte nach Wichtung geordnet. Erfolgt ein Verweis über den Zustand t hinaus, ohne dass der gesuchte Zustand t erreicht ist, wird die Bedingung in Zeile 7 falsch. Ein Wert für h setzt sich aus den drei Komponenten

- Wert für den Eingang, der durch die Schleifenvariable j festgesetzt ist,
- das zu erreichende Ziel t und
- dem kleinstmöglichen Zustand, der eine Auswirkung auf Erfolg hat, Zustand $t-1$,

zusammen. Somit ist in Zeile 9 ein möglicher Minimalwert für h angegeben.

Ein alternativer Ansatz zur Überprüfung ist aufgrund der gegebenen Definition nicht möglich. Auch ist vom Prinzip her keine andere Definition für Normierbarkeit möglich, da die Zustände immer in Abhängigkeit der Eingänge sortiert sein müssen, lediglich die Priorität des Eingangs kann unterschiedlich sein. Es kann sich zwar eine andere Regel mit anderen, normierten Automaten entwickeln, aber die Komplexität der Überprüfung bleibt erhalten. Ein alternatives Verfahren bietet die Definition 3.6 auf Seite 57 mit einer anderen Auswahl von Automaten, bei der sich auch eine andere Sprungberechnung ergibt.

3.4.3 Genutzte Zustände

Mit dem vorherigen Verfahren des Algorithmus 3.15 lässt sich nur für normierte Automaten feststellen, ob alle Zustände genutzt sind, jedoch nicht für nicht-normierte. Daher muss für eine genaue Diagnose ein eigenes Verfahren entwickelt werden.

Basierend auf dem Verfahren Zellularer Automaten wird jeder Zustand durch eine Zelle repräsentiert. Die Nachbarschaftsverbindungen sind durch die Zustandsübergänge des Automaten angegeben. Der Startzustand 0 wird von sich aus erreicht. Danach kann sich die Eigenschaft „erreicht“ auf die Nachbarzustände ausbreiten. Nach spätestens $|\mathbb{S}|$ Übergängen ist das maximale Ergebnis erreicht.

Um einen Präfix zu ermitteln, kann das gleiche Prinzip angewendet werden, die Zustandsübergänge sind dann jedoch in umgekehrter Richtung anzuwenden. Enthält der Automat einen Präfix, so sind lediglich die temporär genutzten Zustände markiert.

Dieses Verfahren, das auch mit Algorithmus 3.16 und Abbildung 3.15 dargestellt ist, zeigt jedoch das gleiche Wirkungsprinzip wie das bereits mit Algorithmus 3.8 vorgestellte Verfahren zur Ermittlung der Transitiven Hülle nach [War62] und den Bedingungen $\forall i \in \mathbb{S}: A[0,i]$ sowie $\forall i \in \mathbb{S}: \neg A[0,i] \vee A[i,0]$. Da zudem noch für die Eigenschaften „vereinfacht“ und „präfixfrei“ zwei getrennte Berechnungen notwendig sind, ergibt sich ein doppelter und damit zu hoher Aufwand für die Umsetzung, so dass dieses Verfahren für eine optimale Ermittlung nicht in Betracht kommt. Allein schon für die Eigenschaft „vereinfacht“ kommen 40 Logikelemente mehr zum Einsatz bei vier Zuständen und zwei

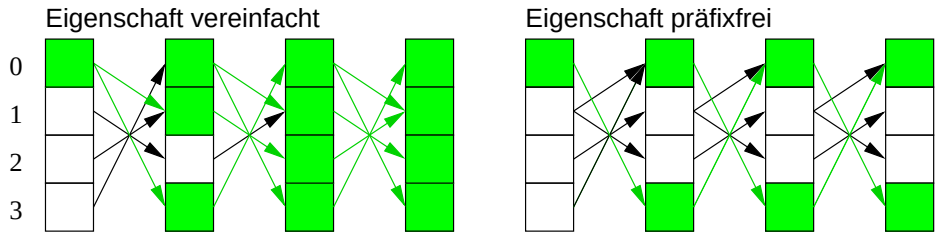


Abbildung 3.15: Erreichbare Zustände für den Automaten aus Abbildung 3.9

Eingängen gegenüber dem Warshall-Algorithmus, da mehr Vergleicher statt Speicher benötigt werden.

Algorithmus 3.16: Erreichbare Zustände nach dem Prinzip eines zellularen Automaten

```
1 function Reachable_CA
2     ∀i ∈ 𝕊: a_i = (i = 0)
3     foreach p ∈ 𝕊 do
4         foreach j ∈ 𝕊 do
5             foreach i ∈ 𝕊 do
6                 a_i := a_i ∨ ⋁_{t∈𝕏} s'_{i,t} = j
7     return ⋀_{i∈𝕊} a_i
```

Eine andere Alternative ist das Verfahren nach Hirschberg, beschrieben in [HCS79] und illustriert mit Algorithmus 3.17. Die Durchführung benötigt $\log_2|\mathbb{S}| \cdot (\log_2|\mathbb{S}| + 4) + 1$ Schritte ($O(\log^2|\mathbb{S}|)$) mit $|\mathbb{S}| \cdot \lceil |\mathbb{S}| \div \lceil \log_2|\mathbb{S}| \rceil \rceil$ Prozessoren. Das Verfahren nach [War62] lässt sich in $|\mathbb{S}|$ Schritten mit $|\mathbb{S}|^2$ Prozessoren gestalten, wobei die „Prozessoren“ jeweils als eine einfache Logik-Operation mit drei Eingängen verwirklicht sind. Somit ist in diesem Fall lediglich für $|\mathbb{S}| \leq 2$ der Algorithmus nach Hirschberg günstiger als der nach Warshall, wie sich empirisch berechnen lässt.

Es gibt auch noch andere Verfahren, um eine transitive Hülle zu bestimmen. Diese sind z. B. auf große Datenmengen optimiert, die nicht gleichzeitig im Hauptspeicher vorliegen können. Beispiele hierfür finden sich in [Nuu95]. Die Algorithmendurchführungen sind jedoch komplex. Die Anzahl der Zustände eines betrachteten Automaten, also die Anzahl der Knoten eines Graphen, ist jedoch gering. Das gleiche Ergebnis ist auch in [Pur70, Seite 82] dokumentiert.

Im Vergleich ist das Verfahren nach Warshall hinreichend effizient, um die beiden Kriterien „vereinfacht“ und „präfixfrei“ für die Automaten zu bestimmen.

Das Überspringen bei der Aufzählung ist für das Kriterium „vereinfacht“ nicht notwendig, da h bereits einen Wert in Zeile 15 des Algorithmus 3.15 erhält. Konkret lässt sich aus dem bisher Festgestellten der Satz 3.4 ableiten:

Satz 3.4 (Wert für h aufgrund von „vereinfacht“) *Wenn die Bedingung* $A[0,i] \vee \forall j \in \mathbb{X}: (s'_{i,j} = 0 \wedge y_{i,j} = 0)$ *für ein* $i \in \mathbb{S}$ *verletzt ist, dann erhält* h *den Wert* $\min\{h, (k-1, |\mathbb{X}|-1, k)\}$ *mit* $k = \min\{k \leq i \mid \neg A[0,k]\}$. *Soll nur eine Überprüfung für Nicht-Startzustände erfolgen, dann hat* i *stattdessen den Wertebereich* $\mathbb{S}\backslash\mathbb{E}$. *Die Bedingung des* $\forall$*-Quantors kann in beiden Fällen entfallen, wenn alle Zustände zu erreichen sind.*

Anders sieht es für die Präfixermittlung aus. Hierfür muss h im Bedarfsfall noch gesetzt werden. Da ein Verweis auf einen Zustandswert an beliebiger Stelle, der wieder

Algorithmus 3.17: Hirschberg-Verfahren, angepasst an die aktuellen Werte, zur Ermittlung der transitiven Hülle in $O(\log^2 n)$ mit $n\lceil n/\lceil \log_2 n \rceil \rceil$ Prozessoren, wobei $n = |\mathbb{S}|$

1 $\forall i, j \in \mathbb{S}$: *Verbindung von i nach* $j \vee i = j \Leftrightarrow A[i,j] = \text{true}$
2 $\forall i \in \mathbb{S}: D[i] = i$
3 **for** $c_1 := 1$ **to** $\lceil \log_2 |\mathbb{S}| \rceil$ **do**
4 **foreach** $i \in \mathbb{S}$ **do**
5 $\mathbb{C} := \{D[j] \mid j \in \mathbb{S} \wedge A[i,j] = 1 \wedge D[j] \neq D[i]\}$
6 $C[i] := \begin{cases} \min \mathbb{C} & \text{wenn } \mathbb{C} \neq \varnothing \\ D[i] & \text{wenn } \mathbb{C} = \varnothing \end{cases}$
7 **foreach** $i \in \mathbb{S}$ **do**
8 $\mathbb{C} := \{C[j] \mid j \in \mathbb{S} \wedge D[j] = i \wedge C[j] \neq i\}$
9 $C[i] := \begin{cases} \min \mathbb{C} & \text{wenn } \mathbb{C} \neq \varnothing \\ D[i] & \text{wenn } \mathbb{C} = \varnothing \end{cases}$
10 **foreach** $i \in \mathbb{S}$ **do**
11 $D[i] := C[i]$
12 **for** $c_2 := 1$ **to** $\lceil \log_2 |\mathbb{S}| \rceil$ **do**
13 **foreach** $i \in \mathbb{S}$ **do**
14 $T[i] := C[C[i]]$
15 $C := T$
16 **foreach** $i \in \mathbb{S}$ **do**
17 $D[i] := \min\{C[i], D[C[i]]\}$
18 **foreach** $i \in \mathbb{S}$ **do**
19 **foreach** $j \in \mathbb{S}$ **do**
20 $A[i,j] := D[i] = D[j]$

auf Zustand 0 zurückverweist, wieder zu einem erfolgreichen Automaten führen kann, bleibt nur die Möglichkeit offen, den höchsten Eingangswert zu verändern, also $h_2 = |\mathbb{X}| - 1$. Für die beiden anderen Werte h_1 und h_3 gestaltet sich die Sache schwieriger.

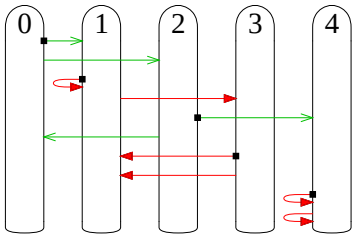


Abbildung 3.16: Nicht-triviales Beispiel für die Ermittlung von $h = (3,1,2)$

Angenommen, in einem Automaten gibt es zwei Zyklen, die von einem Präfix aus erreicht werden können, wie in Abbildung 3.16 zu sehen. Dann führt eine Änderung des höher priorisierten Zustands zu einem größeren Erfolg. Demnach ist im ersten Schritt der kleinste Wert $i \in \mathbb{S}$ zu suchen, der die Präfixbedingung $\neg A[0,i] \vee A[i,0]$ nicht erfüllt. Als nächstes ist der größte Wert $j \in \mathbb{S}$ zu finden, der von i aus erreichbar ist, für den also $A[i,j]$ gilt. j ist der Wert für h_1.

Der kleinstmögliche Wert $k \in \mathbb{S}$, für den $i \leq s'_{h_1,h_2} = s'_{j,|\mathbb{X}|-1} < k$ und $A[k,0]$ gilt, ergibt den Wert für h_3. Da aber mit Algorithmus 3.12 alle Zustandsübergänge für Zustände $> h_1 = j$ ihren Maximalwert erhalten, verweist nach der darauf unabdingbar folgenden

Erhöhung bereits der Zustand $j+1$ auf den Zustand 0. Für den Fall $k > h_3$ ist h_3 stattdessen auf den Wert $j+1$ als kleinsten Zielwert zu setzen.

Der Wert von $s'_{j,|\mathbb{X}|-1}$ ist kleiner als $j+1$, da j der maximale Zustand im Zyklus nach dem Präfix ist und aus obiger Vorbedingung kein Zustand k dazwischen liegt, der auf Zustand 0 verweist. Des weiteren folgt aus $\neg A[i,0]$ und $A[i,j]$ die Aussage $\neg A[j,0]$. Somit ist sichergestellt, dass in der Aufzählung kein gültiger Automat übersprungen wird.

$$\begin{aligned} i &:= \min\{i \in \mathbb{S} \mid A[0,i] \wedge \neg A[i,0]\} \\ j &:= \max\{j \in \mathbb{S} \mid A[i,j]\} \\ k &:= \min\left(\{k \in \mathbb{S} \mid s'_{j,|\mathbb{X}|-1} < k \wedge A[k,0]\} \cup \{j+1\}\right) \\ h &:= \min\{h, (j, |\mathbb{X}|-1, k)\} \end{aligned} \tag{3.3}$$

Um eine aufwendige Ermittlung mit den Zuweisungsgleichungen 3.3 zu vermeiden, aber wenigstens die Ausgabevarianten zu überspringen, ist der Wert $(|\mathbb{S}|-1, |\mathbb{X}|-1, s'_{|\mathbb{S}|-1,|\mathbb{X}|-1}+1)$ für h möglich. Ein weitreichender Sprung ist zeitlich von Vorteil; so reduziert sich bei sieben Zuständen die Anzahl um 943 492 367 Berechnungen, dies entspricht einer Zeitersparnis von 0,033 %.

Des weiteren ist zu Beginn eine weitere Optimierung möglich, um nicht für alle Zustände $< |\mathbb{S}|-1$ den Wert h zu setzen. Der Algorithmus 3.2, ergänzt um Algorithmus 3.18, setzt die notwendigen Zustandsübergänge des größten Eingangswertes $|\mathbb{X}|-1$ für einen besseren Start.

Algorithmus 3.18: Initialisieren auf ersten streng verbundenen Automaten als Addendum zu Algorithmus 3.2 von Seite 26

```
6 foreach i ∈ 𝕊\{|𝕊|−1} do
7     s_{i,|𝕏|−1} := i+1
```

3.4.4 Variable Zustandsanzahl

Ist eine Liste von Automaten *bis zu* einer maximalen Anzahl von Zuständen gewünscht, die miteinander streng verbunden sind und dabei die für die Normierung aufgestellten Bedingung einhalten, ist eine andere Überprüfung erforderlich. Orientiert an Definition 3.2 auf Seite 34, in der für ungenutzte Zustände feste Werte vorgegeben sind, lassen sich die anderen Kriterien betrachten.

Für das Kriterium „Reduziert“ ergibt sich ein zusätzliches Problem: Die nicht genutzten Zustände sind identisch und können zu einem Zustand reduziert werden. Damit erfüllt ein solcher Automat nicht mehr die notwendige Bedingung und wird fälschlicher Weise aussortiert.

Das Überspringen eines garantiert reduzierbaren Automaten durch die Zeilen 16 bis 18 des Algorithmus 3.12 hat keinen Einfluss auf die Ergebnisliste, da die verbleibenden Zustände aufgrund ihrer gleichen Ausgabe reduzierbar sind. Anders sieht es mit dem die Initialisierung ergänzenden Algorithmus 3.18 aus. Hier bleiben Zustandsübergänge unbeachtet, die aber relevant sind. Von daher muss Algorithmus 3.2 allein gestellt bleiben.

Auch das Verändern von h durch die verschiedenen Überprüfungen hat keine negativen Auswirkungen, lediglich der Test auf Vollständigkeit, also Nutzung aller Zustände, sollte unterbleiben. Die Überprüfung des Präfix ist kein Problem, da eine Rückkehr zum

Zustand 0 gewährleistet ist und zudem nur die erreichbaren Zustände das Ergebnis beeinflussen.

Basierend auf diesen Rahmenbedingungen lässt sich eine Regel definieren, um streng verbundene Automaten mit weniger Zuständen zu erhalten. Damit die bisherigen Regeln unverändert bestehen bleiben, sind die ungenutzten Zustände im niedrig priorisierten Bereich der hohen Zustandswerte; als Kennzeichen haben diese ihren Anfangswert.

Definition 3.5 (Auswahl genutzter Zustände $\mathbb{S}'$) *Ein Automat ist für eine Auswahl der Zustände $\mathbb{S}' \subsetneq \mathbb{S}$ mit $\mathbb{S}' = \{i \in \mathbb{S} \,|\, A[0,i]\}$ als **teil-verbunden** ausgewählt, wenn für die verbleibenden Zustände $\mathbb{S} \backslash \mathbb{S}' \neq \varnothing$ die Bedingung $\forall i \in \mathbb{S} \backslash \mathbb{S}',\, j \in \mathbb{Y} \colon s'_{i,j} = 0 \,\wedge\, y_{i,j} = 0$ gilt. A bezeichnet dabei das Ergebnis der reflexiven transitiven Hülle durch Algorithmus 3.8.*

Für isomorphe Vertreter bzw. normierte Automaten sind die Werte in $\mathbb{S}'$ zusammenhängend. Die ausgeschlossenen, unerreichbaren Zustände dürfen keine Startzustände sein, so dass sich neben $\mathbb{S}'$ auch $\mathbb{E}' = \{i \in \mathbb{E} \,|\, i \in \mathbb{S}'\}$ ergibt. Alternativ ist bei $\mathbb{E} \neq \mathbb{S}$ denkbar, Startzustände von der Überprüfung auszuschließen, um eine andere Definition zu erhalten; dann lautet die Bedingung $\mathbb{S}' = \{i \in \mathbb{S} \,|\, A[0,i] \,\vee\, i \in \mathbb{E}\}$. Die Reduktion vermeidet nach wie vor automatisch Duplikate der Startzustände.

Um mit dieser Einschränkung die Reduktion überprüfen zu können, ist die Bedingung „$\wedge\, i \in \mathbb{S}'$“ in Zeile 5 des Algorithmus 3.7 anzuhängen. Da i der größere der beiden Werte ist und somit eher in $\mathbb{S}'$, muss j bei normierter bzw. isomorphiefreier Aufzählung nicht betrachtet werden. Alternativ können alle Werte $e[i,j]$ und $e[j,i]$ mit $i \in \mathbb{S} \backslash \mathbb{S}',\, j \in \mathbb{S}$ nach Zeile 8 den Wert „false“ erhalten.

Algorithmus 3.19: Überprüfen und Setzen von h für teilzusammenhängende Automaten

```
1 function PartlyConnected
2     if ∃i ∈ 𝕊\𝕊', j ∈ 𝕏: s'_{i,j} ≠ 0 ∨ y_{i,j} ≠ 0 then
3         h := min{h, (|𝕊'|, 0, |𝕊|)}
4         return false
5     return true
```

Bei verletzter Definition 3.5 ist die nächste Zustandskonstellation herbeizuführen. Algorithmus 3.19 übernimmt diese Aufgabe. Aufgrund der Aufzählungsreihenfolge durch Algorithmus 3.3 ist die $\exists$-Überprüfung mit $\neg A[0, |\mathbb{S}| - 1] \,\wedge\, y_{|\mathbb{S}|-1,|\mathbb{X}|-1} \neq 0$ gleichbedeutend, $|\mathbb{S}'|$ hat dabei den Wert $\min\{i \in \mathbb{S} \,|\, \neg A[0,i]\}$ bzw. $\min\{i \in \mathbb{S} \backslash \mathbb{E} \,|\, \neg A[0,i]\}$.

Mit Bestimmung der Anzahl von miteinander verbundenen Zuständen $|\mathbb{S}'|$ lässt sich zusätzlich auch eine Untergrenze der Automateneigenschaft festlegen und überprüfen. In diesem Fall erhält h bei Verletzen der Regel den Wert $\min\{h, (|\mathbb{S}'| - 1, |\mathbb{X}| - 1, |\mathbb{S}'|)\}$, wie auch schon für Algorithmus 3.15 mit $t = |\mathbb{S}'|$ diskutiert.

3.4.5 Reihenfolge der Permutationen

Eine Überprüfung mit allen Permutationen wie mit Algorithmus 3.9 bedeutet bei z. B. sieben Zuständen ein Test von $7! = 5040$ Möglichkeiten allein nur die Zustände betreffend. In Hardware umgesetzt bedeutet dies eine Zusammenführung von 5040 Einzelergebnissen, deren Werte vorher erst zu erstellen sind. Dies ist effizient nicht handhabbar.

Von daher liegt die Idee nahe, nach einem alternativen Verfahren zu suchen, das die gleichen Ergebnisse mit weniger Überprüfungen liefert. Dafür sind allerdings die unterschiedlichen Permutationen σ, χ und υ zuerst getrennt voneinander zu betrachten.

Unter der Einschränkung, dass alle Zustände untereinander stark verbunden sein sollen, also von jedem Zustand zu jedem anderen Zustand ein Weg existiert und damit die Eigenschaften Vereinfacht und Präfixfrei erfüllt sind, lässt sich die Zustandspermutation einfacher überprüfen, indem z. B. die Zustandsübergänge eine festgelegte Reihenfolge aufweisen. Die nachfolgenden Abschnitte befassen sich mit der Feststellung dieser Reihenfolge.

Erfolgt die Überprüfung nur für eine Art der Permutation, z. B. nur für Ausgabewerte, bestehen die anderen Permutationen nur aus ihrer identischen Abbildung. Eine entsprechende Vorinitialisierung nimmt der Algorithmus 3.20 vor. Die für einen Vergleich zu erstellenden permutierten Automaten als Zwischenergebnis, dargestellt mit den fett gedruckten Bezeichnern für Zustandsübergänge $\mathbf{s}'$ und Ausgabewerte $\mathbf{y}$, zeigt der Algorithmus 3.21 nur zur Rekapitulation, da die Permutationen selbst in stark abgeänderter Form aufgebaut werden.

Algorithmus 3.20: Vorinitialisierung der Permutationen

```
1 InitPermutation(|𝕊|, σ)
2 InitPermutation(|𝕏|, χ)
3 InitPermutation(|𝕐|, υ)
```

Algorithmus 3.21: Zustandsübergänge und Ausgaben permutiert kopieren

```
1 foreach i ∈ 𝕊 do
2     foreach j ∈ 𝕏 do
3         s'_{σ(i),χ(j)} := σ(s'_{i,j})
4         y_{σ(i),χ(j)} := υ(y_{i,j})
```

3.4.5.1 Zustandsübergänge bei einem Startzustand

Ausgehend von der Aufzählung gemäß Algorithmus 3.3 hat die Änderung bei den Eingangswerten eine geringere Auswirkung als eine Modifizierung von Zustandsübergängen. In der Aufzählung sind auch kleine Werte zuerst aufgeführt. Von daher müssen die kleineren Werte eher bei den kleineren Zuständen und dort bei den kleineren Eingangswerten stehen. Daraus ergibt sich die nachfolgende Definition 3.6, um Vertreter von zustandsisomorphen Automaten zu bestimmen.

Definition 3.6 (Arrangierte Zustandsübergangsabfolge) *Beginnend bei Zustand* 0 *erhält der Folgezustand des höchstpriorisierten Eingangswertes die maximale nächste Zustandsnummer* 1*, die weiteren Nummern werden nach aufsteigenden Eingangswerten und in zweiter Linie nach aufsteigenden Zustandsnummern vergeben.*

Im Umkehrschluss heißt dies, dass kein Zustandsübergang einen Zustand übergehen darf. Möglich ist dies zum Beispiel mit einem Zielzähler *t* (für *target*), der neu erreichte

Zustände markiert. Alle Zustände $\leq t$ gelten als markiert. Gibt es einen Sprung bei geordneter Aufzählung der Zustände i und Eingangswerte j, ist also die obige Definition 3.6 verletzt, ist der Automat irrelevant.

Bleibt die Frage nach den zu überspringenden Automaten. Da für den Eingangswert i bei Zustand j bereits ein zu hoher Wert in $s'_{i,j}$ vorliegt, kann für diese Stelle ein Überlauf erfolgen. Die entsprechende Zuweisung befindet sich in Zeile 6 des Algorithmus 3.22, der vollständig die Auswirkung der Definition 3.6 überprüft.

Algorithmus 3.22: Zustandsisomorphietest bei einem Startzustand ohne Terminalzustände

```
1  function SortedStates1
2      t := 1
3      for i := 0 to |𝕊| − 1 do
4          for j := 0 to |𝕏| − 1 do
5              if s'_{i,j} > t then
6                  h := min{h, (i, j, |𝕊|)}
7                  return false
8              if s'_{i,j} = t then
9                  t := t + 1
10     return true
```

Ist nach Ausführen des Algorithmus 3.22 der Wert von $t < |\mathbb{S}| - 1$, so gibt es unerreichbare Zustände. Ein vorzeitiges Ergebnis der Unerreichbarkeit ergibt $i > t$ innerhalb der Algorithmusdurchführung. Dies mit einer eventuellen Nebenbedingung zu inspizieren, bleibt aber dem Kriterium „Vereinfacht“ vorbehalten.

3.4.5.2 Zustandsübergänge ohne Einschränkung

Gibt es nun mehrere Startzustände oder nur teilweise Terminalzustände, ist der Algorithmus 3.22 unzureichend, da z. B. auch vom Startzustand 0 ein Terminalzustand mit wesentlich höherer Zustandsnummer für einen gültigen Automaten erreichbar sein muss. Da es mehrere Zielbereiche gibt, liegt es nahe, den Zähler t durch eine Maske zu ersetzen, der eine Markierung für gültige Zielzustände aufweist.

Wie auch schon zuvor ist der erste Startzustand 0 als auch der Zustand 1 ein jeweils gültiges Ziel. Sollte es weitere Startzustände geben, müssen diese erst im Laufe der Analyse ihre Marke und damit verbunden eine mögliche Erreichbarkeit erhalten, damit nicht eine Permutation innerhalb der Startzustände entsteht. Ebenso sind die ersten Zustände aller anderen Bereiche wie Terminalzustände ohne Startzustände von Anfang an ein gültiges Ziel. Für zehn Zustände ist ein Beispiel in Abbildung 3.17 gegeben.

Die Initialisierung der Marken kann unabhängig von der Existenz der einzelnen Teilmengen erfolgen. So ist bei fehlenden terminierenden Startzuständen $|\mathbb{E} \cap \mathbb{F}| = |\mathbb{E}|$, der betroffene Zustand wird also aus zwei Gründen markiert. Ähnlich ist es für die anderen Bedingungen. Allerdings besteht ein Problem, wenn z. B. nur Startzustände existieren, also $\mathbb{E} = \mathbb{S}$. In einem solchen Fall wird $|\mathbb{E}| = |\mathbb{S}|$ initialisiert, also eine Marke für einen Zustand gesetzt, der nicht existiert. Dies führt aber zu keinen weiteren Problemen, lediglich die Maske muss entsprechend viele Stellen aufweisen.

Der weitere Ablauf ist dem zu Algorithmus 3.22 ähnlich. Bei Erreichen eines Zustands erhält der nachfolgende eine Markierung als gültiges Ziel. Betrachtet man das

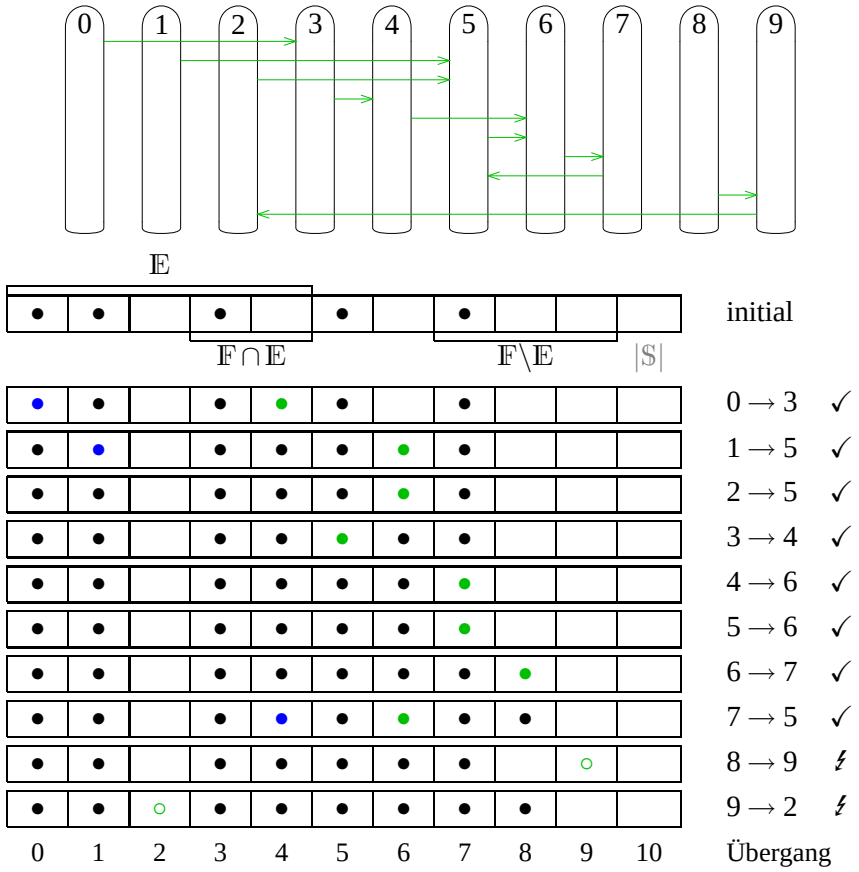


Abbildung 3.17: Zustandsisomorphietest mit $|\mathbb{S}| = 10$ und $|\mathbb{X}| = |\mathbb{Y}| = 1$

Beispiel in Abbildung 3.17 und setzt $s'_{2,0} = 7$ statt 5, so zeigt sich ein Problem bei Zustand 9. Im Gegensatz zum ursprünglichen liegt nun keine Vertauschung der Zustände 1 und 2 vor, lediglich noch von 8 und 9, allerdings ist 2 bisher noch nicht als gültiges Ziel markiert, obwohl kein anderer Startzustand mit Übergang zu einem terminierend Zustand, der nicht gleichzeitig Startzustand ist, vorkommt. Als Lösung bietet sich an, neben dem Ziel auch die Quelle zu markieren. Der aus Algorithmus 3.22 abgeleitete Algorithmus 3.23 setzt dies in Zeile 8 für $t[i]$ um.

Im Vergleich mit Algorithmus 3.22 zeigt sich, dass h im Algorithmus 3.23 den gleichen Wert erhält – mit der gleichen Begründung, wie im vorigem Abschnitt 3.4.5.1 beschrieben. Damit ist die Permutationsüberprüfung vollständig, alle Übergangsmöglichkeiten für einen vereinfachten Automaten sind berücksichtigt. Der Parameter η ist für zukünftige Erweiterungen zum optionalen Setzen des Wertes h; mit Algorithmus 3.30 findet dies Anwendung.

Die Abbildung 3.17 zeigt einen vollständigen Test, obwohl eigentlich nach Zustand $i = 8$ das Ergebnis feststeht. Die letzte, unnötige Kontrolle ist nur der Vollständigkeit halber aufgeführt.

3.4.5.3 Anderer Startzustand an der Stelle von Zustand 0

Schließlich gibt es noch eine weitere Überprüfung: Bisher wurde nur verifiziert, ob eine ideale Anordnung ab Zustand 0 vorliegt, nicht jedoch, ob ein anderer Zustand mit der Zustandsbezeichnung 0 und den daraus folgenden Zustandsübergängen nicht eher auf-

Algorithmus 3.23: Zustandsisomorphietest mit mehreren Start- oder Terminalzuständen

```
1  function SortedStates(η)
2      foreach i ∈ 𝕊 ∪ {|𝕊|} do
3          t[i] := i ∈ {0, 1, |𝔼\𝔽|, |𝔼|, |𝔼 ∪ (𝕊\𝔽)|}
4      for i := 0 to |𝕊| − 1 do
5          for j := 0 to |𝕏| − 1 do
6              if t[s'_{i,χ(j)}] then
7                  if ¬t[s'_{i,χ(j)} + 1] then
8                      t[i] := true
9                      t[s'_{i,χ(j)} + 1] := true
10             else
11                 if η then
12                     h := min{h, (i, j, |𝕊|)}
13                 return false
14     return true
```

gezählt worden wäre. Daher erhält bei mehreren potentiellen Startzuständen jeder dieser Zustände testweise die Position 0. Von da aus erfolgt die Berechnung der Zustandsübergänge nach obigem Muster. Ist der so entstandene Automat bereits vorher aufgezählt worden, liegt eine Zustandspermutation bezüglich des Startzustandes vor. Der Automat aus Abbildung 3.6 hat mit $\mathbb{E} = \mathbb{S}$ und $\mathbb{F} = \varnothing$ als besten Startzustand die ursprüngliche Nummer 3, wie in Abbildung 3.18 zu sehen ist.

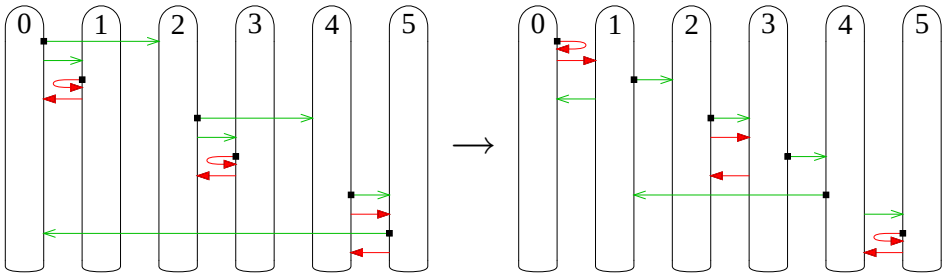


Abbildung 3.18: Bezüglich Zustand 0 optimierter Automat aus Abbildung 3.6

Für einen Vergleich ist als erstes die Zustandspermutation σ zu erstellen. Der erste, ausgewählte Startzustand s erhält die neue Nummer 0, es ist dann also $\sigma\colon s \mapsto 0$. Ein Zustandsübergang sei innerhalb der nicht-terminierenden Startzustände, dann erhält dieser die nächste Nummer 1, und so weiter. Diese Fortzählung übernimmt der Zähler k.

Startzustände ohne und mit gleichzeitigem Terminalzustand sind allerdings aufgrund ihrer Zuordnung der Zustandsnummern unterschiedlich zu behandeln. Genauso verhält es sich auch mit den anderen Zustandsgruppen. Auch hier gibt es unterschiedliche Nummernbereiche. Allerdings ist ein Zustand aus einem Bereich immer in diesem Bereich. Ein permutierter Zustand ist also in der gleichen Teilmenge wie der nicht-permutierte Zustand. Von daher muss es je Bereich unterschiedliche Zähler k geben, die, um einen Index ergänzt, eine Unterscheidung ermöglichen.

Zu sehen ist dies im Algorithmus 3.24, der den Bereich für den im Parameter gegebenen, nicht-permutierten Zustand i durch Mengenzugehörigkeit bestimmt und in g

speichert. Die Permutation erhält dann für einen weiteren Zustand i einen neuen Wert, abhängig von dem Bereich g. Demzufolge erhält auch der Zähler k_g einen inkrementierten Wert. Eine Gültigkeitsprüfung ist nicht erforderlich, da die Funktion nur für unterschiedliche $i \in \mathbb{S}$ aufgerufen werden soll und somit der Wert von k_g nie über den eigenen Bereich hinaus Verwendung findet. Gibt es insgesamt nur einen Bereich, so kann die Ermittlung von g entfallen; der Zähler k findet dann ohne Index Verwendung.

Algorithmus 3.24: Setzen einer neuen Permutation innerhalb eines Bereiches

```
1 function SetScopePermutation(i, k)
2     g := { 0  wenn i ∈ 𝔼\𝔽
           { 1  wenn i ∈ 𝔼∩𝔽
           { 2  wenn i ∈ 𝕊\(𝔼∪𝔽)
           { 3  wenn i ∈ 𝔽\𝔼
3     σ_i := k_g
4     k_g := k_g + 1
```

Die Initialisierung von k erfolgt mit gleichen Werten wie beim Zustandsisomorphietest in Abbildung 3.17: jeweils der minimale Zustandswert des jeweiligen Bereiches. Statt nun den ersten Startzustand über die Funktion „SetScopePermutation“ zu setzen, kann σ_0 den entsprechenden Wert s und k_0 den initialen Wert 1 erhalten.

Um für eine Überprüfung die Eingangswertpermutation zu berücksichtigen, erhält $\dot{\mathbf{s}}'$ eine nur mit dieser Permutation versehene Kopie. So ist sichergestellt, dass die neuen Zustände ihre Nummerierung in der richtigen Reihenfolge erhalten. Im Algorithmus 3.25 ist dies in den Zeilen 3 bis 5 zu sehen.

Der Aufbau der Permutation erfolgt über den Schleifenzähler $\ddot{\imath}$, der die permutierten Zustandswerte enthält. Da jedoch die Zustandsübergänge noch nicht in permutierter Form vorliegen, enthält zusätzlich i die nicht-permutierte Variante, ermittelt über die Umkehrpermutation $\sigma^{-1}(\ddot{\imath})$. Ist das Ziel eines Zustandsübergangs noch nicht in der Permutation enthalten, erfolgt ein Aufruf der Funktion „SetScopePermutation“ zum Setzen des nächstmöglichen Zählerwertes. Erst danach kann der mit einer Zustandspermutation versehene Zustandsübergangswert in $\mathbf{s}'$ einen Wert erhalten. Dies geschieht im üblichen Ablauf, in der äußeren Schleife die Zustandswerte, in der inneren die Eingangswerte, jeweils in aufsteigender Reihenfolge.

Ein Problem ergibt sich bei nicht zusammenhängenden Automaten, wenn kein Zustandsübergang zu einem weiteren Zustand führt, obwohl die Permutation noch nicht vollständig erstellt ist. Ein Beispiel soll die Problematik verdeutlichen. Sei $\mathbb{E} = \mathbb{S}$ und für den aktuell betrachteten Automat gilt $\forall i \in \mathbb{S},\ j \in \mathbb{X}\colon\ s'_{i,j} = i$. Für $s = 1$ und $\ddot{\imath} = 1$ gibt es keinen festgelegten Zustand, alle außer Zustand 1 sind mögliche Kandidaten. Dies sind $|\mathbb{S}| - 2$ Zustände. Ein vollständiger Test hätte $(|\mathbb{S}| - 2)!$ Möglichkeiten für den nachfolgenden Zustand zur Folge. Da der erste Startzustand ebenso durchprobiert wird, bei denen sich das gleiche Problem fortsetzt, ergeben sich insgesamt $(|\mathbb{S}| - 1)!$ Versuche, also genauso viele wie mit Algorithmus 3.9. Es gäbe dann mit diesem Verfahren keine Vorteile.

Allgemein betrachtet ist mit nur einem Startzustand die Bedingung Vereinfacht ausreichend – jeder Zustand ist dann von Zustand 0 ausgehend erreichbar, ein Rückweg ist nicht erforderlich. Anders ist dies bei Automaten mit mehreren Startzuständen und einem Präfix, so dass der Zustand 0 nicht erreichbar ist, sofern sich ein Startzustand außerhalb

Algorithmus 3.25: Permutationstest eines anderen Startzustands als ersten Zustand

```
1  function ImprovableFirstState
2      r := false
3      foreach i ∈ 𝕊 do
4          foreach j ∈ 𝕏 do
5              ṡ'_{i,χ(j)} := s'_{i,j}
6      foreach s ∈ 𝔼\(𝔽 ∪ {0}) do
7          σ := ( 0  1  ⋯  |𝕊|−1 )  mit ε ∉ 𝕊
                ( ε  ε  ⋯    ε   )
8          σ_s := 0 // Startzustand s an Position 0 setzen
9          k_0, k_1, k_2, k_3 := 1, |𝔼\𝔽|, |𝔼|, |𝔼 ∪ (𝕊\𝔽)|
10         for ï := 0 to |𝕊| − 1 do
11             if σ^{-1}(ï) ∉ 𝕊 then // fehlender Zustandsübergang ≡ neuer Zyklus
12                 for i := 0 to |𝕊| − 1 do
13                     if σ(i) ∉ 𝕊 then
14                         SetScopePermutation(i, k)
15                         endloop
16             i := σ^{-1}(ï)
17             for j := 0 to |𝕏| − 1 do
18                 if σ(ṡ'_{i,j}) ∉ 𝕊 then
19                     SetScopePermutation(ṡ'_{i,j}, k)
20                 s'_{i,j} := σ(ṡ'_{i,j})
21         r := r ∨ PermutationTest
22     return r
```

des Präfix befindet; eine durchgehende Nummerierung aller Zustände ist dann, wie eben beschrieben, nicht mehr möglich und die Nummerierung muss bei einem weiteren auszuwählenden Zustand beginnen. Um eine bessere Konstellation auszuschließen, sind alle weiteren Permutationen zu überprüfen. Dies bringt aber keinen Vorteil gegenüber der Überprüfung mit allen Permutationen. Von daher empfiehlt sich dieses Verfahren nur für stark verbundene Automaten.

Von daher muss „Vereinfacht“ vorausgesetzt werden, damit eine Testreduktion möglich wird. Es gibt die Möglichkeit, den Test wegen fehlender anderer Bedingungen abzubrechen oder, wenn die Bedingung aus Abschnitt 3.4.4 eingehalten werden soll, einen beliebigen Vertreter zu bestimmen, so dass der Automat ggf. mit einem anderen Test ausgeschlossen werden kann.

Der fehlende Isomorphiewert $\ddot{\imath}$ erhält den kleinsten Wert i aus σ. Dies entspricht dem der notwendigen Gruppe, da alle anderen Werte zuvor spätestens bereits mit dem Schleifenzähler $\ddot{\imath}$ gesetzt wurden. Demnach ist auch $\ddot{\imath} = k_g$. Diese Konfliktlösung erfolgt in Zeile 11, wenn Zustand $\ddot{\imath}$ nicht erreichbar ist, sei es durch Präfix oder prinzipiell durch keinen anderen Zustand.

Die eigentliche Permutationskontrolle fehlt noch. Da bisher nur die Zustandspermutation feststeht, bietet der Algorithmus 3.4 zum Vergleich zweier Automaten nur unzureichend Hilfe. Die Zeilen 2 bis 5 lassen sich aber übertragen.

Hinzu kommt der Wert für h. Wenn der Automat mit permutiertem ersten Startzustand eine bessere Konstellation aufweist, muss ein nächster ausgewählter Automat nicht

mehr dieses Optimum erfüllen. Da dies nicht unbedingt direkt an den Zustandsübergängen des Startzustandes, sondern auch in Folge der Neunummerierung liegen kann, ist der Zustand des permutierten Automaten zu suchen, der im bisherigen Automaten den größten Zustandswert hat. Eine Änderung bewirkt dann weitere Unterschiede bei den Zustandsübergängen, so dass in Konsequenz der Startzustand s nicht mehr optimal dasteht. Dieser maximale Zustand ist mit m bezeichnet und erhält in Zeile 4 des Algorithmus 3.26 seinen Wert. Das Sprungziel für h, gesetzt in Zeile 7, ist durch das Tupel $(m, |\mathbb{X}|-1, s'_{m,|\mathbb{X}|-1}+1)$ bestimmt. Die Kontrolle erfolgt auch für alle anderen Startzustände als Wert in s, damit sich insgesamt ein minimales h ergibt.

Algorithmus 3.26: Permutationskontrolle zwischen s' und $\mathbf{s}'$ mit Unterstützung von σ

```
1 function PermutationTest
2     m := σ^{-1}(0)
3     for i := 0 to |𝕊| − 1 do
4         m := max{m, σ^{-1}(i)}
5         for j := 0 to |𝕏| − 1 do
6             if s'_{i,j} ≠ s'_{i,j} then
7                 h := min{h, (m, |𝕏| − 1, s'_{m,|𝕏|−1} + 1)}
8                 return s'_{i,j} > s'_{i,j}
9     return OutputPermutationTest
```

Verbleiben noch die Startzustände, die gleichzeitig Terminalzustände sind. Diese können nicht an erster Stelle stehen, wenn es nicht-terminierende Startzustände gibt. Von daher ist ein abgewandeltes Verfahren notwendig. Zur einfacheren Dokumentation seien im Folgenden sowohl $|\mathbb{E}\backslash\mathbb{F}| \geq 1$ als auch $|\mathbb{E}\cap\mathbb{F}| \geq 1$, für die Algorithmusdurchführung ist dies nicht erforderlich.

Die nichtterminierenden Zustände haben sich mit den anderen Überprüfungen als ideal angeordnet erwiesen. Von daher besteht der erste Bereich $\mathbb{E}\backslash\mathbb{F}$ der Permutation aus einer identischen Abbildung. Deren Zustandsübergänge auf Zustände außerhalb dieses Bereiches lassen sich ebenso in der Permutation initial zuweisen. Für diese gilt natürlich rekursiv das gleiche. Sind soweit alle möglichen Zustandspermutationen identischer Abbildung aufgrund der Zustandsübergänge angelegt, erfolgt die Überprüfung der verbliebenen terminierenden Startzustände $\mathbb{E}\cap\mathbb{F}$ nach dem gleichen Prinzip wie mit Algorithmus 3.25. Die bis dahin erstelle Permutation kann als Basis wiederverwendet werden, genauso wie auch der Zähler k. Daher sind diese mit einem Punkt markiert in Kopie als $\dot{\sigma}$ und $\dot{k}$ angelegt. Algorithmus 3.27 zeigt das ganze Prozedere.

Die alternative Methode, anhand logischer Vergleiche den optimalen Startzustand zu ermitteln, liefert keine vollständigen Ergebnisse, wie Experimente gezeigt haben. Als Kriterien für einen besten Startzustand an Stelle 0 galten dabei die Anzahl der Referenzen auf einen Zustand, um insgesamt einen niedrigen Zählerstand zu erreichen, sowie die Anzahl der Zustandsübergänge auf den eigenen Zustand oder, wenn ein vorheriger Zyklus existiert, die Anzahl der Übergänge auf die vorherigen Zustände, die ja einen niedrigeren Zahlenwert haben. Falls dann noch immer gleichwertige Kandidaten vorhanden sein sollten, mussten die Ausgabewerte eine Entscheidung herbeiführen. Bereits bei $|\mathbb{S}| = 5$ mit $|\mathbb{X}| = 1$ gab es Konstellationen, die jeweils einer speziellen Regel bedurft hätten, um Duplikate zu verhindern. Diese Problemfälle zeigt Abbildung 3.19.

Algorithmus 3.27: Permutationstest für terminierende Startzustände

```
1  function ImprovableTerminalStartState
2      foreach i ∈ 𝕊 do
3          foreach j ∈ 𝕏 do
4              ṡ'_{i,χ(j)} := s'_{i,j}
5      σ := ( 0  1  ⋯  |𝔼\𝔽|−1  |𝔼\𝔽|  ⋯  |𝕊|−1 )
            ( 0  1  ⋯  |𝔼\𝔽|−1     ε     ⋯     ε  )  mit ε ∉ 𝕊
6      k̇0, k̇1, k̇2, k̇3 := |𝔼\𝔽|, |𝔼\𝔽|, |𝔼|, |𝔼 ∪ (𝕊\𝔽)|
7      foreach ï ∈ 𝕊 do
8          if σ^{-1}(ï) ∈ 𝕊 then
9              i := σ^{-1}(ï)
10             foreach j ∈ 𝕏 do
11                 if σ(ṡ'_{i,j}) ∉ 𝕊 then
12                     SetScopePermutation(ṡ'_{i,j}, k̇)
13                 s'_{i,j} := σ(ṡ'_{i,j})
14     if k̇1 = |𝔼| then
15         return false
16     σ̇ := σ
17     r := false
18     foreach s ∈ (𝔼 ∩ 𝔽)\{0} do
19         σ, k := σ̇, k̇
20         if σ(s) ∉ 𝕊 then
21             σ_s := k1
22             k1 := k1 + 1
23             for ï := 0 to |𝕊| − 1 do
24                 if σ^{-1}(ï) ∉ 𝕊 then
25                     for i := 0 to |𝕊| − 1 do
26                         if σ(i) ∉ 𝕊 then
27                             SetScopePermutation(i, k)
28                             endloop
29                 i := σ^{-1}(ï)
30                 for j := 0 to |𝕏| − 1 do
31                     if σ(ṡ'_{i,j}) ∉ 𝕊 then
32                         SetScopePermutation(ṡ'_{i,j}, k)
33                     s'_{i,j} := σ(ṡ'_{i,j})
34             r := r ∨ PermutationTest
35     return r
```

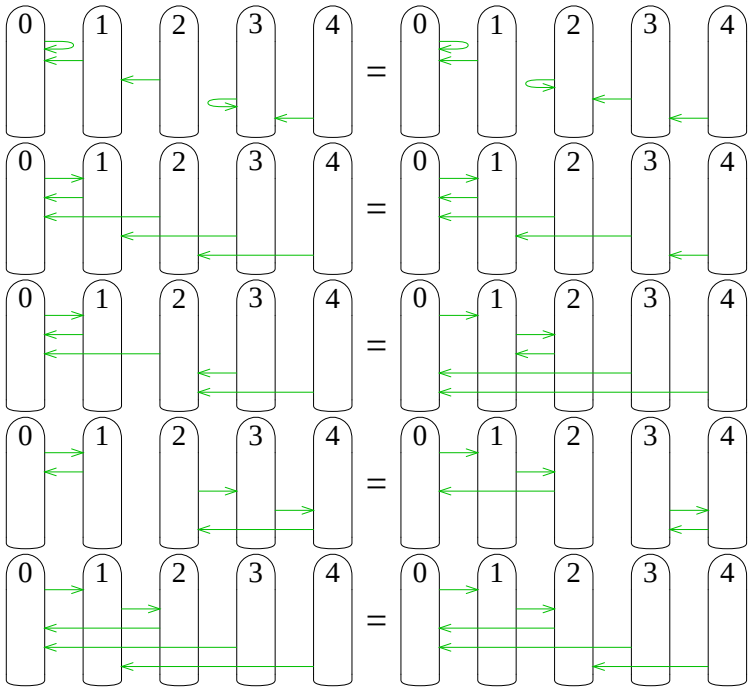


Abbildung 3.19: Problemfälle bei Startzustandsverifikation

3.4.5.4 Eingangswertpermutation

Soll des weiteren auch noch eine Eingangswertisomorphie gelten, so ist jede Permutation zu testen. Anders als bei den Zuständen gibt es keine direkte Relation durch Zustandsübergänge, aus der sich eine Reihenfolge ableiten lässt. Bei den Ausgaben ist eine Grundordnung durch die Zustände gegeben, aufgrund derer die Werte im Automatenzähler anpassbar sind. Die Eingangswerte sind aber nicht im Zähler aufgeführt. Aufgrund dessen kann es keine Sortierregel geben.

Unterliegen alleinig die Eingangswerte einer Permutation, kann eine Überprüfung mit den bisherigen Methoden aus Abschnitt 3.2 erfolgen, allerdings erhält h dort keinen neuen Wert. Dies ist aber durchaus sinnvoll, wenn sich für die Zustandsübergänge eine bessere Konstellation durch eine nicht-identische Abbildung der Eingangswerte ergibt. Eine Angabe ist aber nur für den Zustand aufgrund des Schleifenzählers i möglich, da bereits eine Änderung des Zustandsübergangs bei jedem Eingangswert die ursprüngliche Bedingung nicht mehr erfüllt und so, zumindest für diesen Zustand i, keine gleich geartete Eingangswertpermutation mehr vorliegt. In Algorithmus 3.28 ist diese Variante dargestellt, angelehnt an den Algorithmus 3.4.

3.4.5.5 Ausgabewertpermutation

An dritter Stelle steht die Ausgabewertisomorphie. Da Zustands- und Eingabewertisomorphie aufgrund der Aufzählungsreihenfolge durch Algorithmus 3.3 höher gewichtet sind, erfolgt ein Vergleich zwischen permutierten und nicht-permutierten Automaten lediglich bei identischen Zustandsübergängen aller Zustände und Eingangskonstellationen. Andernfalls hat sich bereits ein Unterschied gezeigt, so dass die Ausgabewerte keine Relevanz haben.

Gilt es, nur die Ausgabewertpermutation zu überprüfen, so lässt sich eine Regel erkennen: Ausgaben mit niedrigem Wert sollten zuerst und ggf. häufiger auftreten als Aus-

Algorithmus 3.28: Test der Eingangswertpermutation als alleinige Permutation

```
1  function ImprovableWithInputPermutation
2      InitPermutation(|𝕏| − 1, χ)
3      while NextPermutation(|𝕏| − 1, χ) do
4          for i := 0 to |𝕊| − 1 do
5              for j := 0 to |𝕏| − 1 do
6                  if s'_{i,j} ≠ s'_{i,χ(j)} then
7                      if s'_{i,j} > s'_{i,χ(j)} then
8                          h := min{h, (i, |𝕏| − 1, s'_{i,|𝕏|−1} + 1)}
9                          return true
10                     else
11                         return false
12         for i := 0 to |𝕊| − 1 do
13             for j := 0 to |𝕏| − 1 do
14                 if y_{i,j} ≠ y_{i,χ(j)} then
15                     return y_{i,j} > y_{i,χ(j)}
16     return false
```

gaben mit hohem Wert. Ist die Situation umgekehrt, so liegt eine Permutation der Ausgabewerte vor, die nachfolgend in der Aufzählungsliste erscheint und damit kein Kandidat für die gesuchten Automaten ist.

Algorithmus 3.29: Permutationstest der Ausgabe

```
1  function OutputPermutationTest
2      foreach i ∈ 𝕊 do
3          foreach j ∈ 𝕏 do
4              y_{σ(i),χ(j)} := y_{i,j}
5      υ := ( 0   1   ···   |𝕐| − 1 )  mit ε ∉ 𝕐
            ( ε   ε   ···      ε    )
6      k := 0
7      for i := 0 to |𝕊| − 1 do
8          for j := 0 to |𝕏| − 1 do
9              if υ(y_{i,j}) ∉ 𝕐 then
10                 υ_{y_{i,j}} := k
11                 k := k + 1
12             y_{i,j} := υ(y_{i,j})
13     for i := 0 to |𝕊| − 1 do
14         for j := 0 to |𝕏| − 1 do
15             if y_{i,j} ≠ y_{i,j} then
16                 return y_{i,j} > y_{i,j}
17     return false
```

Für das Zusammenspiel mit den anderen Permutationsmöglichkeiten müssen die zueinander gehörenden Ausgaben verglichen werden – mit und ohne Permutation von Zustand und Eingang. Um nicht alle Ausgabewertpermutationen überprüfen zu müssen, können nach obiger Regel die Ausgabewerte sortiert werden. Im Programmbeispiel des

Algorithmus 3.29 zur Überprüfung, ob eine Permutation der Zustände mittels σ und der Eingangswerte mittels χ bereits als Kandidat aufgeführt ist, sind die Zeilen 5 bis 12 ausschließlich für die Bestimmung bei Ausgabewertpermutation notwendig und können gegebenenfalls ersatzlos entfallen, wenn die Ausgabewerte nicht isomorph vorliegen. In Zeile 2 bis 4 werden die beiden Permutationen σ und χ verwendet, falls diese Permutationen ungleich der identischen Abbildung sein sollten, also ohne eine Vertauschung.

Der Aufbau der Permutation υ verwendet für die höchstpriorisierteste Stelle den kleinsten Wert, $\upsilon(y_{0,0})$ ist also immer gleich 0. Erscheint im Ablauf ein anderer Ausgabewert, erhält dieser den nächsten Ausgabewert, bestimmt durch den Zähler k. So kann es bei vollendeter Permutation vorkommen, dass nicht alle Werte gesetzt sind, wenn einige Ausgabewerte keinen Gebrauch gefunden haben. Dies behindert aber nicht die Überprüfung, die analog zu den Zeilen 6 bis 9 des Algorithmus 3.4 fungiert.

3.4.5.6 Zusammensetzung der Permutationen

Da die einzelnen Permutationen nur im Zusammenhang überprüfbar sind, wie sich bereits mit dem Beispiel in Abbildung 3.10 gezeigt hat, erfordert es eine spezifische Kombination der Einzelalgorithmen. Die Eingangswertpermutation ist die einzige, die nicht optimierbar ist. Von daher bildet diese die äußere Schleife. Bei der Zustandspermutation für den Startzustand hat sich gezeigt, dass erst danach bei gesetzter Zustandspermutation σ die Ausgabewertpermutation erfolgt. Von daher ist auch hier die Reihenfolge festgelegt, die in Algorithmus 3.30 zusammengetragen ist, mit der Funktion „ImprovableFirstState“ wird σ für den eigentlichen Automaten gesetzt.

Algorithmus 3.30: Kombination der einzelnen Permutationen

```
1  function PermutationRepresentative
2      InitPermutation(|𝕏| − 1, χ)
3      if ¬SortedStates(true) then
4          return false
5      if ImprovableFirstState ∨ ImprovableTerminalStartState then
6          return false
7      InitPermutation(|𝕊| − 1, σ)
8      while NextPermutation(|𝕏| − 1, χ) do
9          if SortedStates(false) then
10             return false
11         if (∀i ∈ 𝕊, j ∈ 𝕏: s'_{i,j} = s'_{i,χ(j)}) ∧ OutputPermutationTest then
12             return false
13     return true
```

Wie schon für Algorithmus 3.21 oder den jeweiligen Permutationen selbst beschrieben, können die einzelnen Permutationen entfallen, wenn an deren Stelle ihre identische Abbildung tritt. Die notwendigen Nebenbedingungen für eine reduzierte Permutationsprüfung sind in Tabelle 3.3 aufgeführt.

Tabelle 3.3: Nebenbedingung für die einfachere Permutationsprüfung

σ	**notwendige Bedingung**
$\lvert\mathbb{E}\rvert < \lvert\mathbb{S}\rvert$	vereinfacht oder
	einheitliche Werte ($= 0$) der nicht erreichten Zustände
$\mathbb{E} = \mathbb{S}$	stark zusammenhängend (vereinfacht und präfixfrei)

3.4.6 Reduktion

Das Aussortieren von bedeutungsäquivalenten Zuständen ist bereits im Abschnitt 3.2.3 ab Seite 35 umfassend behandelt. Eine weitere Vereinfachung des Algorithmus 3.7 ist nicht mehr möglich, lediglich die Zeilen 24 bis 30 entfallen, wie bereits beschrieben.

Ein Überspringen von Automaten aufgrund der Konstellation von Zustandsübergängen ist nicht möglich, da bereits die geänderte niedrigste Zählerstelle $y_{|\mathbb{S}|-1,|\mathbb{Y}|-1}$ zu einem nächsten erfolgreichen Automaten führen kann. Als Anfangspunkt für einen kürzeren Sprung über mehrere Ausgabewerte hinweg bei gleicher Zustandsübergangskonstellation interessiert der höchste reduzierbare Zustand $j = \max\{\, j \in \mathbb{S} \mid e[i,j]\}$.

Dessen Zustandsübergänge verweisen im Vergleich mit dem anderen identischen Zustand entweder auf gleiche Zustände oder auf gleiche Zustandspaare, dessen beteiligter Zustand jedoch nicht größer als j sein kann. Somit verbleibt lediglich die Erhöhung des Ausgabewertes von Zustand j, um einen Unterschied zu dem zuvor identischen Zustand herzustellen. Um dieses Ziel zu erreichen, erhalten alle Ausgabewerte der Zustände größer j ihren Maximalwert, damit der nächste aufgezählte Automaten die ursprüngliche Bedingung nicht mehr verletzt. In Analogie zur Bestimmung des Tupels h zum Überspringen irrelevanter Zustandsübergänge, wie im Abschnitt 3.4.1 ab Seite 46 beschrieben, speichert h_y den gefundenen Zustand j.

Algorithmus 3.31 zeigt die Analyse zum gleichzeitigen Setzen von h_y alternativ zu der mit Algorithmus 3.7 auf Seite 37 durchgeführten Phase 3. Schließlich führt der Algorithmus 3.32 den Sprung mit h_y durch, sofern die anderen Analysen keinen gültigen Wert für h ergeben haben. Daher ist dies als Ergänzung zu Algorithmus 3.12, Seite 47 aufgeschrieben, der auch die Sprungausführung für h enthält. Der Erfolg lässt sich aus der Tabelle 3.4 ablesen.

Algorithmus 3.31: Berechnung von h_y als Ersatz ab Zeile 20 des Algorithmus 3.7

```
20    for j := |𝕊| − 1 to 1 do
21        foreach i ∈ {0, 1, ..., j − 1} do
22            if e[i, j] then
23                h_y := j
24                return true
25    return false
```

Algorithmus 3.32: h_y anwenden, ergänzend zu Algorithmus 3.12 vor Zeile 19

```
19        else if h_y ≠ 0 then
20            foreach i ∈ 𝕊\{0, 1, ..., h_y} do
21                foreach j ∈ 𝕏 do
22                    y_{i,j} := |𝕐| − 1
23        h_y := 0 // oder der Zeile 3 hinzufügen
```

Tabelle 3.4: Erfolg von h_y mit den Kriterien arrangiert, reduziert, vereinfacht und präfixfrei

	Anzahl der erfolglosen Tests		
$\lvert\mathbb{S}\rvert$	**nur mit** h	**mit** h **und** h_y	**Reduzierung**
2	37	37	0,0 %
3	1 357	1 214	10,5 %
4	95 921	70 527	26,5 %
5	10 117 265	6 242 743	38,3 %
6	1 420 495 677	751 539 239	47,1 %

3.4.7 Berücksichtigung der Semantik der Ausgabe

Das Ausgabeverhalten eines Automaten ist ebenfalls von Relevanz. Allerdings hängt dies wesentlich vom Kontext ab. Daher wird als Beispiel wie schon für Abbildung 3.6 ein Automat verwendet, der zwei Eingangswerte ($|\mathbb{X}| = 2$) und (mindestens) zwei Ausgabewerte ($|\mathbb{Y}| \geq 2$) hat.

Ziel des Automaten ist es, möglichst oft den Wert $x = 1$ zu erhalten. Allerdings muss für $x = 1$ die Ausgabe y alternierend sein. Angelehnt an die Darstellung in Abbildung 3.1 hat der Automat in der Umwelt eine Richtung gespeichert, die über den Aktor als Drehung verarbeitet wird; y gibt also eine Drehrichtung an. Weiterhin gibt es in dem Modell nur vier Richtungen, die mit den vier Himmelsrichtungen Norden, Osten, Süden und Westen bezeichnet werden können.

Bei $x = 0$ ist die Eingabe $x = 1$ zu erreichen. Dazu muss oft in die gleiche Richtung gedreht werden, um alle vier möglichen Richtungen zu testen. Bei einem effizienten Algorithmus sollen z. B. nach maximal fünf Schritten alle Richtungen getestet worden sein – zwei Schritte mehr, als minimal benötigt werden. Ein einfaches Verfahren der Überprüfung ist mit Algorithmus 3.33 umgesetzt, bei dem diese fünf Schritte durch ausprobieren getestet werden.

Algorithmus 3.33: Gleich bleibender Ausgabezyklus für $x = 0$

```
1  function OutputSemantic0
2      foreach i ∈ 𝕊 do
3          s := i
4          d := 0
5          ∀k ∈ {0,1,2,3}: a_k := false
6          for j := 0 to 4 do
7              d := rotate(d, y_{s,0})   (mod 4)
8              a_d := true
9              s := s'_{s,0}
10         if ∃k ∈ {1,2,3}: ¬a_k then
11             return false
12     return true
```

Für $x = 1$ ist es nicht relevant, in welcher Reihenfolge die Ausgaben y erfolgen. Von daher kann eine maximale Zykluslänge nicht vorgegeben werden, so dass ein anderer Test notwendig ist. Als erstes wird ein Zyklus ermittelt. Analog zu Algorithmus 3.8 zur Bestimmung der reflexiven transitiven Hülle wird die initiale Matrix O nur für Verbindungen mit $x > 0$ durchgeführt. Danach kann im zweiten Schritt ermittelt werden, ob für jeden Zyklus die notwendigen Ausgabemöglichkeiten erfolgen. Mit dem in Algorithmus 3.34 dargestellten Verfahren ist es sogar möglich, auch $|\mathbb{X}| \geq 2$ zu überprüfen.

Algorithmus 3.34: Veränderliche Ausgabezyklen für $x > 0$

```
1 function OutputSemantic1
2     ∀i, j ∈ 𝕊: (Verbindung von i nach j für x > 0 ∨ i = j) ⇔ O[i, j] = true
3     foreach j ∈ 𝕊 do
4         foreach i ∈ 𝕊\{j} do
5             if O[i, j] then
6                 foreach k ∈ 𝕊 do
7                     if O[j,k] then
8                         O[i,k] := true
9     return ∀i ∈ 𝕊: {0,1} ⊆ {v ∈ 𝕐 | ∃j ∈ 𝕊, k ∈ 𝕏\{0}: O[i, j] ∧ y_{j,k} = v}
```

Auf Automaten mit zwei Zuständen angewendet ergibt sich, dass für die 108 von 256 relevanten Automaten (normiert, alle Zustände genutzt, ohne Präfix, nicht reduzierbar) nur 28 Automaten den Kriterien der Algorithmen 3.33 und 3.34 entsprechen. Es werden also 80 Automaten (74 %) zusätzlich gefiltert.

Wichtig ist dabei, dass man als „Versuchsleiter“ nicht zu viele Einschränkungen vornimmt. Es ist wichtig, die einschränkenden Regeln genau aufzustellen. Andernfalls könnten interessante Algorithmen fälschlicherweise aussortiert werden.

3.4.8 Nutzung aller Eingangsmöglichkeiten

Neben den Ausgaben sind auch die Eingaben zu betrachten. Ein Eingangswert ist genutzt, wenn kein anderer Eingangswert die gleichen Ausgaben und die gleichen Zustandswechsel aufweist. Dies lässt sich mit Algorithmus 3.35 überprüfen. Es kann allerdings auch sein, dass spezielle Werte gleich sein dürfen, wenn z. B. die Bits der Binärdarstellung relevant sind. Dann ist eine Abwandlung des Algorithmus erforderlich.

Algorithmus 3.35: Nutzung aller Eingänge

```
1 function InputSemantic
2     foreach i ∈ 𝕏\{|𝕏| − 1} do
3         if ∃j ∈ 𝕏\{0, 1, ..., i}: ∀k ∈ 𝕊: y_{k,i} = y_{k,j} ∧ s'_{k,i} = s'_{k,j} then
4             return false
5     return true
```

Mit dem im vorigen Abschnitt 3.4.7 betrachteten Fall für $|\mathbb{X}| = 2$ ist eine solche Überprüfung allerdings nicht notwendig, da bereits die Kombination der Algorithmen 3.33 und 3.34 ergibt, dass der Eingang berücksichtigt werden muss.

3.4.9 Moore-Automaten

Sind statt Mealy- nur Moore-Automaten aufzuzählen, vereinfacht sich die Aufzählung, nicht aber die Anzahl der Überprüfungen. Die Ausgaben sind dann nur noch vom Zustand und nicht mehr vom Eingangswert abhängig. Von daher verkürzen sich die Zählerstellen in Abbildung 3.5 von ursprünglich $|\mathbb{S}| \cdot |\mathbb{X}|$ auf $|\mathbb{X}|$ Werte für die Ausgabe y. Um dies zu verdeutlichen, besteht der Index nur aus dem aktuellen Zustand, z. B. y_s statt $y_{s,x}$.

Automaten mit funktionsgleichen Zuständen kann es ebenso geben, so dass die Reduktion aus Abschnitt 3.4.6 durchzuführen ist. Die Bedingung in Zeile 5 des Algorithmus 3.7 vereinfacht sich allerdings zu $y_i = y_j$.

Bei der Isomorphieüberprüfung, z. B. mit Algorithmus 3.9, ändert sich nur der Umfang der Tests in Zeile 27, da der Index entfällt. Die Permutation der Eingangswerte wird noch für die Zustandsübergänge benötigt. Gleiches gilt für die Zeilen 3, 8 und 14 des Algorithmus 3.29.

Um nur eine provisorische Umsetzung vorzunehmen, kann sich auch die Aufzählung durch Algorithmus 3.3 ändern. Statt einem Überlauf über verschiedene Eingangswerte vorzunehmen, erfolgt die Inkrementierung nur für den Eingangswert 0, für die anderen wird der Wert kopiert. Algorithmus 3.36 zeigt dies als Ersatz für die Zeilen 2 bis 8. Dadurch bleibt dann insbesondere die Ordinalzahlbestimmung durch Algorithmus 3.5 gleich, genauso wie auch der Vergleich zweier Automaten z. B. durch Algorithmus 3.4.

Algorithmus 3.36: Aufzählung der Ausgabewerte für Moore-Automaten als Ersatz für die Zeilen 2 bis 8 des Algorithmus 3.3, Seite 27

```
2   for i := |𝕊| − 1 to 0 do
3       if y_{i,0} < |𝕐| − 1 then
4           y_{i,0} := y_{i,0} + 1
5           foreach j ∈ 𝕏\{0} do
6               y_{i,j} := y_{i,0}
7           return true
8       else
9           foreach j ∈ 𝕏 do
10              y_{i,j} := 0
```

3.4.10 Zusammenspiel der Überprüfungen

Vom Prinzip her sind alle Überprüfungsverfahren miteinander kombinierbar, mit Ausnahme von Normiert und Isomorphie, die eine unterschiedliche Priorisierung von Eingangswerten und Zustandsabfolge vornehmen und sich daher gegenseitig ausschließen. Dies betrifft auch die Eigenschaft „arrangiert“ aus Satz 3.6 auf Seite 57, die eine Überprüfung der Isomorphie vornimmt.

Für den Fall, dass genau ein Startzustand und keine Terminalzustände vorliegen, also $\mathbb{E} = \{0\}$ und $\mathbb{F} = \varnothing$, stellt sich die Frage, ob normiert oder arrangiert als Kriterium dienen soll. Das Untersuchungsergebnis streng verbundener, nicht reduzierbarer Automaten für $|\mathbb{X}| = |\mathbb{Y}| = 2$ ist in Tabelle 3.5 zu sehen. In den Spalten „erfolglos N“ und „erfolglos A“ ist die Anzahl der Automaten für normiert und arrangiert aufgeführt, die während der Aufzählung überprüft wurden, aber nicht alle Kriterien erfüllt haben. Neben der Zustandsanordnung kamen die Kriterien reduziert, vereinfacht und präfixfrei zum Einsatz. Das Ergebnis spricht für die Normierung im Gegensatz zur universelleren Isomorphie bzw. der arrangierten Zustandsübergänge.

Zusätzlich zeigen die Werte der Tabelle 3.5 den Unterschied zwischen gefundenen zu möglichen Automaten bei steigender Zustandsanzahl $|\mathbb{S}|$ auf. Abbildung 3.21 illustriert dieses Verhältnis auf einer logarithmischen Skala.

Da mit *h* immer das Sprungziel berechnet wird, das die Verletzung eines bestimmten Kriteriums auflöst, weist ein kürzerer Sprung nur auf nicht-relevante Automaten. Eine

Tabelle 3.5: Anzahl gefundener, interessanter Automaten für $|\mathbb{X}| = |\mathbb{Y}| = 2$ mit den Kriterien normiert bzw. arrangiert, reduziert, vereinfacht und präfixfrei gegenüber den dabei erfolglos getesteten Automaten

$\|\mathbb{S}\|$	$(2 \cdot \|\mathbb{S}\|)^{2 \cdot \|\mathbb{S}\|}$	**gefunden**	**erfolglos N**	**erfolglos A**
1	4	4	0	0
2	256	108	37	37
3	46 656	8 124	1 356	1 357
4	16 777 216	798 384	95 922	95 921
5	10 000 000 000	98 869 740	10 117 247	10 117 265
6	8 916 100 448 256	14 762 149 668	1 420 495 700	1 420 495 677
7	11 112 006 825 558 016	2 581 401 130 308	241 972 998 408	241 973 005 093

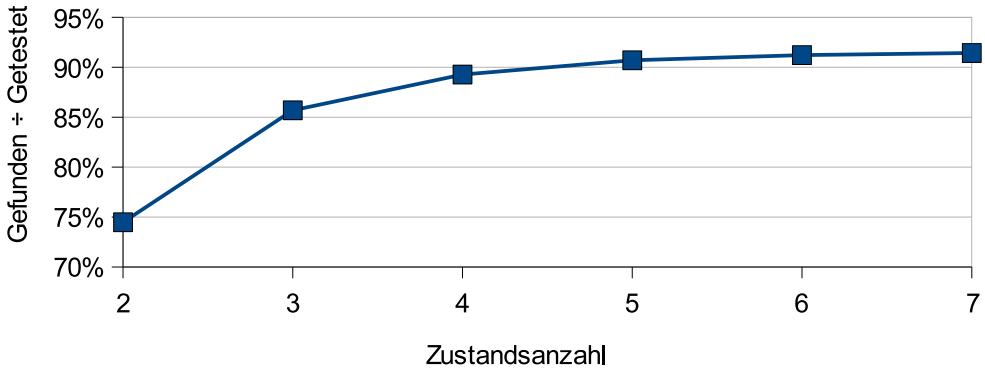


Abbildung 3.20: Relation gefundene zu getesteten Automaten mit den Kriterien arrangiert, reduziert, vereinfacht und präfixfrei

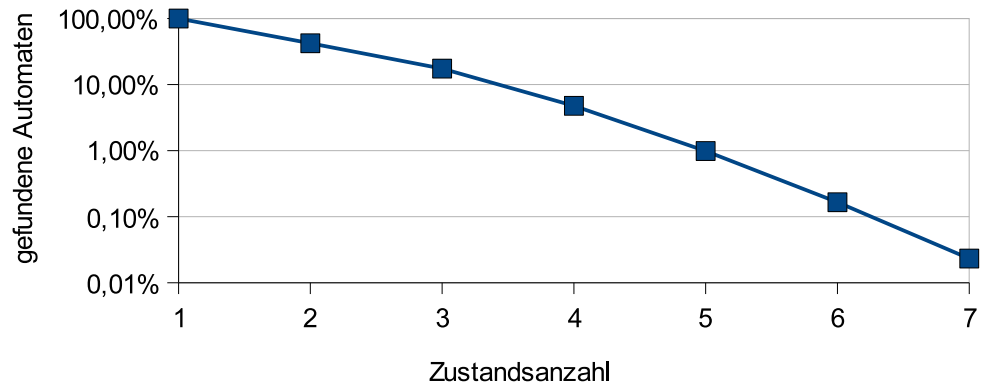


Abbildung 3.21: Anteil gefundener Automaten an der Gesamtanzahl $(2 \cdot |\mathbb{S}|)^{2 \cdot |\mathbb{S}|}$

Kombination verschiedener Kriterien mit maximaler Sprungweitenbestimmung führt daher immer nur zum nächsten potentiell interessanten Automaten bei Betrachtung aller Kriterien.

Für die Wahl der Algorithmen ist immer die Notwendigkeit der Automaten erforderlich, für die die Aufzählung erfolgen soll. Von daher ist keine pauschale Aussage für alle Fälle möglich. Allerdings ist es wenig sinnvoll, die Bedingungen vereinfacht und teilverbunden zu kombinieren, da vereinfacht schärfer ist und somit die nur teilverbundenen, aber nicht vereinfachten Automaten nicht als Ergebnis erscheinen.

Die Kombination von Präfixfreiheit und Vereinfachung ergibt zusammengenommen den in der Automatentheorie verwendeten Begriff „stark zusammenhängend“. Allerdings bleibt es erforderlich, diesen Sachverhalt getrennt zu betrachten, da sich der Sprungwert

h unterschiedlich berechnet. Die Bedingung selbst ist mit $\forall i \in \mathbb{S} : A[0,i] \wedge A[i,0]$ relativ einfach untersucht.

3.5 Hardware-Pipeline

Aufgrund der zahlreichen, unterschiedlichen Kriterien aus Abschnitt 3.4 bietet es sich an, die Überprüfung zu parallelisieren, um so schneller den Algorithmus einteilen zu können und um die Aufzählung zu vervollständigen. Dafür bietet sich eine Umsetzung in Hardware an.

Theoretisch ist es möglich, die Kriterien in Schaltkreise umzusetzen, so dass es nur einen Takt für den Zähler aus Abbildung 3.5, Seite 29 gibt. Dies ist allerdings nicht effizient, da die Signallaufzeiten durch die vielen Berechnungen zu lang wären und sich dadurch ein geringer Takt ergibt. Alternativ existiert die Technik einer Pipeline. Dazu gliedern sich die Überprüfungsalgorithmen in kleine Teilschritte mit Zwischenergebnissen auf, so dass dann ein höherer Takt der Berechnung z. B. mit einem FPGA möglich ist. Bevor ein Ergebnis feststeht, kann die Pipeline bereits den nächsten Automaten in die frei gewordenen Stufen aufnehmen.

3.5.1 Automatenzähler

Unabhängig von Pipeline- oder Schaltkreisumsetzung existiert der Zähler mit seinen einzelnen Stellen. Eine Stelle für die Ausgabe benötigt $\lceil \log_2 |\mathbb{Y}| \rceil$ Bits, die für einen Zustandsübergang $\lceil \log_2 |\mathbb{S}| \rceil$ Bits, so dass sich insgesamt $|\mathbb{S}| \cdot |\mathbb{X}| \cdot (\lceil \log_2 |\mathbb{S}| \rceil + \lceil \log_2 |\mathbb{Y}| \rceil)$ Bits für einen Automaten ergeben. Mit einer Pipeline multipliziert sich dieser Wert mit der Länge der Pipeline, da jede Pipelinestufe Zugriff auf den jeweiligen Automaten benötigt. Hinzu kommen die Zwischenergebnisse der einzelnen Pipelineberechnungen, die Einfluss auf den Gesamtspeicherbedarf haben.

Nicht zu verachten ist auch der Zugriff auf Bestandteile des Automaten in der Pipeline durch die Untersuchungskriterien. In der Softwareentwicklung ist Multiplikation kein Problem, in der Hardwareumsetzung allerdings bedeutet dies viel Aufwand. Eine andere Lösung für den Zugriff mit mehrdimensionalen Arrays getrennt für Pipelinestufe, Zustand und Eingangswert ist aufgrund fehlender Unterstützung durch das Synthesewerkzeug von Xilinx nicht möglich. Verkettung und Addition für die Indexberechnung bringt Abhilfe; die Pipelinestufen sind dann als Speicheradressen organisiert, die Wortbreite entspricht der eines Automaten. Der Index setzt sich aus Eingangs- und Zustandswert durch Konkatenation zusammen, der Speicherbedarf ändert sich dadurch zu $|\mathbb{S}| \cdot \lceil \log_2 |\mathbb{X}| \rceil^2 \cdot (\lceil \log_2 |\mathbb{S}| \rceil + \lceil \log_2 |\mathbb{Y}| \rceil)$. Für $|\mathbb{X}| = 2$ ergibt sich kein Unterschied, dafür verringert sich der Ressourcenverbrauch in der Umsetzung wesentlich im Gegensatz zum multiplizierten Index.

Da die Pipeline von Beginn an existiert und für mehrere Takte ein Ergebnis liefert, obwohl noch keine Überprüfung dafür erfolgt ist, erfordert dies noch ein „Gültigkeits“-Signal (*valid*), um die Interpretation des ausgegebenen Automaten als Ergebnis zu verhindern. Da nur zu Beginn dieses Gültigkeitssignal gesetzt werden kann, ist dieses, als Bit gespeichert, über die gesamte Pipelinelänge für jede einzelne Stufe vorhanden.

Die einzelnen Zählerstellen inkrementieren sich nach dem Verfahren aus Algorithmus 3.3, Seite 27. Um nun gleichzeitig den Wert h zu setzen, der ja auch Einfluss auf die gleichen Zählerstellen hat, muss der Ablauf unterschieden werden, da nach den bernsteinschen Bedingungen aus [Ber66] keine Variable aus unterschiedlichen Quellen par-

allel einen neuen Wert erhalten darf. Von daher gibt es einen Multiplexer, der zwischen Überspringen und Inkrementieren schaltet, mit Priorität für *h*. Dadurch geht zwar ein Takt verloren, andererseits erspart das Übergehen von Zählerständen viele Takte unnötiger Berechnung.

Ein anderes Problem mehrfacher Quellen hat die Minimalbestimmung von *h*. Da unterschiedliche Überprüfungen unterschiedliche Werte für *h* liefern können, ist die Bestimmung von *h* nach dem Prinzip des Identifikationsbusses aus [Fli01, Seite 313f.] umgesetzt. Dabei müssen die Einzelwerte von *h* aus unterschiedlichen Quellen in Registern gespeichert vorliegen, um nachfolgend mit brauchbarer Laufzeit das Minimum bestimmen zu können.

Hinzu kommt, dass nicht einfach das Maximum wie bei einem Identifikationsbus zu bestimmen ist, sondern nach Formel 3.2 teilweise ein Minimum für h_1 und h_2, sowie teilweise ein Maximum für h_3 und – ganz zu Beginn, quasi h_0 – der Wahrheitswert, ob der Wert für *h* überhaupt gültig ist. Um trotzdem die Maximumbestimmung zu verwenden, liegen h_1 und h_2 invertiert als Einskomplement am Identifikationsbus an. Anschließend verarbeitet dann der Automatenzähler den für *h* gefundenen Wert, dabei erhalten alle Pipelinestufen ein Signal, dass der gespeicherte Automatenwert ungültig ist, damit nicht nochmals ein Wert *h* fälschlicherweise Einfluss nehmen kann. Anschließend erfolgt über die normale Inkrementierung die Bestimmung des nächsten gültigen Automaten.

3.5.2 Überprüfungskriterien

Einige Kriterien basieren auf der reflexiven transitiven Hülle, bestimmt durch den Warshall-Algorithmus 3.8. Von daher konzentriert sich eine erste Betrachtung auf diesen Algorithmus, um das Prinzip der Pipelinefunktionsweise zu erörtern und die notwendige Anzahl an Stufen festzulegen.

In einem ersten Schritt übertragen sich die Verbindungen in den Speicher für *A*, dargestellt in Zeile 2 des Algorithmus 3.8 auf Seite 38. Dies geschieht parallel. Basierend auf diesen in Registern abgespeicherten Werten können die Schleifen ab Zeile 3 ablaufen. Aufgrund der Datenabhängigkeit ist es jedoch erforderlich, für den ersten Wert der Schleifenvariablen *j* das Resultat zu speichern, um danach in einer nächsten Stufe das Ergebnis wiederzuverwenden. Damit ergeben sich die weiteren Pipelinestufen im Umfang der Anzahl von Zuständen, bis das Ergebnis der reflexiven transitiven Hülle in Stufe $|\mathbb{S}|+2$ feststeht.

Dann ist das Ergebnis verwendbar. Nachdem der Automat ausgewertet ist, kann schließlich in einem weiteren Schritt gegebenenfalls der Wert für *h* gemäß Satz 3.4 und Gleichung 3.3 ermittelt werden, wobei die Berechnung von *k* und die Speicherung in *h* in der gleichen Stufe geschehen kann, die Auswertung des Minimums über alle Kandidaten für *h* erfolgt dann in einer weiteren Stufe. Insgesamt ergeben sich somit sechs weitere Schritte zusätzlich zur Anzahl der Zustände, veranschaulicht in Abbildung 3.22. Für die Ausgabe, behandelt in Abschnitt 3.5.3, kommt eine weitere Pipeline-Stufe hinzu, da erst in der Stufe $|\mathbb{S}|+6$ feststeht, ob der behandelte Automat relevant und damit auszugeben ist. Um den Mechanismus des Kopierens von Daten innerhalb der Pipeline beizubehalten, ist die abschließende Pipelinestufe $|\mathbb{S}|+7$ hinzugefügt, die eigentliche Berechnung erfordert dieses Addendum jedoch nicht.

Damit eine länger andauernde Ausgabe weitere Berechnungen nicht unnötig aufhält, besteht die Möglichkeit, dass diese Stufe $|\mathbb{S}|+7$ nur als Puffer dient. Erst wenn die Ausgabe noch aktiv ist, in Stufe $|\mathbb{S}|+7$ das nächste Ergebnis bereitsteht und in der Stufe

$|\mathbb{S}|+6$ ein weiterer, relevanter Automat anliegt, muss die Pipeline die Arbeit unterbrechen.

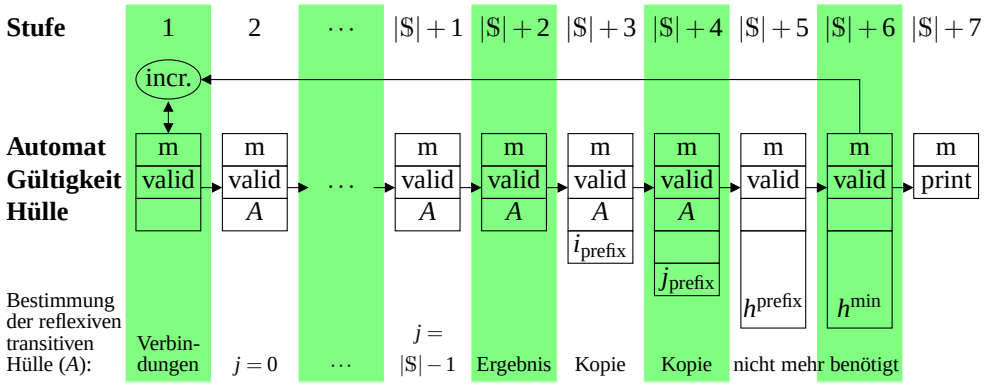


Abbildung 3.22: Einteilung der Pipelinestufen anhand der Präfixbestimmung

Es bietet sich an, die anderen Kriterien gleichartig einzuteilen, zumal diese ebenso auf einem Zustandszähler in der äußeren Schleife basieren. Von daher ergeben sich keine Ergänzungen oder sonstigen Abänderungen an der Pipeline. Um die Implementierung variabel zu halten, verwendet die Hardwareumsetzung mit Verilog HDL die „**generate**“-Konstrukte, zu sehen im Algorithmus D.4 ab Seite 185.

Als Beispiel sei die Zustandsreduktion nach Algorithmus 3.7 auf der Seite 37 aufgeführt, die einer ähnlichen Technik wie die Bestimmung der reflexiven transitiven Hülle bedarf. In der ersten Stufe werden die Ausgaben ausgelesen und dann in Stufe 2 als Ergebnis gespeichert. Danach folgen $|\mathbb{S}|$ Schleifendurchläufe in Form von aufeinander folgenden Pipelinestufen, so dass dann das Ergebnis in Stufe $|\mathbb{S}|+2$ feststeht. Anschließend reicht es aus, nur eine Hälfte der Matrix auszuwerten. Hierfür bieten sich $|\mathbb{S}|-1$ Leitungen an, die jeweils eine Zeile zusammenfassen; anfangs $|\mathbb{S}|-1$ Elemente ohne die Diagonale, also ohne $e[0,0]$, am Ende verbleibt nur noch das Element $e[|\mathbb{S}|-2,|\mathbb{S}|-1]$. Diese $|\mathbb{S}|-1$ Zwischenergebnisse fasst ein reduzierender Nicht-Oder-Operator zusammen. Das dabei entstehende Ergebnisbit bleibt bis zum Ende der Pipeline als einziges von der Zustandsreduktion erhalten.

Für die Minimalbestimmung z. B. aus Gleichung 3.3 ist ein Trick hilfreich, um einen konkreten Zahlenwert zu bestimmen. Ein Repräsentant für jedes Bit sei in einem Array angelegt. Solange die inverse Bedingung zutrifft, ist der entsprechende Wert Eins zugewiesen. Für das erste Bit, bei dem die Bedingung nicht mehr gilt, also das minimale Element der ursprünglichen Bestimmung, ändert sich dies; auch die nachfolgenden Bits weisen dann aufgrund einer Verkettung den Wert Null auf. Jetzt muss nur noch die Quersumme der einzelnen Bits errechnet werden, um den gesuchten Minimalwert zu erhalten.

Für die Addition stellt sich die Frage, wie diese effizient durchgeführt werden kann. Im Sinne der parallelen Abläufe lassen sich die einzelnen Bits paarweise addieren, um dann die entstandenen Zwischenergebnisse wieder zu addieren – solange, bis auch das letzte Paar summiert ist. Da jedoch die Bestimmung des gültigen Wertes sequentiell abläuft, ergibt sich durch dieses eigentlich optimale Verfahren eine zusätzliche Verzögerung, da erst mit der Bestimmung des letzten Gültigkeitsbits die Berechnung starten kann. Besser ist es da, die einzelnen Bits nacheinander aufzuaddieren, wie mit Algorithmus 3.37 beispielhaft als Verilog-HDL-Modul inklusive der Minimalwertbestimmung gezeigt. Für $n=8$ ist dies über 10 % schneller als die parallele Variante und braucht zudem ein Logikelement weniger.

Algorithmus 3.37: Index des vordersten, nicht-gesetzten Bits

```
1  module minbit #(parameter n = 8, k = n) (input [n : 1] bits, output [k : 1] sum);
2      wire valid [n : 1];
3      wire [k : 1] s [n : 1];
4      genvar i;
5      assign valid[1] = bits[1];
6      assign s[1] = valid[1];
7      generate
8          for (i = 2; i <= n; i = i + 1) begin: calculate
9              assign valid[i] = valid[i − 1] && bits[i];
10             assign s[i] = s[i − 1] + valid[i];
11         end
12     endgenerate
13     assign sum = s[n];
14 endmodule
```

Für die Bestimmung des Maximums statt des Minimums sind ein paar Änderungen notwendig. Statt bei dem kleinsten Wert bzw. Bit startet die Suche bei dem größten Wert. Des weiteren ist keine Umkehrung der Werte erforderlich, so dass das erste Element ohne Treffer eine Null aufweist. Erst nach einem gefundenen Element ist der Wert Eins weiterzugeben. Dies führt auch dazu, dass dadurch eine Oder- statt einer Und-Verknüpfung bei *valid* notwendig ist. Damit nicht die Subtraktion von Eins bei der Summe erforderlich ist, darf auch nicht für den obersten Wert die Summe gebildet werden, sondern erst für das nachfolgende Element. Dadurch ist auch die Verschiebung der Bitanzahl gewährleistet, da im Gleichklang zur Minimumbestimmung das ursächliche Element nicht mitgezählt werden darf, erst alle niedrigeren Elemente danach. Somit ist es unabhängig, ob das kleinste Element 0 in der Auswahlmenge des Maximums enthalten ist oder nicht – gibt es kein Maximum, dann ist auch kein Bit gesetzt und das Ergebnis ist automatisch Null. Von daher existiert kein spezieller Fehlerwert.

Eine andere Notwendigkeit der Pipeline zeigt Algorithmus 3.33 von Seite 69. Nach einer Initialisierung startet in Zeile 6 eine Simulation mit fünf Durchläufen. Jedes Zwischenergebnis der Simulation erfordert eine Speicherung mindestens des Wertes d. Hierfür ist jedes Mal eine Pipelinestufe notwendig. Zusammen mit der Auswertung in einer weiteren Stufe ergibt dies sieben Pipelinestufen, unabhängig von der Anzahl der Zustände. Ein Konflikt mit der bisherigen Betrachtung entsteht nicht, da die bisherige Anzahl von $|\mathbb{S}| + 6$ Stufen lediglich für $|\mathbb{S}| = 1$ unterschritten wird. Dieser Fall ist aber wegen seiner Trivialität nicht relevant. Bereits bei zwei Zuständen ($|\mathbb{S}| = 2$) sind beide Pipeline-Konstruktionen gleich lang, so dass eine Auswertung problemlos erfolgen kann.

3.5.3 Ausgabe der Ergebnisse

Nach der Berechnung der Ergebnisse muss noch eine Übertragung der Daten vom FPGA zum PC erfolgen. Hierfür bietet sich die serielle Schnittstelle an, da diese auf den verwendeten Testplatinen vorhanden und universell einsetzbar ist.

Da jedoch selbst bei schnellstmöglicher Übertragung mit 115 200 Baud ein Byte 260 Takte bei 50 MHz benötigt, muss die Pipeline während der Übertragung eines Ergebnisses stoppen, da sich während der Übertragung garantiert ein weiteres Ergebnis findet, das

ebenfalls zu übertragen ist – dies resultiert in einem Ressourcenkonflikt, den es zu vermeiden gilt. Ein großer FIFO-Speicher, der die Ergebnisse aufnehmen kann, hilft zwar, die Zeitdauer zu verkürzen. Da jedoch die Beschreibung eines Automaten nicht mit einem Byte erledigt ist, werden auch hier mehrere Takte verwendet. Mit der Darstellung aus Abbildung 3.6a auf Seite 30 ergeben sich für eine Übertragung $(4 \cdot |\mathbb{S}| + 1) \cdot |\mathbb{X}|$ Bytes inklusive Zeilenumbruch, wenn sich sowohl Zustandsübergangsnummer als auch Ausgabewert mit jeweils einem Zeichen darstellen lassen.

Interessiert hingegen nur die Gesamtanzahl der relevanten Automaten, so reicht es, nur diese Anzahl nach Empfang eines vereinbarten Zeichens zu übertragen. Entfällt die aktuelle Übermittlung des Wertes bei Verzicht auf korrekte Darstellung, falls ein Überlauf innerhalb der verwendeten Ziffern auftritt, kann die Pipeline sogar durchgehend arbeiten. Alternativ lässt sich der Wert vor der Übertragung einfach kopieren und dann ohne weitere Auswirkung seriell übertragen.

3.5.4 Gesamtbetrachtung

Für die Aufzählung von Automaten sind nun alle Komponenten beschrieben und die geeignete Länge der Pipeline auf $|\mathbb{S}| + 6$ festgelegt. Für das Zusammenspiel ist es noch wichtig, dass sich die einzelnen Teile nicht gegenseitig behindern.

Für den Fall $|\mathbb{S}| \cdot |\mathbb{Y}| < |\mathbb{S}| + 6$ ist eine Zusatzlogik unter dem Einfluss von h interessant, da in einem solchen Fall die Pipeline einige Zählerschritte weiter vorangeschritten ist, als es das Untersuchungsergebnis ergibt und so h ein Sprungziel aufweisen kann, dessen Berechnung bereits in der Pipeline erfolgt. Für alle anderen Werte sind die Zustandsübergänge in der Pipeline identisch, so dass h keinen Rücksprung zu einem bereits teilüberprüften Automaten bewirken kann.

Falls ein Sprungziel gefunden ist, kann dieses mit einem Vergleich zwischen h nach der Minimalwertberechnung und aktuellem Zählerstand erfolgen. Eine weitere Pipelinestufe ist dafür nicht notwendig, lediglich der Test, ob h gültig ist und ein Sprung erfolgen soll, ist erweitert. Alternativ kann h ohne Rücksichtnahme auf die bereitstehende Berechnung in der Pipeline erfolgen. In jedem Fall ist jedoch als Vergleich zwischen h_3 und s'_{h_1,h_2} gemäß Zeile 14 des Algorithmus 3.12 der Zustandswert aus Stufe $|\mathbb{S}| + 6$ relevant.

Die Ausgabe beeinflusst zwar auch die Berechnung, allerdings ändert sich die Pipeline an für sich deswegen nicht, sondern das bereits beschriebene Gültigkeitssignal ist ausreichend.

Zeitlich ergibt sich mit der Nutzung der Pipeline die Komplexität $O(n)$ für die Klassifizierung eines einzelnen Automaten, wobei n für die Anzahl der Zustände $|\mathbb{S}|$ steht. Mit dem Platzbedarf kommt die Komplexität $O(n^2)$ hinzu, so dass zusammengenommen die Komplexität $O(n^3)$ entsteht, die auch eine reine Softwareimplementierung z. B. mit einem Personal-Computer vom Ablauf her benötigt. Von daher erscheint der zeitliche Vorteil der Hardwareimplementierung deutlich, die bei einer möglichen Eintaktimplementierung theoretisch sogar auf $O(1)$ sinkt.

Die Umsetzung bezieht sich auf eine geringe Anzahl von Eingangswerten $|\mathbb{X}|$. Sollte die entstehende Taktrate aufgrund der Signallaufzeiten nicht mehr den Ansprüchen genügen, so müssen für die Eingangswerte eigene Pipelinestufen entstehen statt eine parallele Berechnung je Zustandswert zu erfahren, so dass sich eine Pipelinelänge von $|\mathbb{S}| \cdot |\mathbb{X}| + 6$ ergibt.

3.5.5 Resultate

Aufgrund gleicher Aufzählungsreihenfolge entsprechen sich die Ergebnisse der Hardware- und Software-Berechnung, ein direkter Vergleich der Berechnungsdauer ist somit ohne weiteres möglich, wenn die Ausgabe von Ergebnissen aufgrund technisch wesentlich unterschiedlicher Möglichkeiten unberücksichtigt bleibt. Für einen ersten Schritt ist die maximal mögliche Taktrate erforderlich, dargestellt in Tabelle 3.6. Zusätzlich ist noch die Anzahl der notwendigen Logikbausteine (LE für Logic Element bzw. LUT für Look Up Table) aufgeführt, wobei die Ausgabe davon etwa 199 Logikbausteine unabhängig von der Anzahl der Zustände benötigt. Ein Logikbaustein bezeichnet bei Altera und Xilinx eine Einheit, bestehend aus einer variablen Logikfunktion mit bis zu vier Eingängen und einem optionalem Flip-Flop zur Speicherung eines Bits, alles beliebig kombinier- und zusammenschaltbar. Die Angaben basieren bei Altera auf dem Cyclone-Baustein EP1C20F324C7, bei Xilinx auf dem FPGA XC3S1000 aus der Serie Spartan 3.

Tabelle 3.6: Logikbausteine und maximale Taktfrequenz für Überprüfung der Kriterien arrangiert, reduziert, vereinfacht und präfixfrei

	Altera		**Xilinx**	
$\lvert\mathbb{S}\rvert$	**LEs**	$f_{\max}$	**LUTs**	$f_{\max}$
2	1 063	90,24 MHz	1 144	46,887 MHz
3	1 358	85,16 MHz	1 552	44,960 MHz
4	1 853	89,54 MHz	2 169	48,174 MHz
5	2 814	86,67 MHz	3 554	43,745 MHz
6	4 200	82,93 MHz	5 599	46,698 MHz
7	6 548	86,25 MHz	10 422	47,551 MHz
8	11 078	83,22 MHz	18 561	46,462 MHz

Der interne Takt lässt sich bei Altera mit der Megafunktion „phase-locked loop“ (PLL) und bei Xilinx mit dem Digital Clock Manager (DCM) einstellen, so dass die am FPGA anliegende Frequenz von 50 MHz bei den Experimentierplatinen (*Evaluation Boards*) gleich bleiben kann, aber trotzdem die maximale Geschwindigkeit in der Berechnung erzielt wird. Für die Nutzung der seriellen Schnittstelle ist dann entweder der Parameter für die Wartezyklen einzustellen oder z. B. ein FIFO-Speicher zu verwenden, der mit unterschiedlichen Taktfrequenzen für Ein- und Ausgangsdaten umgehen kann.

Für einen Vergleich mit der Software ist noch die Anzahl der Takte erforderlich, bis alle relevanten Automaten die Pipeline durchlaufen haben. Zur Veranschaulichung ist in Tabelle 3.7 zusätzlich die rechnerische Zeit ohne optimierende Sprünge angegeben, die sich aus der Frequenz der Tabelle 3.6 und der Anzahl aller möglichen Automaten aus Tabelle 3.5 zusammensetzt. Die Zeit für die Software stammt aus Messungen mit einem Windows-XP-Rechner mit einem Prozessor vom Typ Pentium 4 HyperThreading, betrieben mit 3,2 GHz. Der verwendete C-Compiler stammt aus einer Cygwin-Installation und hat die Versionsnummer „3.4.4 (cygming special, gdc 0.12, using dmd 0.125)“. Die Geschwindigkeitssteigerung (Π_S) errechnet sich aus der Zeit der Software geteilt durch die Zeit der Hardware mit dem Altera-Baustein, jeweils in Sekunden für eine optimale Berechnung mit Sprüngen; kontrolliert wurden Isomorphie der Zustände bei einem Startzustand, Nutzung aller Zustände, Präfixfreiheit und reduzierter Automat.

Die Vorzüge der Hardwareberechnung und der Eingriff in die Aufzählung durch Sprünge gemäß Abschnitt 3.4 ist deutlich. Die Aufzählung bei sieben Zuständen benötigt statt einem Monat lediglich neun Stunden, wenn statt Software die parallel rechnen-

Tabelle 3.7: Berechnungsdauer (Angaben unter Verwendung der Sprungtechnik außer bei $\bar{h}$)

	Software	**Hardware**			Π_S
$\lvert\mathbb{S}\rvert$	**Zeit**	**Takte**	**Zeit**	**Zeit** $\bar{h}$	$^{\mathbf{SW}}/_{\mathbf{HW}}$
2	< 1 ms	187	2,1 μs	2,8 μs	< 480
3	15 ms	9 930	116,6 μs	547,9 μs	128,6
4	453 ms	902 577	10,1 ms	187,4 ms	44,9
5	85,2 s	109 189 318	1,3 s	115,4 s	67,7
6	4,5 h	16 189 070 248	195,2 s	29,9 h	83,6
7	38,3 d	2 823 618 430 488	9,1 h	4,1 a	77,3

de Hardware zum Einsatz kommt. Auch die Sprungtechnik hilft spürbar: Bei effektivem Einsatz der Hardware benötigt die Bestimmung der Anzahl ohne Sprünge das 3 935fache der Zeit, also statt neun Stunden über vier Jahre, bei 50 MHz Taktfrequenz sogar ganze sieben Jahre.

Tabelle 3.8: Logikbausteine und maximale Taktfrequenz für alle Überprüfungen (Kriterien arrangiert, beliebiger Startzustand, reduziert, vereinfacht, präfixfrei, vorüberprüftes Ausgabeverhalten)

	Altera		**Xilinx**	
$\lvert\mathbb{S}\rvert$	**LEs**	$f_{\max}$	**LUTs**	$f_{\max}$
2	1 530	65,54 MHz	1 390	50,292 MHz
3	2 654	54,20 MHz	2 310	44,960 MHz
4	4 642	49,75 MHz	4 328	48,700 MHz
5	8 243	45,90 MHz	7 776	48,723 MHz
6	13 613	45,38 MHz	13 296	46,887 MHz
7	21 672	44,64 MHz	23 522	44,778 MHz

Bei Einsatz weiterer Einschränkungen reduziert sich nochmals die Anzahl möglicher Automaten. Für die zusätzlichen Eigenschaften beliebiger Startzustand und Anforderungen an die Ausgabe ergeben sich für den Hardwarebedarf analog zu Tabelle 3.6 die Werte in Tabelle 3.8. Die errechnete Anzahl der interessanten Automaten hierfür ist in Tabelle 3.9 dargestellt und in Abbildung 3.23 graphisch mit logarithmischer Skala aufgetragen. Die Spalte „$D \cap S \cap V$“ (alle Bedingungen zusammen) bezieht sich auf Automaten mit beliebigem Startzustand und einer Kontrolle des Ausgabeverhaltens sowie den bisherigen Kriterien Isomorphie der Zustände bei einem Startzustand, Nutzung aller Zustände, Präfixfreiheit und reduzierter Automat. Auch hierbei reduziert sich die Anzahl relevanter Automaten merklich – im Vergleich zu allen Automaten ohne einschränkende Kriterien (A) zeigt sich eine deutliche Abflachung der Kurve.

Interessant ist dabei der Vergleich der Anzahl „D“ (ohne Duplikate) und „S“ (beliebiger Startzustand), die sich um den Faktor $\lvert\mathbb{S}\rvert$ voneinander unterscheiden. Ein weiterer Zusammenhang zeigt sich zwischen der letzten Spalte „$D \cap S \cap V$“ (alle Bedingungen zusammen) und der Anzahl davor: $(D \cap S) \cdot (D \cap V) \div (D)$ liefert den gleichen Zahlenwert. Demnach zeigt sich von der Anzahl her ein Zusammenhang, mit aufgelisteten Graphen im direkten Vergleich ist es allerdings nicht erkennbar, auch nicht eine Konstruktionsregel, wie bereits im Abschnitt 3.4.5.3 auf Seite 59 für einen alternativen Startzustand beschrieben.

Um die Werte in Tabelle 3.9 für sieben Zustände zu erhalten, ist eine vereinfachte Schaltung für die 20 060 zur Verfügung stehenden Logikelemente erforderlich. Die Ein-

Tabelle 3.9: Anzahl an Automaten im Vergleich mit den zusätzlichen Kriterien beliebiger Startzustand (*S*) und überprüftes Ausgabeverhalten (*V*) zusammen mit den duplikatfreien Kriterien arrangiert, reduziert, verbunden und präfixfrei (*D*)

$\lvert\mathbb{S}\rvert$	*D*	$D \cap S$	$D \cap V$	$D \cap S \cap V$
2	108	54	28	14
3	8 124	2 708	1 812	604
4	798 384	199 596	158 188	39 547
5	98 869 740	19 773 948	17 039 940	3 407 988
6	14 762 149 668	2 460 358 278	2 175 542 688	362 590 448
7	2 581 401 130 308	368 771 590 044	212 197 728 470	30 337 377 463

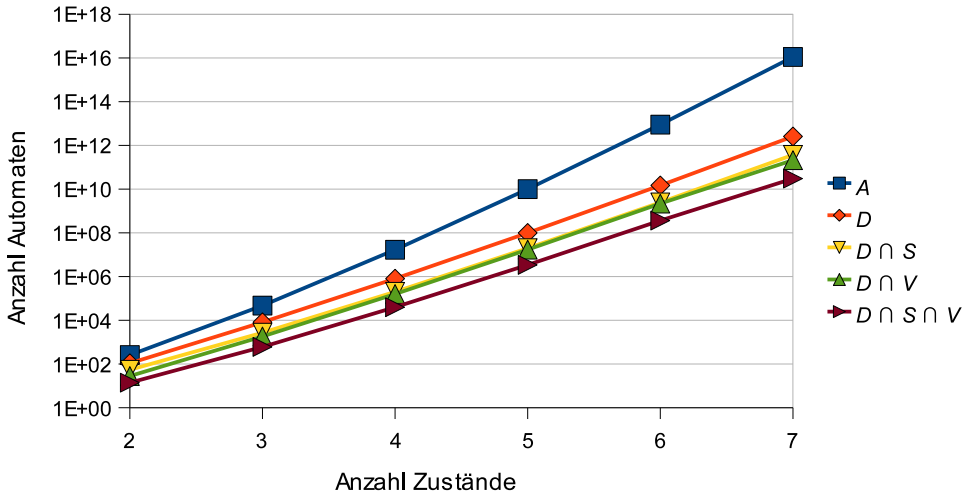


Abbildung 3.23: Automatenanzahl aus Tabelle 3.9 im Vergleich mit allen möglichen Automaten ($A = (|\mathbb{S}| \cdot |\mathbb{Y}|)^{|\mathbb{S}| \cdot |\mathbb{X}|}$)

und Ausgabe über die serielle Schnittstelle sowie die Berechnung von *h* für Präfix gemäß Formel 3.3 ist genauso wie für *h* der optimalen Startzustände durch Zeile 7 des Algorithmus 3.26 entfallen, dafür entstand ein neuer Mechanismus zur Zahlenausgabe über die acht zur Verfügung stehenden LEDs. Jeweils acht Binärziffern als Teil des Zahlenwertes erscheinen für etwa eine Sekunde. Zur Orientierung des Anzeigezyklusbeginns erscheinen statt zweier Ziffern zuerst eine blinkende LED – während der Berechnung mit langer, bei abgeschlossener Zählung mit kurzer Leuchtdauer. Die nächste LED ist für eine Identifikation des Wertes. So ist es möglich, zwei 62-Bit-Werte darzustellen. Mit platzoptimierenden Syntheseeinstellungen bedarf es 20 050 Logikelementen bei einer maximal möglichen Taktrate von 44,64 MHz. Für acht Startzustände ist eine Berechnung für beliebigen Startzustand mit der gegebenen Hardware nicht mehr möglich.

Insgesamt müssen bei der Umsetzung in Verilog HDL für das oberste Modul neben den einzelnen Parametern für Eingangswerte-, Ausgabewerte- und Zustandsanzahl auch die jeweilige Bitanzahl mit angegeben sein, da leider nicht jedes Synthesewerkzeug die in [Ins01, Seite 162] vorgeschlagene Logarithmusberechnung durchführen kann. Des weiteren kann die insbesondere für die Zustandsspeicherung verwendete parametrisierte Bereichsangabe, z. B. „s[1][0 +: outputbits]“, aufgrund fehlender Unterstützung der verwendeten Plattform Xilinx ISE 9.2.03i nicht genutzt werden, für Altera Quartus II Version 7.2 hingegen stellt beides kein Problem dar.

Schlussfolgernd zeigt sich ein besonderer Vorteil produktunabhängig mit wesentlichem Zeitvorteil gegenüber den Möglichkeiten der Software, Automatenkriterien zu überprüfen. Die Unterschiede in der Anzahl der Logikbausteine zwischen Altera und Xilinx sind teilweise durch die technisch unterschiedliche Implementierung der seriellen Schnittstelle und der unterschiedlichen Technologie bedingt.

3.6 Zusammenfassung

Mit dem Ziel, relevante Automaten schnell aufzuzählen, entstanden verschiedene Kriterien. Ein komplexes Gebiet dabei stellt die Auswahl einer Permutation der Zustände dar. Hierzu entstanden drei Variationen: Zum einen Normiert, bei dem die Zustandsübergänge für kleinere Eingangswerte zusammenhängend sein sollten, zum anderen Arrangiert als überprüfbares Verfahren, bei dem zuerst die kleineren Zustandsnummern im Vordergrund stehen und die Eingangswerte nachgeordnet sind. Die dabei ausgewählten Automaten unterscheiden sich voneinander, die Gesamtanzahl ist aber gleich. Die dritte Variante benötigt die Aufzählung aller Permutationen eines Automaten, um dann den Automaten mit der kleinsten Ordinalzahl als Repräsentant auszuwählen – die Ergebnisse entsprechen denen von Arrangiert, das damit eine effizienteres Verfahren darstellt.

Neben der Permutation der Zustände ist auch noch – bei Bedarf – die Permutation von Eingangs- und Ausgabewerten möglich. Zumindest für die Ausgabewerte ist eine Überprüfung möglich, ohne hierfür alle Permutationen aufzählen zu müssen. Alternativ dazu ist auch ein Überprüfen der Semantik dieser Werte möglich, allerdings ist dies vom Einsatzzweck abhängig und ist nicht allgemein angebbar.

Weitere Kriterien stellen die Überprüfung zur Nutzung aller Zustände dar. Relevant ist in erster Linie, ob überhaupt alle Zustände erreichbar sind. Ansonsten käme der Automat auch mit weniger Zuständen aus. Sind auch diese Automaten relevant, so ist ein einheitlicher Wert von 0 für Zustandsübergänge und Ausgabewerte für die nichtgenutzten Zustände festgelegt, um Duplikate zu unterbinden. Interessant ist auch, ob im weiteren Verlauf alle Zustände genutzt werden. Diese Überprüfung erfolgt mittels der Eigenschaft Präfix, die ein Enden in einem Zyklus mit weniger Zuständen unterbindet.

Für einen Automaten ist es auch noch relevant, ob dieser nicht auch mit weniger Zuständen auskommt, ob also Zustände zusammenlegbar sind, ohne dass sich dann das Verhalten ändert. Hierfür bietet sich das Kriterium Reduziert an.

Ein Teil der Kriterien ermöglicht eine Aussage der Eigenschaften der nachfolgenden Automaten, falls ein Kriterium verletzt ist. Daher sind diese Automaten ungeprüft ausschließbar und können somit übersprungen werden – allerdings nur aufgrund der gewählten Aufzählungsreihenfolge. Den Erfolg zeigen die Implementierungen, wobei die Hardwareumsetzung aufgrund der parallel durchführbaren Kriterienüberprüfung schneller ablaufen kann, als dies derzeit mit einem universelleren Standard-Computer möglich ist.

3.7 Retrospektion

Die stark verringerte Laufzeit der Aufzählung mit Sprüngen gegenüber der vollständigen Aufzählung stellt einen deutlichen Vorteil dar. Auch die verringerte Komplexität der Überprüfungsverfahren aus Abschnitt 3.4 im Gegensatz zur Konstruktion eines Repräsentanten in Abschnitt 3.2 reduziert die Laufzeit. Ein alternativer Ansatz, wie im Ab-

schnitt 3.3 betrachtet, hat nicht zu einem Ergebnis geführt. Dies alles spricht für das im Abschnitt 3.5 in Hardware zwecks kürzerer Berechnungszeit umgesetzte Verfahren.

Eine Reihenfolge der einzelnen Kriterien gibt es nicht, wie auch die Parallelität der Hardwarevariante deutlich macht. Für eine Softwareimplementierung ist daher der Ablauf der Überprüfung nicht relevant, zumal für alle Kriterien die Sprungberechnung erfolgen sollte. Es lässt sich jedoch eine Verringerung der Überprüfungen erreichen, indem nur bei veränderten Zustandsübergängen eine entsprechende Berechnung erfolgt. Umgekehrt brauchen die Bedingungen, die keine Sprünge auslösen können, nicht überprüft zu werden, wenn sich bereits eine Irrelevanz ergeben hat. Dies betrifft die Reduzierbarkeit sowie die Ein- und Ausgabesemantik. Auch kann die kostenintensive Überprüfung des optimalen Zustands an Stelle 0 entfallen, wenn sich ein negatives Ergebnis für die Nutzung aller Zustände (Vereinfacht und Präfix) gezeigt hat.

Interessant ist hierbei, dass die bisher übliche Auswahl von Automaten lediglich auf Permutation beruhte. Stattdessen ist die dabei genutzte Isomorphie nicht die einzige interessante Eigenschaft. Die Reduzierbarkeit von Automaten aufgrund ihrer Ausgabe stammt aus der Elektrotechnik, um Kosten bei der Entwicklung und Produktion zu sparen. Wie sich gezeigt hat, ist sie aber auch bei der Aufzählung von Automaten sinnvoll verwendbar, um die Anzahl der Untersuchungsobjekte zu vermindern, also die Anzahl der als interessant ermittelten Automaten. Weitere Eigenschaften neben der Semantik und der Auswertung genutzter Zustände sind eher problembezogen, aber durchaus relevant, um die Anzahl weiter zu verringern.

3.8 Bilanz

Ein Ergebnis lässt sich mit ausreichender Rechenleistung bestimmen. Das erste, mit Hilfe eines Computers gelöste Problem ist 1976 das Vier-Farben-Problem, das in tausende Fälle reduziert und ausprobiert wurde. Gemäß [DH96, Seite 401 bis 409] erregte dies selbst in der New York Times Aufsehen. Das Konzept des mathematischen Beweises und damit die Theorie bleibt dadurch unverändert, lediglich die Praxis der Mathematik verändert sich, nach Meinung von Prof. Haken in einem Interview zu seinem Beweis. Von daher ist die verwendete Vorgehensweise legitim.

Dieses Vier-Farben-Problem wurde 1879 von Kempe das erste Mal gelöst, allerdings entwickelte Heawood 1890 ein Gegenbeispiel, so dass das Problem wieder offen war. Neben der obig beschriebenen Lösung mittels Computerhilfe ist in [SB97, Seite 140ff.] ein 1979 erstmals erschienener mathematischer Beweis dieses Problems aufgeführt, den zuletzt der Autor 1996 überarbeitet hat. Von daher kann es zeitlich durchaus interessant sein, Probleme mit Computerunterstützung anzugehen.

Auch kann nach einer ersten Analyse mit einer einfacher zu schreibenden Softwarelösung – quasi der Prototyp – der Ansatz in eine Hardwareentwicklung münden, um dann für weitere Eingangswerte schneller zu einem Ergebnis zu kommen. Aber auch dabei sollte man die mathematische Theorie nicht aus den Augen verlieren, im einzelnen war sie für diese Analyse, insbesondere für die Permutationen, sehr nützlich, um effizient zu einer Lösung zu gelangen. Eine Mischung aus allem ist das beste Rezept für ähnlich gelagerte Problemlösungsstrategien.

Kapitel 4

Anwendungsbeispiel Kreaturen

Jeder Automat braucht eine Tätigkeit, um sich bewähren zu können. Eine Idee dafür ist die Frage, wie viel Intelligenz ausreicht, damit sich eine Kreatur in einem unbekannten Gebiet zurechtfindet und alle begehbaren Stellen abschreitet. Neben den Eigenschaften der Kreatur ist ein Gebiet festzulegen, um Simulationen durchführen zu können. Dabei gibt es unterschiedliche Techniken der Durchführung, die insbesondere für die variantenreichen Hardwareentwürfe am Ende dieses Kapitels analysiert sind.

4.1 Modellbeschreibung

Die Simulation unterteilt sich in einzelne Funktionsgruppen. Zum einen gibt es die statische Umgebung, die den Rahmen vorgibt, ähnlich einer Welt, auf der die Aktionen stattfinden. Dann gibt es sich darauf bewegende Objekte, die sich dynamisch verhalten. Falls es dabei zu Konflikten kommt, ist ein Mediator von Nöten, der diesen Konflikt beseitigt, so dass die dynamischen Objekte weiter agieren können. Eine Statistik vollendet die Beschreibung, um einen Vergleich zwischen den einzelnen Simulationen zu ermöglichen.

4.1.1 Statische Umgebung

In einer Welt gibt es Dinge, die sich über einen Simulationszeitraum gesehen nicht verändern. Zum Beispiel Gebietsbegrenzungen. Für eine Simulation ist es aufgrund des Rechenbedarfs von Vorteil, wenn für diese unveränderlichen Objekte nicht jedes Mal eine Neuberechnung erfolgt. Daher sei hier eine statische Umgebung eingeführt. Für Objekte, die sich nur gelegentlich ändern, sich ansonsten aber statisch verhalten, gilt der folgende Abschnitt 4.1.2 über dynamische Objekte.

Eine Welt mit ihren Ausmaßen kann mit Hindernissen ähnlich einem Labyrinth ausgestattet sein. Für eine einfache Beschreibung sei die Welt eine ebene, rechteckige mit festem Raster, so dass die Hindernisse des Labyrinths mit ganzzahligen Koordinaten angegeben werden können.

Eine Welt habe die Breite X und Länge Y mit $X,Y \in \mathbb{N}$. Dadurch ergeben sich die Koordinaten $\mathbb{P} := \{c \mid c = (x,y) \in \mathbb{N}_0 \times \mathbb{N}_0 \ \wedge \ 0 \leq x < X \ \wedge \ 0 \leq y < Y\}$ für die Oberfläche der Welt. Hindernisse (*obstacles*) befinden sich an den Positionen $\mathbb{H} \subset \mathbb{P}$, wobei der Einfachheit halber der Rand der Welt $\{c \mid c = (x,y) \in \mathbb{P} \ \wedge \ ((x = 0) \vee (x = X-1) \vee (y = 0) \vee (y = Y-1))\} \subseteq \mathbb{H}$ immer durch Hindernisse dargestellt wird und somit immer vorhanden ist. Das übrig gebliebene ist eine freie, „begehbare“ Fläche $\mathbb{A} \subseteq \mathbb{P} \backslash \mathbb{H}$. Beispiele

für unterschiedliche Welten sind im Anhang C zu finden, die Mengenbeschreibung von $\mathbb{H}$ für eine Welt ist auf Seite 141 angegeben. Gibt es nicht erreichbare Positionen, z. B. bei einer kreisförmigen Umgebung, deren äußerer Bereich theoretisch, aber nicht praktisch begehbare Positionen enthält, ist $\mathbb{A}$ nur eine Teilmenge von $\mathbb{P}\backslash\mathbb{H}$.

Da die Hindernisse in der statischen Umgebung eine feste und begrenzte Position innehaben, lassen sich diese als statische Objekte bezeichnen. Für eine Position p sei h der Wahrheitswert, ob diese Position in der Menge der Hindernisse enthalten ist oder nicht. Es gilt $h_p := (p \in \mathbb{H})$.

4.1.2 Bewegliche Objekte

Für die Simulation wesentlich sind sich bewegende Objekte. Dies können sowohl die eingangs erwähnten Kreaturen, aber auch lediglich mobile Hindernisse sein. Beispiele für deren Darstellung sind in Abbildung 4.1 zu sehen. Daraus lässt sich ableiten, dass es unterschiedliche Regeln für unterschiedliche bewegliche Objekte geben muss.

Abbildung 4.1: Zwei unterschiedliche Kreaturen und ein mobiles Hindernis

In einer einfacher durchführbaren Simulation können die mobilen Hindernisse aber auch erst einmal entfallen. Dadurch entfällt auch die Steuerung dieser Hindernisse, beeinflusst durch außerterritoriale Ereignisse, zufallsgesteuert oder durch Geschehnisse auf dem Simulationsfeld. Vorstellbar sind hier ein Blatt, angetrieben durch zufällig auftretenden Wind unterschiedlicher Intensität, oder ein Ball oder ein Karton, der sich durch Kontakt mit den Kreaturen in Bewegung versetzt.

Für alle mobilen Varianten gilt, dass diese eine Position $p \in \mathbb{P}$ auf dem Feld und eine Bewegungsrichtung r haben. Da sich beide nur durch das Bewegungsmuster voneinander unterscheiden, werden im folgenden nur noch die Kreaturen im Sinne von Agenten behandelt; für die mobilen Hindernisse gilt analoges, nur mit einer anderen Regel und evtl. anderen Umgebungsinformationen. Details sind in Abschnitt 4.1.3 aufgeführt.

Eine Kreatur weiß erst einmal nicht, ob sie sich in der Umgebung bewegt oder die Umgebung nur an ihr vorbeizieht. Relativ betrachtet gibt es auch diesbezüglich keinen Unterschied. In jedem Fall ändert sich die wahrgenommene Umgebung, z. B. resorbiert über den Eingang x. Einfluss nehmen, also steuern, kann die Kreatur mit Anweisungen über die Ausgabe y. Das eigene Gedächtnis lässt sich als Zustand s darstellen. Damit ist ein Zustandsautomat gegeben und in eine Umwelt eingebettet, analog zu Abbildung 3.1.

Die Festlegung, ob dies ein Moore- oder Mealy-Automat sein soll und wie viele Zustände dieser hat, kann von Kreatur zu Kreatur unterschiedlich sein. In Bezug zu Kapitel 3 seien es die allgemeinen Mealy-Automaten mit maximal sieben Zuständen, so dass eine Aufzählung in überschaubarer Zeit möglich ist.

Für die weitere Handhabung seien alle Kreaturen zu einer Menge $\mathbb{I}$ zusammengefasst, die die Indizes aller beweglichen Objekte enthält. Ohne Analogie zu den bisherigen Mengen beginnt die Zählung allerdings bei Eins. Bei beispielsweise drei Kreaturen lautet die Menge also $\mathbb{I} = \{1,2,3\}$ mit $|\mathbb{I}| = 3$.

4.1.3 Verbindung von statischen und dynamischen Objekten

Im einfachen Modell gibt es für bewegliche Objekte nur vier Bewegungsrichtungen, zusammengefasst in der Menge $\mathbb{D} := \{\text{Norden}, \text{Osten}, \text{Süden}, \text{Westen}\}$, deren stellvertretende Zahlenwerte für den Umgang mit Berechnungen in Tabelle 4.1 aufgeführt sind. Um voranzukommen und Hindernisse zu überwinden, reichen zwei Drehbewegungen mit Schritt nach vorne als einzige Bewegungsform aus, repräsentiert mit der Menge $\mathbb{B}$ mit den Elementen „nach rechts“ und „nach links“. Diese lassen sich mit „gerade aus“ und „zurück“ erweitern.

Tabelle 4.1: Transformation der Bewegungsrichtungen

Wert aus $\mathbb{D}$	**äquivalenter Zahlenwert**
Norden	0
Osten	1
Süden	2
Westen	3

Gibt es zusätzlich noch Zwischenwerte der Richtungen $\mathbb{D}$, so ergeben sich auch für die Drehung $\mathbb{B}$ weitere Möglichkeiten, z. B. halb rechts. Zugunsten eines einfachen Modells soll $\mathbb{D}$ aber auf vier Elemente beschränkt bleiben.

Tabelle 4.2: Mögliche Drehungen und deren Auswirkung auf die Richtung

Drehung $\mathbb{B}$	**Zahlenwert $\mathbb{Y}$**	**Richtungsänderung für $\mathbb{D}$**
rechts	0	$+1$
links	1	$-1 = +3 \pmod 4$
gerade aus	2	$+0$
zurück	3	$+2$

Mathematisch lässt sich eine Drehung als Addition mit modulo $|\mathbb{D}| = 4$ auffassen. Die dafür zu verwendenden Werte sind in Tabelle 4.2 zusammengefasst. Bei gleichzeitiger Fortbewegung ändert sich auch die Position der Kreatur in Abhängigkeit ihrer Drehung. Unter der Voraussetzung, dass sich eine Kreatur immer vorwärts bewegen muss, wenn dies möglich ist, also z. B. nicht vor einem Hindernis steht, ergibt sich eine Folgeposition p' nach Formel 4.1 mit $\mathbb{A}'_i = \mathbb{A}$. Da jedoch diese Folgeposition p' und die vorher auf Hindernisse zu begutachtende Position identisch sind, bietet sich die Definition einer Zielposition $\dot{p}$ in die Richtung $r \in \mathbb{D}$ der Kreatur an Position $p \in \mathbb{A} \subset \mathbb{P}$ als Hilfsvariable an.

Um die Positionen der unterschiedlichen Kreaturen voneinander unterscheiden zu können, ist ein Index $i \in \mathbb{I}$ notwendig. Aufgrund der zeitlichen Abfolge ist eigentlich auch ein Zeitindex t erforderlich. Damit jedoch die Anzahl der Indizes gering bleibt, entfällt dieser Index. Nichtsdestominder wandelt sich die Folgeposition p'_i in die aktuelle Position p_i im Übergang eines Zeitschritts, also quasi $p_{i,t+1} = p'_{i,t}$.

$$\begin{aligned} \dot{p}_i &:= \begin{cases} (x, y+1) & \text{für } r = \text{Norden} \\ (x+1, y) & \text{für } r = \text{Osten} \\ (x, y-1) & \text{für } r = \text{Süden} \\ (x-1, y) & \text{für } r = \text{Westen} \end{cases} \quad \text{mit } (x,y) = p_i \\ p'_i &:= \begin{cases} p_i & \text{wenn } \dot{p}_i \notin \mathbb{A}'_i \\ \dot{p}_i & \text{wenn } \dot{p}_i \in \mathbb{A}'_i \end{cases} \end{aligned} \tag{4.1}$$

Um nun noch mit einem einfachen Wert den Erfolg des Vorankommens zu beschreiben, der ja auch für die Bestimmung des Folgezustands s' benötigt wird, erfolgt die Definition des Wertes m_i mit $m_i := (\dot{p}_i \in \mathbb{A}'_i)$.

Gleichzeitig kann bei mehreren vorhandenen Kreaturen diese Zielposition $\dot{p}_i$, die ja jede Kreatur hat, für eine Kollisionsbehandlung herangezogen werden. Um dies zu erreichen, enthält die Menge $\mathbb{A}'_i$ nicht nur die hindernisfreien Positionen $\mathbb{A}$, sondern auch alle Positionen und Zielpositionen der anderen Kreaturen. Damit ergibt sich $\mathbb{A}'_i := \mathbb{A} \setminus \{p_j, \dot{p}_j \mid j \in \mathbb{I} \setminus \{i\}\}$. Die Kollision, eine einzelne Kreatur betreffend, habe die Bezeichnung und Definition $c_i := (\dot{p}_i \in \{p_j, \dot{p}_j \mid j \in \mathbb{I} \setminus \{i\}\})$. Zusätzlich gibt es noch den Wert o_p als Indiz, ob sich an Position p ein (dynamisches) Objekt befindet. Somit gilt $o_p := (p \in \{p_i \mid i \in \mathbb{I}\})$.

Damit sind alle Werte bestimmt, die eine Kreatur bezüglich der Umwelt benötigt. Der Eingang x ist identisch mit m_i, die Steuerung der Drehung $b \in \mathbb{B}$ übernimmt die Ausgabe $y \in \mathbb{Y}$ mit den Äquivalenzen gemäß Tabelle 4.2, die auch die Auswirkung auf die Richtung $r \in \mathbb{D}$ enthält. Für eine erste, einfache Variante sei lediglich $\mathbb{Y} = \{0,1\}$. Das Voranschreiten erledigt die Umwelt gemäß Formel 4.1, somit arbeitet die Simulationsumgebung prinzipiell unabhängig von der Kreatur.

Verbleibt die Frage, ob die Position der Kreaturen nur bei der Umgebung, nur bei der Kreatur, bei beiden kombiniert oder vollständig separiert assoziiert sein soll. Gleiches gilt für die Richtung. Diese Details der Umsetzung behandelt Abschnitt 4.3.3 mit Ergebnissen in Abschnitt 4.3.4.

4.1.4 Verbindung zwischen dynamischen Objekten

Neben der Anbindung eines dynamischen Objektes an eine statische Umgebung ist es auch denkbar, dynamische Objekte direkt untereinander unabhängig von deren Position Informationen austauschen zu lassen. Dies ist als feste Verbindung, quasi eine Standleitung, oder dynamisch, ähnlich einer wählbaren Telefonverbindung, denkbar. Die Informationen, die dabei ausgetauscht werden, können auch nur wenige oder sogar nur ein Bit umfassen.

Für die Wahl eines Kommunikationsempfängers ist die Ausgabemenge $\mathbb{Y}$ zu erweitern. Da nach wie vor eine Bewegungssteuerung notwendig ist, könnte ein Bereich ähnlich dem Prinzip der Dualzahlen für Kommunikation, ein anderer für das Vorankommen zuständig sein. Die Kommunikationsdaten können dann Indikatoren für eine bestimmte Kreatur, eine Gruppe von Kreaturen oder Kreaturen in der näheren Umgebung sein. Zusätzlich sind die eigentlichen Kommunikationsdaten in der Ausgabe notwendig. Die anderen Kreaturen können diese Daten über ihren ebenfalls zu erweiternden Eingang $\mathbb{X}$ aufnehmen.

Statt einer direkten Kommunikation ist auch eine Hinterlassenschaft von Nachrichten in der statischen Umgebung denkbar. Diese muss dann entsprechend zur Aufnahme von Daten erweitert werden. Aber auch hier ist eine Erweiterung der Ein- und Ausgänge $\mathbb{X}$ und $\mathbb{Y}$ notwendig. Das statische Feld dient nur als Mittler.

Auch eine Kombination all dieser Möglichkeiten ist denkbar. Da jedoch unabhängig der Details immer eine Erweiterung der Mengen, insbesondere der Eingangswerte $\mathbb{X}$ notwendig ist, führt eine Aufzählung der dazugehörigen Automaten, generiert mit dem Verfahren aus Kapitel 3, zu einer wesentlichen Erhöhung der Ausführungs- und damit Gesamtsimulationsdauer. Zudem erhöht sich auch wesentlich die Komplexität der Implementierung, die bei gleich bleibenden zur Verfügung stehenden Ressourcen eine kleinere

Umgebung zur Folge hat, wie sich in [Yua06] gezeigt hat. Daher ist dieses Thema nur eine nicht weiter verfolgte Anregung.

4.1.5 Kollisionsbehandlung bei mehreren Objekten

Ein Konflikt entsteht, wenn mehrere Objekte auf die gleiche Zielposition zur gleichen Zeit gelangen wollen. Wenn nur ein Objekt gleichzeitig auf einer Position sein darf, besteht die einzige Lösungsmöglichkeit darin, dass sich maximal ein Objekt bewegt, die anderen dürfen nicht. Für ein Voranschreiten ist ein Objekt auszuwählen. Abbildung 4.2 veranschaulicht die Problematik und die weiteren Konsequenzen für andere bewegliche Objekte, miteinander in verschiedenen Ereignisketten verwoben.

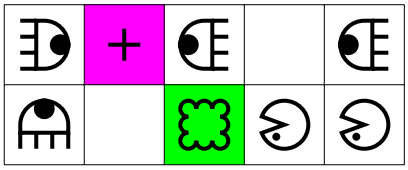


Abbildung 4.2: Beispiel für sich gegenseitig blockierende Kreaturen

Die einzelnen beweglichen Objekte lassen sich untereinander priorisieren. Alternativ lässt sich aufgrund bestimmter Eigenschaften wie Richtung, Objektattribute oder zufällig eine Auswahl treffen. Im Gegensatz dazu könnte auch keinerlei Auswahl erfolgen und alle betroffenen Objekte erhalten ein Konfliktsignal.

Ein Beispiel mit zusätzlicher sprachlicher Unterstützung einer speziellen Konfliktbehandlung bietet [Hoc98, Kapitel 7]. Für die Priorisierung lässt sich ein Auswahlverfahren benutzerspezifisch mittels einer konkreten Regel zur Berechnung angeben. Die anschließende Berechnung unterteilt sich in zwei Phasen. In der ersten erfolgt die Berechnung des Ziels, in der zweiten die der Akzeptanz, in der neben der Auswahl auch die Durchführung erfolgt.

Im Sinne einer einfachen Implementierbarkeit erfolgt keine Bevorzugung irgendeines beweglichen Objektes. Dadurch ist es sogar möglich, die Konfliktbehandlung in nur einer Phase erfolgreich abzuschließen. Dazu melden die Objekte ihren Bewegungswunsch bei ihrer jeweiligen Zielposition an. Diese wertet dann die Signale aus allen Richtungen aus. Ist die Summe aller Signale größer eins, also will mehr als ein Objekt auf die Zielposition, gibt es ein Ablehnungssignal an alle angeschlossenen Positionen. Gleiches gilt, wenn sich auf der Zielposition bereits ein statisches oder dynamisches Objekt befindet; der Maximalwert der Summe verringert sich dann quasi auf Null. Die Konfliktberechnung kann je nach Modell von den Zellen selbst oder an zentraler Stelle erfolgen.

4.1.6 Statistische Auswertung

Die Simulation braucht auch ein bewertbares Ergebnis. Für die Kontrolle, ob die Kreaturen sich auf allen Positionen der Umgebung aus $\mathbb{A}$ aufgehalten haben, gibt es zwei Möglichkeiten. Zum einen lassen sich Marker von der Position Brotkrumen gleich aufsammeln, zum anderen, wie bei Ameisen, Pheromone auftragen. Zur Aufzeichnung diene die Menge $\mathbb{V} \subseteq \mathbb{A}$.

Die erste Methode beginnt bei $\mathbb{V}_{\ominus} = \mathbb{A}$ und alle besuchten Stellen werden abgezogen, $\mathbb{V}'_{\ominus} := \mathbb{V}_{\ominus} \backslash \{p_i \mid i \in \mathbb{I}\}$. Alternativ lässt sich stattdessen ein Marker auftragen, so dass die

Menge $\mathbb{V}_\oplus$ größer wird. Hierfür beginnt $\mathbb{V}_\oplus$ mit der leeren Menge, $\mathbb{V}_\oplus = \varnothing$, um dann anschließend zu wachsen, $\mathbb{V}'_\oplus := \mathbb{V}_\oplus \cup \{p_i \mid i \in \mathbb{I}\}$.

Wie für die Folgeposition $p' \rightsquigarrow p$ gilt auch hier sowohl für $\mathbb{V}'_\ominus$ als auch $\mathbb{V}'_\oplus$, dass diese im Übergang eines Zeitschritts nach $\mathbb{V}_\ominus$ bzw. $\mathbb{V}_\oplus$ übergehen. Für die statistische Erhebung reicht eine der beiden Methoden aus. Im folgenden findet die subtraktive Methode mit $\mathbb{V}_\ominus$ Verwendung.

Um die Statistik zu vervollständigen, ist zusätzlich die Anzahl der Zeitschrittübergänge, die dem Begriff einer Generation entspricht, aufzuzeichnen. Zusammen mit $\mathbb{V}$ und der Anzahl der dafür benötigten Generationen g lässt sich der Erfolg von Algorithmen qualitativ bewerten.

4.2 Intention der Problembeschreibung

Für die Durchführung der Simulation und deren Bewertung gibt es verschiedene Betrachtungsweisen. Zum einen kann die Zeit, also die Anzahl der benötigten Generationen, als Maßstab dienen, zum anderen kann auch ein Zeitpunkt festgelegt werden, zu dem dann die Anzahl der bis dahin besuchten Positionen gewertet wird.

Eine Forderung kann auch sein, bestimmte Aufgaben zu erfüllen. Zum Beispiel Gegenstände verschieben, mit Regeln ähnlich dem Spiel Sokoban, bei dem ein Arbeiter Kisten in einen bestimmten, markierten Bereich schieben muss. Schlimmstenfalls bleiben die Kisten aber in einer Ecke hängen. Dann ist es dem Arbeiter nicht mehr möglich, die Kiste zu bewegen, da er an keine Seite zum Schieben herankommt.

Um die Aufgabe leichter zu gestalten, lassen sich die Kisten durch eine Art Klette ersetzten. Eine Kreatur, statt einem Arbeiter, braucht dann aber auch eine Möglichkeit, die Klette wieder loszuwerden. Entweder durch einen weiteren Ausgabewert in $\mathbb{Y}$ oder durch einen speziellen Zielbereich, den dann nur die Klette erkennen muss. In diesem Zielbereich kann dann die Klette vom Feld verschwinden, quasi in die dritte Dimension aufgesaugt werden.

Dies bedeutet dann für die Klette als dynamisches Objekt eine vermehrte Anzahl an Eingängen $\mathbb{X}$: Zum einen, ob sie sich an eine Kreatur anheften kann, zum anderen, ob sie sich auf einem Zielbereich befindet. Relevant ist auch, wie sie sich, immer an der Kreatur hängend, fortbewegen kann. Evtl. sollen beim Vorbeiziehen an anderen Kletten diese ebenfalls mitgenommen werden. Dies alles ist möglich und zu überlegen.

Alternativ zur Klette wäre ein Ball, bei dem die Kreatur zusätzlich noch den Ball in Bewegung setzen kann. Dieser Ball würde dann über das Feld rollen, bis ein Hindernis auftritt und eine andere Kreatur den Ball mitnimmt bzw. in eine andere Richtung bewegt. Im Falle des Hindernisses stellt sich noch die Frage, wie der Ball darauf reagiert. Zur Wahl stehen liegen bleiben oder – nach den Gesetzen der Physik – in entgegengesetzter Richtung zurückprallen, gemäß dem Impulserhaltungssatz, z. B. beschrieben in [Mes01].

Es zeigt sich, dass die sich bewegenden Objekte unterschiedliche Parameter für Eingangswerte, Ausgabewerte und Zustände haben können. Für einen ersten Entwurf werden nur die Kreaturen als dynamische Objekte verwendet, die allesamt gleich gestaltet sind. Dies vereinfacht die Gegebenheit.

Aber auch so ergeben sich noch zahlreiche Varianten einer Aufgabenstellung. So kann jede Kreatur für sich agieren oder mit den anderen kommunizieren, entweder über zu setzende Marken auf dem Feld oder per „Ferngespräch“ gezielt untereinander, alternativ auch per Funk direkt zwischen allen Kreaturen.

Als Schiedsrichter fungiert das beobachtende Feld. Sobald alle Positionen besucht sind, ist der Versuch erfolgreich verlaufen. Wenn sich jedoch nach einer bestimmten Anzahl an Generationen nichts verändert hat, ist davon auszugehen, dass sich auch weiterhin kein Erfolg ergibt. Eine derartige Konstellation ist dann als gescheitert zu betrachten.

Statt einem voll besuchten Feld ließe sich auch ein teilweise besuchtes Feld als Erfolg verbuchen, dazu muss nur eine Untergrenze existieren, ab der es so gilt. Die Anzahl der besuchten Positionen muss dann nur noch damit verglichen werden. Genauso gut kann dies auch nur ein Schwellenwert sein, ab dem dann die durchgeführte Simulation in einer Erfolgsliste erscheint. Auch kann die Aufgabe die Suche nach einem Weg durch ein Labyrinth sein, also von einem Start- zu einem Zielpunkt zu gelangen. Die Bewertungsmaßstäbe sind frei definierbar, sind aber bisher alle quantitativ formuliert, um einen vergleichbaren Maßstab zu haben.

4.3 Realisierung des Simulationssystem

Um eine weitere Forschung nicht nur in der Theorie, sondern auch in der Praxis durchführen zu können, sind Testsysteme erforderlich. Hierfür gibt es verschiedene Plattformen: Es kann zum einen im Rechner ein paralleler Rechenablauf simuliert werden, es kann aber auch direkt ein Spezialrechner gebaut werden – zum Beispiel unter Verwendung eines FPGA-Bausteins, der parallele Rechenabläufe ermöglicht. Um einen einfachen Start zu haben, empfiehlt sich die Verwendung einer geeigneten Testplatine (Evaluation Board), um nicht zusätzliche Probleme im Hardwaredesign und Bestückung mit hochgepackten Bauteilen zu haben.

Für die Umsetzung der Struktur gibt es verschiedene Ansätze – gleichgültig, ob für Hardware oder Software, das Prinzip ist für beide Arten anwendbar. Allerdings ist im Folgenden der Bezug eher zur Hardware dargestellt, da sich dort mehr Details und Problemfälle ergeben, so dass mitunter die Umsetzung in Software keine Unterschiede aufweist. In jedem Fall lassen sich die Ideen aber relativ einfach so auch in Software implementieren.

Für Software stellt sich der Unterschied eher zwischen einfachen Arrays und komplexen Strukturen bzw. Objekte dar, in der Hardware sind dies dann Register oder externe Speicher. Der Unterschied besteht in den zu speichernden Daten, was insbesondere in der objektorientierten Programmierung zur Geltung kommt.

4.3.1 Fundament Zellularautomat

Für die Simulation, bei der viele Dinge parallel geschehen können, bietet sich das Konzept des Zellularautomaten an. Dieses besteht aus einer Einteilung in einzelne Zellen, die einen eigenen, veränderlichen Zustand und bidirektionale Verbindung zu anderen Zellen haben können. Idealerweise sind diese Zellen in einem regelmäßigen, rechteckigen Raster ähnlich einem kartesischen Koordinatensystem angeordnet; die einzelnen Zellen sollen dann auch nur mit den direkten Nachbarzellen in Austausch stehen, dies ist in Abbildung 4.3 dargestellt. Durch diese Standardisierung kann die Berechnung in einfacher Form ohne großen Aufwand erfolgen. Für eine leichtere Referenzierung sind die Zellen mit Koordinaten versehen.

Eine Zelle hat dabei neben dem eigenen Zustand vier Eingänge als Verbindung zu den Nachbarzellen. Eine solche Verbindung kann den Zustand der Nachbarzelle übermitteln

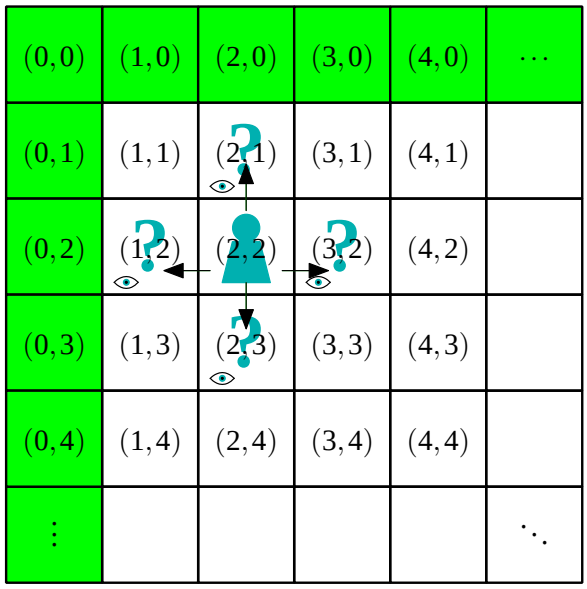


Abbildung 4.3: Nachbarschaftsmodell eines zellularen Automaten

oder auch nur einen bestimmten, ausgewählten Wert. Insgesamt entspricht die Charakteristik einer einzelnen Zelle der eines Automaten, wie in Kapitel 3 vorgestellt. Zusammen mit mehreren Zellen entsteht ein zellularer Automat. Dabei müssen die einzelnen Zellen nicht den gleichen Regeln gehorchen, können also unterschiedliche Automaten beherbergen.

Für die Aufteilung in statische und dynamische Objekte ist die Verwendung unterschiedlicher Automaten interessant. Hindernisse entsprechen einem einfachen Automaten ohne Zustände oder Eingänge, lediglich der Ausgang gibt seine stetige Blockierung bekannt. Die Automaten der übrigen Positionen entsprechen denen der größten Anforderungen der dynamischen Objekte. So zieht der resultierende Automat die Eingangsanzahl eines Objektes und die Zustandsanzahl eines anderen Objektes für seine Anforderung heran, um alle Automaten umsetzen zu können. Um die Eingänge und Ausgaben der Automaten dynamischer Objekte anzubinden, ist ein Mechanismus für Kollisionsbehandlung und Bewegung notwendig, basierend auf den Abschnitten 4.1.3 und 4.1.5.

Zu beachten ist, dass es zum einen den Automaten zur Umsetzung der dynamischen Eigenschaften gibt, zum anderen gibt es auch noch den Zustandsautomaten jeder einzelnen Kreatur bzw., allgemein, jedes dynamischen Objektes. Interpretieren lässt sich beides zusammengenommen als ein Automat, bei dem es einen internen Zustand s und eine Richtung r gibt, die zusammengesetzt den Gesamtzustand (s,r) des für die Umsetzung relevanten Automaten ergeben. Die Übergangsfunktion zur Berechnung des neuen Zustands ändert sich auch dementsprechend, um so die Umgebung mit einzubeziehen.

Basierend auf dem Prinzip, dass eine Zelle nur von anderen Zellen lesen, aber nicht auf diese schreiben darf, ist für eine Konfliktbehandlung ein erweitertes Nachbarschaftsmodell hilfreich. Statt nur eine Zelle weiter zu schauen, wie in Abbildung 4.3 dargestellt, blickt eine Kreatur auf die Nachbarzelle ihrer Drehrichtung und von dort aus in alle vier Richtungen. Will sich noch eine andere Kreatur auf die gewünschte Position bewegen oder ist die Wunschzelle ein Hindernis, so scheitert das Weitergehen, die Drehung erfolgt auf der aktuellen Position. Ist eine Bewegung jedoch möglich, so erhält der Automat über seinen Eingang x diese Information. Da jedoch die Kreatur sich danach nicht mehr auf dieser statischen Zelle befindet, muss die Zielzelle die notwendigen Informationen vor

dem Löschen kopieren. Dazu gehören Status und Drehung, die sich jedoch aufgrund der Übergangstabelle des Automaten ändern. Von daher ist die Speicherung der neuen Werte ausreichend. Wenn es unterschiedliche Automaten für die Kreaturen gibt, so ist auch ein Kopieren der Automaten, d. h. deren Übergangstabelle, nötig. Alternativ ist ein Index mit Verweis auf den zuständigen Automaten denkbar; dann müssen allerdings in jeder Zelle alle Automaten vollständig vorliegen.

4.3.2 Darstellungsdetails

Gerade bei einer Simulation sind die einzelnen Details der Berechnungsschritte interessant. Um nicht nur nackte Zahlen zu haben, sondern eine Umgebung wahrhaftig zu sehen, ist eine Darstellung des zweidimensionalen Feldes in Form einer Grafik wünschenswert. Diese kann verschiedene Gestaltungen aufweisen. Die Illustration analog zu Abbildung 4.3 ist nur eine Variante.

4.3.2.1 System mit Textausgabe

Für einfache Systeme ohne Grafikanzeige bietet sich die stilisierte Textausgabe mit Assoziationen zur Grafik an. Hindernisse erscheinen als Blöcke, Kreaturen mit ihrer Richtung r und internem Zustand s, noch nicht besuchte Positionen aus $\mathbb{V}_{\ominus}$ durch einen Punkt, als Beispiel zu sehen in Abbildung 4.4. Für die Darstellung ist es lediglich erforderlich, von jedem Objekt die Position und Eigenschaft zu haben. Für eine neue Zeile ist zusätzlich die Ausgabe von Wagenrücklauf und Zeilenumbruch (ASCII 13 und 10) erforderlich.

Mit diesem einfachen System kann jedes Terminalprogramm ein Feld von einer in Verilog HDL entwickelte Simulationsausgabe mit Übertragung über serielle Schnittstelle adäquat darstellen. Auch für einfache, konsolenorientierte Programme, vorzugsweise in C oder C++ entwickelt, bietet sich diese simple Repräsentation eines Feldes an.

```
[][][][][][][][][][][][][][][][][][][][][]
[]. .       . . . . . . . . . . . . . . []
[].     .     >2. . . . . . . . . . . . []
[]. . . . .     . . . . . . . . . . . . []
[][][][][][][][][][][][][][][][][][][][][]
```

Abbildung 4.4: Darstellung eines Feldes, basierend auf Textzeichen

4.3.2.2 Java-Programmierumgebung

Für eine anspruchsvollere Grafik mit zusätzlichen interaktiven Möglichkeiten mit visuellen Möglichkeiten der kontextbezogenen Auswertung bietet sich ein Programm, in Java entwickelt, an. Nachteilig ist dann die fehlende Steuerungsmöglichkeit über eine Kommandozeile, so dass mit dem Code keine Statistik über verschiedene Konstellationen von Umgebung, Kreaturanzahl und unterschiedlicher Algorithmen automatisiert erstellt werden kann. Dafür ist eine konkrete Einflussnahme während der Simulation möglich. Eine Bibliothek dafür ist mit [Bot06] entstanden, ein Beispiel ist in Abbildung 4.5 zu sehen.

4.3.2.3 Simulation mit PostScript-Ausgabe

Um erst nach einer Berechnung die Simulation zu betrachten, ist die Aufzeichnung des Ablaufs notwendig. Dafür ist ein später abrufbares Dateiformat notwendig. Dies könnte

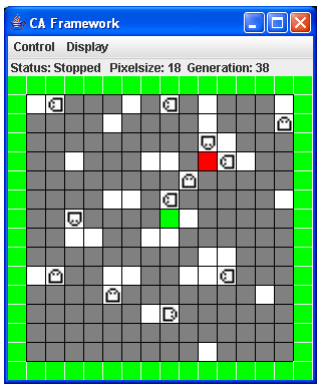


Abbildung 4.5: Simulationszwischenergebnis einer Java-Simulation

durch ein selbst entwickeltes Java-Programm mit einem eigenen Dateiformat geschehen. Alternativ ist ein Dateiformat, das auf verschiedenen Rechnersystemen mit Standardprogrammen gelesen werden kann.

Letztere Variante bietet die Programmiersprache PostScript von Xerox, mit der es möglich ist, vektororientiert eine Seite beliebiger Größe zu beschreiben. Mit üblichen Konvertern ist auch eine Umwandlung in ein beliebig anderes Dateiformat möglich, um zum Beispiel auch Videos graphisch ansprechend zu erstellen. Mit Hilfe von [Sök92] ist eine neue Schrift entstanden, mit der nur ein einziges Zeichen das komplexe Aussehen einer Kreatur repräsentiert. Dadurch entsteht mit einfachen Mitteln ein Abbild einer Generation; Beispiele sind in Anhang C und in Abbildung 4.6 zu sehen.

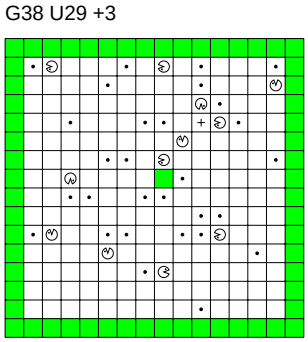


Abbildung 4.6: Simulationszwischenergebnis einer PostScript-Ausgabe

4.3.2.4 Analysewerkzeuge

Um verschiedene Algorithmen und Umgebungen miteinander vergleichen zu können, ist eine statistische Basis notwendig, in der die Ergebnisse verschiedener Simulationen zusammengetragen sind. Dafür ausreichend sind die Daten von verwendeten Algorithmen, Identifizierung der Umgebung, Anzahl der Kreaturen und sonstiger variabler Dinge, Anzahl der Generationen und Anzahl der besuchten Positionen, evtl. auch nur auf Vollständigkeit hin ausgewertet. Zusätzlich kann die Anzahl der aufgetretenen Konflikte interessant sein.

Im kommaseparierten Format CSV ausgegeben, lassen sich die Daten zum einen mit einem Tabellenkalkulationsprogramm betrachten und sortieren, zum anderen kann mit einfachen Skripten, z. B. auf der Basis der interpretierten Sprache Perl und einer Anleitung wie [VC97], automatisiert eine Auswertung und Zusammenfassung erfolgen, um so

auch die uninteressanten Daten auszufiltern. Dies ist das eigentliche Fundament, um an relevante Ergebnisse zu gelangen.

4.3.3 Konzepte

Mit Abschnitt 4.1 sind mögliche Implementierungsdetails aufgeführt, die dennoch Spielraum für unterschiedliche Interpretationen lassen. Im Folgenden sind Variationen dargestellt, um eine effektive Simulationsumgebung zu erhalten.

4.3.3.1 Klassische Variante eines Feldes

Nach dem klassischen Modell ist in jeder Zelle alles Notwendige gespeichert. Hierzu zählen die Existenz von Hindernissen *h*, das Vorhandensein von Kreaturen *o* sowie deren aktueller Zustand *s* und deren Richtung *r*. Ebenso ist auch die Automatenübergangstabelle in jeder Zelle gespeichert, die Folgezustand und Ausgabewert vorgibt. Um nur wenige Daten bei Voranschreiten der Kreatur kopieren zu müssen, empfiehlt sich, die Automaten statisch an jeder Position vorzuhalten.

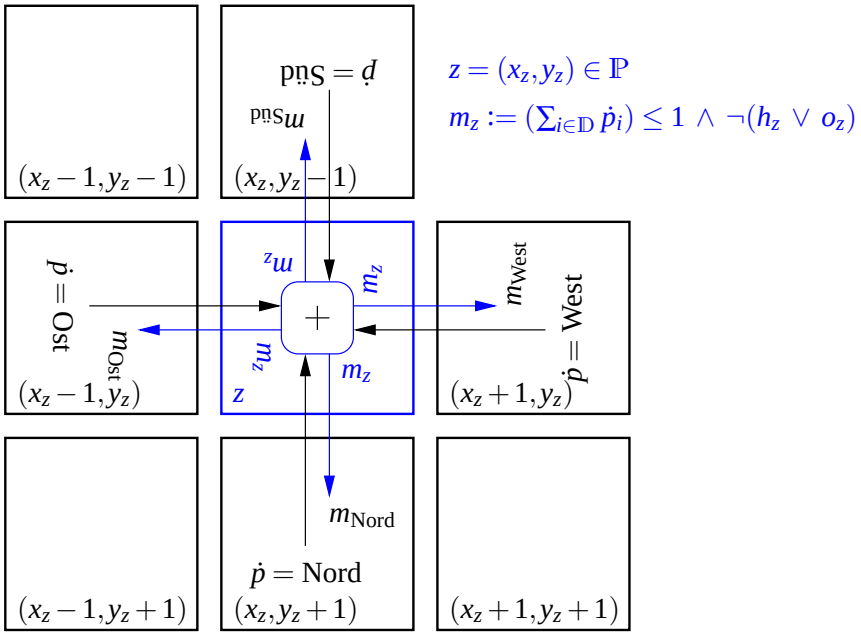


Abbildung 4.7: Ausschnitt der Schaltung zur Konfliktlösung für eine Zielposition *z*

Das Ablehnungssignal aus der jeweiligen Richtung werten die betroffenen Objekte aus. Anschließend ist klargestellt, welche Objekte sich bewegen dürfen. In Hardware sind dafür einfache Schaltkreise ausreichend, das Prinzip illustriert Abbildung 4.7. Wenn sich auf einer Zelle eine Kreatur befindet und ihre Richtung *r* auf die Zielposition *z* weist, sendet diese Zelle ein positives Anforderungssignal an die Zelle *z*. Diese Zelle *z* wertet alle ankommenden Signale aus. Ist die Summe maximal ein Signal, so sendet diese an alle umliegenden Zellen ein Bestätigungssignal. Dieses Signal unterbleibt, wenn sich auf *z* eine Kreatur oder ein Hindernis befindet. Empfängt eine Kreatur dieses Bestätigungssignal aus der richtigen der vier Richtungen, so kann sich diese erfolgreich weiterbewegen. Andernfalls ist *m* negativ und die Kreatur dreht sich nur auf der Stelle und wechselt ihren Zustand gemäß Zustandsübergangstabelle für den Teil $x = 0$.

Da mit Software keine echte massiv-parallele Umsetzung möglich ist, gibt es zwei Felder. Eines enthält eine aktuelle, nur lesbare Variante, das andere Feld eine zukünftige, nur schreibbare Version, deren Bedeutungen nach einem Generationsübergang wechseln, ohne die Daten kopieren zu müssen. Für die eigentliche Berechnung der Regel kann eine Funktion mit der Zielposition als Parameter die Anzahlbestimmung und somit die Ermittlung von m vornehmen. Die gleiche Funktion kann auch die Zelle z mit sich selbst im Parameter verwenden, um herauszufinden, ob sich in der nächsten Generation eine Kreatur auf ihr befindet. Für eine Implementierung eignet sich eine objektorientierte Programmiersprache, wie z. B. Java oder C++ gemäß [Str92].

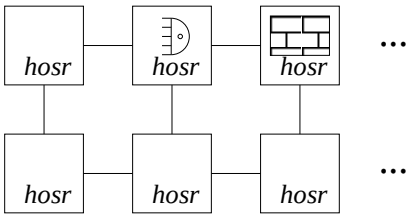


Abbildung 4.8: Einheitliche Umsetzung von Kreatur und Feld

Durch die Abfragetechnik entsteht zwischen den einzelnen Zellen eine Verbindungsstruktur, so dass jede Zelle mit ihren vier Nachbarzellen jeweils über ein eigenes Anforderungssignal und ein eigenes Bestätigungssignal verknüpft ist. Zu sehen ist das Resultat in der schematischen Abbildung 4.8. Jede Zelle speichert dabei, ob sie selbst ein Hindernis h ist oder ob sie eine Kreatur o aufgenommen hat, dabei haben dann auch Zustand s und Richtung r eine Relevanz.

Die Speicherung der Werte für $\mathbb{V}$ erfolgt in jeder Zelle nach dem Prinzip $v' := v \vee o$. Um ein Gesamtbild zu erhalten, gibt es zusätzlich einen globalen Zähler, der die Veränderungen von v und v' wahrnimmt und aufsummiert. Mit der Gesamtzahl freier Positionen verglichen, ergibt sich die Quote der bereits besuchten Positionen.

Bevor alles beginnen kann, ist eine Phase der Initialisierung notwendig. Die Daten können in einem vorher gefüllten Speicher gelagert werden. Sequentiell ausgelesen verteilen sich die Daten auf die jeweiligen Zellen. Damit dies funktioniert, ist eine andere Struktur der Verbindungen notwendig. Statt untereinander im zweidimensionalen Raster verbunden, sind die Zellen nun für diesen Zweck wie an einer Perlenkette aufgereiht.

Die Gültigkeit signalisiert ein von Zelle zu Zelle weitergereichtes Token. Daraus ergibt sich die in Abbildung 4.9 dargestellte Struktur, die zusätzlich auch das gleich ablaufende Ausgabeverfahren enthält. Einzige Ergänzung ist die notwendige Pause bei vollen Ausgabepuffer. Im übrigen verwenden das gleiche Prinzip auch die nachfolgenden Architekturen in den Abschnitten 4.3.3.2 und 4.3.3.3.

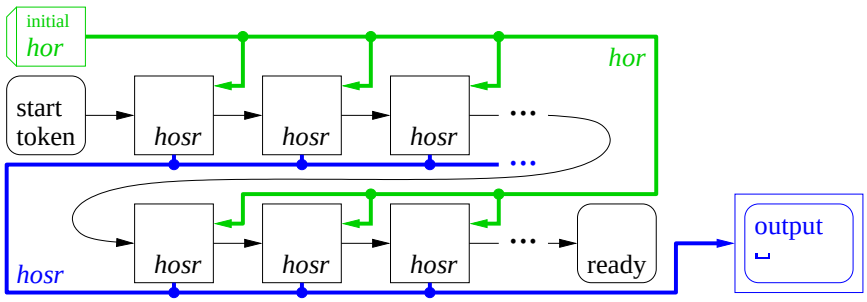


Abbildung 4.9: Verbindungsstruktur während der Initialisierungs- und Ausgabephase

4.3.3.2 Aufteilung in Funktionsgruppen

Eine Möglichkeit, die Anzahl der Berechnungen zu reduzieren, ist, eine Unterteilung in Aktivitätsbereiche vorzunehmen. Nur die Bereiche, in denen eine Änderung zu erwarten ist, werden berechnet. In [Sch98] ist dies für das aus [Con71] bekannte „Game of Life“ durchgeführt, erstmals erschienen in [Gar70]. Für diese Problemstellung ist jenes Verhalten allerdings nicht effektiv, da sich zu viele Stellen ändern. Wenn hingegen nur in der Nähe einer Kreatur eine Berechnung erfolgt, so ist Ähnliches wie mit den Bereichen erreicht. Die Kreaturen selbst steuern dazu die notwendigen Positionen ohne großen Aufwand, indem sie um ihre Position wissen und diese auch den Gegebenheiten entsprechend anpassen.

Dies führt zu einer Auslagerung der Kreaturen aus dem Feld, das nur noch als Speicher für die Hindernisse dient, implementiert als einzelne Register für gleichzeitigen, parallelen Zugriff. Die Auswahl der Zelle zur Hinderniserkennung erfolgt über Multiplexer, der als Eingang alle Hindernisinformationen aller Positionen $h_{\mathbb{P}}$ erhält und über die Zielposition $\dot{p}_i$ eine Auswahl trifft, dargestellt in Abbildung 4.10. Dieses Ergebnis $\neg h_{\dot{p}_i}$ gelangt zu einer Kollisionserkennung, die auch die Positionen und Ziele aller anderen Kreaturen erhält und gemäß Algorithmus 4.1 arbeitet. Aus dem Vergleich aller Werte empfängt jede Kreatur i ihr eigenes Erfolgssignal m_i, um so Folgezustand s'_i und -position p'_i sowie die nächste Richtung r'_i selbstständig zu berechnen.

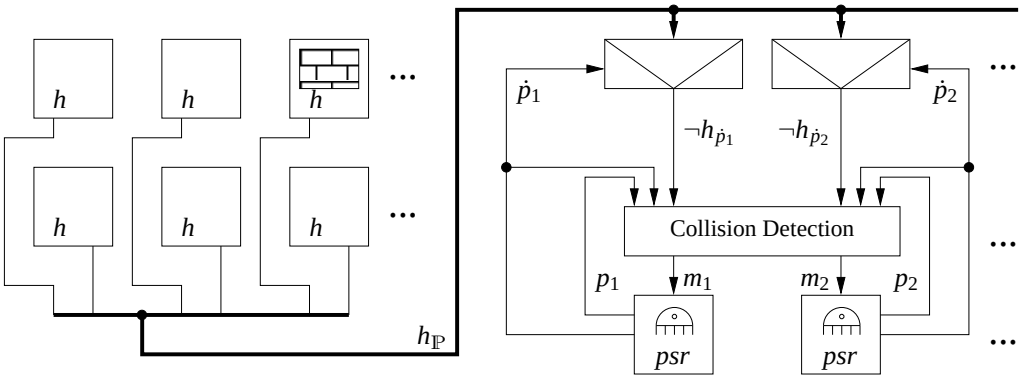


Abbildung 4.10: Einfaches Feld, separate Kollisionserkennungslogik und Kreaturen

Zur Erstellung der Statistik ist für die Zellen des Feldes zusätzlich noch die Information notwendig, ob sich auf ihr eine Kreatur befindet. Also neben $h_{\mathbb{P}}$ in die eine Richtung befinden sich noch die Positionen p aller Kreaturen in die andere Richtung auf dem Verbindungsbus. Jede Zelle gleicht ihre eigene Position mit den Positionen aller Kreaturen ab, um so den Wert o zu erhalten. Hierfür muss jede Zelle ihre eigene Position kennen. Nach dem vorherigen Verfahren aus Abschnitt 4.3.3.1 erfolgt dann die Berechnung der besuchten Positionen.

4.3.3.3 Verlagerung der Berechnung

Statt nun den Kreaturen alles zu überlassen, ist alternativ eine Kollisionsberechnung nach Abbildung 4.7 denkbar. Dies hat zur Folge, dass sich die Position p_i und Richtung r_i der Kreaturen in die Feldzellen verlagert, die Kreaturen selbst geben nur ihre Drehrichtung d_i bekannt, nachdem sie ihre Bewegungsfreigabe m_i erhalten haben.

Die Angabe, welche Kreatur sich auf welcher Feldzelle befindet, muss ebenfalls gespeichert werden. Eine effiziente Lösung stellt ein dekodierter Binärwert o mit $|\mathbb{I}|$ Ziffern

Algorithmus 4.1: Kollisionsberechnung

```
1  wire [I : 1] m;    // I = Anzahl der Kreaturen |𝕀|
2  generate genvar i, j;
3      wire [I : 1] collision [I : 1];
4      for (i = 1; i <= I; i = i + 1) begin: creatureA
5          assign collision[i][i] = h[front[i]];
6          for (j = i + 1; j <= I; j = j + 1) begin: creatureB
7              wire fc = (front[i] == front[j]);
8              assign collision[i][j] = fc || (front[i] == pos[j]);
9              assign collision[j][i] = fc || (front[j] == pos[i]);
10         end
11         assign m[i] = ~|collision[i];
12     end
13 endgenerate
```

dar, bei dem genau dann das Bit i gesetzt ist, wenn auch die Kreatur i sich auf der Zelle befindet. Die Position p ist im Gegenzug nicht mehr erforderlich. Die Richtung r bleibt universell für eine potentiell vorhandene Kreatur vorhanden, die Hindernisspeicherung h bleibt unverändert existent.

Hinzu kommt wieder die Verbindungsstruktur, so dass Abbildung 4.11 als Gesamtbild entsteht. Eine Feldzelle kann nun aufgrund ihrer Informationen bei der zugehörigen Nachbarzelle einen Bewegungsübergang anfordern. Die gefragte Zelle wertet alle diese Anforderungssignale aus und sendet ein entsprechendes Bestätigungssignal. Passend zu dem Bit o_i leitet die ursprüngliche Zelle das Ergebnis über die Leitung m_i an die Kreatur weiter, die dann ihrerseits die Drehung d_i ausgibt und den Folgezustand s' setzt. Die Ermittlung für die Besuchsstatistik ist kein Problem, es gilt dann $v' := v \vee \bigvee_{i \in \mathbb{I}} o_i$.

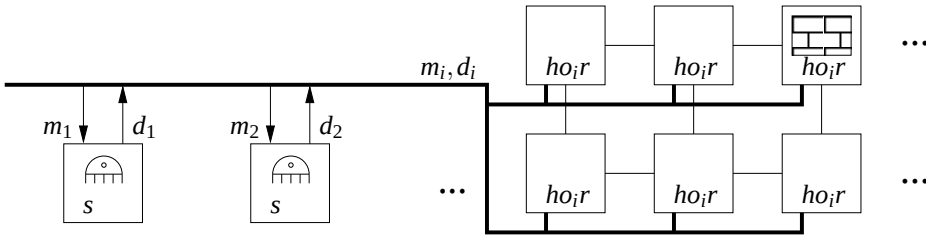


Abbildung 4.11: Einfache Kreaturen für komplexe Berechnung im Feld

Anstatt nun die Kreaturdaten im Feld zu speichern, ist auch eine Zuordnung direkt zu den Kreaturen möglich. Lediglich die Kollisionsberechnung soll in den Zellen erfolgen. Dadurch verlagert sich der Bedarf an Speicherplatz für ungenutzte Drehungen in Bedarf für einen breiteren Datenbus, so dass die Positionsdaten und Richtungen aller Kreaturen statt nur der Drehungen auf dem Bus anliegt, zu sehen in Abbildung 4.12.

Für die Kollisionserkennung ist nun auch eine Nummerierung der Feldzellen notwendig, nicht nur für die Erstellung der Statistik wie in Abschnitt 4.3.3.2. Um die Optimierungen des Syntheseprogramms auszunutzen, empfiehlt sich, diese Nummern statisch und nicht erst zur Laufzeit der Simulation zu vergeben.

Die Ermittlung einer Kollision ist identisch mit der zu Beginn dieses Abschnitts. Allerdings ist für die statistische Auswertung o_i nicht verfügbar, stattdessen berechnet

sich o für eine Zelle z mit $\exists_{i \in \mathbb{I}} : z = p_i$, so dass die ursprüngliche Version für v' wieder Verwendung findet. Die übrigen Variablen verhalten sich in gewohnter Weise.

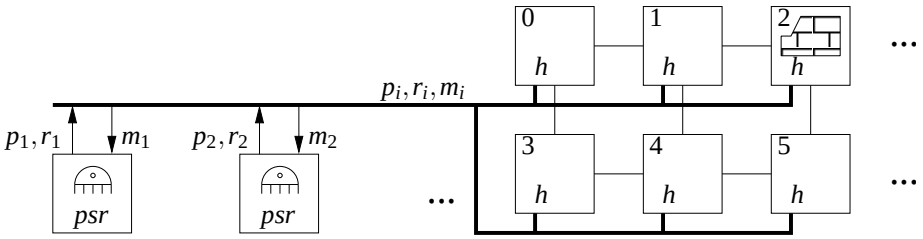


Abbildung 4.12: Kollisionserkennung im indizierten Feld

Dieses Konzept der Berechnung aus Sicht der Kreaturen statt aufgrund der Feldgröße ist auch für die Softwareentwicklung einsetzbar. Statt nun alle Positionen zu überprüfen und zu berechnen, wie bei einem klassischen zellularen Automaten, erfolgt die Berechnung nur an den Zielpositionen der Kreaturen. In einem ersten Schritt markiert jede Kreatur ihre Zielposition; ist diese bereits markiert, so wandelt sich die Markierung in einen Konflikt. Danach kontrolliert jede Kreatur ihre Zielposition und bewegt sich bei Konfliktfreiheit auf die Zielposition. Abschließend löscht die Steuerung alle Markierungen und das zellulare Feld steht für die nächste Generation bereit. Dies ist vorteilhaft für Simulationen, bei denen mehr mögliche Positionen als Kreaturen vorhanden sind, wie es bei dieser Aufgabenstellung stets der Fall sein sollte.

4.3.3.4 Passive Umgebung

Alternativ zur Speicherung der Hindernisse in einem Registerfeld bietet sich ein echter Speicher an. Ein ROM ist hierfür ausreichend. Allerdings stehen auf einem FPGA nur Dual-Port-Speicher zur Verfügung, so dass mehrere ROMs mit gleichem Inhalt benötigt werden, um einen voll-parallelen Zugriff aller Kreaturen gleichzeitig zu ermöglichen.

Ungeachtet dieser Notwendigkeit entsteht eine Schaltung gemäß Abbildung 4.13, die der Abbildung 4.10 entspricht, abgesehen von der Hindernisbestimmung. Die Kollisionsberechnung ist ebenfalls vereinfacht, es werden nun lediglich die Positionen miteinander verglichen, die Hindernisbearbeitung ist ausgelagert, funktional entspricht dies dem vorherigen Vorgehen.

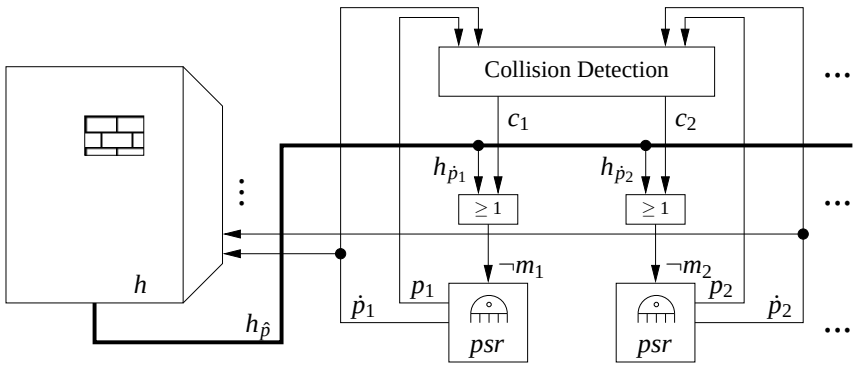


Abbildung 4.13: Feld im ROM, separate Kollisionserkennungslogik und Kreaturen

Durch das geänderte Design ergeben sich auch noch andere Änderungen. So liegen am Bus zwischen Feld und Kreaturen nicht mehr alle Hindernisdaten $h_{\mathbb{P}}$ an, sondern

lediglich die notwendigen für die Zielpositionen, zusammengefasst dargestellt mit $h_{\hat{p}}$. Für ein einzelnes Ergebnis $h_{\dot{p}_i}$ erfolgt die kürzere, objektindizierte Notation $\dot{h}_i$ mit der gleichen Bedeutung.

Die Daten der Kollisionserkennung teilen sich nun auch in c_i für die eigentliche Kollision mit anderen dynamischen Objekten und $\dot{h}_i$ für die Kollision mit einem Hindernis. Das Ergebnis für m_i entsteht, wie zuvor auch, aus einer Oder-Verknüpfung der beiden Teilergebnisse mit anschließender Invertierung, also $m_i = \neg(\dot{h}_i \vee c_i)$.

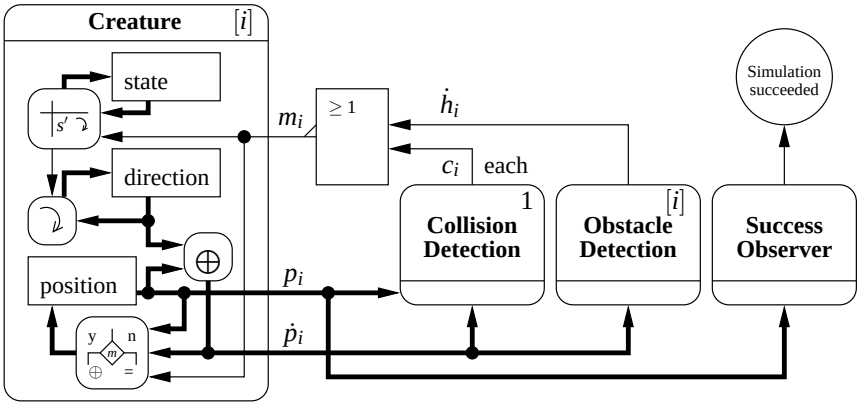


Abbildung 4.14a: Datenfluss aus Sicht einer Kreatur

Durch die geänderte Hinderniserkennung ergibt sich auch für die Besuchsstatistik eine andere Anforderung. Diese ist nun speziell als eigener Prozess zu implementieren. Auch die anderen Feinheiten sind unterschiedlich. Die einzelnen, für ein Gesamtkonzept notwendigen Schritte sind im folgenden mit den Teilabbildungen 4.14 aufgeführt, die Verbindungen untereinander zeigt die Übersichtsabbildung 4.15 anhand von vier Kreaturen.

Beginnend mit einer Kreatur stellt sich das System wie in Abbildung 4.14a dar. Die Kreatur i beinhaltet neben dem Zustand s (*state*) und der daran angeschlossenen Zustandstabelle die Richtung r (*direction*), die Berechnung der sich daraus ergebenden Drehung sowie die aktuelle Position p_i (*position*) und das Ziel $\dot{p}_i$. Der Multiplexer zum Setzen der Folgeposition p' ist ebenfalls enthalten, gesteuert wird er von dem Eingang m_i. Die Positionen entsprechen dabei den Speicheradressen, eine Bewegung nach unten entspricht demnach einer Addition der Breite X auf die aktuelle Position p, also $p' = p + X$. Ein Schritt nach rechts hat lediglich die Inkrementierung um Eins zur Folge, dies resultiert zu $p' = p + 1$.

Collision Detection
$c_i \Leftrightarrow \left(\dot{p}_i \notin \{p_1, p_2, ..., p_{i-1}, p_{i+1}, ..., p_{\lvert\mathbb{I}\rvert}, \dot{p}_1, \dot{p}_2, ..., \dot{p}_{i-1}, \dot{p}_{i+1}, ..., \dot{p}_{\lvert\mathbb{I}\rvert}\}\right)$

Abbildung 4.14b: Modul Konfliktberechnung

Für jede der $|\mathbb{I}|$ Kreaturen ist dieser Mechanismus vorhanden. Das Ergebnis c_i aus der Konfliktberechnung, zu sehen in Abbildung 4.14b, folgt dem gleichen Prinzip wie Algorithmus 4.1, lediglich die Darstellung ist eine andere. Für jede Zielposition $\dot{p}_i$ erfolgt ein Vergleich mit allen anderen Positionen p_j und Zielpositionen $\dot{p}_j$ mit $i \neq j \in \mathbb{I}$.

Die andere Bedingung für das Vorankommen m_i ist die Hinderniserkennung mit einem Ergebnis in h_i. Dieses stammt aus einem Speicher, der für jede mögliche Position

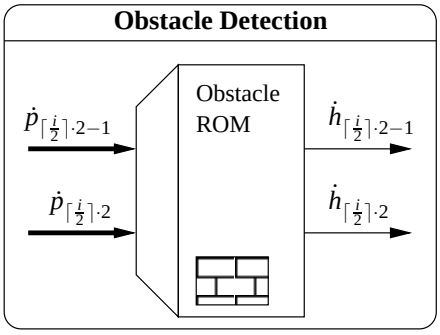


Abbildung 4.14c: Modul Hinderniserkennung

$\dot{p}$ ein Bit liefert, ob sich an dieser Position ein Hindernis befindet. In den verwendeten FPGAs sind Dual-Port-Speicher integriert, bei denen zwei getrennte Adresseingangsleitungen und zwei Datenleitungen vorhanden sind. Da sich die Hindernisse nicht dynamisch ändern können, ist ein Nur-Lesen-Speicher (*ROM*) ausreichend.

Dadurch ist es möglich, an einen Speicher zwei Kreaturen anzuschließen. Die Abbildung 4.14c zeigt die Verteilung für aufeinander folgende Kreaturindizes, beginnend mit einem ungeradem Index. Damit ist die Durchführung der Simulation gewährleistet, die Erkennung des Erfolgs fehlt aber noch.

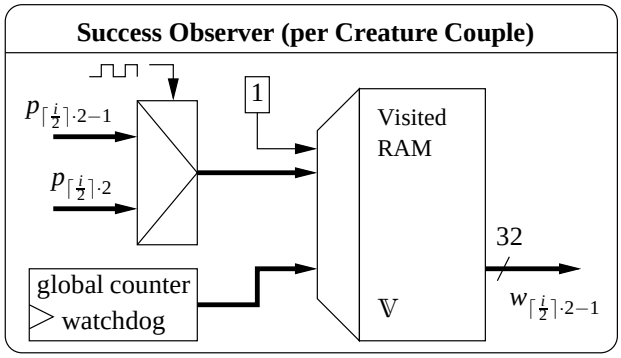


Abbildung 4.14d: Datenaufnahme für die Besuchsstatistik

Die Erfolgsmessung bedarf der Speicherung besuchter Felder. Zum einen hinterlassen die Positionen p der Kreaturen eine Spur im Speicher, zum anderen ist der Speicher auszulesen, um einen Erfolg zu erkennen. Auch hierfür stehen Dual-Port-Speicher zur Verfügung, die ein gleichzeitiges Lesen und Schreiben mit unterschiedlichen Bitbreiten ermöglichen. Die Abbildung 4.14d zeigt das Zusammenspiel.

Da eine Kreatur genau eine Position hat, reicht ein Bit zum Schreiben aus. Eine größere Bitbreite hätte zur Folge, dass zuerst die Daten zu lesen, dann ein Bit maskiert zu verändern und abschließend zu schreiben ist. Daher ist die mögliche Hardwarearchitektur von einem Bit ideal für einen geringen Aufwand.

Es kann zwar kein Schreibkonflikt mit dem zu schreibenden Bit entstehen, wenn zwei Kreaturen sich einen Speicher teilen, trotzdem ist dies aus technischen Gründen nicht möglich. Von daher teilen sich die Kreaturen den Speicher zeitlich getrennt. In der ersten Phase schreibt die Kreatur mit ungeradem Index, in der zweiten dann die mit geradem Index. In einer dritten Phase, nach Abschluss der Berechnung von Zielposition sowie Kollisions- und Hinderniserkennung, erfolgt das Setzen von Position, Richtung und Zu-

stand. So ist die Verzögerung des Speicherbausteins durch die erforderlichen Register zur Pufferung von Adresse und Ausgabedaten berücksichtigt.

Neben dem Beschreiben ist auch das Auslesen des Speichers erforderlich, um den Besuchsstatus $\mathbb{V}$ überblicken zu können. Da nur die Gesamtheit des Speichers erforderlich ist, müssen alle Speicheradressen ausgelesen werden. Um dabei die Anzahl der Adresszugriffe gering zu halten, empfiehlt sich eine andere Bitbreite. Für den ausgewählten Baustein Altera Cyclone I hat sich eine Datenbreite von 32 Bit als Optimal erwiesen.

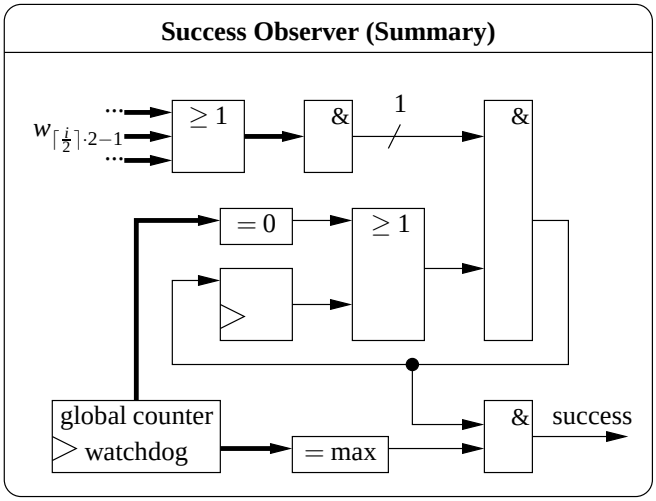


Abbildung 4.14e: Auswertung der Besuchsstatistik

Befinden sich die Hindernisse $\mathbb{A}$ initial im Speicher, so kann eine einfache Und-Zusammenfassung aller Bits je Position aussagen, ob bereits alle Zellen besucht wurden. Da sich jedoch in einem Speicher nur die Werte von zwei Kreaturen befinden, ist erst eine Zusammenfassung aller dafür verwendeten Speicher erforderlich.

Dies erfolgt mit einem Zähler, der wie ein Wachhund durch den Speicher läuft und alle Werte beobachtet, in den Abbildungen 4.14d und 4.14e durch den einen globalen Zähler *watchdog* repräsentiert.

Die Daten aus den verschiedenen, an die Kreaturen angebundenen Speicher der Besuchsstatistik gelangen zu einem Oder-Gatter, in der Abbildung 4.14e sind dies alle Werte von *w*. Dort ist dann für jede Position erstmals eine Aussage über einen erfolgreichen Besuch möglich. Da jedoch nur alle Positionen zusammengefasst relevant sind, erfolgt eine reduzierende Und-Verknüpfung, um zu erfahren, ob alle Bits von *w* des aktuellen Positionsbereichs gesetzt sind. Dieses Ergebnis fließt dann in ein Register mit den vorherigen Positionen ein, so dass eine Aussage bis zur aktuellen, durch den Zähler *watchdog* bestimmten Adresse möglich ist.

Statt $\mathbb{A} = \mathbb{V}$ mit allen auftretenden Lücken in den Positionen durch die Hindernisse zu bestimmen, ist es einfacher, einen Test auf $\mathbb{V} \cup \mathbb{H} = \mathbb{P}$ durchzuführen. Daher ist ein weiterer Speicher notwendig, der die Hindernisse $\mathbb{H}$ nach dem gleichen Prinzip wie die Besuchsdaten enthält. Nur der Zähler *watchdog* liest diesen Speicher aus, eine Veränderung der enthaltenen Werte findet nicht statt.

Wenn der Zähler *watchdog* seinen maximalen Wert erreicht hat, ist die Aussage $\mathbb{V} \cup \mathbb{H} = \mathbb{P}$ möglich, das Ergebnis stellt die Leitung „success“ dar. Um nun den Ablauf effizienter zu gestalten, bietet sich an, vor Erreichen des Maximalwertes den Zähler zurückzusetzen, sobald ein Erfolg nicht gewährleistet werden kann, also mindestens ein

Bit bzw. eine Position nach der Zusammenfassung den Wert Null aufweist. In diesem Fall veranlasst die Steuerung ein Zurücksetzen des Zählers auf die erste Adresse.

Eine weitere Optimierung steht zur Verfügung, wenn der Auswertungsvorgang unabhängig von der Simulationsdurchführung abläuft. Ein Schreiben und Lesen der gleichen Adressen in einem Speicher stellt kein Problem dar, schlimmstenfalls werden alte, noch nicht gesetzte Werte ausgelesen, so dass sich die Analysephase verlängert. Dies hat aber auch den Nachteil, dass sich die exakte Generation nicht mehr bestimmen lässt, zu der die Kreaturen erfolgreich alle Positionen besucht haben. Die Unschärfe des Wertes beträgt maximal die Anzahl Schritte, um alle Beobachtungswerte auszulesen, geteilt durch die Anzahl der Takte zur Berechnung einer Generation. Mit den gewählten Werten drei Takte für eine Generation und einen 4 096 Bit umfassenden Speicher, der sich mit 32 Bit parallel auslesen lässt, ergibt sich ein maximaler Fehler von $\lceil \lceil 4096 \div 32 \rceil \div 3 \rceil = 43$ Generationen, nach denen die Simulation zu spät endet.

Statt nun drei Takte für eine Generation zu benötigen, könnte sich der Multiplexer zugunsten eines geringeren Speicherbedarfs für mehr Kreaturen aufweiten, um so größere Felder zu ermöglichen. Die Anzahl benötigter Takte erhöht sich entsprechend der angeschlossenen Kreaturanzahl, wobei ein Takt für Auslesen und Auswerten des Speicherzugriffs der letzten Kreatur in der Taktfolge reserviert sein muss. Der Speicher für die Hinderniserkennung benötigt zusätzlich auch einen Multiplexer, da kein dritter Zugang zum Speicher möglich ist und somit auch hier eine Reihenfolge entsteht, allerdings für zwei Kreaturen parallel. Geht es soweit, dass sich alle Kreaturen nur noch jeweils einen Speicher für Hinderniserkennung und Besuchsstatistik teilen, bietet sich ein anderes Verfahren in Form eines Busses an, behandelt in Abschnitt 4.3.3.5.

Um eine Simulation zu wiederholen, brauchen die Speicher ihre ursprünglichen Werte. Das Reinitialisieren des Speichers für die Besuchsstatistik erfolgt über einen Adresszähler durch Setzen aller Zellen auf Null, der zusätzliche Speicher für die Hindernisse bleibt unverändert. Die Initialisierung der einzelnen Werte für die Kreaturen erfolgt über im Programmtext angegebene Parameter. Ein Signal löst für alle Kreaturen gleichzeitig den Kopiervorgang der Konstante zum Register für Zustand, Richtung und Position aus.

Die Zusammenfassung der Komponenten zeigt Abbildung 4.15, bei der eine hohe Parallelisierung trotz geringerem Ressourcenverbrauch gewährleistet ist. Benötigt eine Simulation mehr Kreaturen als nach dem bisherigen Verteilungsschema Ressourcen zur Verfügung stehen, vervielfacht sich die Mehrfachnutzung der Speicher, so dass z. B. vier Kreaturen an einen Speicher angeschlossen sind. Dadurch erhöht sich auch die Anzahl notwendiger Takte für die Berechnung einer Generation.

In der dargestellten Konfiguration benötigt das System unterschiedliche Speicher. Zum einen sind dies Nur-Lese-Speicher für die Hinderniserkennung, zum anderen Speicher mit parallelem Lese- und Schreibzugriff für die Besuchsstatistik, wobei bei letzterem die Speicherbreiten unterschiedlich sind: 1 Bit zum Nur-Schreiben, 32 Bit zum Lesen und zum (re-)initialen Schreiben. Beide Speicher benötigen aber die gleiche Anzahl an Bits, nämlich entsprechend der Feldgröße für $X \cdot Y$ Positionen.

Die verwendete Hardware-Plattform bietet Speicher in der Größe von 4 096 Bits an, so dass sich $\lceil \frac{X \cdot Y}{4096} \rceil$ Speicherblöcke für ein Feld ergeben. Zum Einsatz kommen davon zwei Mal $\lceil \frac{|\Pi|}{2} \rceil$ Felder jeweils für Hindernisse und Statistik in unterschiedlicher Bitbreitenkonfiguration sowie ein zusätzliches für die Hindernisse bei der Besuchsstatistik, mit

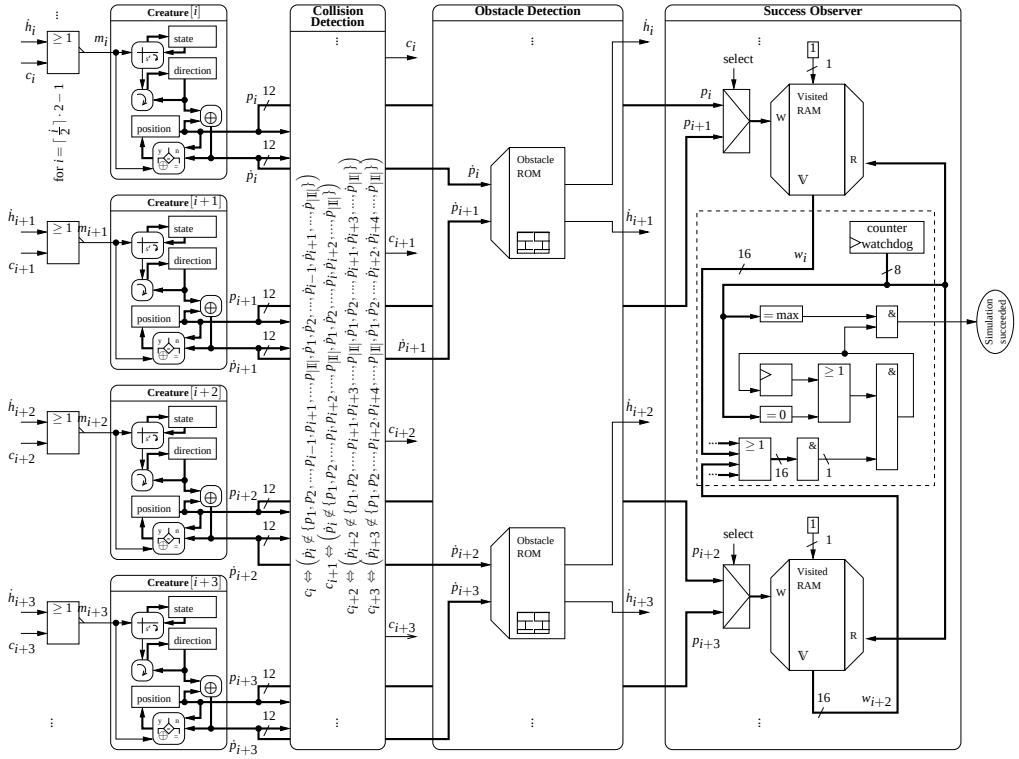


Abbildung 4.15: Zusammenhänge der speicherbasierten Architektur

Konfiguration analog zur Statistik und dem Inhalt der Hindernisauswertung. Insgesamt ergeben sich somit

$$\left\lceil \frac{X \cdot Y}{4\,096} \right\rceil \cdot \left(2 \cdot \left\lceil \frac{|\mathbb{I}|}{2} \right\rceil + 1 \right) \text{ Speicherblöcke,}$$

wenn sich jeweils zwei Kreaturen einen Speicher teilen.

Auch die Durchführung mehrerer Simulationen gleichzeitig stellt kein Problem dar, wenn ausreichend viele Kreaturen verfügbar sind und diese nur auf einem abgegrenzten Gebiet agieren können. Dadurch ist – insbesondere bei kleineren Feldgrößen – der Speicher besser ausgenutzt, da dieser eine Minimalgröße von 4 096 Bytes hat und sich so eine quadratische Feldgröße von $64 \cdot 64$ Zellen ergibt.

4.3.3.5 Bussystem

Statt mit Multiplexern und verschiedenen Speichern zu arbeiten, ist Gleiches mit nur einem Speicher für Hindernisse und einem Speicher für die statistische Auswertung möglich. Durch eine andere Arbeitsweise ergibt sich auch eine andere Infrastruktur für die einzelnen Kreaturen und dadurch evtl. auch eine andere Abhängigkeit für den Gesamtressourcenverbrauch.

Da nun alle Kreaturen am gleichen Speicher angeschlossen sind, wäre eine Steuerung mittels Multiplexern zu aufwendig. Stattdessen kommt für die Durchführung der Berechnung das Prinzip des Tokens zum Zuge, ähnlich dem der Initialisierung aus Abbildung 4.9. Die Kreatur, die ein Token inne hat, darf auf einem Bus Daten für Position p und Zielposition $\dot{p}$ senden. Daraufhin liegt das Ergebnis aus dem Speicher für $h_{\dot{p}}$ auf dem Bus und kann von der Kreatur gelesen werden. Durch die standardmäßige Pufferung

der Ein- und Ausgabedaten des Speichers verzögert sich das Ergebnis auf dem Bus um zwei Takte relativ zur Anfrage. Die aktive Kreatur kann jedoch das Token direkt an die benachbarte Kreatur weitergeben. Dadurch entsteht eine Art Pipeline zwischen Speicheranfragen und Auswertung.

Die Abbildung 4.16 stellt das Prinzip des Bussystems schematisch dar. Dort ist das inkrementelle Verfahren $\mathbb{V}_{\oplus}$ verwendet, da eine 1 die erreichte Position markiert. Die Position p dient während der Berechnungsphase ausschließlich zum Setzen der Statistikdaten für $\mathbb{V}$. Der Speicher für $\mathbb{V}$ enthält im Übrigen initial alle Hindernisse $\mathbb{H}$, so dass der Speicher eigentlich $\mathbb{V} \cup \mathbb{H}$ enthält. Sind dann alle Positionen im Speicher markiert, dies entspricht $\mathbb{V} \cup \mathbb{H} = \mathbb{P}$, so ist die Simulation abgeschlossen.

Die Erkennung einer Kollision erfolgt mit den Daten auf dem Bus. Dazu „belauschen“ die anderen Kreaturen die gesendeten Daten, betreiben also das so genannte *bus snooping*. Stimmt Position oder Zielposition mit ihrer eigenen Zielposition überein, so stellt dies einen Konflikt dar. Da die Kreatur mit erkanntem Konflikt zu einem früheren oder späteren Zeitpunkt ebenfalls die Daten auf dem Bus legt, um zumindest die Position in den Speicher $\mathbb{V}$ einzutragen, erkennt auch die andere betroffene Kreatur einen Konflikt.

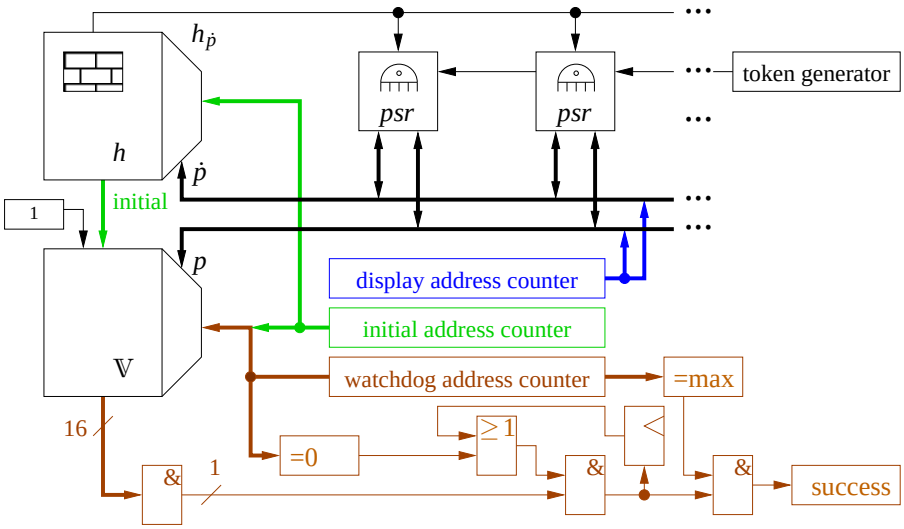


Abbildung 4.16: Prinzip eines Busses für Kreaturen

Da jedoch für einen solchen Bus mit zahlreichen, potentiellen Sendern ein Tri-State-Bus mit einer hochohmigen Ausgabemöglichkeit erforderlich ist, das ganze aber innerhalb eines FPGAs ohne externe Anbindung erfolgen soll, steht diese Tri-State-Technik leider nicht zur Verfügung. Ersatzweise laufen die einzelnen Positionen p_i aller Kreaturen zu einem Oder-Gatter, das die Anfragen zusammenfasst und an den Speicher leitet. Nur die Kreatur mit den Token darf ihre eigentliche Position senden, alle anderen müssen in diesem Fall die logik-neutrale Position 0 dem Bus übergeben.

Um zusätzlich die Verarbeitung zu verbessern, steht ein Register zur Zwischenspeicherung des Oder-Gatterwertes zur Verfügung. Dieses erhalten dann die zahlreichen Kreaturen. Von Vorteil ist die dadurch etwas bessere Taktrate und der gesteigerte Wert für den Fan Out, der die maximal anschließbaren Gatter bzw. Logikelemente bezeichnet.

Alternativ zur lesenden Anbindung von p und $\dot{p}$ jeder Kreatur ist es ausreichend, nur die Zielposition $\dot{p}$ auszuwerten. Eine Kreatur, die eine Kollision erkennt, teilt dies der anfragenden Kreatur über eine eigens dafür geschaffene Leitung c mit, sobald die

Übereinstimmung der eigenen Position oder Zielposition mit dem Wert $\dot{p}$ auf dem Bus vorliegt, die sie nicht selbst gesendet hat. Zu beachten ist dabei, dass der Wert $\dot{p}$ um einen Takt verzögert eintrifft und das gleichartig zu bearbeitende Kollisionssignal *c* um einen weiteren Takt verzögert bei der anfragenden Kreatur erscheint. Dies ist gleichzeitig mit dem Ergebnis aus dem Hindernisspeicher.

Trotz der geringeren Anzahl benötigter Verbindungsleitungen sind lediglich etwa $3^1/_2$ % weniger Logikelemente notwendig, die maximal mögliche Taktrate sinkt dabei aber teilweise um 14 %. Im konkreten Fall haben 64 Kreaturen auf einem 4 096 Zellen umfassenden Feld 10 903 bzw. 10 431 Logikelemente benötigt, die maximale Taktrate betrug dabei 68,19 MHz bzw. 58,50 MHz.

Daher ist in der Tabelle 4.3 nur die Variante ohne gesonderte Kollisionserkennungsleitung aufgeführt, obwohl bei 124 Kreaturen diese mit 20 652 benötigten Logikelementen nicht mehr auf den verwendeten Baustein passt, die andere Variante mit 19 897 Logikelementen hingegen schon. Bei 128 Kreaturen versagen beide Konzepte unter Beibehaltung der ausgewählten Hardware. Aufgrund der symmetrischen Anordnung der Kreaturen in einem Quadrat, um eine einfache, allgemeine Generierung zu ermöglichen, gibt es keine Zwischenwerte.

Das Verfahren der Datenauswertung für $\mathbb{V}$ (*watchdog*) wie im vorigen Abschnitt 4.3.3.4 für Abbildung 4.14e bleibt erhalten, ein Zähler durchläuft alle Adressen des Speichers. Haben alle ausgelesenen Bits den Wert 1, so ist die Simulation erfolgreich beendet. Auch hier gibt es wieder die Möglichkeit, die Generation exakt zu bestimmen oder nur zeitnah das Resultat zu erhalten. Aufgrund der Wahl eines anderen Speichermodells ohne JTAG-Schnittstelle stehen optimal nur 16 Bit zur Verfügung, bei 32 Bit steigt der Ressourcenverbrauch unnötig.

Die gleiche Anbindung an den Speicher (*port*) kann nun anstatt in der Berechnungsphase auch der Initialisierung dienen. Um mehrere Versuche nacheinander mit unterschiedlichen Kreaturkonfigurationen durchzuführen, ist ein Rücksetzen des Speichers $\mathbb{V}$ notwendig. Hierfür müssen alle Adressen beschrieben werden. Um möglichst viele Speicherzellen je Takt zu erreichen, bietet sich die gleiche Bitbreite wie für das Auslesen zur Datenauswertung an.

Da an den Hindernisspeicher *h* bisher nur ein Eingang beschaltet ist, bietet es sich bei einem Dual-Port-Speicher mit möglicher ungleicher Bitbreite für den anderen Port an, die Daten für den $\mathbb{V}$-Speicher in gleicher Bitbreite von 16 Bit auszulesen. Da jedoch die Speicher gepuffert sind, ergibt sich eine Verzögerung für die Eingangsregister des $\mathbb{V}$-Speichers, der die Daten aus dem Ausgaberegister des *h*-Speichers entgegen nimmt. Dies führt zu einer notwendigen Pufferung des initialen Adresszählers für den $\mathbb{V}$-Speicher ähnlich dem Verfahren einer Pipeline.

Das Zurücksetzen der Daten für die Kreaturen erfolgt über ein einfaches Signal, der Startzustand ist auf Null festgelegt, die anderen Werte Position und Richtung sind über Parameter zur Synthesezeit konfigurierbar, Dadurch ist kein weiterer Speicher speziell für die Kreaturen notwendig, die Simulation lässt sich ausreichend ohne Schwierigkeiten durchführen.

Die Ausgabe greift ebenfalls auf alle möglichen Positionen der Speicher zu, allerdings jede Position für sich, da gleichzeitig immer nur ein Zeichen darstellbar ist und ein Mehr an Bits unnötig mehr Organisationsaufwand zur Folge hat. Dies entspricht für den *h*-Speicher dem normalen Simulationsbetrieb. Um gleichzeitig die bereits besuchten Positionen von den noch nicht besuchten unterschiedlich darstellen zu können, erfolgt das Auslesen des $\mathbb{V}$-Speichers über die sonst verwendete Schreibadresse, die ein Bit In-

formation bereitstellt. Damit liegt auf p und $\dot{p}$ während der Ausgabe die gleiche Adresse an.

Um nun auch Auskunft über Existenz, Drehrichtung und Zustandswert der Kreaturen zu erhalten, gibt es einen weiteren Bus zusätzlich zu den Positionen, auf denen diese Daten von den Kreaturen gelegt werden. Auch hierbei darf nur jeweils eine Kreatur Daten unterschiedlich Null von sich geben, um einen Tri-State-Betrieb zu emulieren. Stimmt die Position einer Kreatur mit der Zielposition auf dem Bus überein, so darf diese ihre Daten preis geben. $\hat{p}$ wurde ausgewählt, da in jedem Fall die Hindernisse dargestellt werden, die Statistik ist aber nicht erforderlich. Um bei einer Ressourcensparmaßnahme diesbezüglich nichts umstellen zu müssen, bietet sich $\hat{p}$ gegenüber p an.

Für eine Koordinierung zwischen Ausgabe und Berechnung ist noch ein Steuerwerk notwendig, das zusätzlich auch bei aufeinander folgenden Berechnungen ohne Pause darauf achten muss, dass jeweils nur ein Token zwischen den Kreaturen unterwegs ist und auch die Verzögerungszeiten der Speicher berücksichtigt.

4.3.4 Durchführung

Um eine Wahl zwischen den verschiedenen Hardware-Konzepten treffen zu können, ist ein Vergleich zwischen deren Anforderungen notwendig. Ein einfaches Maß dafür ist die Anzahl der benötigten Logikelemente, um eine Simulation durchzuführen. Neben der eigentlichen Berechnung ist auch die Ausgabe des Feldes und der aktuellen Generation über die serielle Schnittstelle implementiert, um so den Simulationsverlauf kontrollieren zu können.

Tabelle 4.3: Benötigte Ressource „Anzahl Logikelemente“

Feldgröße	$\lvert\mathbb{I}\rvert$	**klassisch** Abb. 4.8	**separat** Abb. 4.10	**komplex** Abb. 4.11	**indiziert** Abb. 4.12	**ROM** Abb. 4.13	**Bus** Abb. 4.16
$4 \cdot 4 = 16$	1	1 807	1 010	1 474	1 013	652	636
$4 \cdot 4 = 16$	2	1 883	1 168	1 493	1 189	769	801
$4 \cdot 4 = 16$	4	2 057	1 520	1 925	1 610	1 105	1 117
$4 \cdot 4 = 16$	8	2 396	2 325	2 577	2 563	1 985	1 796
$4 \cdot 4 = 16$	12	2 755	3 295	3 390	3 607	3 301	2 451
$4 \cdot 4 = 16$	16	3 101	4 390	4 108	4 834	5 164	3 081
$8 \cdot 8 = 64$	1	4 761	1 312	3 181	1 439	652	636
$8 \cdot 8 = 64$	2	5 414	1 623	3 295	1 873	769	801
$8 \cdot 8 = 64$	8	7 687	3 785	6 727	5 674	1 985	1 796
$12 \cdot 12 = 144$	2	11 582	2 515	6 346	2 988	769	801
$16 \cdot 16 = 256$	2	18 883	3 159	11 088	4 846	769	801
4 096	32	—	—	—	—	17 772	5 709
4 096	64	—	—	—	—	—	10 903

Die dabei entstandenen Werte zeigt Tabelle 4.3, wobei die Anzahl für nicht mehr implementierbare Umgebungen nicht mit angegeben ist. Die Ermittlung konzentriert sich auf einen Hersteller und eine Plattform, um vergleichbare Werte zu erhalten. Da sich je nach Konzept unterschiedliche Bedürfnisse aus Feldgröße und Kreaturanzahl ergeben, ist zum einen nur die Größe und damit die Anzahl möglicher Positionen, zum anderen nur die Anzahl der Kreaturen variiert. Graphisch aufbereitet zeigen die Abbildungen 4.17a und 4.17b die Abhängigkeiten bezüglich eines Parameters.

Bei der klassischen Variante zeigt sich deutlich die Abhängigkeit von der Feldgröße, bei den anderen Konzepten ist der beeinflussende Faktor aufgrund der teilweise ausgelagerten Funktionalität geringer. Da bei den Modellen ROM und Bus die Feldgröße jeweils

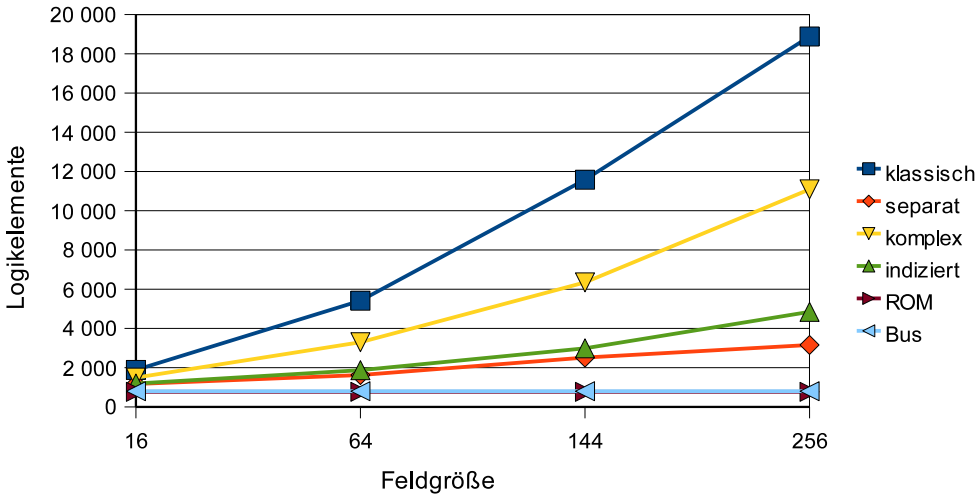


Abbildung 4.17a: Vergleich der Ressource Logikelemente bezüglich der Feldgröße $|\mathbb{P}|$ bei zwei Kreaturen ($|\mathbb{I}| = 2$)

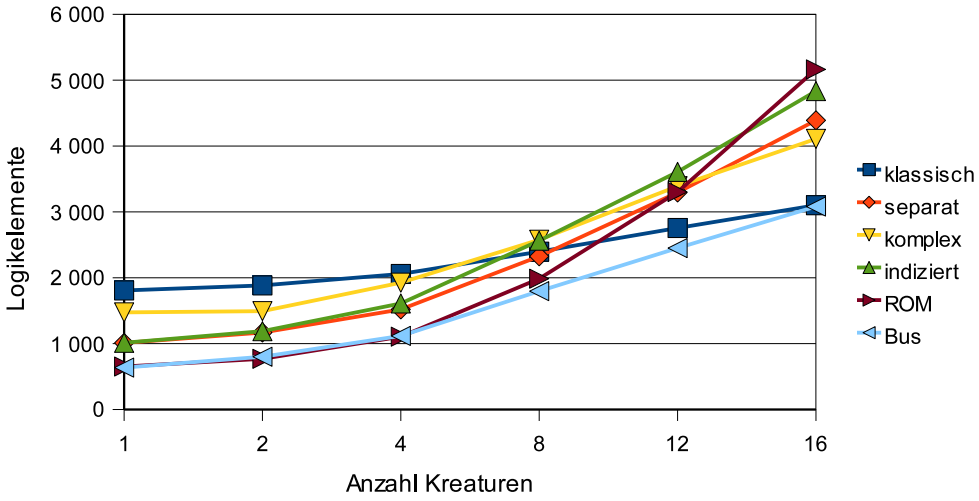


Abbildung 4.17b: Vergleich der Ressource Logikelemente bezüglich der Anzahl von Kreaturen $|\mathbb{I}|$ mit einer Feldgröße $|\mathbb{P}|$ von 16 Zellen

unter der Minimalgröße von 4 096 Positionen liegt, bleibt der Aufwand gleich. Dies alles entspricht den Erwartungen: Je weniger Funktionen die einzelnen Zellen des Feldes vorhalten, um so besser ist das Verhalten bei größeren Feldern, da nur die aktiven Komponenten existieren.

Anders sieht es bei gleich bleibender Feldgröße und variabler Kreaturanzahl aus. Die klassische Variante zeigt den geringsten Anstieg, wohingegen die ROM-Architektur einen steilen Anstieg bei mehr Kreaturen verzeichnet. Die als „komplex“ bezeichnete Architektur präsentiert auch mit diesem Parameter ein gutes Verhalten.

Im kombinierten Vergleich agieren „komplex“ und „separat“ gleichwertig bei der Kreaturanzahl, bei der Feldgröße liegt das Konzept „separat“ bei einem Takt für eine Generationsberechnung vorne. Generell zeigt sich ein Vorteil der langsameren Busarchitektur, dicht gefolgt von der speicherbehafteten Lösung, die mehr als einen Takt je Generation benötigt, jedoch nicht in dem Umfang der Bus-Lösung von 20 ns je zusätzlicher

Kreatur. Dies liegt unter anderem an der Verlagerung des Feldes von den Logikelementen in die nicht erfassten Speicherbereiche.

Das geeignetste Hardwarekonzept hängt somit von den speziellen Bedürfnissen und Anforderung ab, ein allgemein gültiges Rezept kann es demnach nicht geben.

Alternativ, die Simulation mittels Software durchzuführen, bietet eine größere Vielfalt an Möglichkeiten bei wesentlich geringerer Ressourcenbeschränkung. So sind auch die Felder und Ergebnisse im Anhang C allein mit Softwareunterstützung entstanden, erst danach erfolgte die Überprüfung mittels Hardware. Für die zeitoptimale Variante benötigt die Berechnung einer Generation mit acht Kreaturen 709,4 ns nach dem in Abschnitt 4.3.3.3 beschriebenen Verfahren. Mit gleichen Parametern benötigen die Hardwareversionen 100,8 ns im Bus-Verfahren, 20,0 ns mit der ROM-Architektur, bei einer Ein-Takt-Implementierung reichen sogar 15,5 ns für die indizierte Architektur. Eine Hardwareumsetzung ist damit in diesem Fall um den Faktor 7 bis 45,7 schneller als die Softwarevariante auf dem Rechner wie auf Seite 78 beschrieben, wodurch sich der Vorteil spezieller Architekturen gegenüber sequentieller Programmiermethodik zeigt.

4.4 Erweiternde Varianten

Eingangs gab es einige Einschränkungen, um einfache Architekturkonzepte für Simulationen entwerfen zu können. Nun besteht die Möglichkeit, diese Einschränkungen aufzuheben und auf die existierenden Konzepte anzuwenden.

4.4.1 Unterstützung mehrerer unterschiedlicher Kreaturen

Statt nur eine Kreatur bzw. mehrere gleiche Kreaturen zu unterstützen, ist es denkbar, verschiedenartige Kreaturen zu haben, wie schon in Abbildung 4.1 dargestellt. Dafür ist es notwendig, den Kreaturen verschiedene Automaten zuzuordnen. Um die Variationen gering zu halten, empfiehlt sich die Aufzählung mit zwei Automaten. Details zu den Anforderungen und Ergebnissen bietet Abschnitt 5.1, weitere Untersuchungen sind in [EHH08a] und [EHH08b] enthalten.

Unter Verwendung des klassischen Konzeptes ist es erforderlich, die Automaten den Kreaturen mitzuführen. Demnach ist bei einer Bewegung auf eine Nachbarposition nicht nur der Folgezustand und -richtung zu kopieren, sondern auch die vollständige Automatentabelle. Dies führt zu einem sehr hohem Aufwand. Die anderen Konzepte bieten einfachere Unterstützung. Sobald die Kreaturen vom Feld ausgelagert sind, ist ein Zugriff direkt auf diese möglich, eine Anbindung kein Problem mehr. Ein Automat lässt sich direkt den Kreaturen mit statischer Bindung zuordnen, so dass ein Kopieren der Zustandsübergangs- und Ausgabewerttabelle nicht notwendig ist.

4.4.2 Zusätzliche Bewegungsmöglichkeiten

In die Umsetzung eingeflossen sind bisher nur die Werte für „rechts“ und „links“ der Tabelle 4.2. Kommen nun die Werte für „gerade aus“ und „zurück“ hinzu, steigen die Anforderung der Umsetzung und damit die Anzahl der Logikelemente aus Tabelle 4.3.

Die zusätzlichen Möglichkeiten, die ein Wert aus $\mathbb{Y}$ bietet, betrifft jede Position, da jederzeit eine Kreatur mit einer Richtung r und Folgerichtung r' auf ihr zu landen kommen könnte. Da es sich aber nur um eine Modulo-4-Addition handelt, ist lediglich die zu ändernde Tabelle zum Erhalt der Richtungsänderung für $\mathbb{D}$ relevant.

Zusätzlich muss auch noch die Kreatur die weiteren Werte aus $\mathbb{Y}$ ausgeben können. Daher ist auch die interne Automatentabelle anzupassen. Dies betrifft möglicherweise auch die Parameter der Aufzählung, die im Abschnitt 5.1 zur Geltung kommt.

4.4.3 Transport als Ziel

Bisher war nur die Aufgabe betrachtet, ein Feld vollständig abzuschreiten. Mit einer anderen Aufgabe ändern sich auch die Anforderungen. Soll zum Beispiel ein Fußball zum Einsatz kommen, gibt es ein weiteres, sich bewegendes Objekt, das durch Kollision mit anderen Kreaturen seine Richtung ändern kann, und dann weiter, evtl. mit einer anderen Richtung, über das Feld *rollt.*

Alternativ zu einem Ball ist auch eine Kiste denkbar, die wie beim Spiel Sokoban zu einer bestimmten Position zu schieben ist. Das Erreichen könnte der Kreatur z. B. durch eine Glücksempfindungsleitung signalisiert werden. Um es einfacher zu machen, könnte der Kreatur erlaubt werden, die Kiste auch ziehen zu können, um so aus Ecken herauszukommen. Ansonsten dürfte der Erfolg mit einfachen Zustandsautomaten recht gering sein.

Eine dritte Variante ist die einer Klette, die sich an eine vorbeikommende Kreatur anheftet und bei Hindernissen oder speziellen Sammelplätzen hängen bleibt. Zusätzlich könnten die Regeln auch enthalten, dass eine andere Kreatur oder eine andere Klette die transportierte Klette mitreißt oder zum Beibehalten der aktuellen Position animiert.

Dies alles betrifft im wesentlichen die Automatentabelle des sich bewegenden Objektes, aber auch das auszulesende Umfeld. Insbesondere bei der Klette zeigt sich, dass nicht nur die Zielposition in Blickrichtung, die eher zur Anhaftrichtung mutiert, relevant ist, sondern auch alle anderen vier Positionen um die Klette herum. Die Automatentabelle entspricht dann eher einer Umsetzung des vorgegebenen Verhaltens, unterliegt also nicht dem variablen Experimentteil.

Daher eignet sich auch hier das klassische Konzept aufgrund der Anforderungen weniger. Die separate Variante ist wegen Ihrer Art, auf die Feldpositionen lesend zuzugreifen, weniger geeignet, da ansonsten je Klette vier Multiplexer notwendig wären. Bei den komplexen und indizierten Verfahren steigt die Anzahl der notwendigen Leitungen auf dem Bus, aber durch die Auswertung der Zellen könnten Zusatzleitungen weitere Informationen über naheliegende Kletten liefern. Auch die beiden anderen, speicherbehafteten Varianten sind denkbar, hier muss die Auswertung aber über zusätzliche Logik erfolgen.

Die Frage ist dann, ob eine Kreatur eine naheliegende Klette (aktiv) mitnimmt oder umgekehrt, dass eine Klette um die genaue Kreatur weiß, also ihren Index $i \in \mathbb{I}$ kennt. Eine solche Bindung müsste experimentell untersucht werden, um ein Optimum zu finden.

Übrigens muss sowohl eine Klette als auch eine Kiste für eine Kreatur transparent sein, wenn die Kreatur ein Objekt vor sich herschieben kann. Nach den bisherigen Verfahren darf ein solches, eigentlich passives Objekt also nicht auf gewohnte Art in die Hinderniserkennung oder Kollisionserkennung einfließen. Stößt die Kreatur mit einem vorausgeschobenem Objekt auf ein Hindernis, so erfahren die Kreatur und das passive Objekt dies umgehend,

Es sind also je nach Aufgabenstellung spezielle Regeln zu entwerfen. Deren Umsetzung hat auch unterschiedliche Anforderungen an das System zur Folge, so dass keinem Konzept aus Abschnitt 4.3.3 ein Vorzug gegeben werden kann und auch eine generelle Instruktion zur Anpassung der Architektur nicht möglich ist.

4.4.4 Dynamische Anzahl

Eine konstante Anzahl von Kreaturen ist nicht notwendig. Um eine variable Anzahl zu ermöglichen, ist ein Kreaturen-Generator und -Destruktor notwendig. Dies entspricht zum Beispiel einem Parkhaus in der Verkehrssimulation oder einem Klettenaufsauger in eine andere Dimension – gemäß einer Idee aus dem vorherigen Kapitel, allerdings nur als Destruktor.

In der klassischen Variante ist dies einfach durch eine speziell agierenden Zelle möglich, da die Anzahl der Kreaturen nur durch die Anzahl der frei begehbaren Positionen beschränkt ist, also $|\mathbb{I}| \leq |\mathbb{A}|$. Für die anderen Konzepte ist die Anzahl der Kreaturen frei einstellbar, aber dann auf einen Maximalwert beschränkt. Inaktive Kreaturen, die sich z. B. im Parkhaus befinden, lassen sich durch ein zusätzliches Bit kennzeichnen. Die Kollisionserkennung ist entsprechend anzupassen, so dass sich auf den speziellen Positionen auch mehrere Kreaturen aufhalten dürfen.

4.5 Zusammenfassung

Die neu entstandenen Berechnungskonzepte basieren auf bekannten Verfahren, teilweise neu kombiniert. Der Entwurf von Spezialhardware hat für das klassische Prinzip eines universellen Feldes analog zur Abbildung 4.8 auf Seite 94 gezeigt, das bei hoher Auslastung durch viele Kreaturen die Umsetzung interessant ist, zumal viele Abläufe parallel geschehen. Dennoch ist eine Aufteilung in Funktionsgruppen interessant, da sich z. B. mit der Umsetzung der Abbildung 4.13 auf Seite 97 die Berechnung lediglich um einen konstanten Faktor verlängert. Sollten die zur Verfügung stehenden Hardwareressourcen nicht ausreichen, ist die seriell arbeitende Buslösung gemäß Abbildung 4.16 auf Seite 103 von Interesse.

Bei den einzelnen Architekturen zeigen sich verschiedene Komponenten, aufgezeigt in Tabelle 4.4. Eine Kreatur benötigt Zustand und Position mit Richtung, die unterschiedlich gespeichert werden können; Hindernisse und Auswertung sind die anderen Eigenschaften für einen unterschiedlichen Speicherort. Dabei gibt es die Möglichkeiten, die Daten lokal verteilt im Feld oder zentral in einem ROM bzw. RAM unterzubringen. Bei der Kollisionserkennung mehrerer Kreaturen gibt es analog die Berechnungsmöglichkeiten verteilt im Feld oder an zentraler Stelle durch Vergleich der Positionen sowie zusätzlich zeitlich verteilt mit dem Bus-System. Durch die unterschiedlichen Kombinationen ergeben sich die unterschiedlichen Architekturen, die je nach Anforderung unterschiedliche Vorteile haben.

Tabelle 4.4: Übersicht der Hardwarearchitekturen für die Simulation

Architektur	Zustand einer Kreatur	Kreatur-position und -richtung	Kollisions-berechnung	Hindernis und Auswertung
klassisch	im Feld	im Feld	im Feld	im Feld
separat	zentral	zentral	zentral	im Feld
komplex	zentral	im Feld	im Feld	im Feld
indiziert	zentral	zentral	im Feld	im Feld
ROM	zentral	zentral	zentral	zentral
Bus	zentral	zentral	zeitlich verteilt	zentral

Eine Auswirkung auf die Entwicklung der Simulationssoftware haben die Hardwarearchitekturen deutlich gemacht. Statt ein Feld vollständig zu berechnen, ist die Aufteilung in Komponenten einfach möglich. Da dann die Berechnung zielgerichtet angestoßen werden kann, ist eine Simulation mit dieser getrennten Variante schneller durchgeführt.

Auch hier hat sich wieder gezeigt, dass die Basis für Konzept auf einzeln gut herausgearbeiteten Komponenten bestehen sollte, um dann ein ideales Ergebnis erzielen zu können.

Kapitel 5

Ergebnisse

Nachdem die Automaten für sich aufgezählt sind und eine Simulationsdurchführung möglich ist, steht der Schritt der Zusammenführung noch aus. Für die Kombination gibt es unterschiedliche Aspekte und Herangehensweisen.

5.1 Simulation mit Automatenaufzählung

Als erstes ist die Kombination der Ergebnisse aus Kapitel 3 und 4 notwendig – getrennt oder zusammen sowohl für Hardware als auch für Software. Aber bereits vor dem ersten Einsatz gibt es verschiedene Möglichkeiten, mit den Berechnungsmöglichkeiten umzugehen.

5.1.1 Einschränkung

Prinzipiell kann jede Kreatur einen anderen Algorithmus annehmen. Jedoch ergeben sich dann bei zahlreichen Kreaturen auf einem Feld zu viele Möglichkeiten, die nicht in einer überschaubaren Zeit ausreichend simulierbar sind. Zum Beispiel benötigt die Aufzählung mittels Hardware bei drei Zuständen bereits 9 931 Takte. Wenn alle Konstellationen zum Tragen kommen, benötigt allein die Aufzählung aller relevanten Automaten für die Simulation bei nur vier Kreaturen bereits $9\,931^4 = 9\,726\,843\,482\,307\,121$ Takte. Mit der angenommenen Taktfrequenz von 85,16 MHz entspricht dies einer Zeit von etwa 3,6 Jahren. Hinzu kommt noch die eigentliche Simulationsdurchführung als konstanter Faktor. Seien es durchschnittlich eintausend Generationen, so ergibt sich bereits mit diesen geringen Werten eine Laufzeit von 3 600 Jahren. Dieses Verfahren ist demnach auf diese Weise nicht praktikabel.

Daraus ergibt sich, dass mehrere Kreaturen die gleichen Algorithmen verwenden sollten, um die Anzahl der Möglichkeiten zu reduzieren. Für drei unterschiedliche Kreaturen beträgt die Aufzählungszeit nur noch fast 3,2 Stunden, für zwei unterschiedliche Kreaturen sind es sogar nur 1,2 Sekunden allein für die Aufzählung aller Algorithmen. Für Automaten mit mehr Zuständen ist eine Simulation bei zwei unterschiedlichen Kreaturen zeitlich interessant.

Eine Variante ist, der Hälfte der Kreaturen den gleichen Algorithmus zuzuweisen. Um die Durchführung auf zwei verschiedene Algorithmen zu reduzieren, wenn die Kreaturen zu Beginn am Rand angeordnet sind, gibt es die drei symmetrischen Varianten, den

- auf einer Seite gegenüberliegende Kreaturen,

- den über Eck liegende Kreaturen, z. B. obere und linke Seite, sowie
- jeder zweiten Kreatur einer Seite

den gleichen Automaten zuzuweisen und somit die Kreaturen gleichzusetzen. Abbildung 5.1 präsentiert das Resultat für ein einfaches Feld ohne weitere Hindernisse. Die Simulation mit nur einer Art von Automat bleibt davon unberührt relevant.

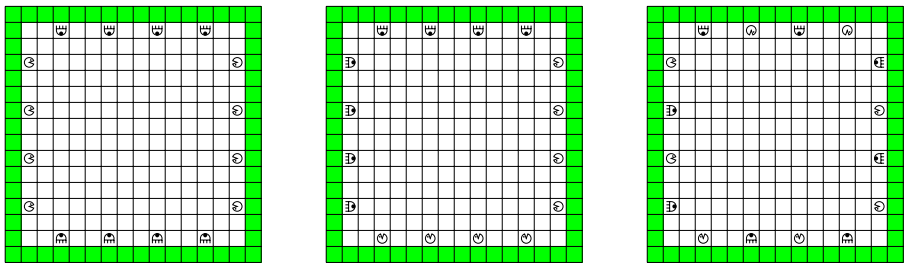


Abbildung 5.1: Anordnung für Kreaturen am Rand mit insgesamt zwei unterschiedlichen Automaten (gegenüberliegend, aneinander liegend, alternierend)

5.1.2 Hardwareentwurf

Es hat sich gezeigt, dass sowohl die Aufzählung von Automaten als auch die Simulationsdurchführung mit einer speziellen Hardwarearchitektur schneller vorangeht als softwareprogrammiert mit einem üblichen Rechner. Nun ist eine Kombination der Automatenaufzählung mit der Simulation dynamischer Objekte erforderlich. Dabei erfordern unterschiedliche Automaten für dynamische Objekte eine mehrfache Aufzählung; bei zwei Typen ergeben sich zwei Automatenzähler. Kommen mehrere Automaten zum Einsatz, ist es irrelevant, ob sich die Automaten in ihrer Konfiguration, z. B. in der Zustandsanzahl, entsprechen.

Für die Kombination ist eine Architektur mit vom Feld getrennten Kreaturen vorteilhaft, da hier ein direkter Zugriff auf die Automatentabelle der Kreatur möglich ist. Basieren alle Kreaturen auf dem gleichen Automaten, erhalten alle ihre Automatentabelle für Zustandsübergänge und Ausgabewerte aus gleicher Quelle. Der dahinter liegende Automatenaufzähler generiert immer dann den nächsten Automaten, wenn die Simulation erfolgreich beendet oder vorzeitig abgebrochen ist. Hierbei entspricht die in Abbildung 3.22 dargestellte Druckausgabe der Weitergabe an die Kreaturen der Simulation.

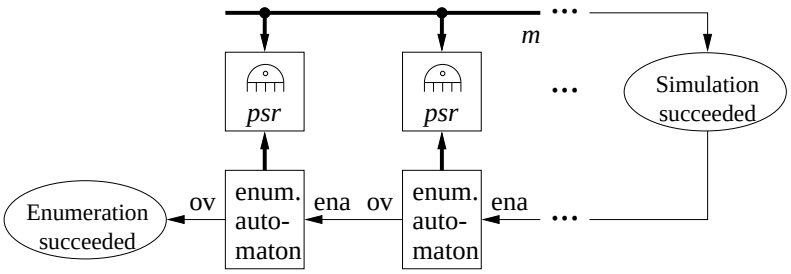


Abbildung 5.2: Anbindung der Automatenaufzählung an die Simulation

Gibt es nun unterschiedliche Kreaturen, sind auch unterschiedliche Automatenaufzähler erforderlich, die koordiniert ihre Aufzählung durchführen. Ein Automat symbolisiert eine Zählerstelle. Gibt es einen Überlauf, beginnt der Zähler also wieder von Vorne,

so ist dieser Überlauf das Signal für den nächsten Zähler. Die Abbildung 5.2 stellt dies für zwei herausgegriffene Zähler dar. Dabei ist ein Überlauf mit „ov“ für *overflow*, das Inkrementierungssignal mit „ena“ für *enable* bezeichnet. Der erste Zähler erhält sein Signal bei Abschluss einer Simulation. Das Verfahren ist beendet, wenn auch der höchststelligste Zähler einen Überlauf erzeugt.

Eine Simulation kann immer dann starten, wenn ein gültiger Automat an allen Zählerständen vorhanden ist.

5.1.3 Softwareentsprechung

Ähnlich wie für den Hardwareentwurf lassen sich die Zähler implementieren. In einer objektorientierten Programmierumgebung sind auch mehrere verkettete Automaten kein Problem; ein Überlauf, z. B. über einen Funktionsrückgabewert mitgeteilt, hat das Inkrementieren des nächsten Automaten zur Folge. Eine Kreatur kann einen Automaten über ein Array auswählen.

5.1.4 Kombinierte Berechnung mit Hard- und Software

Eine andere Art der Kopplung von Aufzählung und Simulation bietet die Aufteilung zwischen Hardware und Software, so dass eine Komponente die Aufzählung durchführt, die andere die Simulation. Die Tabelle 3.7 zeigt einen deutlichen Vorteil für die Hardware, weswegen die Aufzählung dort geschehen sollte. Der Vorsprung der Hardware bezüglich der Simulation ist nicht ganz so herausragend, weshalb sich diese für die Umsetzung in Software eignet, insbesondere für große Felder mit viel Speicheraufwand. Nichtsdestominder ist die Anlieferung der Daten auch umgekehrt möglich.

Verbleibt die Frage, wie die Daten zwischen den Systemen ausgetauscht werden können. Die einfachste Variante ist, die existierende serielle Schnittstelle zu nutzen. Allerdings ist diese mit 115 200 Baud bzw. 11 520 Byte/s nicht besonders schnell. Eine Netzwerkanbindung mit theoretisch bis zu 1 000 MByte/s bietet da weitaus mehr Transfervolumen. Aber auch eine PCI-Schnittstelle zu einem Rechner bietet einen Geschwindigkeitsvorteil, wie die Untersuchung in [Pas07] gezeigt hat; Transferraten von 120 MByte/s sind möglich.

Aber auch ohne das Schema der aufgeteilten Berechnung ist die Übertragung der Daten von der Hardware hin zu einem Rechner notwendig, um so die Ergebnisse dauerhaft protokollieren zu können. Die genauen Details der Aufteilung sind projektabhängig und basieren im wesentlichen auf der zur Verfügung stehenden Hardwareplattform und den zu berechnenden Eigenschaften, insbesondere der Simulationsumgebung.

5.1.5 Vorfilter durch andere Experimente

Statt einer vollständigen Trennung zwischen Hardware und Software bezüglich Aufzählung und Simulation vorzunehmen, könnte die Hardware auch Aufzählung und Vorauswahl durch Simulation durchführen, die dann weniger gewordenen Automaten an einen Rechner übertragen, so dass dann die Software weitere Simulationen durchführen kann. Dieser Filter ist recht effektiv und reduziert die Anzahl der auf dem Rechner zu simulierenden Automaten deutlich, wie auch das Beispiel in Abbildung B.12 zeigt.

Alternativ könnten die vorgefilterten Automaten auch auf einer anderen Spezialhardware zur Ausführung kommen, deren einzelne Bausteine nicht mit der Aufzählung und

Auswertung belegt sind. Die Simulation erhält die Liste der Automaten aus einem vorinitialisiertem ROM oder wieder per Datenübertragung von einem Rechner mit ausreichender Speicherkapazität. In jedem Fall ist ein universeller, softwaregesteuerter Rechner notwendig.

5.2 Erfolgreiche Automaten

Den erfolgreichen Automaten für alle Umgebungen hat es bei den Zwei- bis Sechs-Zustandsautomaten nicht gegeben. Es gab aber interessante Verhaltensmuster und Teilerfolge. Die Berechnung und Simulationsdurchführung der folgenden Automaten stammt – soweit nicht anders vermerkt – für die Felder #1 bis #5 (Abbildung C.1, Seite 141) aus Hardwaresimulationen, die Überprüfung auf anderen Felder und Test der Klassifizierungskriterien erfolgte ausschließlich mittels Software, da zur damaligen Entwicklungsstufe die Hardwarekonzepte noch nicht ausgearbeitet waren.

5.2.1 Ein Automat mit zwei Zuständen

Bei den Zwei-Zustandsautomaten, die auf den Feldern der Abbildung C.1 zur Ausführung kamen, gab es zwei Favoriten. Insgesamt haben auch nur 19 der 256 möglichen Automaten ein brauchbares Verhalten dargestellt und so mindestens 42 % der möglichen Positionen besucht. Die Automaten selbst zeigt Abbildung 5.3, die erreichten Felder stellt Tabelle 5.1 dar. Nur bezüglich ihres Startzustandes unterscheiden sich die beiden Automaten, ansonsten sind sie isomorph zueinander, so dass bei einer überprüften Startzustandsisomorphie nur einer der beiden Automaten die Aussortierung überstanden hätte.

Abbildung 5.3: Die beiden erfolgreichen Zwei-Zustandsautomaten 57 und 108

Tabelle 5.1: Erreichungsgrad bei Automaten mit zwei Zuständen

Automat	#1	#2	#3	#4	#5	Durchschnitt
max.	50	64	58	53	48	
57	50	22	28	34	34	61 %
108	50	28	10	36	41	60 %

5.2.2 Ein Automat mit vier Zuständen

Für Vier-Zustandsautomaten, deren Überprüfung auf die Eigenschaft Präfixfreiheit nicht erfolgt ist, gab es mehr Erfolge zu verbuchen, allerdings sind auch hier nicht alle Felder aus Abbildung C.1 erfolgreich besucht worden. Abbildung 5.4 zeigt die Automaten, Tabelle 5.2 das entstandene Resultat. Der Sortierung liegt der Anteil der besuchten Positionen zu den möglichen Positionen je Feld zugrunde.

Es zeigt sich, dass ein Drei-Zustandsautomat mit Präfix erfolgreicher ist, als die eigentlichen Vier-Zustandsautomaten. Aber auch hier erreicht kein Automat alle Positionen in allen Feldern. Lediglich Automat B4 erreicht im Feld #2 alle Positionen. Interessant

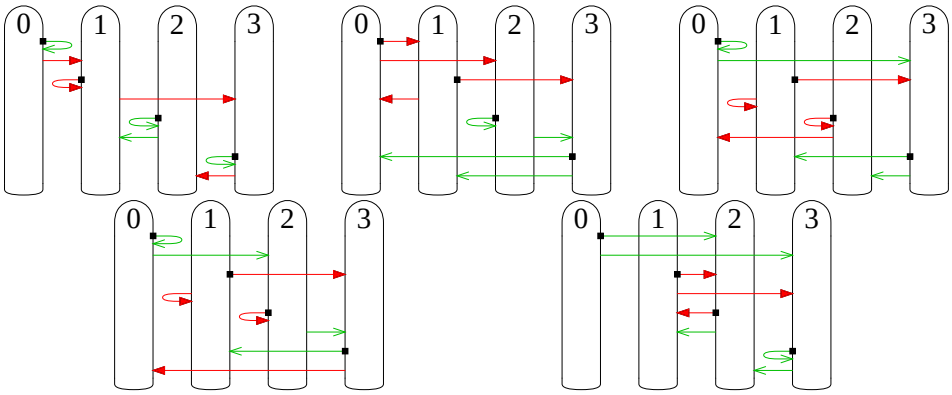


Abbildung 5.4: Die fünf erfolgreichsten Vier-Zustandsautomaten A4 bis E4

Tabelle 5.2: Erreichungsgrad bei Automaten mit vier Zuständen

Automat	#1	#2	#3	#4	#5	Durchschnitt
max.	50	64	58	53	48	
A4	50	60	56	53	48	97 %
B4	50	64	58	50	48	97 %
C4	50	60	58	48	48	96 %
D4	50	60	58	47	48	95 %
E4	48	60	47	48	48	91 %

ist auch der Verlauf von „Anzahl besuchter Positionen“ zur Generation, also Anzahl der Schritte, die sich eine Kreatur bewegt hat. Abbildung 5.5 zeigt beispielhaft den Kurvenverlauf für das Feld #1. Dabei hat Automat E4 eine hohe Anfangsgeschwindigkeit, erreicht aber nicht alle Positionen, während B4 am langsamsten ist, aber dafür erfolgreich.

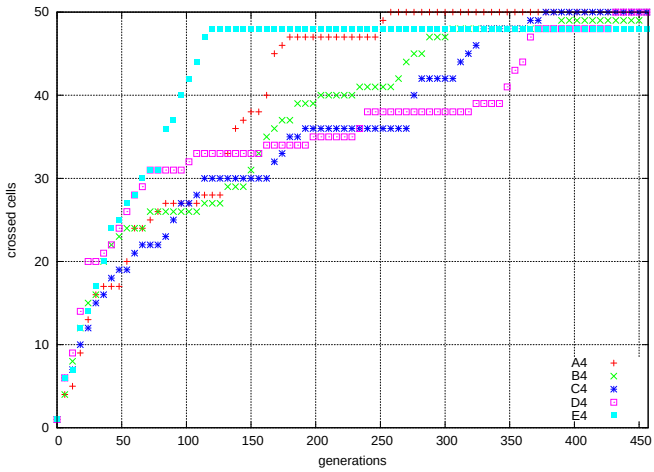


Abbildung 5.5: Verlauf der neu besuchten Positionen für Feld #1 bei den besten Vier-Zustandsautomaten

5.2.3 Ein Automat mit fünf Zuständen

Es gibt insgesamt sechs Fünf-Zustandsautomaten, die alle Positionen der Felder aus Abbildung C.1 erreichen konnten. Diese sind in Abbildung 5.6 aufgeführt. Auch hier sieht

die Geschwindigkeitsverteilung interessant aus, zu sehen exemplarisch wieder für das Feld #1 in Abbildung 5.7, die einen ähnlichen Verlauf aber in kürzerer Zeit wie für die Vier-Zustandsautomaten darstellt. Ab fünf Zuständen lohnt sich auch ein Vergleich der Geschwindigkeit, um neue Felder zu erreichen. Die fünf Felder bieten zusammen 273 freie Positionen an; Automat A5 benötigt hierfür 1 159 Schritte. Dies ergibt eine Durchschnittsgeschwindigkeit von $\bar{v} = \frac{1159}{273} \approx 4,25$ Schritten pro neu besuchter Position.

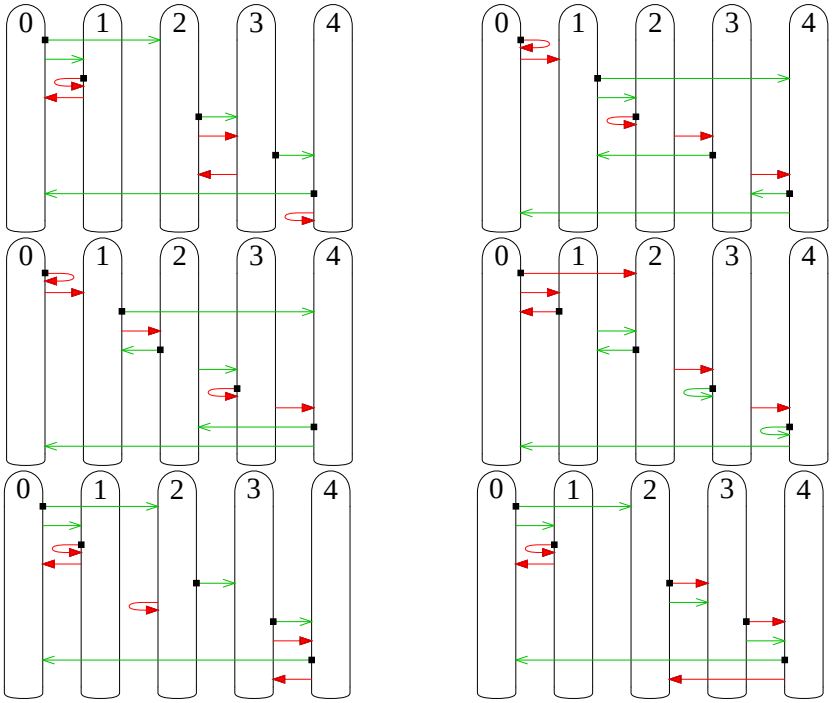


Abbildung 5.6: Die sechs erfolgreichsten Fünf-Zustandsautomaten A5 bis F5

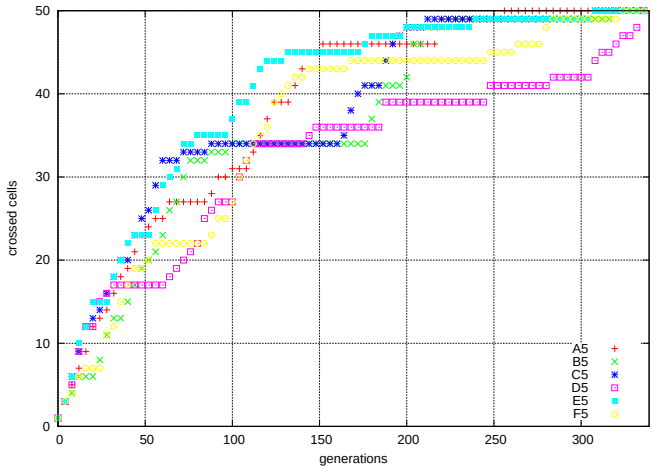


Abbildung 5.7: Verlauf der neu besuchten Positionen für Feld #1 bei den besten Fünf-Zustandsautomaten

5.2.4 Ein Automat mit sechs Zuständen

Bei sechs Zuständen ist die Zahl der erfolgreichen Automaten größer. Um eine weitere Eingrenzung vorzunehmen, durchliefen die automatengesteuerten Kreaturen auf 21 weiteren Feldern die Simulation, aufgeführt in Abschnitt C.1. Die neun besten Automaten sind in Abbildung 5.8 zusammengestellt.

Dabei gab es eine Vorauswahl durch Hardwareberechnung mit den fünf Feldern aus Abbildung C.1 sowie das erste Feld aus Abbildung C.3. Die Kreaturen mussten eine Mindestanzahl an Positionen besucht haben, um in die weitere Auswahl zu kommen. Für die Felder #1 bis #6 sind dies mindestens 48, 60, 47, 47, 48 und 52 Positionen. Demnach mussten also für das Feld #5 alle Positionen besucht werden. Gab es in keinem Feld eine Änderung in der Anzahl besuchter Felder für die letzten 256 Generationen, so wurde die Simulation vorzeitig abgebrochen und der nächste Automat getestet. So kam eine Liste mit 312 948 Automaten zustande, die dann softwareseitig äquivalenzgefiltert mit weiteren Simulationen der Feldern #7 bis #26 auf 64 061 Automaten dezimiert wurde. Eine Auswahl durch die Kriterien im Kapitel 3 hat noch nicht stattgefunden, deren Entwicklung kam erst später zum tragen.

Automat J6 zeigt das beste Verhalten für die Felder #1 bis #5, versagt allerdings mehr als die Automaten A6 bis I6 bei den anderen Feldern. Am besten bewältigt Automat G6 die Anforderungen und erreicht mit 99,9197 % den Spitzenplatz in der Anzahl besuchter Positionen. Knapp dahinter folgt der zu G6 bezüglich der Drehrichtung inverse Automat B6. Interessant ist auch die Symmetrie der einzelnen Zustände zueinander. So bilden Zustände 0 bis 2 eine Gruppe, die zu den Zuständen 3 bis 5 eine unterschiedliche Drehrichtung aufweisen.

Um die invertierte Drehrichtung genauer zu untersuchen, hat ein weiterer Automat, K6, zu sehen in Abbildung 5.9, die Liste der relevanten Sechs-Zustandsautomaten als Pendant zu C6 erweitert. Dieser ist bei gleich vielen, allerdings anderen Feldern erfolgreich und dabei im Durchschnitt schneller, wie die Ergebnisse in Tabelle 5.3 wiedergeben.

Die mittlere Strecke zur Erreichung einer noch nicht besuchten Position, also dem Kehrwert der Geschwindigkeit, bemisst sich aus der Anzahl besuchbarer Positionen $|\mathbb{A}|$ je Feld geteilt durch die Anzahl benötigter Generationen, um all diese Positionen zu erreichen. In Tabelle 5.4 ist dieser Wert in der Spalte $\overline{v}$ für die mittlere Schrittanzahl zur Erreichung einer bisher unbesuchten Position dargestellt. Die Werte in der Tabelle sind für die Automaten nach erfolgreich besuchten Feldern und dem Kehrwert der Geschwindigkeit sortiert.

5.2.5 Mehrere Kreaturen mit gleichem Automaten

Ein weiteres Bewertungskriterium stellt zugleich eine weitere Einsatzmöglichkeit dar. Statt nur eine Kreatur auf einem Feld unterzubringen, interessiert das Verhalten mehrerer Kreaturen, die nach den einfachen Regeln aus Abschnitt 4.1.5 einen Kollisionskonflikt lösen. Um die Ergebnisse vergleichbar zu halten, erfolgt die Anordnung der Kreaturen mit zueinander symmetrischen Startpositionen gemäß Abbildung C.6. Als unterschiedliche Umgebungen kommen die Felder ENV1 bis ENV4 aus Abbildung C.7 zum Einsatz, zum Vergleich mit einem barrierefreien Feld steht zusätzlich auch ENV0 aus bereits erwähnter Abbildung C.6 zur Verfügung.

Die bisher gefundenen Automaten auf die neuen Felder angewendet, führt zu einem differenzierten Ergebnis. So überquert der bisher nicht so erfolgreiche Automat J6 auch mit wenigen Kreaturen am meisten Felder vollständig. Die Werte sind im Detail in den

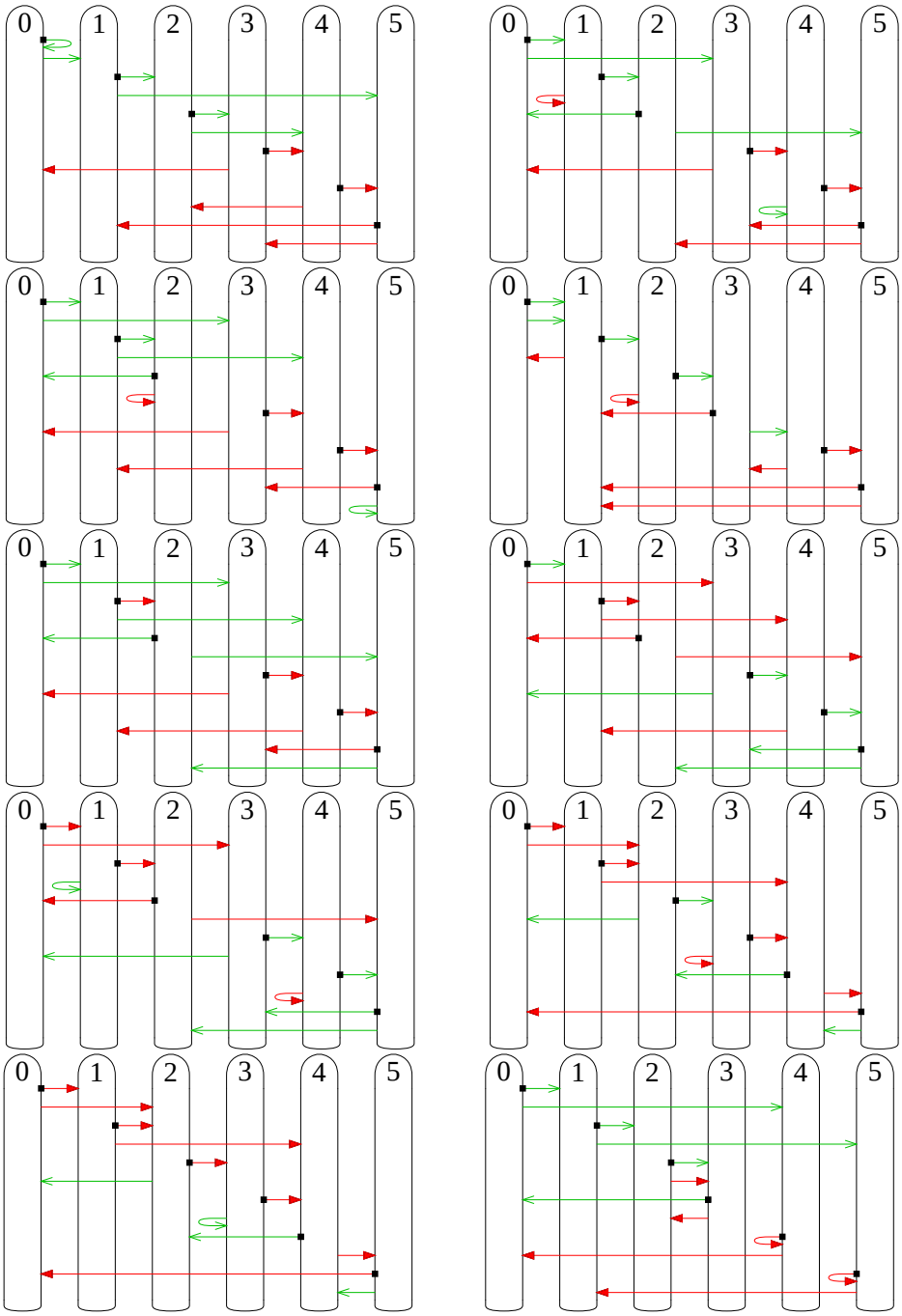


Abbildung 5.8: Die zehn besten Sechs-Zustandsautomaten A6 bis J6

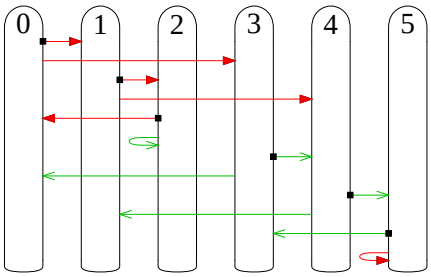


Abbildung 5.9: Zu Vergleichszwecken selektiv bestimmter Automat K6

Tabelle 5.3: Benötigte Generationen bis zum vollständigen Besuch aller Positionen der Automaten A6 bis K6

Feld	$\lvert\mathbb{A}\rvert$	A6	B6	C6	D6	E6	F6	G6	H6	I6	J6	K6
#1	50	630	195	247	216	430	227	257	—	—	240	—
#2	64	353	129	123	383	252	251	219	453	234	123	176
#3	58	—	239	291	—	300	159	185	—	—	236	209
#4	53	283	215	215	—	216	231	137	—	—	165	169
#5	48	461	190	190	154	358	325	157	250	237	190	233
#6	196	2067	694	505	1266	771	772	415	603	603	—	387
#7	54	377	—	—	135	287	217	—	483	275	103	—
#8	342	2104	1438	1286	889	1731	1361	1824	2889	2869	1280	1900
#9	342	2108	1671	1747	1408	2468	2400	1217	1867	1344	1204	1209
#10	342	2431	1292	1368	1221	1261	1483	2216	923	989	1302	2272
#11	342	2942	2063	2139	1244	1332	1432	1279	1979	1970	1050	1211
#12	342	2024	1944	2020	1037	2257	2222	1944	2300	2173	1247	2020
#13	532	3120	3070	2355	2204	1892	2165	2861	2142	2142	2574	2429
#14	532	4170	3327	3404	2125	1781	2236	3020	3202	4205	2742	3175
#15	532	3158	2599	3135	1961	3399	3413	2592	3682	3293	—	2314
#16	266	3540	1518	—	—	—	—	1156	—	—	—	1069
#17	318	2525	1373	1953	—	—	—	2132	—	—	1610	2169
#18	341	2422	1266	1236	1955	1267	1235	1266	2568	1998	—	1236
#19	341	2450	1282	1245	2344	1305	1334	1282	2303	2344	—	1245
#20	341	2402	1262	1225	2324	1285	1377	1325	2600	2370	—	1346
#21	44	348	165	166	252	168	157	157	318	300	—	176
#22	200	1656	747	728	1432	824	788	768	1394	1394	—	705
#23	200	—	689	784	1323	730	755	731	1343	1416	—	730
#24	200	—	770	774	1233	779	736	715	1217	1119	—	685
#25	72	545	187	139	163	371	301	187	651	359	190	139
#26	75	578	—	—	165	380	310	—	679	373	236	—

Tabelle 5.4: Auswertung von Erfolg und Geschwindigkeit

Automat	Anzahl	$\bar{v}$
G6	24	4,17
B6	24	4,24
F6	24	4,41
E6	24	4,80
K6	23	4,13
C6	23	4,28
A6	23	7,75
D6	22	4,58
I6	21	5,63
H6	21	6,54
J6	16	3,65

Tabellen C.1 aufgeführt. Anscheinend gibt es Automaten, die für ein kooperatives Zusammenarbeiten besser geeignet sind als für singuläre Aufgaben. Aber auch die solistisch gut agierenden Automaten sind bei ausreichender Anzahl für ein erfreuliches Gruppenergebnis geeignet.

Es zeigt sich generell, dass mit mehr Kreaturen weniger Generationen benötigt werden, also weniger Zeit erforderlich ist. Allerdings ändert sich dies nicht im direktem Zusammenhang. So gibt es vorteilhafte, aber auch behindernde Kollisionen. Ein besonders erfolgreiches Beispiel zeigt Abschnitt C.3, bei dem sich jede Kollision optimal auf die Raumüberquerung auswirkt.

Einen Vergleichsmaßstab bietet die Kennzahl der *Arbeit*, ein Produkt aus Zeit, also benötigte Generationen, und der Kreaturanzahl $|\mathbb{I}|$. Würde ein Mehr an Kreaturen ohne jegliche Effekte ablaufen, so wäre der Arbeitswert unverändert. Die Arbeit für gleiche Umgebung und gleichen Automaten, aber unterschiedliche Kreaturanzahl stellt die Tabelle 5.5 beispielhaft für einige in der Solo-Version erfolgreiche Automaten dar. Die Umgebung ENV2 ist nicht enthalten, da es bei der Auswahl keinen Automaten gab, der alle Positionen erfolgreich besucht hat.

Tabelle 5.5: Arbeit ausgewählter Fälle

$\lvert\mathbb{I}\rvert$	ENV0:C6	ENV1:C6	ENV3:A6	ENV4:A6
1	3 333	5 214	12 797	18 117
2	6 534	4 986	12 086	15 724
4	4 480	3 512	10 664	10 828
8	5 032	9 896	11 776	27 840
12	7 572	3 492	13 080	24 972
16	24 256	6 320	14 400	13 712
28	6 132	10 976	25 844	14 952
32	7 136	8 928	31 456	35 264
60	3 780	8 340	31 680	29 580
64	7 936	20 032	81 536	37 696

Die logarithmische Darstellung der Arbeitseinsparung bezüglich einer Kreatur in Abbildung 5.10, hervorgegangen aus den Werten der Tabelle 5.5, zeigt, dass es nicht immer von Vorteil ist, mehrere Kreaturen am Start zu haben. Allerdings kann durch die Eigenschaft eines dynamischen Hindernisses, entstanden durch Kollisionskonflikte, dazu beitragen, dass auch ungünstigere Automatenverhalten ein positives Ergebnis erbringen. Besonders beeindruckend ist der Erfolg des Automaten I6, der die beste Effizienz aus den Kollisionskonflikten zieht – zu sehen in Abschnitt C.3 –, allerdings als einzelne Kreatur keines der Felder ENV0 bis ENV4 vollständig abschreiten kann.

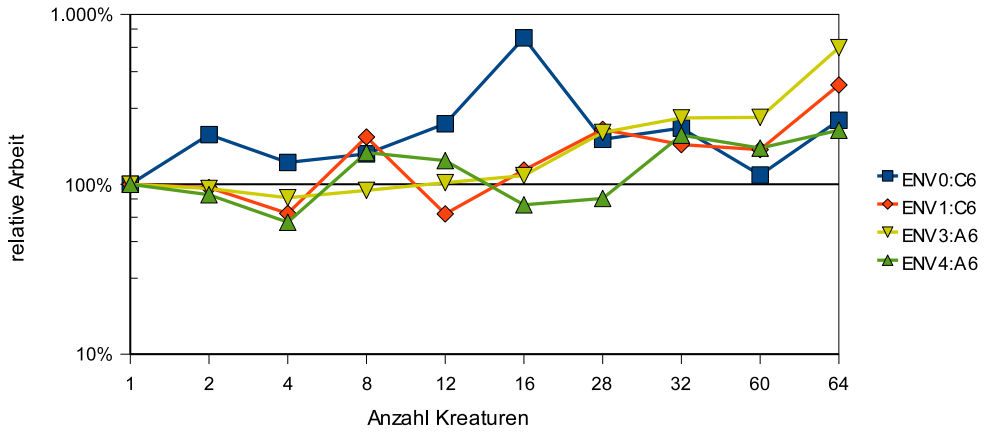


Abbildung 5.10: Arbeitsvergleich relativ zu $|\mathbb{I}| = 1$

Ab 28 Kreaturen ist es jedem Automaten A6 bis J6, also nicht K6, möglich, die Felder ENV0 bis ENV4 vollständig zu begehen. Ein Vergleich der Effizienz anhand der vorhin

definierten mittleren Geschwindigkeiten mit zusätzlicher Einbindung der Kreaturanzahl ergibt die Formel

$$\bar{v} = \frac{g \cdot |\mathbb{I}|}{|\mathbb{A}|}$$

mit g als Repräsentant der benötigten Generationen zum Erreichen aller Positionen. Damit entspricht dieser Wert dem der Arbeit geteilt durch die konstante Anzahl erreichbarer Felder. In Relation zu $|\mathbb{I}| = 1$ betrachtet ergeben sich somit für $\bar{v}$ die gleiche Darstellung wie bereits mit Abbildung 5.10 gezeigt. Daher kann der Wert $\bar{v}$ nur als allgemeiner Vergleichswert z. B. gegenüber der Tabelle 5.4 dienen. Folglich fasst die Tabelle 5.6 alle Umgebungen je Automat und Kreaturanzahl zusammen, wobei wieder wie für den Wert der Fünf-Zustandsautomaten vor der Division erst alle Generationen und freien Felder zusammenaddiert werden.

Es gibt Automaten, die in einen kleinen Bereich schwanken, andere sind extrem differenziert bezüglich der Kreaturanzahl. Insbesondere bei I6 sind weniger Kreaturen im Durchschnitt erfolgreicher als mit doppelt so vielen. Allerdings ist es auch gerade dieser Algorithmus, der auf dem Feld ENV0 am wenigsten Generationen benötigt und daher mit $\bar{v} \approx 2,59$ die geringste mittlere Geschwindigkeit aufweist. Eine generelle Aussage ist daher nicht möglich, da die Schwankungsbreite umgebungsabhängig recht groß ist.

Tabelle 5.6: Mittlere Geschwindigkeit bei unterschiedlicher Kreaturanzahl $\bar{v}_{|\mathbb{I}|}$

Automat	$\bar{v}_{28}$	$\bar{v}_{32}$	$\bar{v}_{60}$	$\bar{v}_{64}$	**Durchschnitt**	**Std.-Abweichung**
A6	15,44	18,65	20,41	29,75	21,06	3,55
B6	15,59	88,79	22,31	21,16	36,96	20,02
C6	16,35	18,37	18,25	29,00	20,49	3,32
D6	12,27	22,04	25,93	19,95	20,05	3,32
E6	44,32	39,81	19,47	13,90	29,37	8,63
F6	19,18	11,86	25,35	20,09	19,12	3,20
G6	98,40	18,18	15,84	33,08	41,37	22,39
H6	94,35	67,13	26,96	46,99	58,85	16,62
I6	27,55	27,11	103,95	68,26	56,72	21,32
J6	9,49	12,01	11,09	17,05	12,41	1,88

Orientiert am Ergebnis für die Felder aus Abbildungen C.6 und C.7 erscheint es plausibel, die Vorauswahl von Sechs-Zustandsautomaten auf anderen Feldern basierend und gleichzeitig für den Einsatz mehrerer Kreaturen optimiert zu bestimmen. Bisher gelang es erst mit acht Kreaturen, alle Felder vollständig abzuschreiten – und dann auch nur für den bisher gefundenen Algorithmus J6, der dafür höchstens 4 831 Generationen benötigte. Von daher beginnt die Suche im ersten Ansatz mit acht Kreaturen pro Feld und einer maximal vorgegebenen Generationsanzahl von $(\lceil 4831 \div 512 \rceil + 1) \cdot 512 = 5632$, die es zu unterschreiten gilt.

Die Durchführung erfolgt mit der Hardwarearchitektur aus Abbildung 5.2, die dafür eingesetzten Automaten mussten neben den üblichen Auswahlkriterien arrangiert, alle Zustände genutzt und reduziert auch dem überprüften Ausgabeverhalten entsprechen – allerdings mit der schärferen Einschränkung, dass eine vollständige Drehung bei $x = 0$ bereits nach vier Testschritten erfüllt sein musste, so dass sich die Anzahl der zu untersuchenden Automaten auf 885 997 824 reduziert hat. In die 32 Automaten umfassende Ergebnisliste kamen mit einer Messungenauigkeit von $\lceil \lceil 8192 \div 16 \rceil \div (8+2) \rceil = 52$ Generationen alle Automaten, die nicht mehr als 512 Generationen des bisher besten Er-

gebnisses benötigten, ein Teil der so entstandenen Automaten ist in Abbildung 5.11 nach Erfolg sortiert dargestellt, die Werte je Felder sind im Detail in Tabelle C.4 aufgeführt.

Die Automaten erhalten ihre Bezeichnung nach der gemessenen Generationsanzahl aufsteigend sortiert und sind einander in Tabelle C.3 zugeordnet. Demnach ist der erste Automat 6-1 der bestgeeignetste Sechs-Zustandsautomat für die Felder aus Abbildungen C.6 und C.7 für jeweils acht Kreaturen mit 1 027 benötigten Generationen zum vollständigen Überschreiten aller Positionen, dicht gefolgt vom zweiten Automaten mit 1 092 Generationen, der sogar auch alle Felder #1 bis #6 vollständig abschreiten kann. Mit Abstand folgt der dritte Automat mit 1 262 Generationen.

Bei genauer Betrachtung hat der Automat 6-3 bei freien Feldern zwei Zyklen: jeweils abwechselnd Rechts- und Linksdrehung. Im einen Fall sind zwei gleiche Drehungen hintereinander ausgeführt, im anderen Fall direkt alternierend. Die Art des Modus wird eher zufällig durch die Umgebung und deren Lage der Hindernisse bestimmt. Im Falle des Blockierens vor einem Hindernis wendet der Automat nur konstant eine Drehrichtung an, auch wenn es über mehrere Zustände hinweg zur Anwendung kommt. Vom anschaulichen her ist es plausibel, dass dieser Automat zu den erfolgreichen zählt. Auch der fünftbeste Automat 6-5 hat diesen Aufbau, allerdings mit einem anderen Wechselmechanismus an anderer Stelle. Die anderen aufgeführten Automaten haben nur einen Zyklus bei freier Bewegung, der aber die Wechselfolgen kombiniert. In blockierter Situation ist die Drehrichtung immer die gleiche – mit einziger Ausnahme des letztes Automaten 6-32, bei dem von Zustand 5 ausgehend initial eine andere Drehung erfolgt.

Bemerkenswert ist auch die Drehrichtung, z. B. der Automaten 6-2 und 6-4. Diese ist invers zueinander, also rechts und links sind getauscht, alle Zustandsübergänge sind aber identisch. Dies ist auch bei den Automaten 6-5 und 6-6, 6-7 und 6-8, 6-9 und 6-10 sowie 6-1 und 6-12 zu sehen. Von daher ähneln sich die Automaten zueinander, allerdings ist ein allgemeines Konstruktionsprinzip z. B. aus Automaten mit weniger Zuständen nicht erkennbar. Auch die allgemeine Funktionsweise ist höchst unterschiedlich – die Zustände sind sehr unterschiedlich ineinander verwoben oder in einer Sequenz aufgereiht –, so dass offensichtlich keine allgemeine Regelmäßigkeit der Automatenstruktur ableitbar ist.

Ein Vergleich mit den bisherigen Feldern #1 bis #6 zeigt kein so gutes Abschneiden der neu gefundenen Automaten. Die Simulationsergebnisse zeigt Tabelle C.5. Neben den bereits erwähnten Automaten 6-2 erreicht nur noch 6-25 alle Positionen. Problematisch sind dabei meist nur ein bis zwei Felder, an denen die meisten Automaten scheitern. Daher scheinen kleinere Felder mit eingeschränkter Bewegungsfreiheit sehr viel problematischer zu sein als die größeren Varianten. Von daher ist das Einsatzgebiet der Kreaturen vorher möglichst präzise abzuschätzen, um eine brauchbare Auswahl an generierten und vorgetesteten Automaten zu erhalten.

5.2.6 Simulation mit zwei Automaten

Abbildung 5.12 zeigt alle Automaten, die nach einer Simulation mit acht Kreaturen für die Felder ENV0 bis ENV4 bei der eingestellten Obergrenze erfolgreich waren. Dabei kamen auf einem Feld zwei unterschiedliche Automaten mit jeweils drei Zuständen zum Einsatz, die gleichen Kreaturen waren gegenüberliegend angeordnet, d. h. oben und unten starteten Kreaturen mit dem ersten Automaten, rechts und links welche mit dem zweiten Automaten. Alle aufgeführten Automatenkombinationen haben die Simulation nach 16 542 Generationen beendet. Dies ist der minimale Wert, der sich nach 2 180 053 Simulationen ergeben hat.

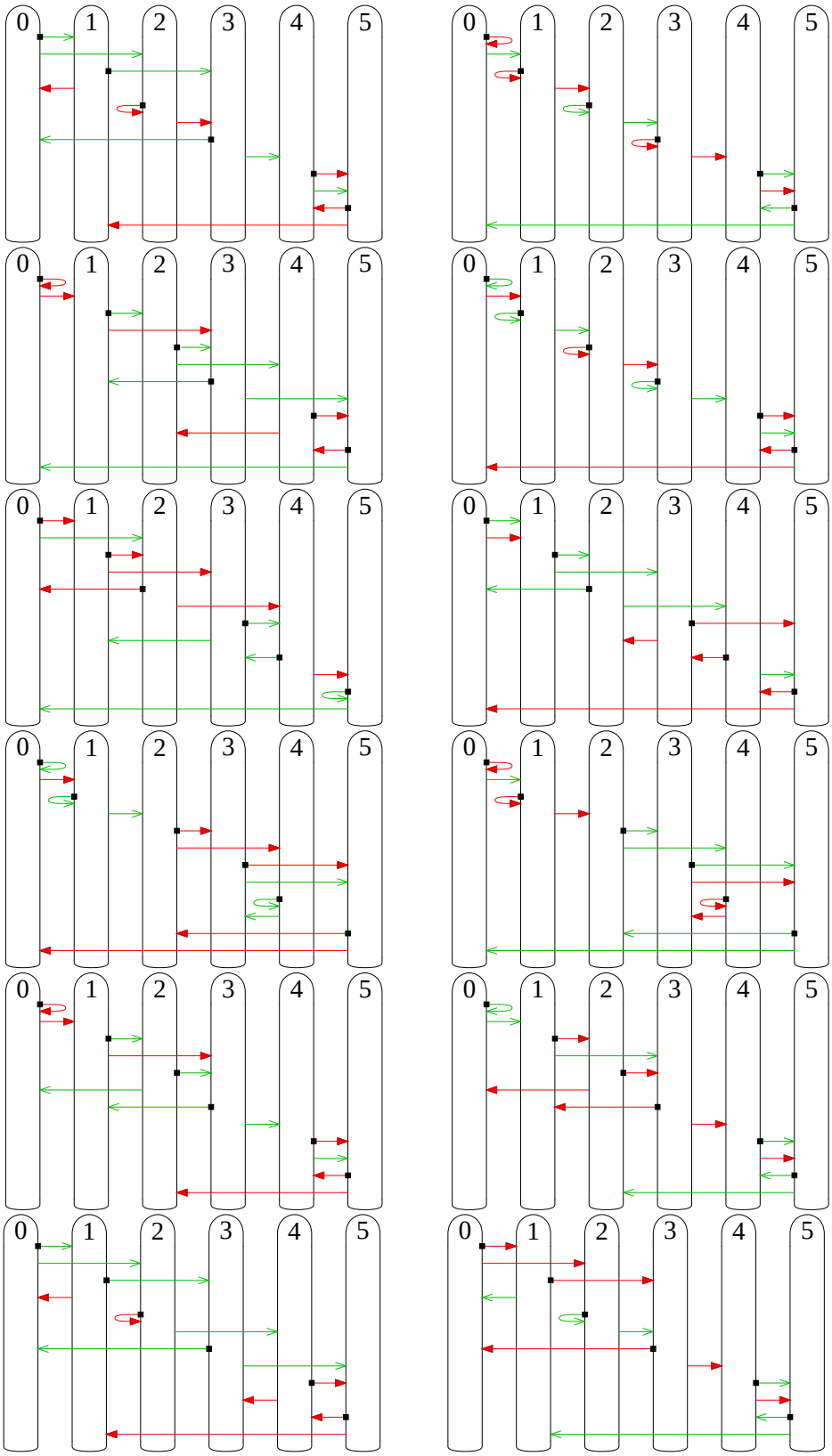


Abbildung 5.11: Die bei acht Kreaturen auf den Feldern ENV0 bis ENV4 erfolgreichsten Automaten 6-1 bis 6-12 aus der Direktsimulation mittels Hardwareberechnung

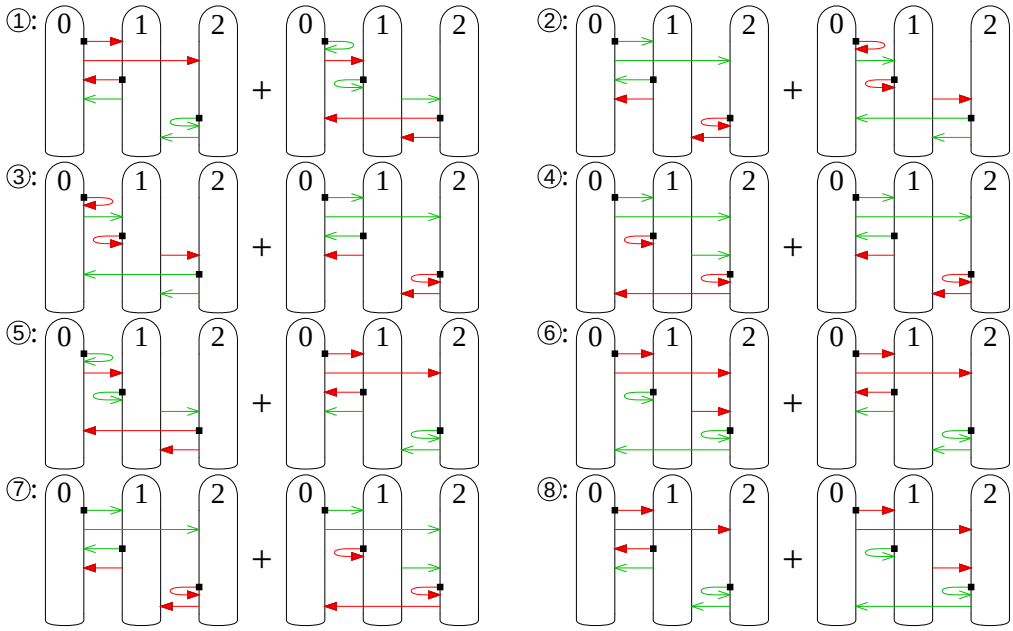


Abbildung 5.12: Kombinierte Drei-Zustandsautomaten

Die Automaten in ① und ⑤ sind lediglich miteinander vertauscht, gleiches gilt für ② und ③, ④ und ⑦ sowie ⑥ und ⑧. Bezüglich der Drehrichtung ist ① zu ②, ③ zu ⑤, ④ zu ⑥ sowie ⑦ zu ⑧ invers. Damit gibt es zwei Gruppen, angeführt durch ① und ④, deren invertierte und vertauschte Varianten die anderen Ergebnisse bilden. Zusätzlich ist auch noch der erste Automat von ① mit dem zweiten Automaten von ⑥ identisch, also dem bezüglich Drehrichtung inversem zu ④. Der andere Automat aus ① ist eine zyklische Permutation bezüglich des Startzustandes des ersten Automaten aus ⑥. Somit verbleibt einzig ① als unikate Lösung, alle anderen Ergebnisse sind Abwandlungen davon.

Es zeigt sich wieder, dass – ähnlich zu den Sechs-Zustandsautomaten – bei blockierter Bewegung hauptsächlich eine Art der Bewegungsrichtung vorherrscht. Bei freiem Voranschreiten wechseln sich die Drehrichtungen ab, allerdings dominiert aufgrund der ungeraden Zustandsanzahl eine der beiden Drehrichtungen.

Diese erfolgreiche Automatenkombination setzt sich auch mit 64 Kreaturen fort. So braucht eine Simulation auf den Feldern ENV0 bis ENV4 89, 158, 297, 145 bzw. 925 Generationen. Abgesehen vom letzten Wert liegen diese relativ nahe an denen eines Sechs-Zustandsautomaten, ersichtlich in Tabelle C.2 – und dies mit nur der halben Zustandsanzahl. Von daher ergibt sich ein interessantes Forschungspotential.

Eine Einschränkung besteht allerdings. Für zwei Automaten mit je sechs Zuständen dauert die Aufzählung bereits etwa 13,8 Jahre, mit je fünf Zuständen sind es unter den gleichen Bedingungen nur knapp 17 Stunden, wenn alle Kriterien inklusive Startzustandsisomorphie und Ausgabeverhalten zum Einsatz kommen. Hinzu kommt die Zeit für die Simulationsdurchführung.

5.2.7 Simulation mit vier Automaten

Mit zwei Zuständen ist aber selbst die Simulation mit vier Automaten von kurzer Dauer. Je Seite ist dann ein anderer Automat für die Kreaturen möglich. Das Ergebnis mit insgesamt acht Kreaturen zeigt die Abbildung 5.13, die Reihenfolge ist die der Startpositionen oben, unten, links und rechts. Es zeigt sich auch hier die Ähnlichkeit der einzelnen Auto-

maten. Unter Einbeziehung der Ausgabewertpermutation setzt sich das Ergebnis aus drei Automatentypen zusammen. Dies sind zufällig die ersten drei Automaten in der ersten Zeile der Abbildung 5.13.

Die Konstellationen der Automatentypen entsprechen je Zeile *abcc*, *bacc*, $\bar{b}\bar{a}\bar{c}\bar{c}$, $\bar{a}\bar{b}\bar{c}\bar{c}$, $\bar{c}\bar{c}\bar{b}\bar{a}$, *ccab*, $\bar{c}\bar{c}\bar{a}\bar{b}$, *ccba*, wobei z. B. $\bar{c}$ für den Automatentypen *c* mit permutierten Ausgabewerten steht. Dieser entstandene Automat $\bar{c}$ entspricht in diesem Fall auch dem Automaten *c* mit permutierten Zuständen, so dass der Zustand 1 zum Startzustand gerät.

Es zeigt sich, dass die Automaten links und rechts oder oben und unten als Startposition im Feld paarweise zusammengehören. Die eine Kombination setzt sich aus den Typen *a* und *b* zusammen, die andere Kombination besteht lediglich aus dem Typ *c*. Bemerkenswert ist auch die Tatsache, dass entweder alle oder kein Automat permutierte Ausgabewerte vorweist.

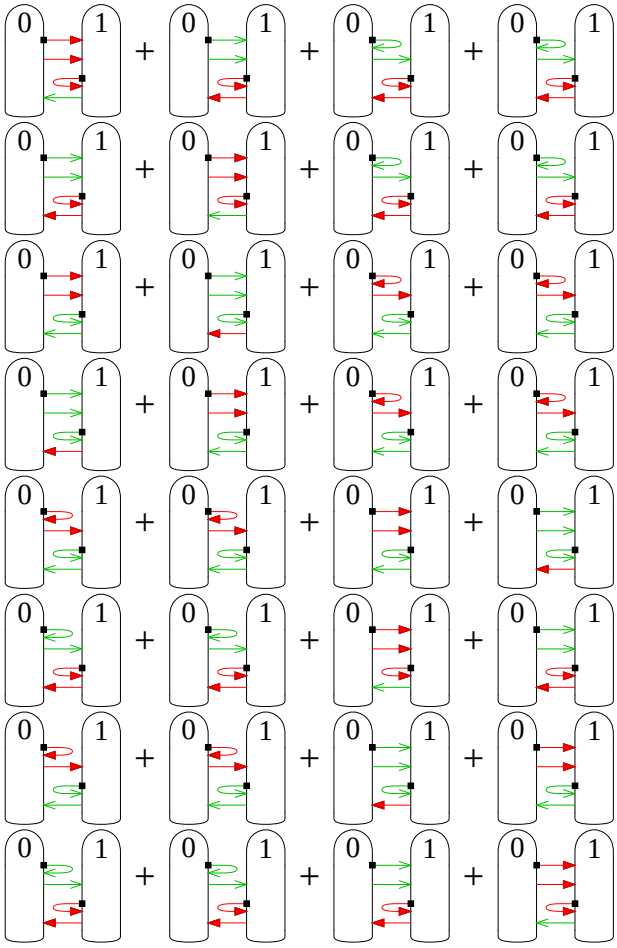


Abbildung 5.13: Kombinierte Zwei-Zustandsautomaten

5.3 Bewertung

Es hat sich zwar eine bestimmte Eigenschaft bei den ausgewählten Automaten gezeigt, die Variationen, die dabei möglich und auch aufgetreten sind, bleiben aber zahlreich. Daher ist die Aufzählung mit entsprechenden Überprüfungen, bevor es zur Simulations-

durchführung kommt, ein ideales Mittel, um garantiert den besten Automaten für die gegebenen Problemlösungen zu gewinnen.

Eine bemerkenswerte Eigenschaft zeigen die kombinierten Automaten. Die Permutation der Ausgabewerte führt zu ähnlich guten Automaten. Daher ist es eine gute Option, für eine erste Näherung die Ausgabewerte als vertauschbar zu betrachten. Mit den dann gewonnenen Simulationsergebnissen lässt sich in weniger Schritten mit den in einer ersten Simulation bewährten Automaten die optimale Ausgabewertkonstellation ermitteln.

Betrachtet man sich die gefundenen Automaten, so fällt auf, dass diese alle einen großen Zyklus bei freier Bewegungsmöglichkeit aufweisen. Zum einen erfüllt es die geforderte Eigenschaft, dass alle Zustände erreichbar und damit genutzt sein müssen, zum anderen wäre dies auch über die blockierenden Zustandsübergänge hinreichend möglich. Daher ergibt sich eine weitere Forderung nach einem Maximalwert der Zyklenanzahl im Falle $x = 1$. Damit sind dann nicht nur die Ausgabewerte, sondern auch die Zustandsübergänge bei freien Positionen beschränkt – und tragen so deutlich zur Verringerung der potentiellen Automatenanzahl bei. Mathematisch stellt die Formel 5.1 eine Überprüfungsmöglichkeit dar, basierend auf den Werten von O aus dem Algorithmus 3.34.

$$|\{i \in \mathbb{S} \mid \forall j < i\colon\ \neg(O[i,j] \vee O[j,i])\}| \in \{1,2\} \tag{5.1}$$

Damit ist zwar eine erhebliche Reduzierung in der Aufzählung erreicht, allerdings führt dies noch nicht zu einem alternativen Verfahren. Regeln für ein heuristisches Verfahren sind hiermit noch nicht möglich, da die Zahl der verbliebenen Zustandsübergänge zahlreich sind, insbesondere bei blockierter Bewegung.

Insgesamt die bisherigen Ergebnisse betrachtend sind entweder die Automaten auf kleineren oder größeren Feldern nicht erfolgreich einsetzbar, wie die Werte in Abschnitt C.2 zeigen. Dies führt zu der Schlussfolgerung, dass Sechs-Zustandsautomaten noch nicht unbedingt universell einsetzbar sind. Ein Experiment für einen Automaten mit sieben Zuständen würde für die notwendigen 103 959 871 652 Experimente zu je geschätzten 1 100 Generationen im Mittel bei angenommenen 40 MHz etwa 33 Tage in der Durchführung dauern, wenn sie denn aufgrund der verfügbaren Kapazität der verwendeten Hardware überhaupt möglich wäre. Nichtsdestoweniger sind die entstandenen Automaten für weiterführende Experimente signifikant.

Kapitel 6

Auswertung

Insgesamt betrachtet hat sich für die Aufzählung von Automaten eine erfolgreiche Eingrenzung ergeben. Die Umsetzung des Anwendungsbeispiels mit den unterschiedlichen Architekturen hat die notwendigen unterschiedlichen Betrachtungsweisen aufgezeigt, die interessante Ergebnisse in der Simulation ermöglichten. Dies bildet aber nur die Basis für eine Weiterentwicklung.

6.1 Zusammenfassung

Mit Automaten ist es möglich, Verhalten zu modellieren. Um diese Automaten mit unterschiedlichen Anforderungen aufzuzählen, kamen die Auswahlkriterien arrangiert, vereinfacht, präfixfrei und reduziert zum Einsatz. Mit „arrangiert“ ist es problemlos möglich, auch eine Isomorphie der Eingangs- bzw. Ausgabewerte einzubinden, daher ist dies universeller als die Alternative „normiert“. Eine schnelle Aufzählung und damit eine wesentliche Verringerung des Suchraums hat die Sprungtechnik bewirkt, bei der in der Aufzählung nachfolgende Automaten ungeprüft einem nicht erfüllten Kriterium zugeordnet sind und daher im Voraus ausgeschlossen werden können. Mit der entwickelten Hardwarearchitektur ist es zusätzlich möglich, die Kriterien parallel zu überprüfen und insgesamt zeitsparend Ergebnisse zu erhalten.

Die gewonnen Automaten können direkt in eine Simulation mittels Hardware gelangen. Dafür stehen unterschiedliche Architekturen zur Verfügung, die, auf Problemgrößen nach vorheriger Analyse angepasst, die Automaten auf ein effizientes Verhalten hin untersuchen. Unterschiedliche Automaten lassen sich leichter in eine Architekturen mit zentral gespeicherten Zustand anbinden, da diese ohne eine Übertragung an eine andere Stelle zusammen mit dem Zustand auskommen.

Für das verwendete Anwendungsbeispiel, bei dem es darauf ankommt, eine oder mehrere Kreaturen in einem unbekannten Areal über jede Position zu leiten, haben sich bestimmte Automaten bei den meisten Feldern als besonders erfolgreich hervorgetan. Weiterhin hat die Kombination bestimmter unterschiedlich verhaltender Kreaturen einen Vorteil bezüglich der Anzahl notwendiger Schritte ergeben.

6.2 Ausblick

Neben den bereits in dem Abschnitt 4.4 beschriebenen Möglichkeiten, gibt es noch weitere Ansatzpunkte, die Experimente fortzuführen oder zu erweitern. So könnte die Auf-

zählung auch Graphen mit variabler Kantenanzahl $\leq |\mathbb{X}|$ umfassen. Dies ist z. B. für Probleme der Graphentheorie interessant. Es gibt aber noch andere Erweiterungen unterschiedlicher Aspekte.

6.2.1 Simulationsumgebung

Eine technische Weiterentwicklung der Problemstellung ist für die Kreaturen die Möglichkeit, untereinander Informationen auszutauschen. Dies könnte direkt über eine stilisierte Funkverbindung geschehen oder über Informationen, die auf den einzelnen Positionen des Feldes zum Liegen kommen. Die Interpretation ist dem Automaten überlassen. Dies kann dazu führen, dass unterschiedliche Kreaturen nicht die gleiche „Sprache“ sprechen.

In jedem Fall erhöht sich die Anzahl der Eingänge und Ausgabewerte. Eine reine Aufzählung aller Automaten ist zu komplex. Von daher empfiehlt sich ein Test nur mit vorausgewählten Automaten, die ohne Kommunikation erfolgreich sind.

Für eine direkte Verbindung der Kreaturen untereinander ist auch das Konzept der Globalen Zellularen Automaten interessant. Das Prinzip ist in [HHH04] angewendet, eine spezielle Architektur ist in [Hee07] enthalten.

6.2.2 Verhalten von Kreaturen

Erfolgt die Drehung einer Kreatur nach rechts oder links nicht durch einen Zustandsautomaten sondern durch eine Zufallssteuerung, so ergibt sich zumindest für einen Teil der Felder aus Abbildung C.1 ein vollständiger Besuch aller Positionen, wie eine Untersuchung in [DSL06] ergeben hat. Damit ist in diesem Fall der Zufall besser als ein Zwei-Zustandsautomat. Zu überlegen ist, ob eine Kombination von Zustandsautomat und Zufallssteuerung ein besseres Ergebnis hervorbringen könnte. Die Variationen sind zahlreich, allerdings die Vergleichbarkeit bei echten Zufallszahlen schwierig, da eine Wiederholbarkeit aufgrund des Zufalls nicht gewährleistet ist.

Betrachtet man die Kreaturen als Tiere und lässt sich von der Biologie inspirieren, so gibt es ein genetisches Basisverhalten, das von Geburt an vorhanden ist, und es gibt erlernbare Fähigkeiten. Eine solche Kombination von Grundverhalten mit sich selbst verändernden Automaten stellt eine interessante Erweiterung in Richtung künstlicher Intelligenz dar, deren variabler Teil gemäß [MSPPU02] erfolgen könnte.

Weitere Unterstützung könnte eine Kreatur über Glücksgefühle erfahren. Dieser Einfluss auf das Lernverhalten setzt ein Bewertungssystem innerhalb der Simulation voraus. Erzielbar ist der Glücksstatus z. B. über einen eigenen Zustand oder Zählerwert. Ein Lernsystem ist Voraussetzung, um auf Glück reagieren zu können.

Die Variationen für ein einfaches Modell zur Nachbildung der Natur sind zahlreich, die vielleicht auch Einfluss in das Themengebiet der Bionik finden könnte, deren Entdeckungen beispielhaft in [CBN05] aufgeführt sind.

Eine andere Herangehensweise zur Automatenfindung wäre eine genetisch optimierte Suche. Als Basis könnten bekannte Automaten mit weniger Zuständen dienen. Diese erfahren dann eine Veränderung durch zufällige Mutationen und eine Erfolgsauslese gemäß den genetischen Prinzipien, unter anderem beschrieben in [Poh00]. So ließe sich das Verhalten von Kreaturen automatisch finden.

Anhang

Anhang A

Semantik des Pseudocodes

Der verwendete Pseudocode soll die Algorithmen auf einfache Weise verdeutlichen. Dazu sind allgemeine Sprachelemente verwendet, die aus Pascal, C oder Perl bekannt sind. Die Operationen sind allein auf mathematischer Beschreibungsebene. Um eine Unterscheidung zwischen Vergleich und Zuweisung zu ermöglichen, ist eine (sequentielle) Zuweisung mit dem Doppelpunkt-Gleich-Operator $:=$ dargestellt.

Die Struktur der Sprache zeigt sich im Einrücken der Blöcke, wie sie bei bedingter Ausführung auftreten. Besondere Schlüsselwörter sind dadurch nicht mehr notwendig, so dass ein ungestörter Lesefluss möglich ist. In der nachfolgenden Befehlsübersicht sind diese Blöcke durch **...** dargestellt und repräsentieren einen eingerückten Block.

Die verschiedenen Anweisungen werden lediglich durch eine neue Zeile voneinander getrennt, ein Symbol wie etwa ein Semikolon ist dann nicht notwendig. Kommentare sind ebenfalls möglich und sind entweder zwischen Schrägstrich-Stern /* und Stern-Schrägstrich */ eingebunden oder werden durch zwei Schrägstriche // eingeleitet und mit einem Zeilenende beendet.

for $i := a$ **to** b **do ...**
: Schleife, in der i anfangs den Wert a erhält und danach in Einer-Schritten sich Richtung b verändert. Inkrementiert wird bei $a \leq b$, dekrementiert bei $a > b$.

foreach $i \in \mathbb{I}$ **do ...**
: Für jeden Schleifendurchlauf wird ein Element der Menge $\mathbb{I}$ verwendet. Die Reihenfolge, in der die Elemente aufgeführt werden, ist nicht relevant.

while c **do ...**
: Solange die Bedingung c erfüllt ist, wird der nachfolgende Block ausgeführt.

do ... while c
: Der Block wird ausgeführt. Dies wiederholt sich, solange die Bedingung c nach Ausführung des Blocks erfüllt ist.

endloop
: Bewirkt ein vorzeitiges Schleifenende. Bei geschachtelten Schleifen gilt dies für die direkt umgebende Schleife. Dies kann z. B. bei einer **for**-Schleife durch Setzen der Schleifenvariablen i auf den maximalen Wert b erreicht werden.

endloop i
: Bewirkt ein vorzeitiges Ende der Schleife mit der Zählervariablen i. Im Gegensatz

zu **endloop** ohne Argument ist dies nur für **for**- und **foreach**-Schleifen möglich, kann dafür aber auch über die unmittelbare Schachtelungsebene hinausgehen.

if *c* **then ...**
Der Block wird nur dann ausgeführt, wenn die Bedingung *c* erfüllt ist.

if *c* **then ... else ...**
Der erste Block wird ausgeführt, wenn die Bedingung *c* erfüllt ist. Andernfalls wird der zweite Block ausgeführt.

function *f*(Parameter) **...**
Die Funktion *f* wird deklariert, das Ende der Funktion wird durch das Ende des Blocks markiert. Die Parameter sind optional und können inklusive der Klammern entfallen; mehrere Parameter werden durch Kommata voneinander getrennt. Eine Änderung des Wertes innerhalb der Funktion wirkt sich auf den aufrufenden Programmteil aus, erfolgt in diesem Fall also als Referenzaufruf (*call by reference* statt *call by value*, siehe z. B. [Kam98, Seite 180]).

return *r*
Innerhalb einer Funktion wird mit dieser Anweisung die Funktion beendet und der optionale Wert *r* als Ergebnis geliefert.

Insbesondere Schleifen verwenden Mengen, angegeben durch einen Buchstaben mit Doppelstrich oder durch einzeln aufgeführte Elemente in geschweiften Klammern, voneinander durch Kommata getrennt, z. B. $\mathbb{I} = \{1,2,\ldots,5\}$. Je ein senkrechter Strich vor und hinter einer Menge bezeichnet die Anzahl der Elemente, die Kardinalzahl; im Beispiel ist dies $|\mathbb{I}| = 5$. Die üblichen Operatoren $\cap$ für Durchschnitt, $\cup$ für Vereinigung und $\setminus$ für Differenz sind untereinander nicht priorisiert. Daher ist die Verwendung von runden Klammern in Zweifelsfällen obligatorisch, z. B. $|(\mathbb{I}\setminus\{4\}) \cup \{8\}| = 5$ und $|\mathbb{I}\setminus(\{4\} \cup \{8\})| = |\{1,2,3,4,5\}\setminus\{4,8\}| = |\{1,2,3,5\}| = 4$.

Anhang B

Automaten mit zwei Zuständen

Um die verschiedenen Kriterien der Automateneigenschaften aus Kapitel 3 zu verdeutlichen, sind alle Automaten nach ihren Eigenschaften in diesem Anhang aufgeführt. Damit die Anzahl überschaubar bleibt, ist dies auf Automaten mit zwei Zuständen sowie zwei Eingangswerte und zwei Ausgabewerte beschränkt, also $|\mathbb{S}| = |\mathbb{X}| = |\mathbb{Y}| = 2$. Des weiteren erfolgt die Untergliederung disjunkt, so dass innerhalb eines jeden der folgenden Abschnitte kein Automat doppelt aufgeführt ist.

B.1 Hauptbewertungskriterien

Durch die in Abschnitt 3.2 ab Seite 32 eingeführten Kriterien

- normiert,
- reduziert,
- vereinfacht und
- präfixfrei

erfolgt die Hauptauswahl, unabhängig der Verwendung eines Automaten. Die Automaten, die allen vier Kriterien entsprechen, bilden die Basis für alles weitere – mit Ausnahme von „normiert“, das durch eine Zustandsisomorphie ersetzt werden kann. Die Anzahl bleibt dann zwar die gleiche, jedoch enthält das Ergebnis zum Teil andere Automaten.

Die Tabelle B.1 zeigt die Zahlen im Einzelnen, inspiriert durch das Präsentationsverfahren mittels KV-Diagramm gemäß Abbildung 2.3. Die nachfolgenden Abbildungen B.1 bis B.8 stellen dann die hinter diesen Zahlen liegenden Automaten dar.

Tabelle B.1: Anzahl der Automaten für zwei Zustände, zwei Eingangs- und zwei Ausgabemöglichkeiten in Abhängigkeit der Klassifizierung

		reduziert			
normiert	0	0	3	1	
	12	36	**108**	36	*vereinfacht*
	0	0	0	0	
	0	0	45	15	
			präfixfrei		

Abbildung B.1a: Die ersten 84 von 108 normierten, vereinfachten, reduzierten und präfixfreien Algorithmen eines Zwei-Zustandsautomaten

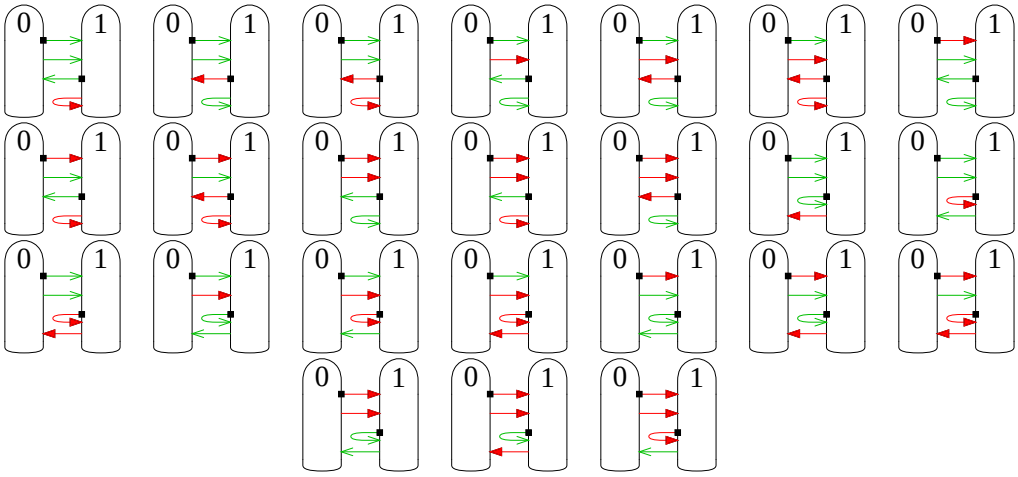


Abbildung B.1b: Die restlichen 24 von 108 normierten, vereinfachten, reduzierten und präfixfreien Algorithmen eines Zwei-Zustandsautomaten

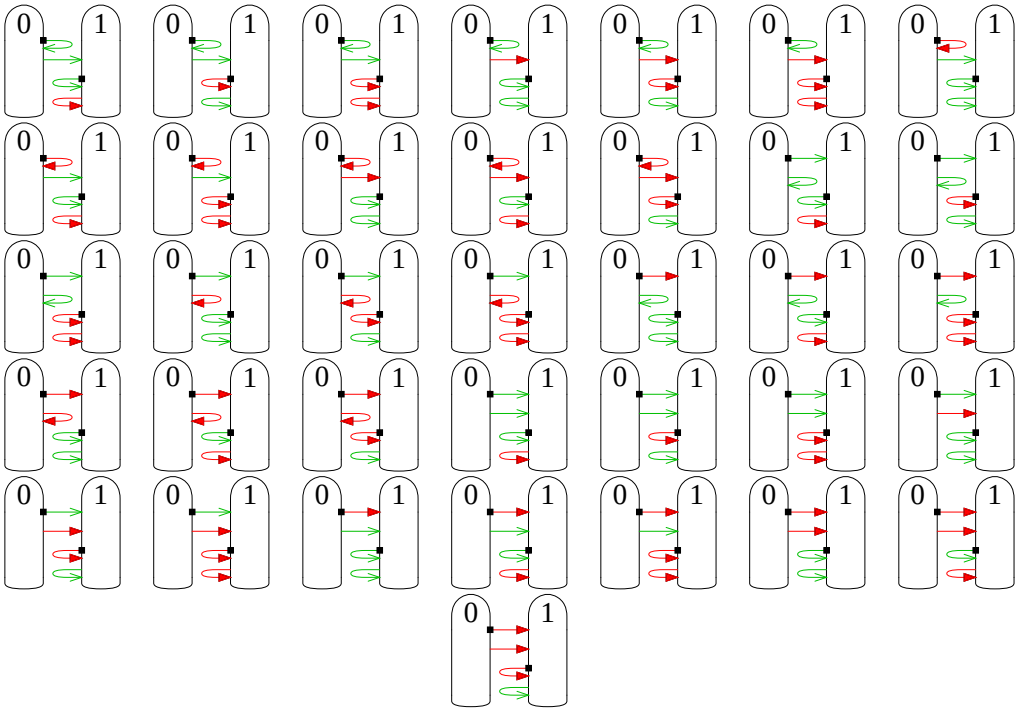


Abbildung B.2: Die 36 normierten, vereinfachten und reduzierten Algorithmen mit Präfix eines Zwei-Zustandsautomaten

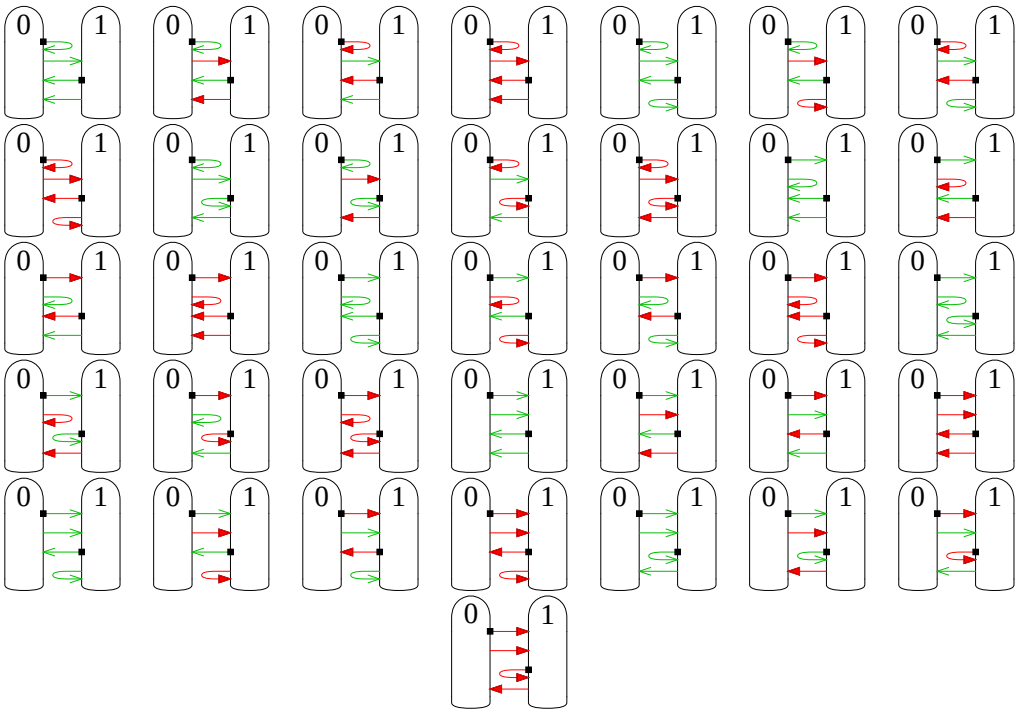


Abbildung B.3: Die 36 normierten, vereinfachten, nicht-reduzierten und präfixfreien Algorithmen eines Zwei-Zustandsautomaten

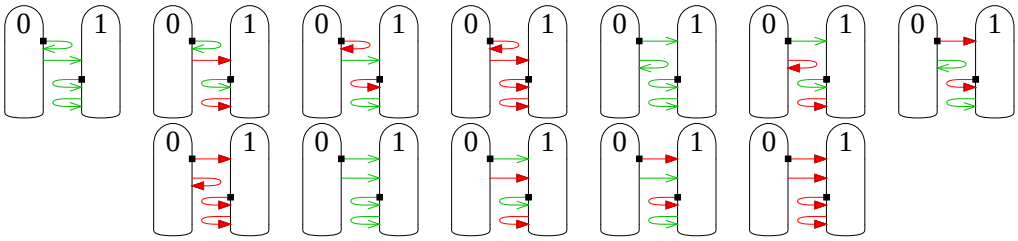


Abbildung B.4: Die zwölf normierten, vereinfachten und nicht-reduzierten Algorithmen mit Präfix eines Zwei-Zustandsautomaten

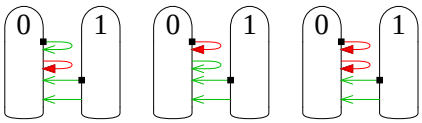


Abbildung B.5: Die drei normierten, unvereinfachten, reduzierten und präfixfreien Algorithmen eines Zwei-Zustandsautomaten

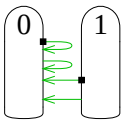


Abbildung B.6: Der einzige normierte, unvereinfachte, nicht-reduzierte und präfixfreie Algorithmus eines Zwei-Zustandsautomaten

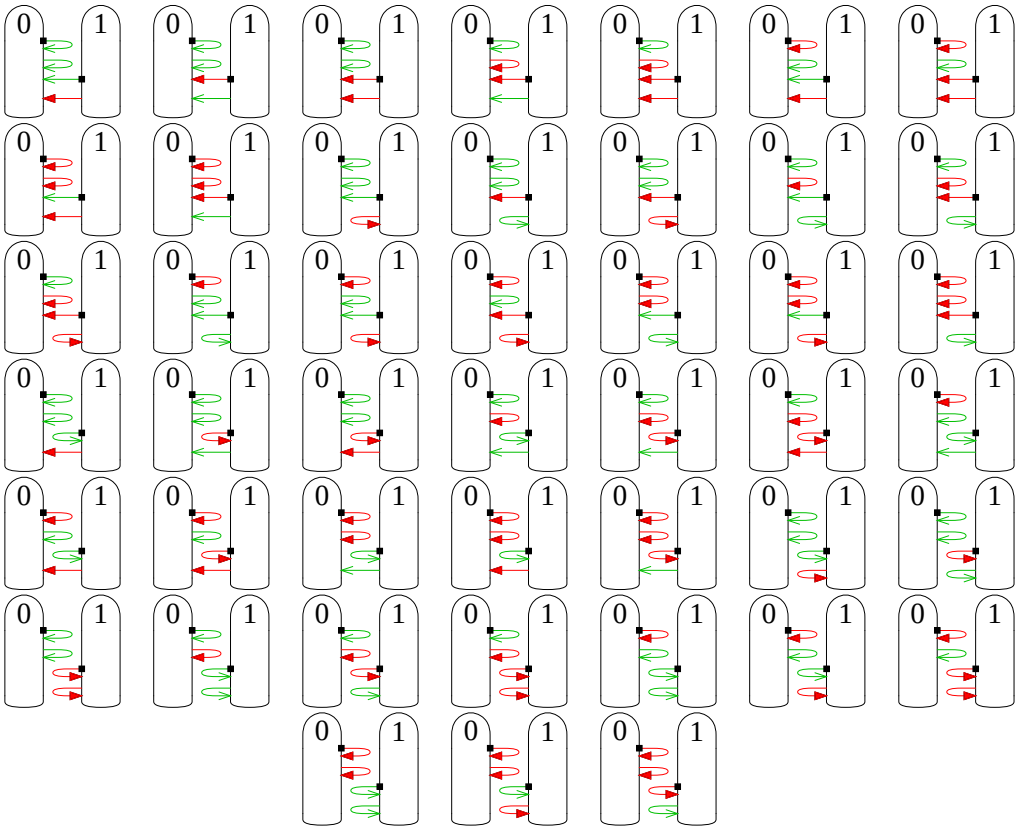


Abbildung B.7: Die 45 nicht-normierten, unvereinfachten, reduzierten und präfixfreien Algorithmen eines Zwei-Zustandsautomaten

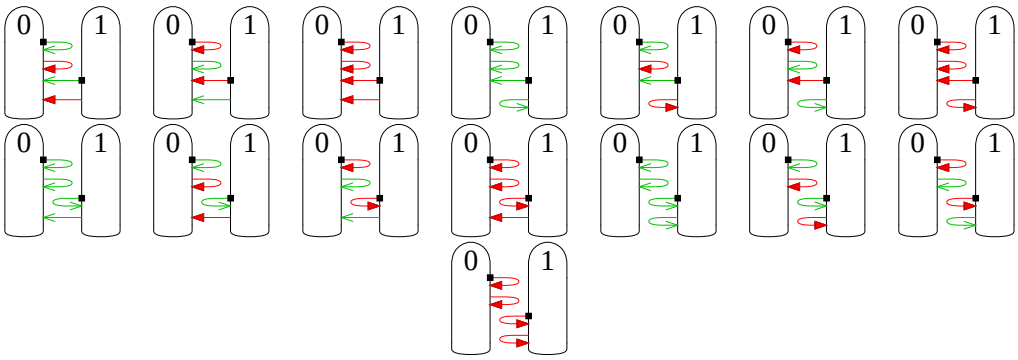


Abbildung B.8: Die 15 nicht-normierten, unvereinfachten, nicht-reduzierten und präfixfreien Algorithmen eines Zwei-Zustandsautomaten

B.2 Betrachtung des Ausgabezyklus

Für den Einsatz der Automaten zum vollständigen Abschreiten eines Gebietes gibt es weitere Kriterien zur Selektion von Automaten. Die Automaten der Abbildungen B.9 und B.10 haben nicht beide erforderlichen Bedingung bezüglich der Eingangswerte, in Abbildung B.11 sind die relevanten Automaten aufgeführt.

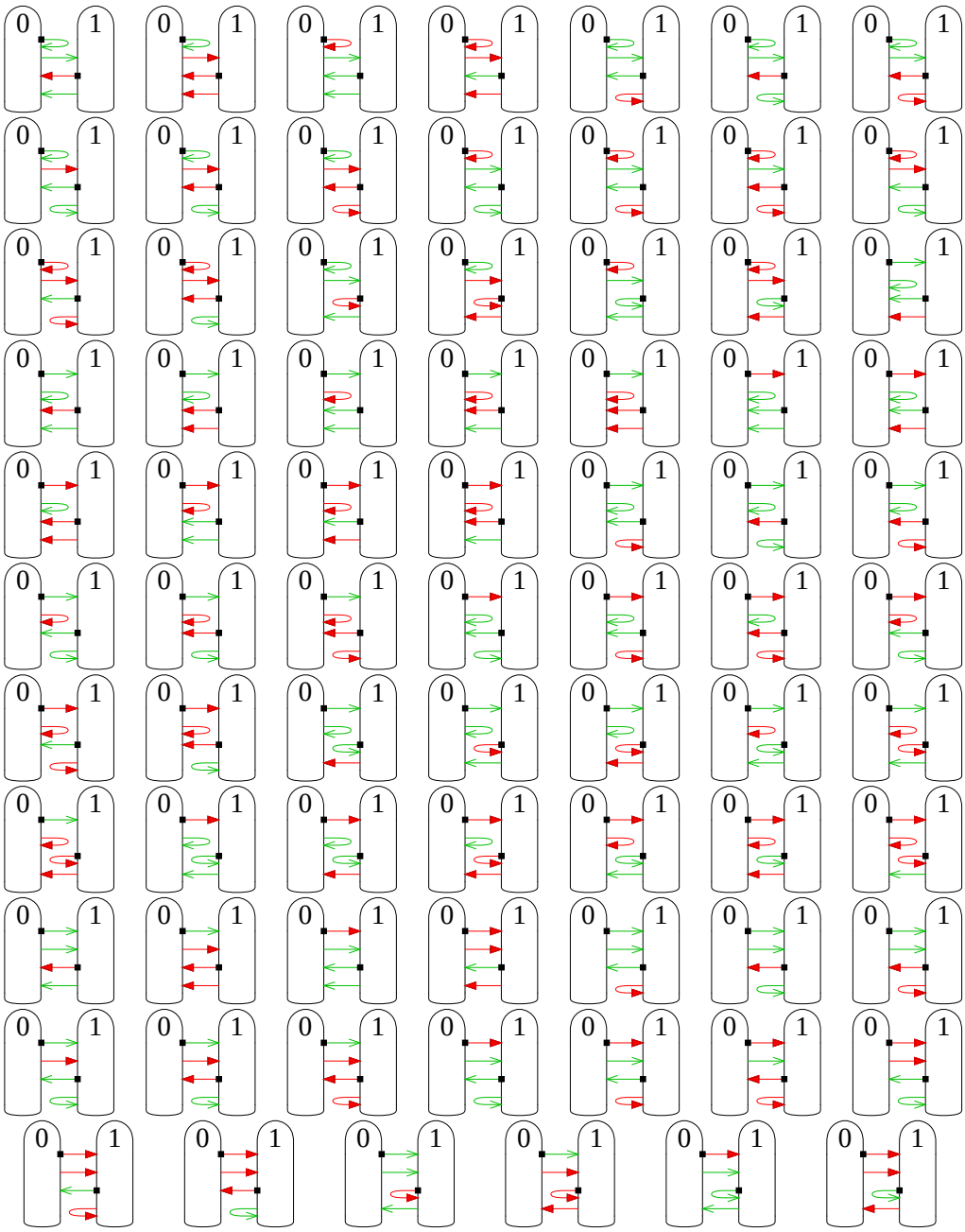


Abbildung B.9: Konstanter Ausgabezyklus bei $x > 0$

Abbildung B.10: Alternierender Ausgabezyklus, insbesondere bei $x = 0$

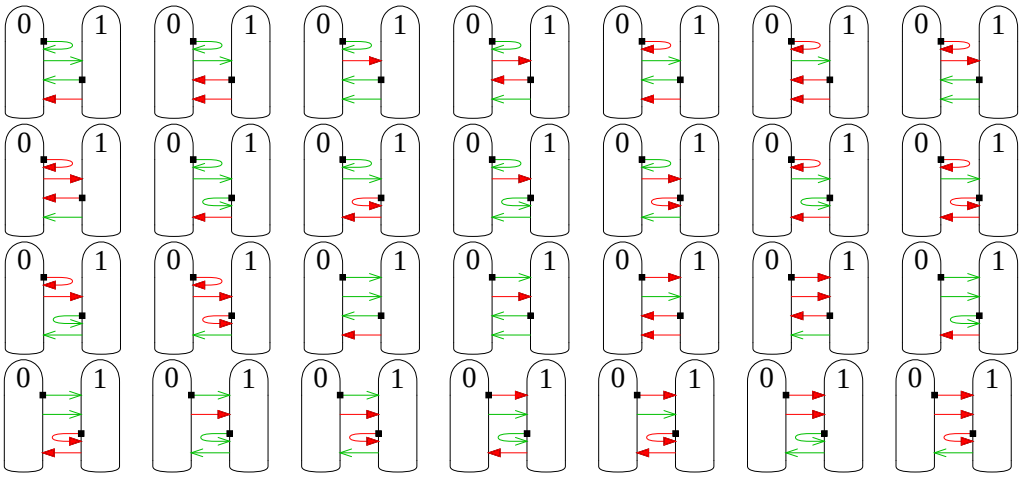


Abbildung B.11: Interessante Zwei-Zustandsautomaten

B.3 Bewertung durch Simulation

Die 28 Automaten aus Abbildung B.11 haben die Simulation mit den Umgebungen aus Abbildung C.1 durchlaufen. Dabei hat nur ein Teil der Automaten eine Kreatur bei mindestens vier Feldern über mindestens zehn Positionen bewegen können. Diese sind in Abbildung B.12 dargestellt. Zum Vergleich sind die Automaten, bei denen sich die Kreatur bei mindestens zwei Feldern nicht fortbewegt hat, in Abbildung B.13 angegeben.

Nach diesen Bewertungskriterien haben die erfolgreichen Automaten 79 bis 169 Positionen, die erfolglosen Automaten lediglich nur 18 bis 70 Positionen erreicht. Maximal möglich sind 273 Positionen. Dieses unzureichende Ergebnis rührt – neben der geringen Zustandsanzahl – von der zu hohen Hindernisdichte, so dass sich die Kreaturen eher nur an den Hindernissen entlanghangeln, statt sich frei zu bewegen. Der beste Automat ohne die Vorauswahl des Ausgabeverhaltens erreicht nur 77 Positionen und scheitert bei drei Feldern. Von daher lohnt sich die Vorauswahl zur Reduzierung von 108 auf 28 Simulationsdurchführungen.

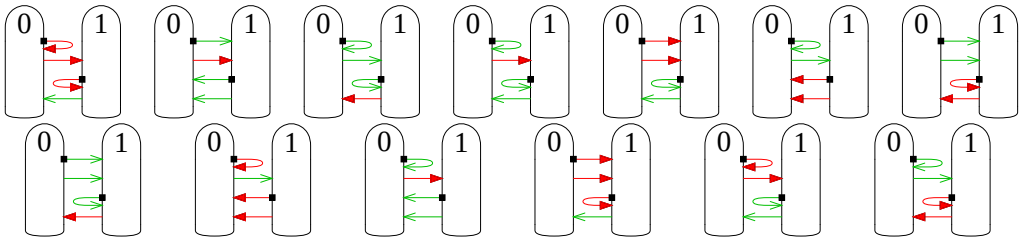


Abbildung B.12: 13 Erfolgreiche Zwei-Zustandsautomaten

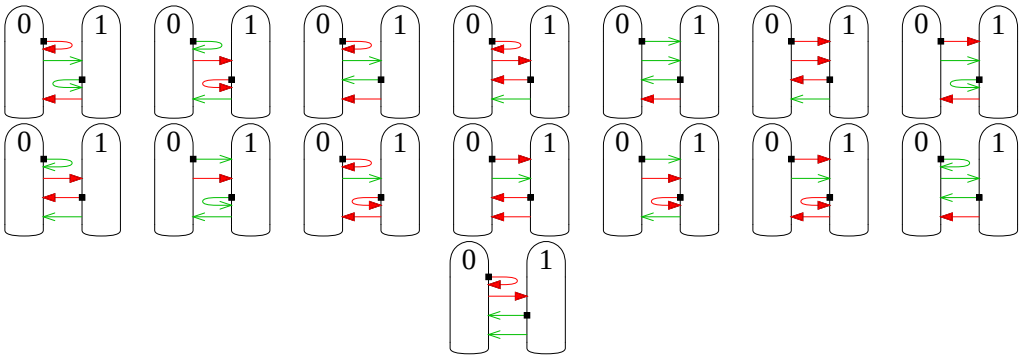


Abbildung B.13: 15 Erfolglose Zwei-Zustandsautomaten, die aber alle Auswahlkriterien erfüllen

B.4 Startzustandsisomorphie

Ein weiteres Ausschlusskriterium ist die Startzustandsisomorphie, bei der beide Zustände potentielle Startzustände sind. Die 28 Automaten aus Abbildung B.11 teilen sich zu gleichen Teilen nach den Kriterien der Abbildungen B.14 und B.15 auf.

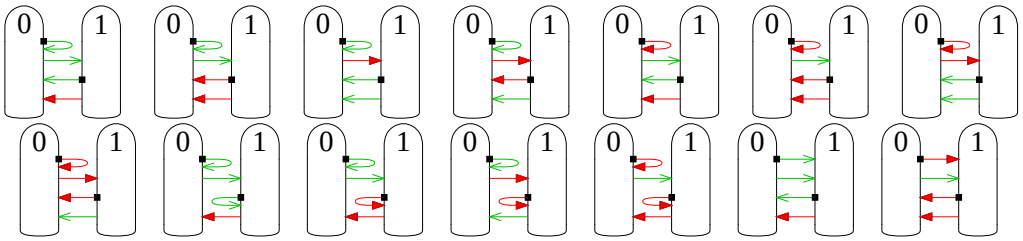


Abbildung B.14: Repräsentant bezüglich Startzustand

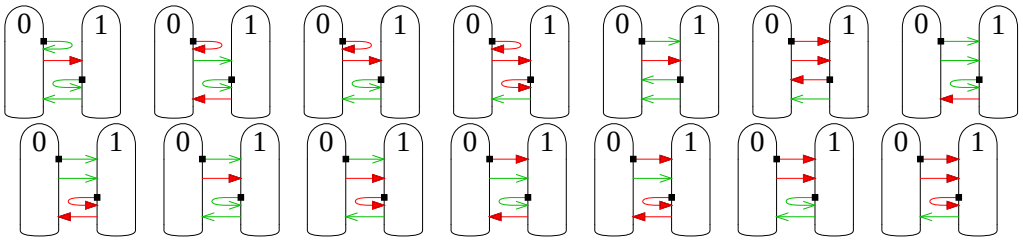


Abbildung B.15: Startzustandspermutation

Anhang C

Simulationsumgebung

Für die Simulation aus Kapitel 4 ist eine Grundlage in Form von Feldgrößen und Anordnung der Hindernisse notwendig. Die Anforderungen reichen von kleinen Feldern, insbesondere für Automaten mit wenig Zuständen, bis hin zu großen Feldern mit vielen Möglichkeiten für die Architekturen mit RAM-Unterstützung bzw. basierend auf Software. Die entstandenen Ergebnisse sind in Kapitel 5 erörtert.

C.1 Für eine Kreatur

Zum Testen einer einzelnen Kreatur in kleiner Umgebung reichen kleine Felder aus. In der Abbildung C.1 sind verschiedene Beispiele für $X = Y = 10$ aufgeführt, die erstmals miteinander in [HS03] Verwendung gefunden haben. Um die Welten unterscheiden zu können, wurden die Konfigurationen mit Nummern versehen, wobei die Welt links die Bezeichnung #1 hat. Diese hat die weiteren Werte $p_0 = (2,2)$, $r_0 =$ West und $\mathbb{H} = \{(0,0), (9,0), (0,9), (0,1), (1,0), (9,1), (1,9), (0,2), (2,0), (9,2), (2,9), (0,3), (3,0), (9,3), (3,9), (0,4), (4,0), (9,4), (4,9), (0,5), (5,0), (9,5), (5,9), (0,6), (6,0), (9,6), (6,9), (0,7), (7,0), (9,7), (7,9), (0,8), (8,0), (9,8), (8,9), (0,9), (9,0), (9,9), (4,1), (5,2), (7,2), (1,3), (2,4), (6,4), (7,4), (4,5), (2,6), (7,6), (3,7), (6,7), (7,7), (8,7)\}$.

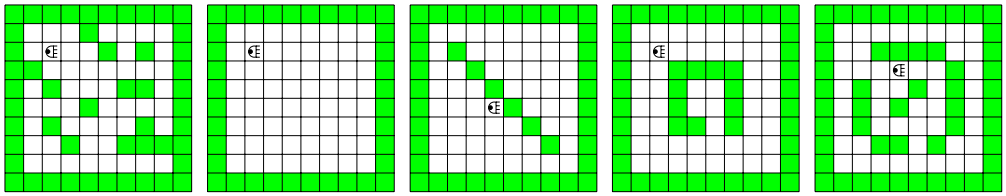


Abbildung C.1: Der Anfangszustand der Welten #1 bis #5 mit gleichem $\mathbb{P}$ und unterschiedlichem $\mathbb{H}$

Zum Tragen gekommen sind auch noch weitere, größere Felder mit unterschiedlichen Formen, zu sehen in den Abbildungen C.2, C.3, C.4 und C.5. Bei den schmalen Feldern #7 und #26 der Abbildung C.5 sind als Besonderheit bei den Automaten B6, C6 und G6 ein oder zwei Positionen, je nach Breite des Feldes, unberührt geblieben. Die Kreatur hat sich dabei lediglich auf zwei festen Routen am Rand entlanggehangelt und ist so nie in die Mitte vorgestoßen. Die problematischen Stellen sind mit einem Kreuz dargestellt. Die anderen Automaten haben das Feld problemlos bewältigt.

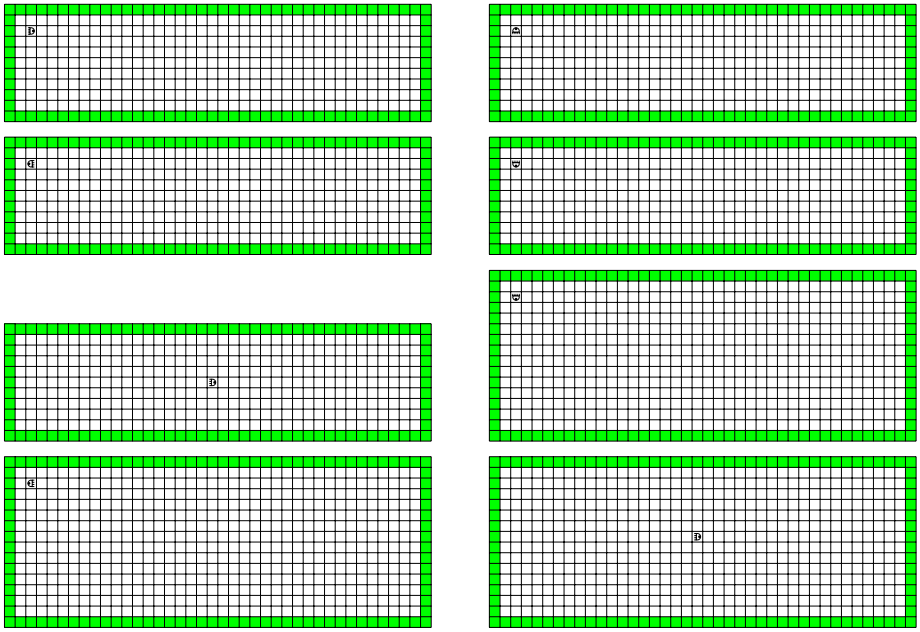


Abbildung C.2: Rechteckige Felder #8 bis #15

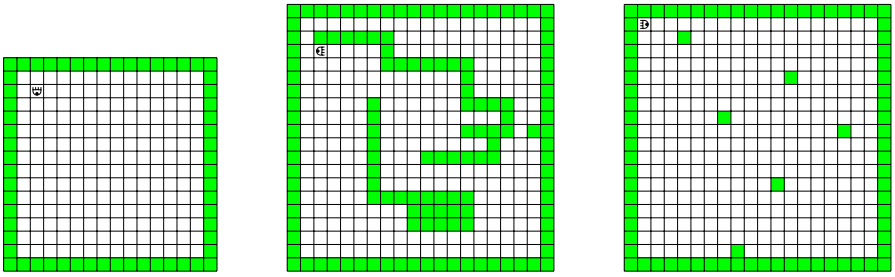


Abbildung C.3: Quadratische Felder #6, #16 und #17

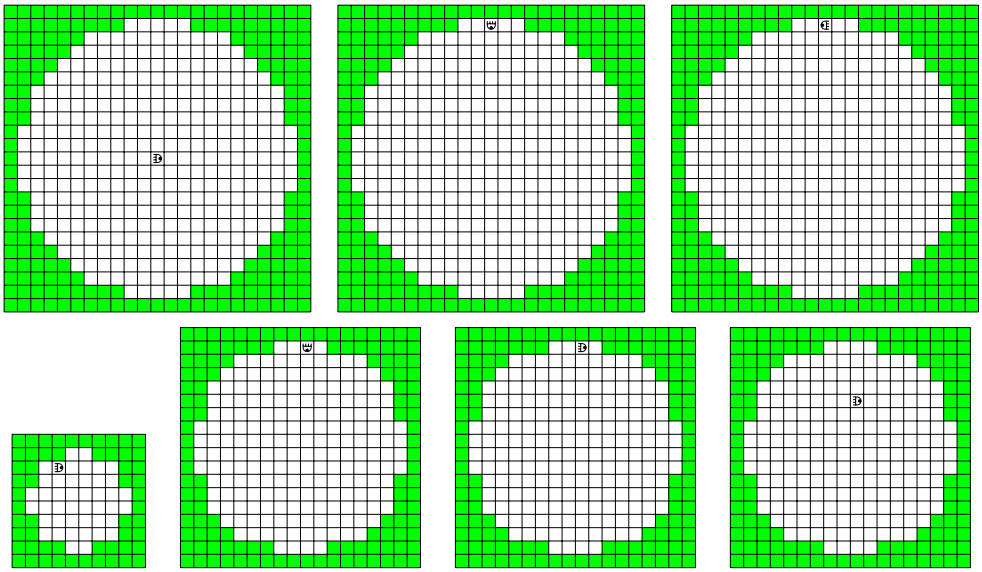


Abbildung C.4: Kreisförmige Umgebungen #18 bis #24

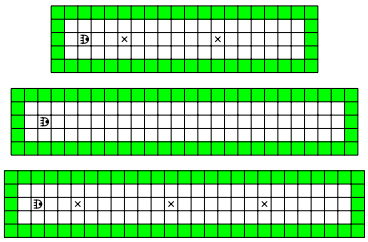


Abbildung C.5: Schmale, rechteckige Areale #7, #25 und #26

C.2 Multi-Kreatur-Felder

Basis für die größeren Felder, um mehr Kreaturen frei bewegen lassen zu können, ist ein Areal der Breite und Höhe von je 35. Dies ergibt eine Innenfläche von $33^2 = 1\,089$ Positionen. Da sich in ersten Versuchen jedoch gezeigt hat, dass die mittlere Position $(17, 17)$ von den Kreaturen nicht erreicht werden konnte, ist zur Erreichung brauchbarer Ergebnisse diese Stelle ein Hindernis geworden.

Die Kreaturen selbst sind symmetrisch am Rand angeordnet, wie die Abbildung C.6 zeigt. Dadurch gibt es keinen Unterschied zwischen oben, unten, rechts und links. Um dies auch mit Hindernissen beizubehalten, sind auch die Hindernisse in den weiteren Feldern symmetrisch angeordnet. Abbildung C.7 zeigt die entworfenen Labyrinthe, die jeweils 129 Hindernisse beinhalten. Somit besteht jeweils eine begehbare Fläche von $|\mathbb{A}| = 960$ Positionen.

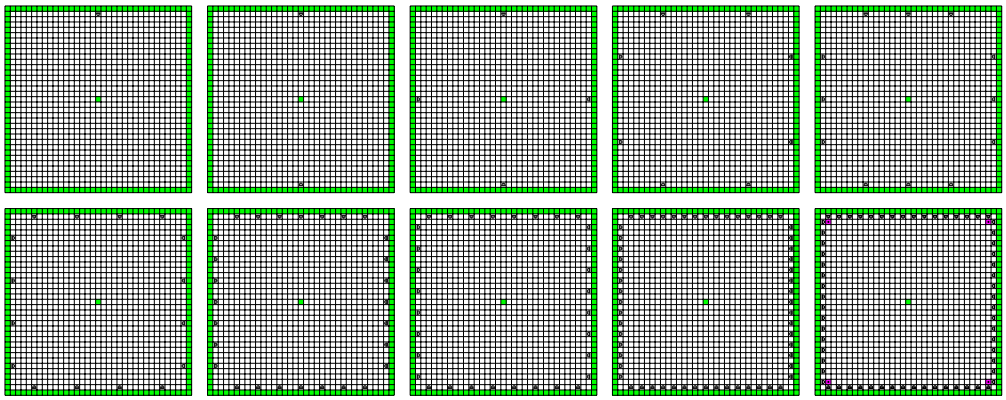


Abbildung C.6: ENV0 mit 1, 2, 4, 8, 12, 16, 28, 32, 60 und 64 Kreaturen

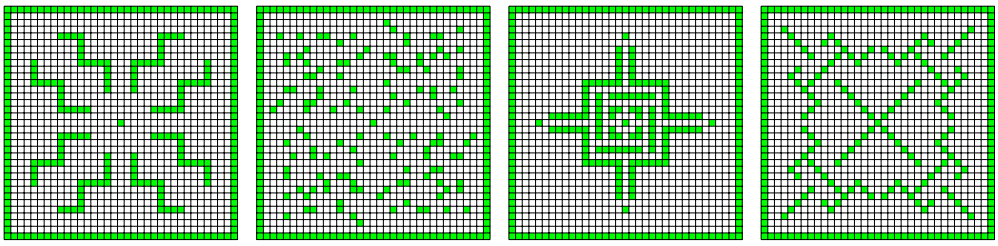


Abbildung C.7: Umgebungen ENV1 bis ENV4 ohne Kreaturen

Die Tabellen C.1 zeigen die erzielten Ergebnisse der einzelnen Umgebungen mit den jeweilig verwendeten Automaten. Jede Kreatur verwendet dabei den gleichen Automaten mit gleichem Startzustand 0. Ein Strich „—“ statt eines Generationswertes ist dann auf-

geführt, wenn selbst bei unendlicher Simulationsdurchführung kein vollständiger Besuch aller Positionen erzielbar ist.

Mit den Tabellen C.1j und C.6 ergeben sich die minimal möglichen Generationen bei 64 Kreaturen mit identischem Automaten, zusammengetragen in Tabelle C.2. Es zeigt sich eine Häufung für den Automaten B6, der somit am ehesten universell einsetzbar ist. Für größere Flächen zeigen zwar die explizit gesuchten Automaten in einigen Fällen ein besseres Verhalten, bei erhöhter Kreaturanzahl verringert sich jedoch der Vorteil, so dass keine explizite Empfehlung eines bestimmten Automaten möglich ist.

Tabelle C.1a: Generationen zum Abschreiten aller Positionen für eine Kreatur

Automat	ENV0	ENV1	ENV2	ENV3	ENV4
A6	—	—	—	12 797	18 117
B6	3 166	—	—	—	—
C6	3 333	5 214	—	—	—
D6	7 525	—	—	—	—
E6	4 169	—	—	—	—
F6	4 213	—	—	—	—
G6	3 166	—	—	—	—
H6	8 009	—	—	—	—
I6	7 168	—	—	—	—
J6	—	—	—	—	—
K6	—	—	—	—	—

Tabelle C.1b: Generationen zum Abschreiten aller Positionen für zwei Kreaturen

Automat	ENV0	ENV1	ENV2	ENV3	ENV4
A6	1 895	—	—	6 043	7 862
B6	3 100	—	—	—	—
C6	3 267	2 493	—	—	—
D6	3 675	—	—	—	—
E6	2 033	—	—	—	—
F6	2 039	—	—	—	—
G6	3 100	—	—	—	—
H6	3 655	—	—	—	—
I6	2 814	—	—	—	—
J6	—	—	—	—	—
K6	—	—	—	—	—

Tabelle C.1c: Generationen zum Abschreiten aller Positionen für vier Kreaturen

Automat	ENV0	ENV1	ENV2	ENV3	ENV4
A6	647	—	—	2 666	2 707
B6	971	—	811	—	—
C6	1 120	878	1 154	—	—
D6	1 750	—	—	—	—
E6	1 120	1 308	1 813	—	—
F6	971	1 488	1 806	—	—
G6	971	—	971	—	—
H6	1 478	—	—	—	—
I6	1 990	—	—	—	—
J6	—	—	1 521	—	2 023
K6	—	—	—	—	—

Tabelle C.1d: Generationen zum Abschreiten aller Positionen für acht Kreaturen

Automat	ENV0	ENV1	ENV2	ENV3	ENV4
A6	718	—	1 500	1 472	3 480
B6	2 012	569	1 155	—	—
C6	629	1 237	883	—	3 266
D6	2 080	1 607	—	—	1 927
E6	1 291	705	—	3 672	—
F6	—	807	1 048	8 563	1 035
G6	2 012	569	815	—	—
H6	918	1 262	—	—	—
I6	1 777	1 710	—	—	930
J6	4 831	1 256	1 157	2 512	1 977
K6	—	—	—	—	—

Tabelle C.1e: Generationen zum Abschreiten aller Positionen für zwölf Kreaturen

Automat	ENV0	ENV1	ENV2	ENV3	ENV4
A6	489	990	723	1 090	2 081
B6	864	909	869	9 556	6 734
C6	631	291	608	8 921	2 860
D6	1 256	2 268	—	—	—
E6	1 414	509	709	—	2 607
F6	1 032	1 040	1 686	1 097	1 087
G6	864	909	702	—	6 734
H6	1 116	836	—	—	524
I6	963	2 352	2 163	—	777
J6	2 607	1 419	912	818	1 418
K6	—	—	—	—	—

Tabelle C.1f: Generationen zum Abschreiten aller Positionen für 16 Kreaturen

Automat	ENV0	ENV1	ENV2	ENV3	ENV4
A6	375	737	533	900	857
B6	476	545	1 002	4 748	1 434
C6	1 516	395	412	5 708	4 226
D6	763	964	—	8 295	2508
E6	1 544	681	429	1 316	461
F6	736	1 185	858	10 864	1225
G6	476	545	339	4 618	1 434
H6	892	1 136	18 529	—	1 063
I6	449	499	3 859	—	1 525
J6	435	473	400	457	1074
K6	—	—	—	—	—

Tabelle C.1g: Generationen zum Abschreiten aller Positionen für 28 Kreaturen

Automat	ENV0	ENV1	ENV2	ENV3	ENV4
A6	380	396	484	923	534
B6	383	415	411	1 045	490
C6	219	392	256	1 273	738
D6	266	464	420	459	550
E6	428	310	474	6 126	463
F6	258	531	281	1 358	948
G6	383	415	255	15 775	490
H6	815	421	1 186	13 636	547
I6	373	342	1 728	1 946	460
J6	376	231	183	228	652
K6	—	—	—	—	—

Tabelle C.1h: Generationen zum Abschreiten aller Positionen für 32 Kreaturen

Automat	ENV0	ENV1	ENV2	ENV3	ENV4
A6	196	198	393	983	1 102
B6	332	435	206	12 427	273
C6	223	279	274	651	1 402
D6	508	459	1 000	631	796
E6	587	642	260	4 196	445
F6	282	499	316	462	268
G6	332	435	239	1 521	273
H6	385	606	1 038	7 783	526
I6	296	703	995	1 493	688
J6	525	308	267	263	487
K6	—	—	—	—	—

Tabelle C.1i: Generationen zum Abschreiten aller Positionen für 60 Kreaturen

Automat	ENV0	ENV1	ENV2	ENV3	ENV4
A6	72	410	173	528	493
B6	63	138	139	1 211	281
C6	63	139	151	570	576
D6	47	336	247	1 150	350
E6	75	262	111	878	273
F6	128	275	150	1 156	373
G6	63	138	107	712	281
H6	304	157	340	1 072	341
I6	47	535	463	7 211	282
J6	63	254	293	117	184
K6	—	—	—	—	—

Tabelle C.1j: Generationen zum Abschreiten aller Positionen für 64 Kreaturen

Automat	ENV0	ENV1	ENV2	ENV3	ENV4
A6	60	152	216	1 274	589
B6	66	253	120	998	192
C6	124	313	122	1 399	275
D6	64	260	323	413	476
E6	58	370	169	180	293
F6	130	173	289	306	649
G6	66	253	207	1 829	192
H6	221	237	377	1 782	1 001
I6	44	275	255	4 164	518
J6	64	131	141	522	455
K6	—	—	—	—	—

Tabelle C.2: Automaten für minimale Überquerung

	8 Kreaturen				64 Kreaturen			
Feld	**Generationen**		**bester Automat**		**Generationen**		**bester Automat**	
ENV0	629	541	C6	6-2, 6-4	44	35	I6	6-24, 6-25
ENV1	569	659	B6, G6	6-14, 6-27	131	84	J6	6-17
ENV2	815	643	G6	6-29	120	105	B6	6-4
ENV3	1 472	470	A6	6-22	180	78	E6	6-2
ENV4	930	858	I6	6-5	192	240	B6, G6	6-6
Alle	4 831	1 027	J6	6-1	370	240	E6	6-6

Tabelle C.3: Automatenliste des vollständigen Berechnungsergebnisses gemäß Beschreibung auf Seite 121, sortiert nach Erfolg

Automatenbeschreibung	**Codierung**
1/0 3/0 2/1 0/0 5/1 4/1 - 2/0 0/1 3/1 4/0 5/0 1/1	6-1
0/1 1/1 2/0 3/1 5/0 4/0 - 1/0 2/1 3/0 4/1 5/1 0/0	6-2
0/1 2/0 3/0 1/0 5/1 4/1 - 1/1 3/1 4/0 5/0 2/1 0/0	6-3
0/0 1/0 2/1 3/0 5/1 4/1 - 1/1 2/0 3/1 4/0 5/0 0/1	6-4
1/1 2/1 0/1 4/0 3/0 5/0 - 2/0 3/1 4/1 1/0 5/1 0/0	6-5
1/0 2/0 0/0 5/1 3/1 4/1 - 1/1 3/0 4/0 2/1 5/0 0/1	6-6
0/0 1/0 3/1 5/1 4/0 2/1 - 1/1 2/0 4/1 5/0 3/0 0/1	6-7
0/1 1/1 3/0 5/0 4/1 2/0 - 1/0 2/1 4/0 5/1 3/1 0/0	6-8
0/1 2/0 3/0 1/0 5/1 4/1 - 1/1 3/1 0/0 4/0 5/0 2/1	6-9
0/0 2/1 3/1 1/1 5/0 4/0 - 1/0 3/0 0/1 4/1 5/1 2/0	6-0
1/0 3/0 2/1 0/0 5/1 4/1 - 2/0 0/1 4/0 5/0 3/1 1/1	6-11
1/1 3/1 2/0 0/1 5/0 4/0 - 2/1 0/0 4/1 5/1 3/0 1/0	6-12
1/1 3/1 2/0 0/1 5/0 4/0 - 2/1 0/0 3/0 4/1 5/1 1/0	6-13
0/1 1/0 2/0 3/1 4/0 5/1 - 1/1 2/1 3/0 4/1 5/0 0/0	6-14
0/0 2/1 4/1 4/1 1/1 5/0 - 1/0 3/0 4/0 5/1 5/1 0/1	6-15
1/1 0/1 3/0 5/0 4/1 2/0 - 2/1 0/0 4/0 1/0 5/1 3/1	6-16
1/1 0/1 4/0 3/0 2/0 5/1 - 2/1 3/1 4/1 0/0 5/0 1/0	6-17
1/0 0/0 3/1 4/1 2/1 5/0 - 2/0 3/0 0/1 4/0 5/1 1/1	6-18
0/1 1/1 3/0 2/0 4/1 5/0 - 1/0 2/1 3/1 4/0 5/1 0/0	6-19
1/0 0/0 3/1 5/1 4/0 2/1 - 2/0 0/1 4/1 1/1 5/0 3/0	6-20
1/0 0/0 2/1 3/0 4/1 5/1 - 1/1 2/0 3/1 4/0 5/0 0/1	6-21
1/1 0/1 2/0 3/1 4/0 5/0 - 1/0 2/1 3/0 4/1 5/1 0/0	6-22
1/1 0/1 2/0 3/0 4/1 5/0 - 2/1 0/0 3/1 4/0 5/1 1/0	6-23
0/0 2/1 3/1 1/1 5/0 4/0 - 1/0 3/0 4/1 5/1 2/0 0/1	6-24
1/0 0/0 2/1 3/1 4/0 5/1 - 2/0 0/1 3/0 4/1 5/0 1/1	6-25
0/1 1/1 2/0 3/1 4/0 5/0 - 1/0 2/1 3/0 4/1 5/1 0/0	6-26
0/0 1/1 2/1 3/0 4/1 5/0 - 1/0 2/0 3/1 4/0 5/1 0/1	6-27
0/0 1/0 2/1 3/1 4/0 5/1 - 1/1 2/0 3/0 4/1 5/0 0/1	6-28
0/1 1/1 2/0 3/0 4/1 5/0 - 1/0 2/1 3/1 4/0 5/1 0/0	6-29
0/0 1/0 2/1 3/0 4/1 5/1 - 1/1 2/0 3/1 4/0 5/0 0/1	6-30
0/0 1/0 2/1 3/0 4/1 2/1 - 1/1 2/0 3/1 4/0 5/0 0/1	6-31
0/1 0/1 2/0 1/1 0/1 3/0 - 1/0 2/1 3/0 4/0 5/0 0/0	6-32

Tabelle C.4: Simulationsergebnisse (Generationen) bei acht Kreaturen mit den großen Feldern ENV0 bis ENV4

Automat	ENV0	ENV1	ENV2	ENV3	ENV4	max.
6-1	874	837	859	725	1 027	1 027
6-2	541	1 005	1 092	800	1 007	1 092
6-3	1 242	909	871	1 209	1 262	1 262
6-4	541	1 005	1 278	1 213	1 007	1 278
6-5	1 297	927	1 019	591	858	1 297
6-6	1 253	1 284	1 324	935	1 358	1 358
6-7	924	1 359	882	877	1 329	1 359
6-8	924	1 359	1 124	880	1 329	1 359
6-9	1 376	1 316	1 129	1 063	1 021	1 376
6-0	1 376	1 316	1 435	1 063	1 021	1 435
6-11	620	885	925	581	1 449	1 449
6-12	620	885	685	645	1 449	1 449
6-13	874	837	1 456	1 094	1 027	1 456
6-14	953	659	1 034	500	1 492	1 492
6-15	542	1 282	950	609	1 507	1 507
6-16	1 384	773	1 516	1 019	1 515	1 516
6-17	554	1 249	1 365	874	1 517	1 517
6-18	924	840	922	1 527	1 210	1 527
6-19	786	1 142	1 544	631	1 149	1 544
6-20	1 384	773	1 328	1 561	1 515	1 561
6-21	1 399	1 214	1 074	1 332	1 580	1 580
6-22	1 399	1 214	787	470	1 580	1 580
6-23	769	1 452	1 034	719	1 602	1 602
6-24	769	1 452	1 327	1 031	1 602	1 602
6-25	1 242	909	1 520	1 630	1 262	1 630
6-26	731	1 195	1 321	1 193	1 826	1 826
6-27	953	659	2 205	488	1 492	2 205
6-28	959	2 314	931	1 348	1 095	2 314
6-29	959	2 314	643	1 547	1 095	2 314
6-30	731	1 195	1 214	2 500	1 826	2 500
6-31	1 127	3 500	904	1 455	1 371	3 500
6-32	3 138	4 164	3 529	1 209	3 114	4 164

Tabelle C.5: Simulationsergebnisse (Generationen) bei acht Kreaturen zum Vergleich mit den ursprünglichen Feldern

Automat	#1	#2	#3	#4	#5	#6
6-1	225	—	—	187	286	—
6-2	324	241	399	244	204	1 152
6-3	—	—	—	—	264	—
6-4	474	—	396	492	—	1 247
6-5	256	178	197	176	—	—
6-6	562	—	302	—	212	921
6-7	366	257	333	—	492	—
6-8	428	230	—	—	313	—
6-9	—	—	193	271	262	—
6-0	415	—	216	198	189	—
6-11	—	170	392	—	182	—
6-12	—	204	564	—	223	—
6-13	364	—	—	189	256	—
6-14	—	158	—	—	—	—
6-15	—	—	—	—	249	—
6-16	—	—	—	—	299	—
6-17	294	—	—	259	—	—
6-18	340	—	437	250	258	—
6-19	650	213	402	240	—	1 060
6-20	—	—	—	297	180	—
6-21	314	153	374	238	—	1 481
6-22	327	218	414	—	301	1 332
6-23	—	151	358	—	—	1 502
6-24	—	—	431	—	319	1 646
6-25	259	211	322	243	251	532
6-26	387	232	—	—	196	—
6-27	—	—	—	—	302	—
6-28	—	—	—	—	192	—
6-29	—	193	—	—	—	—
6-30	—	—	—	—	—	—
6-31	—	155	213	252	—	—
6-32	183	—	—	223	—	—

Tabelle C.6: Simulationsergebnisse (Generationen) ENV0 bis ENV4 bei 64 Kreaturen

Automat	ENV0	ENV1	ENV2	ENV3	ENV4	max.
6-1	95	132	154	101	447	447
6-2	124	227	134	78	299	299
6-3	81	132	215	202	344	344
6-4	124	227	105	81	299	299
6-5	170	221	126	192	501	501
6-6	79	124	123	206	240	240
6-7	141	169	158	200	422	422
6-8	141	169	132	104	422	422
6-9	211	127	207	131	335	335
6-10	211	127	192	141	335	335
6-11	56	111	129	266	385	385
6-12	95	132	117	108	447	447
6-13	56	111	144	171	385	385
6-14	104	157	186	192	347	347
6-15	69	109	152	342	314	342
6-16	112	246	117	150	289	289
6-17	151	84	157	166	375	375
6-18	77	142	128	421	687	687
6-19	127	273	218	111	248	273
6-20	144	186	127	182	889	889
6-21	127	273	265	202	248	273
6-22	144	186	163	171	889	889
6-23	81	132	132	226	344	344
6-24	35	96	117	154	330	330
6-25	35	96	113	170	330	330
6-26	165	121	163	119	358	358
6-27	69	109	115	637	314	637
6-28	56	101	128	213	494	494
6-29	56	101	167	221	494	494
6-30	165	121	135	106	358	358
6-31	136	131	113	516	285	516
6-32	86	282	250	269	405	405

C.3 Simulationsergebnis mit vorteilhafter Kollision

Mit der Umgebung ENV0 zeigt sich bei 64 Kreaturen mit dem Algorithmus I6 eine besonders effiziente Überquerung des Feldes mit nur 44 benötigten Generationen. Dabei sind die Generationen 0, 12, 24, 36 und 44 die einzigen mit einer Kollision, die zudem vorteilhaft für den Ablauf ist. Dargestellt ist in Abbildung C.8 die Konfliktsituation und die darauf folgende Generation – mit Ausnahme der letzten, abschließenden Generation 44.

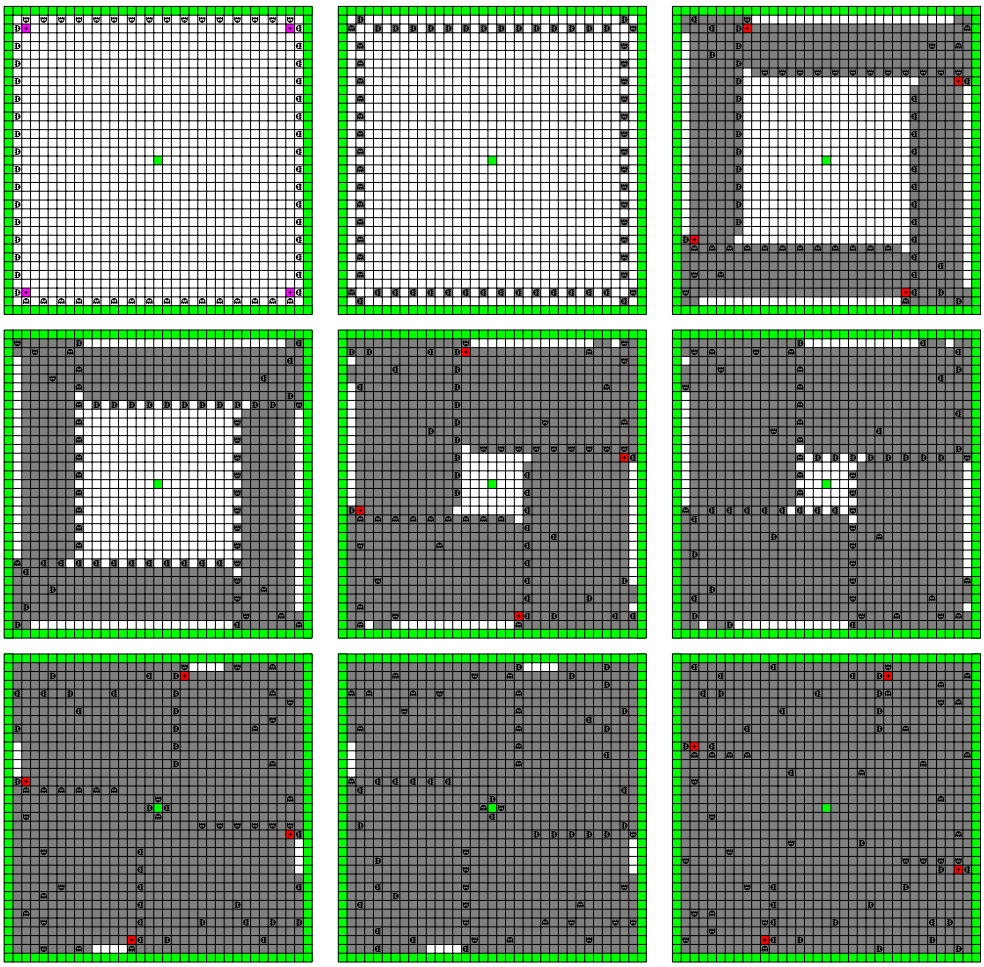


Abbildung C.8: Simulation: Generationen mit Konflikten

Anhang D

Programmcode

Im folgenden ist der Quellcode ausgewählter Programme aufgeführt, die eine Basis für die zuletzt entstandenen Ergebnisse bildet. Es kamen noch weit mehr selbst entwickelte Programme und Hardwaredesigns zum Einsatz, die teilweise aber nur Vorentwicklungsstufen darstellen.

Das Modul des Algorithmus D.4 ist mit veränderter Schnittstelle in Algorithmus D.6 verwendet. Die dafür notwendigen Details zur notwendigen Anpassung gibt Algorithmus D.5 wieder.

Algorithmus D.1: Aufzählung von Automaten in Software

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <signal.h>
5
6  #define states 5
7  #define inputs 2
8  #define outputs 2
9  #define terminals 0
10 #define starts states
11 #define terminalstarts 0
12 #undef LIST_UNREACHED
13 #undef STATE_ISOMORPHISM
14 #undef INPUT_ISOMORPHISM
15 #undef OUTPUT_ISOMORPHISM
16 #undef CHECK_NORMALIZED
17 #define CHECK_ARRANGED
18 #define BEST_0_START
19 #define CHECK_PREFIX
20 #define CHECK_REDUCTION
21 #define OUTPUT_SEMANTIC
22 #undef INPUT_SEMANTIC
23 #undef MOORE_AUTOMATON
24 #define PRINT_AUTOMATON
25 #undef PRINT_ADDITION_AUTOMATON_INFORMATION
26 #undef SIMULATE
27
28 #define min(a,b) (((a) < (b))? (a) : (b))
29
30 typedef enum { false = 0, true = 1 } bool;
31 int s[states][inputs];
```

```
32 int y[states][inputs];
33 int terminal[terminals + 1];
34 bool a[states][states], s_unevaluated;
35 int h1 = states, h2, h3, hy = 0;
36 unsigned long long int tests = 0llu, amount = 0llu;
37
38 const char *AutomatonString();
39
40
41 void set_a()
42 {
43     register int i, j, k;
44     for (i = 0; i < states; i ++)
45         for (j = 0; j < states; j ++) {
46             a[i][j] = i == j;
47             for (k = 0; k < inputs; k ++)
48                 a[i][j] |= s[i][k] == j;
49         }
50     for (j = 0; j < states; j ++)
51         for (i = 0; i < states; i ++)
52             if (a[i][j])
53                 for (k = 0; k < states; k ++)
54                     if (a[j][k])
55                         a[i][k] = true;
56 }
57
58
59 void set_h(int i, int j, int t)
60 {
61     if (j < 0) {
62         --i;
63         j += inputs;
64     }
65     if (h1 > i) {
66         h1 = i;
67         h2 = j;
68         h3 = t;
69     } else if (h1 == i && h2 > j) {
70         h2 = j;
71         h3 = t;
72     } else if (h1 == i && h2 == j && h3 < t)
73         h3 = t;
74 }
75
76
77 void swap(int *a, int *b)
78 {
79     register const int t = *a;
80     *a = *b;
81     *b = t;
82 }
83
84
85 bool next_permutation(int n, int list[])
86 {
87     register int i = n - 2, j = n - 1;
88     if (i < 0)
89         return false;
90     while (list[i] > list[i + 1])
```

```
91              if (i −− == 0) {
92                  for (i = n / 2 − 1; i >= 0; i −−)
93                      swap(&list[i], &list[j − i]);
94                  return false;
95              }
96          while (list[j] <= list[i])
97              j −−;
98          swap(&list[i ++], &list[j]);
99          for (j = n − 1; i < j;)
100             swap(&list[i ++], &list[j −−]);
101         return true;
102 }
103
104
105 bool isOutputPermutation(int sp[], int xp[])
106 {
107         #if (outputs > 1)
108         register int i, j;
109         int yp[states][inputs];
110         for (i = 0; i < states; i ++)
111             for (j = 0; j < inputs; j ++)
112                 yp[sp[i]][xp[j]] = y[i][j];
113         #ifdef OUTPUT_ISOMORPHISM
114         {
115             int yperm[outputs], nexty = 0;
116             for (i = 0; i < outputs; i ++)
117                 yperm[i] = −1;
118             yperm[yp[0][0]] = nexty;
119             for (i = 0; i < states; i ++)
120                 for (j = 0; j < inputs; j ++) {
121                     register int ∗ypij = &yp[i][j];
122                     register const int vypij = ∗ypij;
123                     if (yperm[vypij] == −1)
124                         yperm[vypij] = ++ nexty;
125                     ∗ypij = yperm[vypij];
126                 }
127         }
128         #endif
129         for (i = 0; i < states; i ++)
130             for (j = 0; j < inputs; j ++) {
131                 register const int a = y[i][j], b = yp[i][j];
132                 if (a > b)
133                     return true;
134                 if (a < b)
135                     return false;
136             }
137         #endif
138         return false;
139 }
140
141
142 bool isTerminalPermutation(int sp[])
143 {
144         #if (terminals > 0)
145         register int i, t = 0;
146         bool tp[states];
147         for (i = 0; i < states; i ++)
148             tp[i] = false;
149         for (i = 0; i < terminals; i ++)
```

```
150             tp[sp[terminal[i]]] = true;
151         for (i = 0; i < states; i ++)
152             if (tp[i]) {
153                 register const int a = terminal[t ++];
154                 if (a > i)
155                     return true;
156                 if (a < i)
157                     return false;
158             }
159         #endif
160         return false;
161 }
162
163
164 bool PermutationCandidate()
165 {
166         register int i, j;
167         register bool equal;
168         int sp[states], xp[inputs];
169         int rsp[states], rxp[inputs];
170         for (i = 0; i < states; i ++)
171             sp[i] = i;
172         for (j = 0; j < inputs; j ++)
173             xp[j] = j;
174         #ifdef STATE_ISOMORPHISM
175         #if starts - terminalstarts > 1
176         next_permutation(starts - terminalstarts, sp);
177         #elif terminalstarts > 1
178         next_permutation(terminalstarts, sp + starts - terminalstarts);
179         #elif states - starts - terminals + terminalstarts > 1
180         next_permutation(states - starts - terminals + terminalstarts, sp + starts);
181         #elif terminals - terminalstarts > 1
182         next_permutation(terminals - terminalstarts, sp + states - terminals + terminalstarts);
183         #endif
184         #elif defined(INPUT_ISOMORPHISM)
185         next_permutation(inputs, xp);
186         #endif
187         #ifdef INPUT_ISOMORPHISM
188         do {
189         #endif
190             for (j = 0; j < inputs; j ++)
191                 rxp[xp[j]] = j;
192         #ifdef STATE_ISOMORPHISM
193         do {
194         #endif
195             for (i = 0; i < states; i ++)
196                 rsp[sp[i]] = i;
197             equal = true;
198             for (i = 0; equal && i < states; i ++)
199                 for (j = 0; equal && j < inputs; j ++) {
200                     register const int a = s[i][j], b = sp[s[rsp[i]][rxp[j]]];
201                     if (a > b) {
202                         set_h(states - 1, inputs - 1, s[states - 1] [inputs - 1] + 1);
203                         return false;
204                     } else
205                         equal = a == b;
206                 }
207             if (equal && (isOutputPermutation(sp, xp) || isTerminalPermutation(sp)))
208                 return false;
```

```
209     #ifdef STATE_ISOMORPHISM
210     } while (next_permutation(starts − terminalstarts, sp) ||
211         next_permutation(terminalstarts, sp + starts − terminalstarts) ||
212         next_permutation(states − starts − terminals + terminalstarts, sp + starts) ||
213         next_permutation(terminals − terminalstarts, sp + states − terminals + terminalstarts));
214     #endif
215     #ifdef INPUT_ISOMORPHISM
216     } while (next_permutation(inputs, xp));
217     #endif
218     return true;
219 }
220
221
222 bool Normalized()
223 {
224     register int t, i, j;
225     for (t = 1; t < states; t ++) {
226         register bool match = false, below = true;
227         for (j = 0; j < inputs; j ++) {
228             for (i = 0; i < t; i ++) {
229                 match |= (s[i][j] == t) && below;
230                 below &= (s[i][j] <= t);
231             }
232             if (!match && !below) { /* Überspringen von t */
233                 set_h(t − 1, j, t);
234                 return false;
235             }
236         }
237         if (!match) { /* nicht alle Zustände erreicht */
238             #ifdef LIST_UNREACHED
239             for (i = t; i < states; i ++)
240                 for (j = 0; j < inputs; j ++)
241                     if (s[i][j] || y[i][j]) {
242                         set_h(t − 1, inputs − 1, t);
243                         return false;
244                     }
245             return true;
246             #else
247             set_h(t − 1, inputs − 1, t);
248             return false;
249             #endif
250         }
251     }
252     return true;
253 }
254
255
256 bool Arranged()
257 #if (terminals > 0 && terminals < states − 1) || (starts > 1 && starts < states − 1)
258 {
259     bool t[states + 1];
260     register int i, j;
261     for (i = 0; i <= states; i ++)
262         t[i] = false;
263     t[0] = t[1] = true;
264     t[starts − terminalstarts] = true;
265     t[starts] = true;
266     t[states − terminals + terminalstarts] = true;
267     for (i = 0; i < states; i ++) {
```

```
268             for (j = 0; j < inputs; j ++) {
269                 register const int sij = s[i][j];
270                 if (t[sij]) {
271                     if (!t[sij + 1]) {
272                         t[i] = true;
273                         t[sij + 1] = true;
274                     }
275                 } else {
276                     set_h(i, j, states);
277                     return false;
278                 }
279             }
280         }
281         return true;
282 }
283 #else
284 {
285         register int i, j, t = 1;
286         for (i = 0; i < states; i ++) {
287             for (j = 0; j < inputs; j ++) {
288                 register const int sij = s[i][j];
289                 if (sij > t) {
290                     set_h(i, j, states);
291                     return false;
292                 }
293                 if (sij == t)
294                     t ++;
295             }
296         }
297         return true;
298 }
299 #endif
300
301
302 bool BestStartFirst()
303 {
304         register int s0;
305         bool result = true;
306         for (s0 = 1; s0 < starts; s0 ++) {
307             int o[states], n[states], max_reached = -1;
308             bool pvalid[states];
309             register int i, j, t, c = 0;
310             register bool eq = true;
311             for (i = states - 1; i >= 0; i --) {
312                 n[i] = -1;
313                 o[i] = -1;
314                 pvalid[i] = false;
315             }
316             n[c] = s0;
317             o[s0] = c;
318             pvalid[s0] = true;
319             c ++;
320
321             /* Permutation setzen */
322             for (t = 0; t < states; t ++) {
323                 i = n[t];
324                 if (i == -1) {
325                     set_h(states - 1, inputs - 1, s[states - 1][inputs - 1] + 1);
326                     result = false;
```

```
327                     break;
328                 }
329                 for (j = 0; j < inputs; j ++) {
330                     register const int sij = s[i][j];
331                     if (o[sij] == -1) {
332                         n[c] = sij;
333                         o[sij] = c;
334                         pvalid[sij] = true;
335                         c ++;
336                     }
337                 }
338             }
339 
340             /* Permutation bezüglich Zustände testen */
341             for (i = 0; eq && i < t; i ++)
342                 for (j = 0; eq && j < inputs; j ++) {
343                     register const int a = s[i][j], b = o[s[n[i]][j]];
344                     if (max_reached < n[i])
345                         max_reached = n[i];
346                     if (a > b) {
347                         if (max_reached >= 0)
348                             set_h(max_reached, inputs - 1, s[max_reached][inputs - 1] + 1);
349                         else
350                             set_h(states - 1, inputs - 1, s[states - 1][inputs - 1] + 1);
351                         result = false;
352                     }
353                     eq = a == b;
354                 }
355 
356             /* Permutation bezüglich Ausgaben testen */
357             #if (outputs > 1)
358             for (i = 0; eq && i < t; i ++)
359                 for (j = 0; eq && j < inputs; j ++) {
360                     register const int a = y[i][j], b = y[n[i]][j];
361                     if (a > b)
362                         result = false;
363                     eq = a == b;
364                 }
365             #endif
366         }
367         return result;
368     }
369 
370 
371     bool AllReachable()
372     {
373         register int i;
374         register bool result = true;
375         for (i = 1; i < states; i ++) {
376             register const bool v = a[0][i];
377             result &= v;
378             if (!v) {
379                 set_h(i - 1, inputs - 1, i);
380                 break;
381             }
382         }
383         return result;
384     }
385 
```

```
386
387 bool AllConnected()
388 {
389     register int i;
390     register bool result = true;
391     for (i = 1; i < states; i ++) {
392         register const bool v = a[0][i] || a[i][0];
393         result &= v;
394         if (!v) {
395             set_h(i + 1, inputs - 1, i);
396             break;
397         }
398     }
399     return result;
400 }
401
402
403 bool PrefixFree()
404 {
405     register int i, j = states;
406     register bool result = true;
407     for (i = 0; i < states; i ++) {
408         register const bool v = !a[0][i] || a[i][0];
409         result &= v;
410         if (!v)
411             j = min(j, i);
412     }
413     if (!result) {
414         register int k, l;
415         k = j;
416         for (i = j + 1; i < states; i ++)
417             if (a[j][i])
418                 k = i;
419         l = k + 1;
420         for (i = s[k][inputs - 1] + 1; i < k; i ++)
421             if (a[i][0]) {
422                 l = i;
423                 break;
424             }
425         set_h(k, inputs - 1, l);
426     }
427     return result;
428 }
429
430
431 bool Reduced()
432 {
433     bool e[states][states];
434     register int i, j, k;
435     register bool c;
436     for (i = states - 2; i >= 0; i --) {
437         for (j = i + 1; j < states; j ++) {
438             #if (terminals > 0)
439             register bool ti = false, tj = false;
440             for (k = terminals - 1; k >= 0; k --) {
441                 ti |= terminal[k] == i;
442                 tj |= terminal[k] == j;
443             }
444             c = ti == tj;
```

```
445                 #else
446                 c = true;
447                 #endif
448                 #if (outputs > 1)
449                 for (k = inputs − 1; k >= 0; k −−)
450                     c &= y[i][k] == y[j][k];
451                 #endif
452                 e[i][j] = e[j][i] = c;
453             }
454             e[i][i] = true;
455         }
456         e[states − 1][states − 1] = true;
457         do {
458             c = false;
459             for (i = states − 2; i >= 0; i −−)
460                 for (j = i + 1; j < states; j ++)
461                     if (e[i][j]) {
462                         register bool eq = true;
463                         for (k = inputs − 1; k >= 0; k −−)
464                             eq &= e[s[i][k]][s[j][k]];
465                         if (!eq) {
466                             e[i][j] = e[j][i] = false;
467                             c = true;
468                         }
469                     }
470         } while (c);
471         for (j = states − 1; j > 0; j −−)
472             for (i = 0; i < j; i ++)
473                 if (e[i][j]) {
474                     hy = j;
475                     return false;
476                 }
477         return true;
478 }
479
480
481 bool output_semantic_x1()
482 {
483         register int i, j, k;
484
485         /* Schnelltest für x>0 */
486         for (i = 0; i < states; i++)
487             for (k = inputs − 1; k > 0; k−−)
488                 if (s[i][k] == i) {
489                     set_h(i, inputs − 1, s[i][inputs − 1] + 1);
490                     return false;
491                 }
492
493         /* ausführlicher Test für x>0 ... transitive Hülle erstellen */
494         bool oa[states][states];
495         for (j = states − 1; j >= 0; j−−)
496             for (i = states − 1; i >= 0; i−−) {
497                 oa[i][j] = i == j;
498                 for (k = inputs − 1; k > 0; k−−)
499                     oa[i][j] |= s[i][k] == j;
500             }
501         for (j = 0; j < states; j++)
502             for (i = 0; i < states; i++)
503                 if (oa[i][j])
```

```
504                     for (k = 0; k < states; k++)
505                         if (oa[j][k])
506                             oa[i][k] = true;
507     /* ... transitive Hülle auswerten */
508     for (i = 0; i < states; i++) {
509         bool output_used[outputs];
510         for (k = outputs - 1; k >= 0; k--)
511             output_used[k] = false;
512         for (j = 0; j < states; j++)
513             if (oa[i][j]) {
514                 for (k = inputs - 1; k > 0; k--)
515                     output_used[y[j][k]] = true;
516             }
517         register int amount = 0;
518         for (k = outputs - 1; k >= 0; k--)
519             if (output_used[k])
520                 ++ amount;
521         if (amount < 2)
522             return false;
523     }
524
525     return true;
526 }
527
528
529 bool output_semantic_x0()
530 {
531     register int i;
532     for(i = 0; i < states; i ++) {
533         register int k = i, dir = 0, alldir = 1, j;
534         for (j = 4; j >= 0; j --) {
535             switch (y[k][0]) {
536             case 0:
537                 dir = (dir + 1) % 4;
538                 break;
539             case 1:
540                 dir = (dir + 3) % 4;
541                 break;
542             case 3:
543                 dir = (dir + 2) % 4;
544                 break;
545             case 2:
546             default:
547                 break;
548             }
549             alldir |= 1 << dir;
550             k = s[k][0];
551         }
552         if (alldir != 0xf)
553             return false;
554     }
555     return true;
556 }
557
558
559 bool input_semantic()
560 {
561     register bool result = true;
562     register int i, j, k;
```

```
563     for (i = 0; i < inputs − 1; i ++)
564         for (j = i + 1; j < inputs; j ++)
565             for (k = 0; k < states; ++ k)
566                 result &= y[k][i] == y[k][j] &&
567                     s[k][i] == s[k][j];
568     return !result;
569 }
570
571
572 bool Validate()
573 {
574     register bool result = true;
575     ++ tests;
576     if (s_unevaluated) {
577         set_a();
578         #ifdef CHECK_NORMALIZED
579         result &= Normalized();
580         #endif
581         #ifdef CHECK_ARRANGED
582         result &= Arranged();
583         #endif
584         #ifndef LIST_UNREACHED
585         result &= AllReachable();
586         #endif
587         #ifdef CHECK_PREFIX
588         result &= PrefixFree();
589         #endif
590         #if defined(STATE_ISOMORPHISM) || defined(INPUT_ISOMORPHISM)
591         if (result)
592             result &= PermutationCandidate();
593         #endif
594         s_unevaluated = !result;
595     } else {
596         #if defined(STATE_ISOMORPHISM) || defined(INPUT_ISOMORPHISM)
597         result = PermutationCandidate();
598         #endif
599     }
600     #ifdef BEST_0_START
601     if (result)
602         result &= BestStartFirst();
603     #endif
604     #if defined(CHECK_REDUCTION) && ((outputs > 1) || (terminals > 0))
605     if (result)
606         result &= Reduced();
607     #endif
608     #ifdef OUTPUT_SEMANTIC
609     if (result)
610         result &= output_semantic_x1() && output_semantic_x0();
611     #endif
612     #ifdef INPUT_SEMANTIC
613     if (result)
614         result &= input_semantic();
615     #endif
616     return result;
617 }
618
619
620 void InitiateAutomaton()
621 {
```

```
622     register int i, j;
623     for (i = states − 1; i >= 0 ; i −−) {
624         for (j = inputs − 1; j >= 0; j −−) {
625             s[i][j] = 0;
626             y[i][j] = 0;
627         }
628     }
629     j = 0;
630     for (i = starts − terminalstarts; i < starts; i ++)
631         terminal[j ++] = i;
632     for (i = states − terminals + terminalstarts; i < states; i ++)
633         terminal[j ++] = i;
634     terminal[terminals] = states;
635     s_unevaluated = true;
636     #ifndef LIST_UNREACHED
637     for (i = states − 2; i >= 0 ; i −−)
638         s[i][inputs − 1] = i + 1;
639     #endif
640 }
641
642
643 bool NextAutomaton()
644 {
645     register int i, j;
646     #ifdef OUTPUT_ISOMORPHISM
647     int maxy[states][inputs];
648     register int premaxy = 0, prey = y[0][0];
649     maxy[0][0] = premaxy;
650     for (j = 0; j < inputs; j ++)
651         for (i = 0; i < states; i ++) {
652             premaxy = maxy[i][j] = premaxy > prey? premaxy : prey;
653             prey = y[i][j] + 1;
654         }
655     #define maxyij min(maxy[i][j], outputs − 1)
656     #else
657     #define maxyij outputs − 1
658     #endif
659     for (i = states − 1; i >= 0 ; i −−) {
660         #ifdef MOORE_AUTOMATON
661         j = 0;
662         #else
663         for (j = inputs − 1; j >= 0; j −−)
664         #endif
665             if (y[i][j] < maxyij) {
666                 ++ y[i][j];
667                 #ifdef MOORE_AUTOMATON
668                 for (j = inputs − 1; j > 0; j −−)
669                     y[i][j] = y[i][0];
670                 #endif
671                 return true;
672             } else {
673                 #ifdef MOORE_AUTOMATON
674                 for (j = inputs − 1; j >= 0; j −−)
675                     y[i][j] = 0;
676                 #else
677                 y[i][j] = 0;
678                 #endif
679             }
680     }
```

```
681     s_unevaluated = true;
682     for (i = states − 1; i >= 0 ; i −−)
683         for (j = inputs − 1; j >= 0; j −−)
684             if (s[i][j] < states − 1) {
685                 ++ s[i][j];
686                 return true;
687             } else
688                 s[i][j] = 0;
689     return false;
690 }
691
692
693 bool NextValidAutomaton()
694 {
695     while (NextAutomaton()) {
696         h1 = states; /∗ h2 = inputs − 1; h3 = 0; ∗/
697         if (Validate())
698             return true;
699         if (h1 < states) {
700             register int i, j;
701             if (h1 < 0) {
702                 puts("Abbruch");
703                 return false;
704             }
705             for (j = inputs − 1; j >= 0; j −−) {
706                 for (i = states − 1; i >= 0 ; i −−)
707                     y[i][j] = outputs − 1;
708                 for (i = states − 1; i > h1 ; i −−)
709                     s[i][j] = states − 1;
710             }
711             for (j = inputs − 1; j > h2; j −−)
712                 s[h1][j] = states − 1;
713             if (s[h1][h2] >= h3)
714                 s[h1][h2] = states − 1;
715             else
716                 s[h1][h2] = h3 − 1;
717             #if (outputs > 1) && defined(CHECK_REDUCTION)
718             if (!NextAutomaton())
719                 break;
720             #endif
721         } else if (hy != 0) {
722             register int i, j;
723             for (i = states − 1; i > hy ; i −−)
724                 for (j = inputs − 1; j >= 0; j −−)
725                     y[i][j] = outputs − 1;
726         }
727         hy = 0;
728     }
729     return false;
730 }
731
732
733 const char ∗AutomatonString()
734 {
735     static char automaton[states ∗ inputs ∗ 4 + inputs ∗ 2 + (terminals? 3 : 0) + terminals ∗ 2 + (
            starts? 3 : 0) + starts ∗ 2];
736     register int i, j;
737     register char ∗a = automaton;
738     for (j = 0; j < inputs; j ++) {
```

```
739             if (j != 0) {
740                 *(a++) = '-';
741                 *(a++) = ' ';
742             }
743             for (i = 0; i < states; i ++)
744                 a += sprintf(a, "%d/%d ", s[i][j], y[i][j]);
745         }
746         #ifdef PRINT_ADDITION_AUTOMATON_INFORMATION
747         #if (terminals > 0)
748             *(a++) = ' ';
749             *(a++) = 'T';
750             *(a++) = ' ';
751             for (i = 0; i < terminals; i ++)
752                 a += sprintf(a, "%d ", terminal[i]);
753         #endif
754         #if (starts > 0)
755             *(a++) = ' ';
756             *(a++) = 'S';
757             *(a++) = ' ';
758             for (i = 0; i < starts; i ++)
759                 a += sprintf(a, "%d ", i);
760         #endif
761         #endif
762         return automaton;
763 }
764
765
766 #ifdef SIMULATE
767 #define zmax ((1 << (outputs >> 1)) - 1)
768 unsigned maxgeneration = 50 * (zmax + 1), found = 0u;
769 void IssueAutomaton()
770 {
771     register unsigned generation = 0u;
772     register int ziel;
773     for (ziel = 0; ziel <= zmax && generation <= maxgeneration; ++ ziel) {
774         register int umwelt = 0, state = 0;
775         while (ziel != umwelt && generation <= maxgeneration) {
776             const register int x = ziel > umwelt? 1 : 0;
777             const register int cy = y[state][x];
778             state = s[state][x];
779             if ((cy & 1) == 0) {
780                 umwelt -= 1u << (cy >> 1);
781                 if (umwelt < 0)
782                     umwelt = 0;
783             } else {
784                 umwelt += 1u << (cy >> 1);
785                 if (umwelt > zmax)
786                     umwelt = zmax;
787             }
788             ++ generation;
789         }
790     }
791     if (generation < maxgeneration) {
792         maxgeneration = generation;
793         printf("%d %s\n", generation, AutomatonString());
794         found = 1;
795     } else if (generation == maxgeneration)
796         ++ found;
797 }
```

```
798 #endif
799
800
801 void print_status(int sig)
802 {
803     fprintf(stderr, "@ %s: a(%d, %d, %d, %d, %d, %d) = %llu\nwith %llu tests, i.e. %llu hops
            (%.1f %%)\n", AutomatonString(), states, inputs, outputs, terminals, starts, terminalstarts
            , amount, tests, tests − amount, 100. ∗ (tests − amount + .0) / tests);
804 }
805
806
807 int main()
808 {
809     register const clock_t start = clock();
810     signal(SIGUSR1, print_status);
811     InitiateAutomaton();
812     if (Validate()) {
813         #ifdef PRINT_AUTOMATON
814         puts(AutomatonString());
815         #endif
816         #ifdef SIMULATE
817         IssueAutomaton();
818         #endif
819         ++ amount;
820     }
821     while (NextValidAutomaton()) {
822         #ifdef PRINT_AUTOMATON
823         puts(AutomatonString());
824         #endif
825         #ifdef SIMULATE
826         IssueAutomaton();
827         #endif
828         ++ amount;
829     }
830     fprintf(stderr, "a(%d, %d, %d, %d, %d, %d) = %llu\nwith %llu tests, i.e. %llu hops (%.1f %%)
            in %ld / %d seconds\n", states, inputs, outputs, terminals, starts, terminalstarts, amount,
            tests, tests − amount, 100. ∗ (tests − amount + .0) / tests, clock() − start,
            CLOCKS_PER_SEC);
831     #ifdef SIMULATE
832     fprintf(stderr, "zmax = %d, maxgeneration = %u, found = %u\n", zmax, maxgeneration, found
            );
833     #endif
834     return 0;
835 }
```

Algorithmus D.2: Simulation in Software

```
1  #include <iostream>
2  #include <fstream>
3  #include <stdlib.h>
4  #include <time.h>
5
6  class Field
7  {
8  public:
9      enum celltype {
10         empty, obstacle, frontcell, conflict, creatureN, creatureE, creatureS, creatureW
11     };
12     struct coordinate {
```

```
13              unsigned int x, y;
14              unsigned int state, algorithm_select;
15          };
16
17      protected:
18          const unsigned int size, creatures;
19          enum celltype **area;
20          bool **visited, front_set;
21          unsigned int unvisited, generation, conflicts;
22          struct coordinate *creature;
23          int nextstate[4][2][6], output[4][2][6];
24          celltype next(bool, int, celltype, int) const;
25          int next(bool, int, int) const;
26
27          class ConflictList {
28          protected:
29              unsigned int *x, *y, n;
30          public:
31              ConflictList(int creatures);
32              ~ConflictList();
33              void append(const coordinate *item);
34              unsigned erase(celltype **area);
35          } conflict_list;
36
37          void set_frontcell(celltype &c);
38          unsigned calculate_cells();
39          coordinate *get_front(const coordinate *c, const celltype *a) const;
40          coordinate *get_front(const coordinate *c) const;
41
42      public:
43          Field(unsigned s, unsigned c);
44          ~Field();
45          unsigned calculate();
46          void set_frontcells();
47          unsigned clear_conflicts();
48          unsigned clear_conflicts_by_area();
49
50          unsigned get_generation() const { return generation; }
51          unsigned get_conflicts() const { return conflicts; }
52          unsigned get_unvisited() const { return unvisited; }
53          unsigned int get_size() const { return size; }
54          unsigned int get_creatures() const { return creatures; }
55          enum celltype **get_area() const { return area; }
56          bool **get_visited() const { return visited; }
57          struct coordinate *get_creature() const { return creature; }
58          int get_creature_alg(unsigned int x, unsigned int y) const;
59
60          void select_alg(char nr, int target);
61          void select_alg(const char *nr, int target);
62          unsigned load_field(std::ifstream &i);
63          void set_creature(unsigned x, unsigned y, int dir, int id, int alg);
64      };
65
66
67      Field::ConflictList::ConflictList(int creatures):
68          x(new unsigned int[creatures]),
69          y(new unsigned int[creatures]),
70          n(0u)
71      {
```

```
72 }
73
74
75 Field::ConflictList::~ConflictList()
76 {
77     delete[] x;
78     delete[] y;
79 }
80
81
82 inline void Field::ConflictList::append(const coordinate *item)
83 {
84     if (item -> state != 0) {
85         x[n] = item -> x;
86         y[n] = item -> y;
87         n ++;
88     }
89 }
90
91
92 inline unsigned Field::ConflictList::erase(celltype **area)
93 {
94     register unsigned amount = 0u;
95     register unsigned int *px = x, *py = y;
96     for (register unsigned int i = 0; i < n; i ++) {
97         register celltype *a = &area[*(px ++)][*(py ++)];
98         if (*a == conflict) {
99             *a = empty;
100            ++ amount;
101        }
102    }
103    n = 0;
104    return amount;
105 }
106
107
108 Field::Field(unsigned s, unsigned c):
109     size(s), creatures(c),
110     area(new celltype*[s]),
111     visited(new bool*[s]), front_set(false),
112     unvisited(0), generation(0u), conflicts(0u),
113     creature(new coordinate[c]),
114     conflict_list(creatures)
115 {
116     for (unsigned i = 0; i < c; i ++)
117         creature[i].state = 0;
118     for (unsigned x = 0; x < s; x ++) {
119         area[x] = new celltype[s];
120         visited[x] = new bool[s];
121         for (unsigned y = 0; y < s; y ++)
122             visited[x][y] = false;
123     }
124 }
125
126
127 Field::~Field()
128 {
129     for (register unsigned i = 0; i < size; i ++) {
130         delete[] area[i];
```

```
131             delete[] visited[i];
132         }
133         delete[] area;
134         delete[] visited;
135         delete[] creature;
136 }
137
138
139 inline void Field::set_frontcell(celltype &c)
140 {
141         switch (c) {
142         case empty:
143             c = frontcell;
144             break;
145         case frontcell:
146             c = conflict;
147             break;
148         default:
149             break;
150         }
151 }
152
153
154 inline Field::coordinate *Field::get_front(const Field::coordinate *cc) const
155 {
156         return get_front(cc, &area[cc -> x][cc -> y]);
157 }
158
159
160 Field::coordinate *Field::get_front(const Field::coordinate *cc, const Field::celltype *a) const
161 {
162         static coordinate result;
163         switch (*a) {
164         case creatureN:
165             result.state = cc -> y > 0;
166             result.x = cc -> x;
167             result.y = cc -> y - 1;
168             break;
169         case creatureE:
170             result.state = cc -> x < size - 1;
171             result.x = cc -> x + 1;
172             result.y = cc -> y;
173             break;
174         case creatureS:
175             result.state = cc -> y < size - 1;
176             result.x = cc -> x;
177             result.y = cc -> y + 1;
178             break;
179         case creatureW:
180             result.state = cc -> x > 0;
181             result.x = cc -> x - 1;
182             result.y = cc -> y;
183             break;
184         default:
185             result.state = 0;
186         }
187         return &result;
188 }
189
```

```
190
191 void Field::set_frontcells()
192 {
193     if (front_set)
194         return;
195     register coordinate *cc = creature;
196     for (register unsigned c = 0; c < creatures; c ++) {
197         switch (area[cc -> x][cc -> y]) {
198         case creatureN:
199             if (cc -> y > 0)
200                 set_frontcell(area[cc -> x][cc -> y - 1]);
201             break;
202         case creatureE:
203             if (cc -> x < size - 1)
204                 set_frontcell(area[cc -> x + 1][cc -> y]);
205             break;
206         case creatureS:
207             if (cc -> y < size - 1)
208                 set_frontcell(area[cc -> x][cc -> y + 1]);
209             break;
210         case creatureW:
211             if (cc -> x > 0)
212                 set_frontcell(area[cc -> x - 1][cc -> y]);
213             break;
214         default:
215             break;
216         }
217         cc ++;
218     }
219     front_set = true;
220 }
221
222
223 unsigned Field::clear_conflicts()
224 {
225     front_set = false;
226     return conflict_list.erase(area);
227 }
228
229
230 unsigned Field::clear_conflicts_by_area()
231 {
232     register unsigned amount = 0u;
233     register celltype **ax = area;
234     for (register unsigned int x = 0; x < size; x ++) {
235         register celltype *a = *ax;
236         for (register unsigned int y = 0; y < size; y ++) {
237             switch (*a) {
238             case conflict:
239                 ++ amount;
240                 *a = empty;
241                 break;
242             case frontcell:
243                 *a = empty;
244                 break;
245             default:
246                 break;
247             }
248             a ++;
```

```
249             }
250             ax ++;
251         }
252         front_set = false;
253         return amount;
254     }
255
256
257     unsigned Field::calculate_cells()
258     {
259         register unsigned new_visited = 0u;
260         register coordinate *cc = creature;
261         for (register unsigned c = 0; c < creatures; c ++) {
262             register celltype *a = &area[cc -> x][cc -> y];
263             register const coordinate *front = get_front(cc, a);
264             register const bool free = (front -> state != 0)? (frontcell == area[front -> x][front -> y
                    ]) : false;
265             register const celltype newturn = next(free, cc -> state, *a, cc -> algorithm_select);
266             if (free) {
267                 area[front -> x][front -> y] = newturn;
268                 *a = empty;
269                 cc -> x = front -> x;
270                 cc -> y = front -> y;
271             } else {
272                 conflict_list.append(front);
273                 *a = newturn;
274             }
275             if (!visited[cc -> x][cc -> y]) {
276                 visited[cc -> x][cc -> y] = true;
277                 ++ new_visited;
278             }
279             cc -> state = next(free, cc -> state, cc -> algorithm_select);
280             cc -> algorithm_select ^= 1;
281             cc ++;
282         }
283         return new_visited;
284     }
285
286
287     unsigned Field::calculate()
288     {
289         set_frontcells();
290         register const unsigned new_visited = calculate_cells();
291         conflicts += clear_conflicts();
292         unvisited -= new_visited;
293         ++ generation;
294         return new_visited;
295     }
296
297
298     void Field::set_creature(unsigned x, unsigned y, int dir, int id, int alg)
299     {
300         switch (dir) {
301         case 0: area[x][y] = creatureN; break;
302         case 1: area[x][y] = creatureE; break;
303         case 2: area[x][y] = creatureS; break;
304         case 3: area[x][y] = creatureW; break;
305         }
306         creature[id].x = x;
```

```
307     creature[id].y = y;
308     creature[id].state = 0;
309     creature[id].algorithm_select = alg? 2 : 0;
310 }
311
312
313 unsigned Field::load_field(std::ifstream &init)
314 {
315     register unsigned cid = 0;
316     unvisited = 0;
317     for (register unsigned y = 0; y < size; y ++)
318         for (register unsigned x = 0; x < size; x ++) {
319             register char c;
320             do {
321                 init >> c;
322             } while (c == 10 || c == 13);
323             switch (c) {
324             case '^': case 'n':
325                 if (cid < creatures)
326                     set_creature(x, y, 0, cid ++, c == 'n');
327                 unvisited ++;
328                 break;
329             case '>': case 'e':
330                 if (cid < creatures)
331                     set_creature(x, y, 1, cid ++, c == 'e');
332                 unvisited ++;
333                 break;
334             case 'v': case 's':
335                 if (cid < creatures)
336                     set_creature(x, y, 2, cid ++, c == 's');
337                 unvisited ++;
338                 break;
339             case '<': case 'w':
340                 if (cid < creatures)
341                     set_creature(x, y, 3, cid ++, c == 'w');
342                 unvisited ++;
343                 break;
344             case '+':
345                 area[x][y] = conflict;
346                 unvisited ++;
347                 break;
348             case '*':
349                 area[x][y] = obstacle;
350                 break;
351             case '.':
352                 unvisited ++;
353             default:
354                 area[x][y] = empty;
355                 break;
356             }
357             visited[x][y] = false;
358         }
359     generation = 0u;
360     conflicts = 0u;
361     return unvisited;
362 }
363
364
365 inline Field::celltype Field::next(bool free, int s, Field::celltype c, int alg) const
```

```
366 {
367     register const int rotate = output[alg][free? 1 : 0][s];
368     register int dir;
369     switch (c) {
370     case creatureN: dir = 0; break;
371     case creatureE: dir = 1; break;
372     case creatureS: dir = 2; break;
373     case creatureW: dir = 3; break;
374     default: return empty;
375     }
376     switch ((dir + rotate + 1) & 3) {
377     case 0: return creatureN;
378     case 1: return creatureE;
379     case 2: return creatureS;
380     case 3: return creatureW;
381     default: return empty;
382     }
383 }
384
385
386 inline int Field::next(bool free, int s, int alg) const
387 {
388     return nextstate[alg][free? 1 : 0][s];
389 }
390
391
392 void Field::select_alg(char nr, int target)
393 {
394     #define A(a,b,c,d,e,f) ((int []){a,b,c,d,e,f})[s]
395     #define CPY(SB,SF,OB,OF) \
396         nextstate[target][0][s] = SB; output[target][0][s] = OB; \
397         nextstate[target][1][s] = SF; output[target][1][s] = OF; \
398         break;
399     for (register unsigned s = 0; s < 6; s ++)
400         switch (nr) {
401         case 'a': case 'A': CPY(A(0,2,3,4,5,1), A(1,5,4,0,2,3), A(0,0,0,2,2,2), A(0,0,0,2,2,2));
402         case 'b': case 'B': CPY(A(1,2,0,4,5,3), A(3,1,5,0,4,2), A(0,0,0,2,2,2), A(0,2,0,2,0,2));
403         case 'c': case 'C': CPY(A(1,2,0,4,5,3), A(3,4,2,0,1,5), A(0,0,0,2,2,2), A(0,0,2,2,2,0));
404         case 'd': case 'D': CPY(A(1,2,3,1,5,1), A(1,0,2,4,3,1), A(0,0,0,2,2,2), A(0,2,2,0,2,2));
405         case 'e': case 'E': CPY(A(1,2,0,4,5,3), A(3,4,5,0,1,2), A(0,2,0,2,2,2), A(0,0,0,2,2,0));
406         case 'f': case 'F': CPY(A(1,2,0,4,5,3), A(3,4,5,0,1,2), A(0,2,2,0,0,0), A(2,2,2,0,2,0));
407         case 'g': case 'G': CPY(A(1,2,0,4,5,3), A(3,1,5,0,4,2), A(2,2,2,0,0,0), A(2,0,2,0,2,0));
408         case 'h': case 'H': CPY(A(1,2,3,4,2,0), A(2,4,0,3,5,4), A(2,2,0,2,0,2), A(2,2,0,2,2,0));
409         case 'i': case 'I': CPY(A(1,2,3,4,2,0), A(2,4,0,3,5,4), A(2,2,2,2,0,2), A(2,2,0,0,2,0));
410         case 'j': case 'J': CPY(A(1,2,3,0,4,5), A(4,5,3,2,0,1), A(0,0,0,0,2,2), A(0,0,2,2,2,2));
411         default:
412             for (register unsigned i = 0; i < 2; i ++) {
413                 nextstate[target][i][s] = 0;
414                 output[target][i][s] = 0;
415             }
416             break;
417         }
418     #undef A
419     #undef CPY
420 }
421
422
423 void Field::select_alg(const char *nr, int target)
424 {
```

```
425     for (register int i = 5; i >= 0; i −−)
426         for (register int j = 1; j >= 0; j −−) {
427             register const int index = i ∗ 4 + j ∗ (6 ∗ 4 + 2);
428             char state[2] = {nr[index], 0};
429             nextstate[target][j][i] = atoi(state);
430             output[target][j][i] = nr[index + 2] == '0'? 0 : 2;
431         }
432 }
433
434
435 int Field::get_creature_alg(unsigned int x, unsigned int y) const
436 {
437     register coordinate ∗c = creature;
438     register int i = creatures;
439     for (; i −− > 0; c ++)
440         if (c −> x == x && c −> y == y)
441             return c −> algorithm_select >> 1;
442     return −1;
443 }
444
445
446 class PrintField
447 {
448 protected:
449     std::ostream &out;
450
451 public:
452     PrintField(std::ostream &o): out(o) { }
453     virtual std::ostream &print(const Field &) = 0;
454 };
455
456
457 class PrintCRTField: public PrintField
458 {
459 public:
460     PrintCRTField(std::ostream &o);
461     std::ostream &print(const Field &);
462     std::ostream &print_creatures(const Field &);
463     std::ostream &print_creature(const Field &, unsigned id);
464 };
465
466
467 PrintCRTField::PrintCRTField(std::ostream &o):
468     PrintField(o)
469 {
470     out << "\033[2J";
471 }
472
473
474 std::ostream &PrintCRTField::print(const Field &f)
475 {
476     out << "\033[f";
477     for (register unsigned y = 0; y < f.get_size(); y ++) {
478         for (register unsigned x = 0; x < f.get_size(); x ++) {
479             switch (f.get_area()[x][y]) {
480             case Field::obstacle: out << '∗'; break;
481             case Field::creatureN: out << '^'; break;
482             case Field::creatureE: out << '>'; break;
483             case Field::creatureS: out << 'v'; break;
```

```
484                 case Field::creatureW: out << '<'; break;
485                 case Field::conflict: out << '+'; break;
486                 case Field::empty:
487                 case Field::frontcell:
488                     if (f.get_visited()[x][y])
489                         out << ' ';
490                     else
491                         out << '.';
492                     break;
493                 }
494             }
495             out << '\n';
496         }
497         out << "\n"
498             << "G" << f.get_generation()
499             << " U" << f.get_unvisited()
500             << " K" << f.get_conflicts() << " \n";
501         return out;
502 }
503
504
505 std::ostream &PrintCRTField::print_creatures(const Field &f)
506 {
507         for (register unsigned c = 0; c < f.get_creatures(); c ++)
508             print_creature(f, c);
509         return out;
510 }
511
512
513 inline std::ostream &PrintCRTField::print_creature(const Field &f,
514         unsigned id)
515 {
516         out << f.get_creature()[id].x << ", "
517             << f.get_creature()[id].y << " @"
518             << f.get_creature()[id].state << '\n';
519         return out;
520 }
521
522
523 class PrintPSField: public PrintField
524 {
525 protected:
526         const bool eps, colored, offset;
527         const int mode;
528         unsigned pspages;
529
530         PrintPSField(const Field &f, std::ostream &o, bool eps, bool color, bool offset, int mode, bool
                 simple);
531         void initialize(const Field &, bool simple);
532
533 public:
534         PrintPSField(const Field &f, std::ostream &o, bool eps = false, bool color = false, bool offset
                 = true, int mode = 4);
535         virtual ~PrintPSField();
536         std::ostream &print(const Field &);
537 };
538
539
```

```
540 PrintPSField::PrintPSField(const Field &f, std::ostream &o, bool e, bool col, bool os, int m, bool
        simple):
541     PrintField(o),
542     eps(e), colored(col), offset(os), mode(m),
543     pspages(0u)
544 {
545     initialize(f, simple);
546 }
547
548
549 PrintPSField::PrintPSField(const Field &f, std::ostream &o, bool e, bool col, bool os, int m):
550     PrintField(o),
551     eps(e), colored(col), offset(os), mode(m),
552     pspages(0u)
553 {
554     initialize(f, false);
555 }
556
557
558 void PrintPSField::initialize(const Field &f, bool simple)
559 {
560     if (eps)
561         out << "%!PS-Adobe-3.0 EPSF-3.0\n";
562     else
563         out << "%!PS-Adobe-3.0\n";
564     out << "%%Title: (Simulation World of Creatures)\n"
565     "%%Creator: (Mathias Halbach, wocsim)\n"
566     "%%CreationDate: (Fri Nov 11 10:00:00 2006)\n"
567     "%%Copyright: (mathias.halbach@informatik.tu-darmstadt.de)\n"
568     "%%BoundingBox: " << (offset? "50 50 " : "0 0 ")
569         << (f.get_size() + (offset? 5 : 0)) << "0 "
570         << (f.get_size() + (offset? 5 : 0) + (eps? 0 : 2)) << "0\n"
571     "%%DocumentData: Clean7Bit\n"
572     "%%LanguageLevel: 2\n"
573     "%%Orientation: Portrait\n"
574     "%%PageOrder: Ascend\n";
575     if (eps)
576         out << "%%Pages: 1\n";
577     else
578         out << "%%Pages: (atend)\n";
579     out << "%%DocumentNeededResources: font wocfont\n"
580     "%%+ font Helvetica\n"
581     "%%EndComments\n\n"
582     "%%BeginProlog\n"
583     "% convert -negate -density 200 $i.ps $i.png\n"
584     "/width " << f.get_size() << "0 def\n"
585     "/height width def\n"
586     "%%BeginResource: wocfont\n"
587     "/wocfont <<\n"
588     "/FontName (World Of Creatures)\n"
589     "/isFixedPitch true\n"
590     "/FontType 3\n"
591     "/FontMatrix [0.1 0 0 0.1 0 0]\n"
592     "/FontBBox [0 0 10 10]\n"
593     "/Encoding\n"
594     "0 1 41 { pop /.notdef } for\n"
595     "/obstacle /conflict /.notdef /.notdef /cookie\n"
596     "47 1 59 { pop /.notdef } for\n"
597     "/smileW /.notdef /smileE\n"
```

```
598     "63 1 68 { pop /.notdef } for\n"
599     "/AmoebaE 70 1 77 { pop /.notdef } for\n"
600     "/AmoebaN /.notdef /.notdef /.notdef /.notdef\n"
601     "/AmoebaS /.notdef /.notdef /.notdef /AmoebaW /visited\n"
602     "/.notdef /.notdef /.notdef /.notdef /.notdef /smileN\n"
603     "95 1 100 { pop /.notdef } for\n"
604     "/amoebaE 102 1 109 { pop /.notdef } for\n"
605     "/amoebaN /.notdef /.notdef /.notdef /.notdef\n"
606     "/amoebaS /.notdef /.notdef /smileS /amoebaW\n"
607     "120 1 255 { pop /.notdef } for\n"
608     "256 array astore\n"
609     "/CharProcs <<\n"
610     "/.notdef { }\n"
611     "/smileN { 5 5 moveto 5 5 3 110 70 arc closepath stroke\n"
612     "3.5 6 .5 0 359 arc fill }\n"
613     "/smileE { 5 5 moveto 5 5 3 20 340 arc closepath stroke\n"
614     "6 6.5 .5 0 359 arc fill }\n"
615     "/smileS { 5 5 moveto 5 5 3 290 250 arc closepath stroke\n"
616     "6.5 4 .5 0 359 arc fill }\n"
617     "/smileW { 5 5 moveto 5 5 3 200 160 arc closepath stroke\n"
618     "4 3.5 .5 0 359 arc fill }\n"
619     "/AmoebaN { newpath 5 7 1.3 0 359 arc fill\n"
620     "8 4 moveto 8 5 lineto 5 5 3 0 180 arc 2 4 lineto closepath\n"
621     "2 4 moveto 1 2 lineto\n"
622     "4 4 moveto 4 2 lineto\n"
623     "6 4 moveto 6 2 lineto\n"
624     "8 4 moveto 9 2 lineto\n"
625     "stroke }\n"
626     "/AmoebaE { newpath 7 5 1.3 0 359 arc fill\n"
627     "4 2 moveto 5 2 lineto 5 5 3 270 90 arc 4 8 lineto closepath\n"
628     "2 1 moveto 4 2 lineto\n"
629     "2 4 moveto 4 4 lineto\n"
630     "2 6 moveto 4 6 lineto\n"
631     "2 9 moveto 4 8 lineto\n"
632     "stroke }\n"
633     "/AmoebaS { newpath 5 3 1.3 0 359 arc fill\n"
634     "2 6 moveto 2 5 lineto 5 5 3 180 0 arc 8 6 lineto closepath\n"
635     "1 8 moveto 2 6 lineto\n"
636     "4 8 moveto 4 6 lineto\n"
637     "6 8 moveto 6 6 lineto\n"
638     "9 8 moveto 8 6 lineto\n"
639     "stroke }\n"
640     "/AmoebaW { newpath 3 5 1.3 0 359 arc fill\n"
641     "6 8 moveto 5 8 lineto 5 5 3 90 270 arc 6 2 lineto closepath\n"
642     "8 1 moveto 6 2 lineto\n"
643     "8 4 moveto 6 4 lineto\n"
644     "8 6 moveto 6 6 lineto\n"
645     "8 9 moveto 6 8 lineto\n"
646     "stroke }\n"
647     "/amoebaN { newpath 5 7 1.3 0 359 arc fill\n"
648     "8 4 moveto 8 5 lineto 5 5 3 0 180 arc 2 4 lineto closepath\n"
649     "2 2 8 { dup 2 moveto 4 lineto } for stroke }\n"
650     "/amoebaE { newpath 7 5 1.3 0 359 arc fill\n"
651     "4 2 moveto 5 2 lineto 5 5 3 270 90 arc 4 8 lineto closepath\n"
652     "2 2 8 { dup 2 exch moveto 4 exch lineto } for stroke }\n"
653     "/amoebaS { newpath 5 3 1.3 0 359 arc fill\n"
654     "2 6 moveto 2 5 lineto 5 5 3 180 0 arc 8 6 lineto closepath\n"
655     "2 2 8 { dup 8 moveto 6 lineto } for stroke }\n"
656     "/amoebaW { newpath 3 5 1.3 0 359 arc fill\n"
```

```
657     "6 8 moveto 5 8 lineto 5 5 3 90 270 arc 6 2 lineto closepath\n"
658     "2 2 8 { dup 8 exch moveto 6 exch lineto } for stroke }\n"
659     "/obstacle { newpath 2 setlinejoin\n"
660     "3 7 1 0 270 arc 3 5 1 90 270 arc 3 3 1 90 0 arc 5 3 1 180 0 arc 7 3 1"
661     " 180 90\n"
662     "arc 7 5 1 270 90 arc 7 7 1 270 180 arc 5 7 1 0 180 arc stroke }\n"
663     "/conflict { 5 3 moveto 5 7 lineto 3 5 moveto 7 5 lineto stroke }\n"
664     "/visited { 3 3 moveto 7 7 lineto 3 7 moveto 7 3 lineto stroke }\n"
665     "/cookie { newpath 5 5 1 0 360 arc closepath fill }\n"
666     ">>\n"
667     "/BuildChar { 1 index begin Encoding exch get BuildGlyph end } bind\n"
668     "/BuildGlyph { 10 0 0 0 10 10 setcachedevice exch begin CharProcs exch"
669     " get end exec }\n"
670     ">> definefont pop\n"
671     "%%EndResource\n\n";
672     if (colored) out <<
673     "/visited { moveto 0 10 rlineto 10 0 rlineto 0 −10 rlineto\n"
674     "closepath 0.5 setgray fill 0 setgray } def\n\n"
675     "/conflict { moveto 0 10 rlineto 10 0 rlineto 0 −10 rlineto\n"
676     "closepath 1 0 1 setrgbcolor fill 0 setgray } def\n"
677     "/vconflict { moveto 0 10 rlineto 10 0 rlineto 0 −10 rlineto\n"
678     "closepath 1 0 0 setrgbcolor fill 0 setgray } def\n\n";
679     else out <<
680     "/visited { moveto 0 10 rlineto 10 0 rlineto 0 −10 rlineto\n"
681     "closepath 0.8 setgray fill 0 setgray } def\n\n";
682     if (!simple) {
683         out << "/showobstacles {\n";
684         register unsigned ty = offset? f.get_size() + 3 : f.get_size() − 2;
685         for (unsigned y = 1; y < f.get_size() − 1; y ++, ty −−)
686             for (unsigned x = 1; x < f.get_size() − 1; x ++)
687                 if (f.get_area()[x][y] == Field::obstacle)
688                     out << (x + (offset? 5 : 0)) << "0 " << ty << "0 10 10 rectfill\n";
689         out << "} def\n\n";
690     }
691     out << "/showworld { ";
692     if (eps)
693         out << "pop ";
694     else
695         out << "/Helvetica findfont 11 scalefont setfont 0 setgray\n"
696         << (offset? "5" : "")
697         << "0 height 10 add " << (offset? "50 add " : "")
698         << "moveto show \n";
699     if (!simple) {
700         out << (colored? "0 1 0 setrgbcolor\n" : "0.5 setgray\n");
701         if (offset)
702             out << "50 50 width 10 rectfill\n"
703                 "50 50 10 height rectfill\n"
704                 "width 10 sub 50 add 50 10 height rectfill\n"
705                 "50 height 10 sub 50 add width 10 rectfill\n";
706         else
707             out << "0 0 width 10 rectfill\n"
708                 "0 0 10 height rectfill\n"
709                 "width 10 sub 0 10 height rectfill\n"
710                 "0 height 10 sub width 10 rectfill\n";
711     }
712     out << ".5 setlinewidth /wocfont findfont 10 scalefont setfont\n";
713     if (!simple)
714         out << "showobstacles ";
715     out << "0.1 setgray\n"
```

```
716         "} def\n\n"
717         "/showgrid { 0 setgray 0 setlinewidth\n"
718         << (offset? "5" : "") << "0 10 width " << (offset? "50 add " : "")
719         << "{ " << (offset? "5" : "") << "0 moveto 0 height rlineto } for\n"
720         << (offset? "5" : "") << "0 10 height " << (offset? "50 add " : "")
721         << "{ " << (offset? "5" : "") << "0 exch moveto width 0 rlineto } for\n"
722         "stroke } def\n"
723         "%%EndProlog\n"
724         "\n"
725         "%%BeginSetup\n"
726         ".5 setlinewidth\n"
727         "%%EndSetup\n\n";
728 }
729
730
731 PrintPSField::~PrintPSField()
732 {
733     if (!eps)
734         out << "%%Trailer\n%%Pages: " << pspages << "\n";
735     out << "%%EOF\n";
736 }
737
738
739 std::ostream &PrintPSField::print(const Field &f)
740 {
741     const char *creature = "^>v<" "nesw" "NESW" + mode;
742     out << "%%Page: (G" << f.get_generation() << ") "
743         << (eps? 1 : (++ pspages))
744         << "\n(G" << f.get_generation()
745         << " U" << f.get_unvisited()
746         << " +" << f.get_conflicts()
747         << ") showworld\n";
748     register unsigned ty = offset? f.get_size() + 3 : f.get_size() - 2;
749     register unsigned const ox = offset? 5 : 0;
750     for (unsigned y = 1; y < f.get_size() - 1; y ++, ty --)
751         for (unsigned x = 1; x < f.get_size() - 1; x ++)
752             if (colored && f.get_area()[x][y] == Field::conflict)
753                 out << (x + ox) << "0 " << ty << "0 " << (f.get_visited()[x][y]? "v" : "") << "
                        conflict\n";
754             else if (f.get_visited()[x][y])
755                 out << (x + ox) << "0 " << ty << "0 visited\n";
756     ty = offset? f.get_size() + 3 : f.get_size() - 2;
757     for (unsigned y = 1; y < f.get_size() - 1; y ++, ty --) {
758         out << (ox + 1) << "0 " << ty << "0 moveto (";
759         for (unsigned x = 1; x < f.get_size() - 1; x ++)
760             switch (f.get_area()[x][y]) {
761             case Field::creatureN:
762                 out << creature[0 - (f.get_creature_alg(x, y)? 4 : 0)];
763                 break;
764             case Field::creatureE:
765                 out << creature[1 - (f.get_creature_alg(x, y)? 4 : 0)];
766                 break;
767             case Field::creatureS:
768                 out << creature[2 - (f.get_creature_alg(x, y)? 4 : 0)];
769                 break;
770             case Field::creatureW:
771                 out << creature[3 - (f.get_creature_alg(x, y)? 4 : 0)];
772                 break;
773             case Field::conflict:
```

```
774                     out << '+';
775                     break;
776                 case Field::frontcell:
777                 case Field::obstacle:
778                 case Field::empty:
779                 default:
780                     out << ' ';
781                     break;
782                 }
783             out << ") show\n";
784         }
785         if (eps)
786             out << "showgrid\n";
787         else
788             out << "showgrid showpage\n\n";
789         return out;
790 }
791
792
793 class PrintSimplePSField: public PrintPSField
794 {
795 public:
796         PrintSimplePSField(const Field &f, std::ostream &o, bool eps = false, bool color = false, bool
                offset = true, int mode = 4);
797         std::ostream &print(const Field &);
798 };
799
800
801 PrintSimplePSField::PrintSimplePSField(const Field &f, std::ostream &o, bool e, bool c, bool os,
          int m):
802         PrintPSField(f, o, e, c, os, true, m)
803 {
804 }
805
806
807 std::ostream &PrintSimplePSField::print(const Field &f)
808 {
809         const char *creature = "^>v<" "nesw" "NESW" + mode;
810         out << "%%Page: (G" << f.get_generation() << ") " << (++ pspages)
811             << "\n(G" << f.get_generation()
812             << " U" << f.get_unvisited()
813             << " +" << f.get_conflicts()
814             << ") showworld\n";
815
816         out << (offset? "50 " : "0 ") << (f.get_size() + 4) << "0 moveto (";
817         for (unsigned x = 0; x < f.get_size(); x ++)
818             out << '*';
819         out << ") show\n";
820
821         register unsigned ty = offset? f.get_size() + 3 : f.get_size() - 2;
822         for (unsigned y = 1; y < f.get_size() - 1; y ++, ty --) {
823             out << (offset? "50 " : "0 ") << ty << "0 moveto (*";
824             for (unsigned x = 1; x < f.get_size() - 1; x ++)
825                 switch (f.get_area()[x][y]) {
826                 case Field::creatureN:
827                     out << creature[0 - (f.get_creature_alg(x, y)? 4 : 0)];
828                     break;
829                 case Field::creatureE:
830                     out << creature[1 - (f.get_creature_alg(x, y)? 4 : 0)];
```

```
831                     break;
832                 case Field::creatureS:
833                     out << creature[2 - (f.get_creature_alg(x, y)? 4 : 0)];
834                     break;
835                 case Field::creatureW:
836                     out << creature[3 - (f.get_creature_alg(x, y)? 4 : 0)];
837                     break;
838                 case Field::conflict:
839                     out << '+';
840                     break;
841                 case Field::obstacle:
842                     out << '*';
843                     break;
844                 case Field::frontcell:
845                 case Field::empty:
846                 default:
847                     if (f.get_visited()[x][y])
848                         out << ' ';
849                     else
850                         out << '.';
851                     break;
852                 }
853             out << "*) show\n";
854         }
855
856         out << (offset? "50 50" : "0 0") << " moveto (";
857         for (unsigned x = 0; x < f.get_size(); x ++)
858             out << '*';
859         out << ") show\nshowgrid" << (eps? "\n" : " showpage\n\n");
860         return out;
861 }
862
863
864 void printeps(Field &f, bool simple = false)
865 {
866         char filename[12];
867         sprintf(filename, "g%06d.eps", f.get_generation());
868         std::ofstream out;
869         out.open(filename, std::ios::out);
870         PrintField *p;
871         if (simple)
872             p = new PrintSimplePSField(f, out, true);
873         else
874             p = new PrintPSField(f, out, true, true);
875         p -> print(f);
876         delete p;
877         out.close();
878 }
879
880
881 int main(int argc, char *argv[])
882 {
883         if (argc != 3 && argc != 4) {
884             std::cerr << "Aufruf: " << argv[0] << " Datei.ini Algorithmus[Algorithmus] [Algorithmen
                        2. Creature]\n";
885             return 1;
886         }
887         std::ifstream init(argv[1]);
888         if (init.fail()) {
```

```
889         std::cerr << "Fehler beim Öffnen der Datei '" << argv[1] << "'.\n";
890         return 2;
891     }
892     unsigned int size, creatures;
893     init >> size >> creatures;
894     Field f(size, creatures);
895     if (argv[2][1] == '/') {
896         f.select_alg(argv[2], 0);
897         f.select_alg(argv[2], 1);
898         if (argv[3]) {
899             f.select_alg(argv[3], 2);
900             f.select_alg(argv[3], 3);
901         } else {
902             f.select_alg(argv[2], 2);
903             f.select_alg(argv[2], 3);
904         }
905     } else {
906         f.select_alg(argv[2][0], 0);
907         f.select_alg(argv[2][argv[2][1]? 1 : 0], 1);
908         if (argv[3]) {
909             f.select_alg(argv[3][0], 2);
910             f.select_alg(argv[3][argv[3][1]? 1 : 0], 3);
911         } else {
912             f.select_alg(argv[2][0], 2);
913             f.select_alg(argv[2][argv[2][1]? 1 : 0], 3);
914         }
915     }
916     f.load_field(init);
917 
918     f.set_frontcells();
919     PrintPSField ps(f, std::cout, false, true);
920     ps.print(f);
921     printeps(f);
922 
923     register unsigned long lastchange = 400u;
924     register const clock_t start = clock();
925     while (f.get_unvisited() != 0 && lastchange != 0) {
926         register const unsigned nv = f.calculate();
927         if (nv == 0)
928             lastchange --;
929         else
930             lastchange = 100000u;
931         f.set_frontcells();
932         ps.print(f);
933     }
934     ps.print(f);
935     register const clock_t end = clock();
936     std::clog << argv[1] << ';' << argv[2];
937     if (argv[3])
938         std::clog << '.' << argv[3];
939     std::clog << ';'
940         << creatures << ';'
941         << f.get_generation() << ';'
942         << f.get_conflicts() << ';'
943         << f.get_unvisited() << ';'
944         << (end - start) << ';';
945     if (f.get_unvisited() == 0)
946         std::clog << f.get_generation();
947     else
```

```
948             std::clog << "\"---\"";
949         std::clog << std::endl;
950         return 0;
951 }
```

```
35 0
**********************************
*................................*
*................................*
*................................*
*.......****..........****.......*
*..........*..........*..........*
*..........*..........*..........*
*..........*..........*..........*
*...*......*****...*****......*...*
*...*...........*...*...........*...*
*...*...........*...*...........*...*
*...*****......*...*......*****...*
*........*.....*...*.....*........*
*........*.....*...*.....*........*
*........*.....*...*.....*........*
*........*****.....*****.........*
*................................*
*................*...............*
*................................*
*........*****.....*****........*
*........*.....*...*.....*........*
*........*.....*...*.....*........*
*........*.....*...*.....*........*
*...*****......*...*......*****...*
*...*...........*...*...........*...*
*...*...........*...*...........*...*
*...*......******...*****......*...*
*..........*..........*..........*
*..........*..........*..........*
*..........*..........*..........*
*........****..........****.......*
*................................*
*................................*
*................................*
**********************************
```

Abbildung D.1: Initialdatei einer Umgebung

Algorithmus D.3: Skript zum Positionieren von Kreaturen für die Softwaresimulation

```
1  #!/usr/bin/perl
2  foreach $file (@ARGV) {
3      $file =~ s/\.ini$//;
4
5      $_ = <>;
6      next unless (/^35 0$/);
7      while (<>) {
8          s/[\n\r]+//;
9          push @field, $_;
10     }
11
12     foreach $d (("vv<<^^>>", "vs<wn^e>alternate", "ss<<nn>>side", "ssww^^>>corner")) {
13         &printfield( 1, 17, 33, 16, $d);
14         &printfield( 2, 17, 33, 16, $d);
15         &printfield( 4, 17, 17, 16, $d);
16         &printfield( 8, 9, 9, 16, $d);
17         &printfield(16, 5, 5, 8, $d);
18         &printfield(32, 3, 3, 4, $d);
19         &printfield(64, 2, 2, 2, $d);
20         &printfield(60, 3, 3, 2, $d);
21         &printfield(28, 5, 5, 4, $d);
22         &printfield(12, 9, 9, 8, $d);
23     }
24 }
```

```
25 exit;
26
27 sub printfield
28 {
29     # @_ = (creatures, offset_x, offset_y, increment, distribution)
30     @f = @field;
31     @d = split //, $_[4];
32     $j = 0;
33     for ($i = $_[1]; $i < 33; $i += $_[3]) {
34         $f[1] =~ s/^(.{$i})\./$1$d[$j]/;
35         $f[33] =~ s/^(.{$i})\./$1$d[4 + $j]/ if ($_[0] > 1);
36         $j = ($j + 1) & 1;
37     }
38     for ($i = $_[2]; $i < 33; $i += $_[3]) {
39         $f[$i] =~ s/^\*.(.+)\.\*/*$d[6 + $j]$1$d[2 + $j]*/;
40         $j = ($j + 1) & 1;
41     }
42     open(OUT, "> " . $file . "c" . substr($_[4], 8, 1) . "$_[0].ini");
43     print OUT "35 $_[0]\n";
44     print OUT join("\n", @f) . "\n";
45     close(OUT);
46 }
```

Algorithmus D.4: Autarke Aufzählung und Ausgabe von Automaten in Hardware

```
1  //‘define Xilinx
2  ‘define use_normal_clock
3  //‘define print_amount_only
4  //‘define compressed_output
5  //‘define display_only
6
7  ‘ifdef Xilinx
8  ‘define resetsignal clr
9  ‘define resetedge posedge clr
10 ‘define reset clr
11 ‘else
12 ‘define resetsignal clrn
13 ‘define resetedge negedge clrn
14 ‘define reset !clrn
15 ‘endif
16
17 module automatonlist(clka, ‘resetsignal, led, rxd, txd);
18 parameter states = 3’d6, inputs = 2’d2, outputs = 2’d2;
19 parameter statebits = 3, outputbits = 1, inputbits = 1;
20 localparam amountbits = 64; // clogb2(((states * outputs) ** (states * inputs)))
21 input clka, ‘resetsignal, rxd;
22 output txd;
23 output reg [7 : 0] led;
24 wire clk, clock_ready;
25 ‘ifdef use_normal_clock
26 assign clk = clka;
27 assign clock_ready = 1’b1;
28 ‘else
29 ‘ifdef Xilinx
30  DCM #(// 50 MHz -> 40 MHz
31         .CLKFX_DIVIDE(5), // Can be any integer from 1 to 32
32         .CLKFX_MULTIPLY(4), // Can be any integer from 2 to 32
33         .CLKIN_DIVIDE_BY_2("FALSE"), // TRUE/FALSE to enable CLKIN divide by two feature
34         .CLKIN_PERIOD(20.000), // Specify period of input clock
```

```
35          .CLK_FEEDBACK("NONE"), // Specify clock feedback of NONE, 1X or 2X
36          .FACTORY_JF(16'h8080) // FACTORY JF values
37      ) DCM_inst (
38          .CLKFX(clk), // DCM CLK synthesis out (M/D)
39          .LOCKED(clock_ready), // DCM LOCK status output
40          .CLKIN(clka), // Clock input (from IBUFG, BUFG or DCM)
41          .RST(clr) // DCM asynchronous reset input
42      );
43  `else
44  clkdata pll1(!clrn, clka, clk, clock_ready);
45  `endif
46  `endif
47
48  reg [7 : 0] tdata;
49  wire [7 : 0] rdata;
50  wire rready, tena, tready;
51  `ifdef display_only
52  assign {rready, tready} = 2'b00;
53  assign txd = rxd;
54  `else
55  uart com(clk, clka, !(`reset), 1'b0, rxd, txd, rdata, rready, tdata, tena, tready);
56  `ifndef Xilinx
57  defparam com.use_tmem = "on", com.use_rmem = "on";
58  `endif
59  `endif
60
61  reg [1 : 0] calculate = 2'b00;
62  reg printing;
63  wire overflow, q_valid, running;
64  wire [states * (2 ** inputbits) * (statebits + outputbits) - 1 : 0] q;
65  wire [amountbits - 1 : 0] amount, tests, hops, outs, beststarts, duplicate_free, clock_meter;
66  automaton #(states, inputs, outputs, statebits, outputbits, inputbits) a(clk, `resetsignal, calculate[0], !
        printing, overflow, q, q_valid, running, amount, tests, hops, outs, beststarts, duplicate_free,
        clock_meter);
67  always @(posedge clk or `resetedge)
68      if (`reset)
69          calculate <= 2'b00;
70      else if (clock_ready)
71          calculate <= overflow? 2'b10 : 2'b01;
72
73
74  // output led
75  `ifdef display_only
76  reg [31 : 0] outsel;
77  always @(posedge clk or `resetedge)
78      if (`reset)
79          outsel <= 0;
80      else
81          outsel <= outsel + 1;
82  wire [amountbits - 1 : 0] display1, display2;
83  assign display1 = amount;
84  assign display2 = outs;
85  wire signal = running ^ &outsel[24 : 23];
86  always @(posedge clk)
87      case (outsel[31 : 28])
88        0: led <= {signal, 1'b0, display1[61 : 56]};
89        1: led <= display1[55 : 48];
90        2: led <= display1[47 : 40];
91        3: led <= display1[39 : 32];
```

```
92          4: led <= display1[31 : 24];
93          5: led <= display1[23 : 16];
94          6: led <= display1[15 : 8];
95          7: led <= display1[ 7 : 0];
96          8: led <= {signal, 1'b1, display2[61 : 56]};
97          9: led <= display2[55 : 48];
98         10: led <= display2[47 : 40];
99         11: led <= display2[39 : 32];
100        12: led <= display2[31 : 24];
101        13: led <= display2[23 : 16];
102        14: led <= display2[15 : 8];
103        15: led <= display2[ 7 : 0];
104        default led <= 8'h00;
105        endcase
106 `else
107 always @* led <= {running, amount[6 : 0]};
108 `endif
109
110
111 // output serial
112 initial printing = 1'b0;
113 assign tena = printing;
114 `ifndef display_only
115 reg silent = 0, amount_request = 0, amount_printing = 0;
116 reg [statebits - 1 : 0] print_i = 0;
117 reg [inputbits - 1 : 0] print_j = 0;
118 reg [amountbits - 3 : 0] print_k = 0;
119 reg [2 : 0] print_c = 3'd0;
120 always @(posedge clk or `resetedge)
121     if (`reset) begin
122         printing <= 1'b0;
123         amount_printing <= 1'b0;
124         print_c <= 0;
125     end else if (printing) begin
126         printing <= !(print_c == 3'h7 && tready);
127         if (tready)
128             `ifdef compressed_output
129             if (!amount_printing && print_c == 0)
130             `else
131             if (!amount_printing && print_c == 3)
132             `endif
133                 if (print_i < states - 1) begin
134                     print_i <= print_i + 1'd1;
135                     print_c <= 3'd0;
136                 end else begin
137                     print_i <= 0;
138                     if (print_j < inputs - 1) begin
139                         print_j <= print_j + 1'd1;
140                         print_c <= 3'd4;
141                     end else begin
142                         print_j <= 0;
143                         print_c <= 3'd6;
144                     end
145                 end
146             `ifdef compressed_output
147             else if (!amount_printing && print_c == 4)
148             `else
149             else if (!amount_printing && print_c == 5)
150             `endif
```

```
151                     print_c <= 3'd0;
152                 else if (amount_printing && (print_c == 0 || print_c == 2 || print_c == 4)) begin
153                     if (print_k == 0) begin
154                         print_c <= print_c + 3'd1;
155                         print_k <= amountbits / 4 - 1;
156                     end else
157                         print_k <= print_k - 1;
158                 end else
159                     print_c <= print_c + 3'd1;
160         end else begin
161             printing <= (!silent && q_valid) || amount_request;
162             amount_printing <= amount_request;
163             print_i <= 0;
164             print_j <= 0;
165             print_k <= amountbits / 4 - 1;
166             print_c <= 0;
167         end
168 
169 always @(posedge clk or `resetedge)
170         if (`reset) begin
171             silent <= 1'b0;
172             amount_request <= 1'b0;
173         end else begin
174             if (rready)
175                 if (silent)
176                     silent <=!(!rdata[7] && rdata[6:5] != 2'b01 && rdata[4:0] == 5'h11); // Q, q, ^Q
177                 else
178                     silent <= (!rdata[7] && rdata[6:5] != 2'b01 && rdata[4:0] == 5'h13) || ~|rdata; //
                            S, s, ^S, ^@
179             if (amount_printing)
180                 amount_request <= 1'b0;
181             else if (rready)
182                 amount_request <= (rdata == 8'd65) || (rdata == 8'd97); // A, a
183         end
184 
185 wire [7 : 0] print_s, print_y, print_a, print_t, print_h, print_m, print_b, print_d;
186 localparam alloutputbits = (states * (2 ** inputbits) * outputbits);
187 function [statebits + inputbits - 1 : 0] index;
188 input [statebits - 1 : 0] i;
189 input [inputbits - 1 : 0] j;
190 begin
191     index = {i, j};
192 end
193 endfunction
194 `ifdef Xilinx
195 `define sij(ij,p) {p[(ij)+2], p[(ij)+1], p[(ij)]}
196 `define yij(ij,p) {p[(ij)]}
197 `define s(i,j,p) `sij((statebits * (index(states - 1 - (i), inputs - 1 - (j)))) + alloutputbits, p)
198 `define y(i,j,p) `yij((index(states - 1 - (i), inputs - 1 - (j))), p)
199 `else
200 `define s(i,j,p) p[(statebits * index(states - 1 - (i), inputs - 1 - (j))) + alloutputbits +: statebits]
201 `define y(i,j,p) p[index(states - 1 - (i), inputs - 1 - (j)) +: outputbits]
202 `endif
203 `ifdef compressed_output
204 digit2char sDigit(print_s, {`y(print_i, print_j, q), `s(print_i, print_j, q)});
205 assign print_y = 8'd63;
206 `else
207 assign print_s = {4'h3, {4 - statebits{1'b0}}, `s(print_i, print_j, q)};
208 assign print_y = {4'h3, {4 - outputbits{1'b0}}, `y(print_i, print_j, q)};
```

```
209 ‘endif
210 ‘ifdef Xilinx
211 ‘define d(p) {p[{print_k, 2’d0}+3],p[{print_k, 2’d0}+2],p[{print_k, 2’d0}+1],p[{print_k, 2’d0}]}
212 digit2char aDigit(print_a, ‘d(amount));
213 digit2char tDigit(print_t, ‘d(tests));
214 digit2char mDigit(print_m, ‘d(clock_meter));
215 digit2char dDigit(print_d, ‘d(duplicate_free));
216 digit2char hDigit(print_h, ‘d(outs));
217 digit2char bDigit(print_b, ‘d(beststarts));
218 ‘else
219 digit2char aDigit(print_a, amount[{print_k, 2’d0} +: 4]);
220 digit2char tDigit(print_t, tests[{print_k, 2’d0} +: 4]);
221 digit2char mDigit(print_m, clock_meter[{print_k, 2’d0} +: 4]);
222 digit2char dDigit(print_d, duplicate_free[{print_k, 2’d0} +: 4]);
223 digit2char hDigit(print_h, beststarts[{print_k, 2’d0} +: 4]);
224 digit2char bDigit(print_b, outs[{print_k, 2’d0} +: 4]);
225 ‘endif
226
227 ‘ifdef print_amount_only
228 always @∗
229     case (print_c)
230     3’d0: tdata = print_a;
231     3’d1: tdata = 8’d32;
232     3’d2: tdata = print_d;
233     3’d3: tdata = 8’d32;
234     3’d4: tdata = print_m;
235     3’d5: tdata = 8’d32;
236     3’d6: tdata = 8’d13;
237     3’d7: tdata = 8’d10;
238     default tdata = 8’d63;
239     endcase
240 ‘else
241 always @∗
242     case (print_c)
243     3’d0: tdata = amount_printing? rdata[5]? print_a : print_d : print_s;
244     3’d1: tdata = amount_printing? 8’d32 : 8’d47;
245     3’d2: tdata = amount_printing? rdata[5]? print_t : print_h : print_y;
246     3’d3: tdata = 8’d32;
247     3’d4: tdata = amount_printing? rdata[5]? print_m : print_b : 8’d45;
248     3’d5: tdata = 8’d32;
249     3’d6: tdata = 8’d13;
250     3’d7: tdata = 8’d10;
251     default tdata = 8’d63;
252     endcase
253 ‘endif
254 ‘endif
255
256 endmodule
```

Algorithmus D.5: Untermodul der Aufzählung von Automaten in Hardware

```
1 //‘define Xilinx
2 //‘define use_conditional_h
3 ‘define any_start_state
4 ‘define use_start_state_skip
5 //‘define simple_h_prefix
6
7 ‘ifdef Xilinx
8 ‘define resetsignal clr
```

```
9  ‘define resetedge posedge clr
10 ‘define reset clr
11 ‘else
12 ‘define resetsignal clrn
13 ‘define resetedge negedge clrn
14 ‘define reset !clrn
15 ‘endif
16
17 module automaton(clk, ‘resetsignal, active, next, overflow, q, q_valid, running, amount, tests, hops,
       outs, beststarts, duplicate_free, clock_meter);
18 parameter states = 3’d7, inputs = 2’d2, outputs = 2’d2;
19 parameter statebits = 3, outputbits = 1, inputbits = 1;
20 localparam pipelinelength = states + 6;
21 ‘ifdef any_start_state
22 localparam investigations = 3 + states − 1;
23 ‘else
24 localparam investigations = 3;
25 ‘endif
26 localparam amountbits = 64; // clogb2(((states ∗ outputs) ∗∗ (states ∗ inputs)))
27 input clk, ‘resetsignal;
28
29 input active, next;
30 output reg overflow;
31 output reg [states ∗ (2 ∗∗ inputbits) ∗ (statebits + outputbits) − 1 : 0] q;
32 output reg q_valid;
33 output running;
34 output reg [amountbits − 1 : 0] amount = 0, tests = 0, hops = 0, outs = 0, beststarts = 0,
      duplicate_free = 0, clock_meter = 0;
35
36 localparam alloutputbits = (states ∗ (2 ∗∗ inputbits) ∗ outputbits);
37 ‘define size(s,d) ({s{1’b0}} | (d))
38 ‘ifdef Xilinx
39 // für Xilinx ISE 9.2.03i mit outputbits = 1, statebits = 3:
40 ‘define sij(ij,p) {p[(ij)+2], p[(ij)+1], p[(ij)]}
41 ‘define yij(ij,p) {p[(ij)]}
42 ‘define s(i,j,p) ‘sij((statebits ∗ index(states − 1 − (i), inputs − 1 − (j))) + alloutputbits, p)
43 ‘define y(i,j,p) ‘yij((index(states − 1 − (i), inputs − 1 − (j))), p)
44 ‘define v(i,p) ‘sij(statebits ∗ (i), p)
45 ‘else
46 ‘define s(i,j,p) p[(statebits ∗ index(states − 1 − (i), inputs − 1 − (j))) + alloutputbits +: statebits]
47 ‘define y(i,j,p) p[index(states − 1 − (i), inputs − 1 − (j)) +: outputbits]
48 ‘define v(i,p) p[statebits ∗ (i) +: statebits]
49 ‘endif
50 ‘define a(i,j,p) A[p][(‘size(2 ∗ statebits, i) << statebits) | (j)]
51 function [statebits + inputbits − 1 : 0] index;
52 input [statebits − 1 : 0] i;
53 input [inputbits − 1 : 0] j;
54 begin
55     index = {i, j};
56 end
57 endfunction
58
59 reg next_automaton;
60 always @(posedge clk or ‘resetedge)
61     if (‘reset)
62         next_automaton <= 1’b0;
63     else if (active)
64         next_automaton <= next;
65
```

```
66  reg [investigations : 1] hvalid;
67  reg hminvalid;
68  reg [statebits − 1 : 0] h1 [investigations : 1], h1min;
69  reg [inputbits − 1 : 0] h2 [investigations : 1], h2min;
70  reg [statebits − 0 : 0] h3 [investigations : 1], h3min;
71  wire [states ∗ (2 ∗∗ inputbits) ∗ (statebits + outputbits) − 1 : 0] hresult;
72  reg [states ∗ (2 ∗∗ inputbits) ∗ (statebits + outputbits) − 1 : 0] m [1 : pipelinelength + 1];
73  reg [2 ∗∗ (statebits ∗ 2) − 1 : 0] A [2 : pipelinelength − 2];
74  reg [1 : pipelinelength + 1] valid;
75  reg [1 : pipelinelength + 1] fin;
76  initial begin: init
77      integer i;
78      for (i = 1; i <= pipelinelength + 1; i = i + 1)
79          m[i] = {states ∗ (2 ∗∗ inputbits) ∗ (statebits + outputbits){1'b0}};
80      for (i = 1; i <= investigations; i = i + 1)
81          {hvalid[i], h1[i], h2[i], h3[i]} = 0;
82      for (i = 2; i <= pipelinelength − 2; i = i + 1)
83          A[i] = {2 ∗∗ (statebits ∗ 2){1'b0}};
84      {hminvalid, h1min, h2min, h3min} = 0;
85  end
86  wire ena, issue_queue_ready, use_h;
87  assign ena = active && (next_automaton || !valid[pipelinelength + 1] || !valid[pipelinelength]);
88  assign issue_queue_ready = active && (next_automaton || !valid[pipelinelength + 1]);
89  `ifdef use_conditional_h
90  assign use_h = hminvalid && (m[1] < hresult);
91  `else
92  assign use_h = hminvalid;
93  `endif
94
95  wire ena_state [states − 1: 0][inputs − 1 : 0];
96  wire ena_out [states − 1: 0][inputs − 1 : 0];
97  wire overflow0;
98  assign ena_state[states − 1][inputs − 1] = ena_out[0][0] && (`y(0,0,m[1]) == outputs − 1'b1);
99  assign ena_out[states − 1][inputs − 1] = ena;
100 assign overflow0 = ena_state[0][0] && (`s(0,0,m[1]) == states − 1'b1);
101
102 genvar i, j, k, l;
103 generate
104     // machine memory pipeline
105     for (i = 2; i <= pipelinelength; i = i + 1) begin: pipeline
106         always @(posedge clk or `resetedge)
107             if (`reset)
108                 m[i] <= {states ∗ (2 ∗∗ inputbits) ∗ (statebits + outputbits){1'b0}};
109             else if (ena)
110                 m[i] <= m[i − 1];
111     end
112
113     // increment or hop
114     for (i = 1; i < states; i = i + 1) begin: carry0
115         assign ena_state[i − 1][inputs − 1] = (`s(i,0,m[1]) == states − 1'b1) && ena_state[i][0];
116         assign ena_out[i − 1][inputs − 1] = (`y(i,0,m[1]) == outputs − 1'b1) && ena_out[i][0];
117     end
118     for (i = 0; i < states; i = i + 1) begin: stateloop
119         for (j = 0; j < inputs; j = j + 1) begin: inputloop
120             wire overflow_state, overflow_out;
121             assign overflow_state = `s(i,j,m[1]) == states − 1'b1;
122             assign overflow_out = `y(i,j,m[1]) == outputs − 1'b1;
123             if (j > 0) begin: carryj
124                 assign ena_state[i][j − 1] = overflow_state && ena_state[i][j];
```

```
125                 assign ena_out[i][j − 1] = overflow_out && ena_out[i][j];
126             end
127             assign ‘s(i,j,hresult) = (h1min == i)? ((h2min == j)?
128                     ((‘s(i,j,m[pipelinelength]) >= h3min)? ‘s(i,j,m[pipelinelength]) : h3min − 2’
                            d1) :
129                     ((h2min < j)? states − 2’d1 : ‘s(i,j,m[pipelinelength]))) :
130                     ((h1min < i)? states − 2’d1 : ‘s(i,j,m[pipelinelength]));
131             assign ‘y(i,j,hresult) = outputs − 2’d1;
132             always @(posedge clk or ‘resetedge)
133                 if (‘reset)
134                     {‘s(i,j,m[1]), ‘y(i,j,m[1])} <= {statebits + outputbits{1’b0}};
135                 else if (!active)
136                     {‘s(i,j,m[1]), ‘y(i,j,m[1])} <= {statebits + outputbits{1’b0}};
137                 else if (ena && use_h)
138                     {‘s(i,j,m[1]), ‘y(i,j,m[1])} <= {‘s(i,j,hresult), ‘y(i,j,hresult)};
139                 else begin
140                     if (ena_state[i][j])
141                         if (overflow_state)
142                             ‘s(i,j,m[1]) <= {statebits{1’b0}};
143                         else
144                             ‘s(i,j,m[1]) <= ‘s(i,j,m[1]) + 1’d1;
145                     if (ena_out[i][j])
146                         if (overflow_out)
147                             ‘y(i,j,m[1]) <= {outputbits{1’b0}};
148                         else
149                             ‘y(i,j,m[1]) <= ‘y(i,j,m[1]) + 1’d1;
150                 end
151         end
152     end
153
154     // min h
155     wire [statebits + inputbits + statebits : 0] hmin;
156     wire [investigations : 1] hbus [statebits + inputbits + statebits : 0];
157     always @(posedge clk or ‘resetedge)
158         if (‘reset)
159             {hminvalid, h1min, h2min, h3min} <= 0;
160         else if (ena) begin
161             hminvalid <= hmin[statebits + inputbits + statebits] && valid[pipelinelength − 1] &&
                        !hminvalid;
162             h1min <= ~hmin[statebits + inputbits + statebits − 1 : inputbits + statebits];
163             h2min <= ~hmin[inputbits + statebits − 1 : statebits];
164             h3min <= hmin[statebits − 1 : 0];
165         end
166     for (i = 1; i <= investigations; i = i + 1) begin: hopI
167         wire [statebits + inputbits + statebits : 0] elected;
168         assign hbus[statebits + inputbits + statebits][i] = hvalid[i];
169         assign elected[statebits + inputbits + statebits] = hvalid[i];
170         for (j = 0; j < statebits; j = j + 1) begin: hopState
171             assign hbus[j + inputbits + statebits][i] = !h1[i][j] && elected[j + inputbits + statebits
                        + 1];
172             assign elected[j + inputbits + statebits] = elected[j + inputbits + statebits + 1] && (h1
                        [i][j] != hmin[j + inputbits + statebits]);
173             assign hbus[j][i] = h3[i][j] && elected[j + 1];
174             assign elected[j] = elected[j + 1] && (h3[i][j] == hmin[j]);
175         end
176         for (j = 0; j < inputbits; j = j + 1) begin: hopInput
177             assign hbus[j + statebits][i] = !h2[i][j] && elected[j + statebits + 1];
178             assign elected[j + statebits] = elected[j + statebits + 1] && (h2[i][j] != hmin[j +
                        statebits]);
```

```
179         end
180     end
181     for (i = 0; i <= statebits + inputbits + statebits; i = i + 1) begin: hopBit
182         assign hmin[i] = |hbus[i];
183     end
184
185     // reflexive transitive closure
186     for (j = 0; j < states; j = j + 1) begin: targetA
187         for (i = 0; i < states; i = i + 1) begin: sourceA
188             wire [inputs - 1 : 0] conjunction;
189             for (k = 0; k < inputs; k = k + 1) begin: prepareA
190                 assign conjunction[k] = `s(i,k,m[1]) == j;
191             end
192             always @(posedge clk or `resetedge)
193                 if (`reset)
194                     `a(i,j,2) <= 1'b0;
195                 else if (ena)
196                     `a(i,j,2) <= |conjunction || (i == j);
197             for (k = 0; k < states; k = k + 1) begin: connection
198                 always @(posedge clk or `resetedge)
199                     if (`reset)
200                         `a(i,k,j + 3) <= 1'b0;
201                     else if (ena)
202                         `a(i,k,j + 3) <= `a(i,k,j + 2) || (`a(i,j,j + 2) && `a(j,k,j + 2));
203             end
204         end
205     end
206     for (i = states + 3; i <= pipelinelength - 2; i = i + 1) begin: copy_A
207         always @(posedge clk or `resetedge)
208             if (`reset)
209                 A[i] <= 0;
210             else if (ena)
211                 A[i] <= A[i - 1];
212     end
213 endgenerate
214
215 // connected
216 wire [states - 1 : 0] connection;
217 wire [statebits - 1 : 0] connection_digitsum [0 : states - 1];
218 always @(posedge clk or `resetedge)
219     if (`reset)
220         {hvalid[1], h1[1], h2[1], h3[1]} <= 0;
221     else if (ena) begin
222         hvalid[1] <= !connection[states - 1];
223         h1[1] <= connection_digitsum[states - 1] - 2'd1;
224         h2[1] <= inputs - 1'd1;
225         h3[1] <= connection_digitsum[states - 1];
226     end
227 generate
228     assign connection[0] = 1'b1;
229     assign connection_digitsum[0] = connection[0];
230     for(i = 1; i < states; i = i + 1) begin: h_connected
231         assign connection[i] = connection[i - 1] && `a(0,i,pipelinelength - 2);
232         assign connection_digitsum[i] = connection_digitsum[i - 1] + connection[i];
233     end
234 endgenerate
235
236 // prefix
237 reg [statebits - 1 : 0] prefix_i, prefix_j;
```

```
238 wire [states − 1 : 0] prefix_i_bit, prefix_j_bit, prefix_k_bit;
239 wire [states − 1 : 1] prefixfree;
240 wire [statebits − 1 : 0] prefix_i_sum [0 : states − 1], prefix_j_sum [0 : states − 1];
241 wire [statebits − 0 : 0] prefix_k_sum [0 : states − 1];
242 always @(posedge clk or `resetedge)
243     if (`reset)
244         {hvalid[2], h1[2], h2[2], h3[2]} <= 0;
245     else if (ena) begin
246         hvalid[2] <= ~&prefixfree;
247         `ifdef simple_h_prefix
248         h1[2] <= states − 1;
249         h2[2] <= inputs − 1'd1;
250         h3[2] <= `s(states − 1, inputs − 1'd1, m[pipelinelength − 2]) + 1;
251         `else
252         h1[2] <= prefix_j;
253         h2[2] <= inputs − 1'd1;
254         h3[2] <= prefix_k_sum[states − 1];
255         `endif
256         prefix_j <= prefix_j_sum[0];
257         prefix_i <= prefix_i_sum[states − 1];
258     end
259 generate
260     assign prefix_i_bit[0] = 1'b1;
261     assign prefix_i_sum[0] = prefix_i_bit[0];
262     assign prefix_j_bit[states − 1] = `a(prefix_i,states − 1,pipelinelength − 3);
263     assign prefix_j_sum[states − 1] = 0;
264     assign prefix_k_bit[0] = 1'b1;
265     assign prefix_k_sum[0] = prefix_k_bit[0];
266     for (i = 1; i < states; i = i + 1) begin: h_prefix
267         assign prefix_i_bit[i] = prefix_i_bit[i − 1] && !(`a(0,i,pipelinelength − 4) && !`a(i,0,
                pipelinelength − 4));
268         assign prefix_i_sum[i] = prefix_i_sum[i − 1] + prefix_i_bit[i];
269         assign prefix_j_bit[i − 1] = prefix_j_bit[i] || (`a(prefix_i,i − 1,pipelinelength − 3));
270         assign prefix_j_sum[i − 1] = prefix_j_sum[i] + prefix_j_bit[i];
271         assign prefix_k_bit[i] = prefix_k_bit[i − 1] && !(((`s(prefix_j, inputs − 1, m[
                pipelinelength − 2]) < i) && `a(i,0,pipelinelength − 2)) || (i − 1 == prefix_j));
272         assign prefix_k_sum[i] = prefix_k_sum[i − 1] + prefix_k_bit[i];
273         assign prefixfree[i] = !`a(0,i,pipelinelength − 2) || `a(i,0,pipelinelength − 2);
274     end
275 endgenerate
276
277 // isomorphism
278 reg [statebits − 1 : 0] isomorphism_t [2 : states + 2];
279 reg isomorphism_result [2 : pipelinelength − 2];
280 reg [statebits − 1 : 0] isomorphism_i [2 : pipelinelength − 2];
281 reg [inputbits − 1 : 0] isomorphism_j [2 : pipelinelength − 2];
282 generate
283     always @(posedge clk or `resetedge)
284         if (`reset)
285             isomorphism_t[2] <= 0;
286         else
287             isomorphism_t[2] <= 1;
288     for (i = 0; i < states; i = i + 1) begin: isomorphism // pipeline access: write[i + 3] <= read[i +
            2]
289         wire [statebits − 1 : 0] t [0 : inputs];
290         wire [inputs − 1 : 0] violation;
291         wire [inputbits − 1 : 0] violation_digitsum [0 : inputs];
292         assign t[0] = isomorphism_t[i + 2];
293         assign violation_digitsum[0] = 0;
```

```
294         for (j = 0; j < inputs; j = j + 1) begin: x
295             assign t[j + 1] = (‘s(i,j,m[i+2]) == t[j])? t[j] + 2’d1 : t[j];
296             assign violation[j] = ‘s(i,j,m[i+2]) > t[j];
297             assign violation_digitsum[j + 1] = violation_digitsum[j] + ~|violation[j : 0];
298         end
299         always @(posedge clk or ‘resetedge)
300             if (‘reset) begin
301                 isomorphism_t[i + 3] <= 0;
302                 isomorphism_result[i + 3] <= 0;
303                 isomorphism_i[i + 3] <= 0;
304                 isomorphism_j[i + 3] <= 0;
305             end else if (ena) begin
306                 isomorphism_t[i + 3] <= t[inputs];
307                 if (isomorphism_result[i + 2] || ~|violation) begin
308                     isomorphism_result[i + 3] <= isomorphism_result[i + 2];
309                     isomorphism_i[i + 3] <= isomorphism_i[i + 2];
310                     isomorphism_j[i + 3] <= isomorphism_j[i + 2];
311                 end else begin
312                     isomorphism_result[i + 3] <= 1’b1;
313                     isomorphism_i[i + 3] <= i;
314                     isomorphism_j[i + 3] <= violation_digitsum[inputs];
315                 end
316             end
317     end
318     for (i = states + 3; i <= pipelinelength − 2; i = i + 1) begin: isomorphism_copy
319         always @(posedge clk or ‘resetedge)
320             if (‘reset)
321                 {isomorphism_result[i], isomorphism_i[i], isomorphism_j[i]} <= 0;
322             else if (ena) begin
323                 isomorphism_result[i] <= isomorphism_result[i − 1];
324                 isomorphism_i[i] <= isomorphism_i[i − 1];
325                 isomorphism_j[i] <= isomorphism_j[i − 1];
326             end
327     end
328     always @(posedge clk or ‘resetedge)
329         if (‘reset)
330             {isomorphism_result[2], isomorphism_i[2], isomorphism_j[2], hvalid[3], h1[3], h2
                    [3], h3[3]} <= 0;
331         else if (ena) begin
332             isomorphism_result[2] <= 1’b0;
333             isomorphism_i[2] <= 1’b0;
334             isomorphism_j[2] <= 1’b0;
335             hvalid[3] <= isomorphism_result[pipelinelength − 2];
336             h1[3] <= isomorphism_i[pipelinelength − 2];
337             h2[3] <= isomorphism_j[pipelinelength − 2];
338             h3[3] <= states;
339         end
340 endgenerate
341
342 // start state isomorphism
343 reg [1 : states − 1] startperm = 0;
344 ‘ifdef any_start_state
345 generate
346     for (l = 1; l < states; l = l + 1) begin: start_state
347         reg [states ∗ statebits − 1 : 0] perm [1 : states + 1], reperm [1 : states + 1];
348         reg [states − 1 : 0] permvalid [1 : states + 1], repermvalid [1 : states + 1];
349         reg [statebits − 1 : 0] k0 [1 : states + 1];
350         reg [1 : pipelinelength − 2] valid;
351         reg [1 : states + 1] equal_s, equal_y, improvable_y;
```

```
352         reg [1 : pipelinelength − 2] improvable_s;
353         reg [statebits − 1 : 0] h1perm [1 : pipelinelength − 2];
354         wire [statebits − 1 : 0] new_perm_state = l;
355         always @(posedge clk) begin
356             h1perm[1] <= new_perm_state; // = ‘v(0, reperm[l])
357             perm[1] <= {states ∗ statebits{1’b0}};
358             reperm[1] <= {{(states − 1) ∗ statebits{1’b0}}, new_perm_state};
359             permvalid[1] <= {{states − 1 − l{1’b0}}, 1’b1, {l{1’b0}}};
360             repermvalid[1] <= {{states − 1{1’b0}}, 1’b1};
361             k0[1] <= 1;
362             valid[1] <= 1’b1;
363             {equal_s[1], improvable_s[1], equal_y[1], improvable_y[1]} <= 4’b1010;
364         end
365         for (i = 0; i < states; i = i + 1) begin: s
366             wire [states ∗ statebits − 1 : 0] new_perm [0 : inputs], new_reperm [0 : inputs];
367             wire [states − 1 : 0] new_permvalid [0 : inputs], new_repermvalid [0 : inputs];
368             wire [statebits − 1 : 0] new_k0 [0 : inputs];
369             wire [inputs : 0] new_equal_s, new_improvable_s, new_equal_y, new_improvable_y;
370             assign {new_perm[0], new_reperm[0], new_permvalid[0], new_repermvalid[0],
                        new_k0[0]} = {perm[i + 1], reperm[i + 1], permvalid[i + 1], repermvalid[i + 1],
                        k0[i + 1]};
371             assign {new_equal_s[0], new_improvable_s[0], new_equal_y[0], new_improvable_y
                        [0]} = {equal_s[i + 1], improvable_s[i + 1], equal_y[i + 1], improvable_y[i +
                        1]};
372             for (j = 0; j < inputs; j = j + 1) begin: x
373                 wire [statebits − 1 : 0] oldstate = ‘v(i,reperm[i + 1]);
374                 wire [statebits − 1 : 0] candidate = ‘s(oldstate, j, m[i + 1]);
375                 wire next_permutation = !new_permvalid[j][candidate];
376
377                 assign new_perm[j + 1] = new_perm[j] | (‘size(states ∗ statebits, ({statebits{
                            next_permutation}} & new_k0[j])) << (statebits ∗ (candidate)));
378                 assign new_reperm[j + 1] = new_reperm[j] | (‘size(states ∗ statebits, ({statebits{
                            next_permutation}} & candidate)) << (statebits ∗ (new_k0[j])));
379                 assign new_permvalid[j + 1] = new_permvalid[j] | (‘size(states, next_permutation
                            ) << candidate);
380                 assign new_repermvalid[j + 1] = new_repermvalid[j] | (‘size(states,
                            next_permutation) << new_k0[j]);
381                 assign new_k0[j + 1] = next_permutation? (new_k0[j] + {{statebits − 1{1’b0
                            }},1’b1}) : new_k0[j];
382
383                 wire [statebits − 1 : 0] oris, prms;
384                 assign oris = ‘s(i, j, m[i + 1]);
385                 assign prms = ‘v(candidate, new_perm[j + 1]);
386                 assign new_equal_s[j + 1] = new_equal_s[j] && (oris == prms);
387                 assign new_improvable_s[j + 1] = new_improvable_s[j] || (new_equal_s[j] && (
                            oris > prms));
388                 wire [outputbits − 1 : 0] oriy, prmy;
389                 assign oriy = ‘y(i, j, m[i + 1]);
390                 assign prmy = ‘y(oldstate, j, m[i + 1]);
391                 assign new_equal_y[j + 1] = new_equal_y[j] && (oriy == prmy);
392                 assign new_improvable_y[j + 1] = new_improvable_y[j] || (new_equal_y[j] && (
                            oriy > prmy));
393             end
394             always @(posedge clk or ‘resetedge)
395                 if (‘reset) begin
396                     {perm[i + 2], reperm[i + 2], permvalid[i + 2], repermvalid[i + 2], k0[i + 2],
                                valid[i + 2]} <= 0;
397                     {equal_s[i + 2], improvable_s[i + 2], equal_y[i + 2], improvable_y[i + 2]}
                                <= 4’b0000;
```

```
398                     h1perm[i + 2] <= 0;
399                 end else if (ena) begin
400                     {perm[i + 2], reperm[i + 2], permvalid[i + 2], repermvalid[i + 2], k0[i + 2]}
                            <= {new_perm[inputs], new_reperm[inputs], new_permvalid[inputs],
                            new_repermvalid[inputs], new_k0[inputs]};
401                     valid[i + 2] <= valid[i + 1] && repermvalid[i + 1][i];
402                     {equal_s[i + 2], improvable_s[i + 2], equal_y[i + 2], improvable_y[i + 2]}
                            <= {new_equal_s[inputs], new_improvable_s[inputs], new_equal_y[
                            inputs], new_improvable_y[inputs]};
403                     h1perm[i + 2] <= equal_s[i + 1] && (h1perm[i + 1] < `v(i, reperm[i + 1]))?
                            `v(i, reperm[i + 1]) : h1perm[i + 1];
404                 end
405             end
406             for (i = states + 2; i <= pipelinelength - 2; i = i + 1) begin: copy
407                 always @(posedge clk or `resetedge)
408                     if (`reset)
409                         h1perm[i] <= 0;
410                     else if (ena)
411                         h1perm[i] <= h1perm[i - 1];
412             end
413             always @(posedge clk or `resetedge)
414                 if (`reset) begin
415                     valid[states + 2 : pipelinelength - 2] <= 0;
416                     improvable_s[states + 2 : pipelinelength - 2] <= 0;
417                     startperm[l] <= 0;
418                     {hvalid[3 + l], h1[3 + l], h2[3 + l], h3[3 + l]} <= 0;
419                 end else if (ena) begin
420                     valid[states + 2] <= valid[states + 1] && (improvable_s[states + 1] || (equal_s[
                            states + 1] && improvable_y[states + 1]));
421                     valid[states + 3 : pipelinelength - 2] <= valid[states + 2 : pipelinelength - 3];
422                     improvable_s[states + 2 : pipelinelength - 2] <= improvable_s[states + 1 :
                            pipelinelength - 3];
423                     startperm[l] <= valid[pipelinelength - 2];
424                     `ifdef use_start_state_skip
425                     hvalid[3 + l] <= valid[pipelinelength - 2] && improvable_s[pipelinelength - 2];
426                     h1[3 + l] <= h1perm[pipelinelength - 2];
427                     h2[3 + l] <= inputs - 1;
428                     h3[3 + l] <= `s(h1perm[pipelinelength - 2], inputs - 1, m[pipelinelength - 2]) +
                            1;
429                     `endif
430                 end
431     end
432 endgenerate
433 `else
434 always @(posedge clk or `resetedge)
435     if (`reset)
436         startperm <= 0;
437     else if (ena)
438         startperm <= 0;
439 `endif
440
441 // reduced
442 wire [states - 2 : 0] reducable;
443 reg [states + 3 : pipelinelength - 1] reduced = 0;
444 reg [((states - 1) << statebits) + states - 1 : 0] equclass [2 : states + 2];
445 initial begin: init_unused_equclass
446     integer i;
447     for (i = 2; i <= states + 2; i = i + 1)
448         equclass[i] = 0;
```

```
449 end
450 generate
451     for (i = 0; i < states - 1; i = i + 1) begin: lower_s
452         for (j = i + 1; j < states; j = j + 1) begin: upper_s
453             wire [inputs - 1 : 0] eij;
454             for (k = 0; k < inputs; k = k + 1) begin: each_x
455                 assign eij[k] = `y(i,k,m[1]) == `y(j,k,m[1]);
456             end
457             always @(posedge clk or `resetedge)
458                 if (`reset)
459                     {equclass[2][(i << statebits) + j], equclass[2][(j << statebits) + i]} <= 2'd0;
460                 else if (ena)
461                     {equclass[2][(i << statebits) + j], equclass[2][(j << statebits) + i]} <= {2{&
                            eij}};
462         end
463         always @(posedge clk or `resetedge)
464             if (`reset)
465                 equclass[2][(i << statebits) + i] <= 1'b0;
466             else if (ena)
467                 equclass[2][(i << statebits) + i] <= 1'b1;
468     end
469     always @(posedge clk or `resetedge)
470         if (`reset)
471             equclass[2][((states - 1) << statebits) + states - 1] <= 1'b0;
472         else if (ena)
473             equclass[2][((states - 1) << statebits) + states - 1] <= 1'b1;
474     for (l = 3; l <= states + 2; l = l + 1) begin: reducing
475         for (i = 0; i < states; i = i + 1) begin: lower_s
476             for (j = i + 1; j < states; j = j + 1) begin: upper_s
477                 wire [inputs - 1 : 0] eij;
478                 for (k = 0; k < inputs; k = k + 1) begin: each_x
479                     assign eij[k] = equclass[l - 1][{`s(i,k,m[l-1]), `s(j,k,m[l-1])}];
480                 end
481                 always @(posedge clk or `resetedge)
482                     if (`reset)
483                         {equclass[l][(i << statebits) + j], equclass[l][(j << statebits) + i]} <= 2'
                                b00;
484                     else if (ena)
485                         {equclass[l][(i << statebits) + j], equclass[l][(j << statebits) + i]} <= {2{
                                equclass[l - 1][(i << statebits) + j] && &eij}};
486             end
487             always @(posedge clk or `resetedge)
488                 if (`reset)
489                     equclass[l][(i << statebits) + i] <= 1'b0;
490                 else if (ena)
491                     equclass[l][(i << statebits) + i] <= 1'b1;
492         end
493     end
494     for (i = 0; i < states - 1; i = i + 1) begin: summary
495         assign reducable[i] = |equclass[states + 2][(i << statebits) + states - 1 : (i << statebits) + i
                + 1];
496     end
497     always @(posedge clk or `resetedge)
498         if (`reset)
499             reduced[states + 3] <= 1'b0;
500         else if (ena)
501             reduced[states + 3] <= ~|reducable || (outputs == 1);
502     for (i = states + 4; i < pipelinelength; i = i + 1) begin: reduce_copy
503         always @(posedge clk or `resetedge)
```

```
504                 if (‘reset)
505                     reduced[i] <= 1’b0;
506                 else if (ena)
507                     reduced[i] <= reduced[i − 1];
508     end
509 endgenerate
510
511 // output, x = 0
512 reg constant_output_x0 [7 : pipelinelength − 1];
513 wire [states − 1 : 0] cout;
514 generate
515     for (i = 0; i < states; i = i + 1) begin: cout_s
516         reg [3 : 0] d [1 : 6];
517         reg [3 : 0] a [1 : 6];
518         reg [statebits − 1 : 0] s [1 : 6];
519         always @(posedge clk or ‘resetedge)
520             if (‘reset) begin
521                 d[1] <= 4’b0000;
522                 a[1] <= 4’b0000;
523                 s[1] <= 0;
524             end else if (ena) begin
525                 d[1] <= 4’b0001;
526                 a[1] <= 4’b0000;
527                 s[1] <= i;
528             end
529         for (j = 2; j < 7; j = j + 1) begin: out0
530             always @(posedge clk or ‘resetedge)
531                 if (‘reset) begin
532                     d[j] <= 4’b0000;
533                     a[j] <= 4’b0000;
534                     s[j] <= 0;
535                 end else if (ena) begin
536                     if (‘y(s[j − 1], 0, m[j − 1]))
537                         d[j] <= {d[j − 1][0], d[j − 1][3 : 1]};
538                     else
539                         d[j] <= {d[j − 1][2 : 0], d[j − 1][3]};
540                     a[j] <= a[j − 1] | d[j−1];
541                     s[j] <= ‘s(s[j − 1], 0, m[j − 1]);
542                 end
543         end
544         assign cout[i] = &(a[6] | d[6]);
545     end
546     always @(posedge clk or ‘resetedge)
547         if (‘reset)
548             constant_output_x0[7] <= 1’b0;
549         else if (ena)
550             constant_output_x0[7] <= &cout;
551     for (i = 8; i < pipelinelength; i = i + 1) begin: out0_copy
552         always @(posedge clk or ‘resetedge)
553             if (‘reset)
554                 constant_output_x0[i] <= 1’b0;
555             else if (ena)
556                 constant_output_x0[i] <= constant_output_x0[i − 1];
557     end
558 endgenerate
559
560 // output, x = 1
561 reg alternate_output_x1 [states + 3 : pipelinelength − 1];
562 reg [2 ** (statebits * 2) − 1 : 0] O [2 : states + 2];
```

```
563 ‘define o(i,j,p) O[p][(‘size(2 * statebits, i) << statebits) | (j)]
564 generate
565     for (j = 0; j < states; j = j + 1) begin: targetO
566         for (i = 0; i < states; i = i + 1) begin: sourceO
567             wire [inputs - 1 : 1] conjunction;
568             for (k = 1; k < inputs; k = k + 1) begin: prepareO
569                 assign conjunction[k] = ‘s(i,k,m[1]) == j;
570             end
571             always @(posedge clk or ‘resetedge)
572                 if (‘reset)
573                     ‘o(i,j,2) <= 1’b0;
574                 else if (ena)
575                     ‘o(i,j,2) <= |conjunction || (i == j);
576             for (k = 0; k < states; k = k + 1) begin: connection
577                 always @(posedge clk or ‘resetedge)
578                     if (‘reset)
579                         ‘o(i,k,j + 3) <= 1’b0;
580                     else if (ena)
581                         ‘o(i,k,j + 3) <= ‘o(i,k,j + 2) || (‘o(i,j,j + 2) && ‘o(j,k,j + 2));
582             end
583         end
584     end
585     wire [states - 1 : 0] aout;
586     for (i = 0; i < states; i = i + 1) begin: aout_s
587         wire [1 : 0] out;
588         for (l = 0; l < outputs; l = l + 1) begin: equal_output_selection
589             wire [states - 1 : 0] v;
590             for (j = 0; j < states; j = j + 1) begin: any_state
591                 wire [inputs - 1 : 1] condition;
592                 for (k = 1; k < inputs; k = k + 1) begin: any_input1
593                     assign condition[k] = ‘o(i,j,states + 2) && (‘y(j,k,m[states + 2]) == l);
594                 end
595                 assign v[j] = |condition;
596             end
597             assign out[l] = |v;
598         end
599         assign aout[i] = &out;
600     end
601     always @(posedge clk or ‘resetedge)
602         if (‘reset)
603             alternate_output_x1[states + 3] <= 1’b0;
604         else if (ena)
605             alternate_output_x1[states + 3] <= &aout;
606     for (i = states + 4; i < pipelinelength; i = i + 1) begin: ao_copy
607         always @(posedge clk or ‘resetedge)
608             if (‘reset)
609                 alternate_output_x1[i] <= 1’b0;
610             else if (ena)
611                 alternate_output_x1[i] <= alternate_output_x1[i - 1];
612     end
613 endgenerate
614
615
616 // operation control
617 initial fin = 0;
618 always @(posedge clk or ‘resetedge)
619     if (‘reset)
620         fin <= 0;
621     else if (!active)
```

```
622             fin <= 0;
623         else begin
624             if (ena) begin
625                 fin[1] <= overflow0;
626                 fin[2 : pipelinelength - 1] <= fin[1 : pipelinelength - 2];
627                 fin[pipelinelength] <= fin[pipelinelength] || fin[pipelinelength - 1];
628             end
629             if (issue_queue_ready)
630                 fin[pipelinelength + 1] <= fin[pipelinelength];
631         end
632
633     initial valid = 0;
634     reg valid_result = 0;
635     reg is_out = 0, is_beststart0 = 0, is_duplicate_free = 0;
636     wire base_selection = valid[pipelinelength - 1] && ~|hvalid && reduced[pipelinelength - 1];
637     always @(posedge clk or `resetedge)
638         if (`reset) begin
639             valid <= 0;
640             valid_result <= 0;
641             m[pipelinelength + 1] <= 0;
642         end else if (!active)
643             {valid_result, valid[2 : pipelinelength + 1], valid[1]} <= 'd1;
644         else begin
645             if (ena)
646                 if (use_h)
647                     {valid_result, valid[1 : pipelinelength], is_out, is_beststart0, is_duplicate_free}
                            <= 0;
648                 else begin
649                     valid[1] <= 1'b1;
650                     valid[2] <= valid[1] && !fin[1];
651                     valid[3 : pipelinelength - 1] <= valid[2 : pipelinelength - 2];
652                     valid[pipelinelength] <= valid[pipelinelength - 1] && ~|hvalid && ~|startperm
                            && reduced[pipelinelength - 1] && constant_output_x0[pipelinelength -
                            1] && alternate_output_x1[pipelinelength - 1];
653                     valid_result <= valid[pipelinelength - 1];
654                     is_out <= base_selection && constant_output_x0[pipelinelength - 1] &&
                            alternate_output_x1[pipelinelength - 1];
655                     is_beststart0 <= base_selection && ~|startperm;
656                     is_duplicate_free <= base_selection;
657                 end
658             if (issue_queue_ready) begin
659                 valid[pipelinelength + 1] <= valid[pipelinelength];
660                 m[pipelinelength + 1] <= m[pipelinelength];
661             end
662         end
663
664
665 // counting
666 always @(posedge clk or `resetedge)
667     if (`reset) begin
668         amount <= 0;
669         tests <= 0;
670         hops <= 0;
671         outs <= 0;
672         beststarts <= 0;
673         duplicate_free <= 0;
674         clock_meter <= 0;
675     end else if (ena && !fin[pipelinelength]) begin
676         if (use_h)
```

```
677                 hops <= hops + 1;
678             if (valid_result)
679                 tests <= tests + 1;
680             if (is_out)
681                 outs <= outs + 1;
682             if (is_beststart0)
683                 beststarts <= beststarts + 1;
684             if (is_duplicate_free)
685                 duplicate_free <= duplicate_free + 1;
686             if (valid[pipelinelength])
687                 amount <= amount + 1;
688             clock_meter <= clock_meter + 1;
689         end
690
691
692 // output
693 always @(posedge clk or `resetedge)
694         if (`reset) begin
695             q <= 0;
696             q_valid <= 1'b0;
697             overflow <= 1'b0;
698         end else if (next_automaton || (active && !q_valid)) begin
699             q <= m[pipelinelength + 1];
700             q_valid <= valid[pipelinelength + 1];
701             overflow <= fin[pipelinelength + 1];
702         end
703 assign running = ena;
704
705 endmodule
```

Algorithmus D.6: Simulationshardware gemäß Abschnitt 4.3.3.5

```
1  `define use_normal_clock
2  //`define single_field
3  //`define use_success_improbable
4
5  `define resetsignal clrn
6  `define resetedge negedge clrn
7  `define reset !clrn
8
9  module wocbus(clka, `resetsignal, led, rxd, txd);
10 parameter ObjectsPerArea = 8, areas = 5;
11 parameter I = ObjectsPerArea * areas;
12 parameter states = 3'd4, inputs = 2'd2, outputs = 2'd2;
13 parameter automatons = 2;
14 localparam statebits = 3, outputbits = 1, inputbits = 1;
15 localparam ld_pos = 12 + 1;
16 localparam [ld_pos - 1 : 0] width = 35, height = 5 * width - 4;
17 localparam obj_offset_x = 9, obj_offset_y = 9, obj_step = 16;
18 localparam amountbits = 96;
19 input clka /* synthesis altera_chip_pin_lc="@j3" */;
20 input `resetsignal /* synthesis altera_chip_pin_lc="@c4" */;
21 output [7 : 0] led /* synthesis altera_chip_pin_lc="@p7,@r8,@t8,@t10,@r10,@t11,@r11,@u12"
       */;
22 input rxd /* synthesis altera_chip_pin_lc="@h3" */;
23 output txd /* synthesis altera_chip_pin_lc="@h2" */;
24 wire clk, clock_ready;
25 `ifdef use_normal_clock
26 assign clk = clka;
```

```
27 assign clock_ready = 1'b1;
28 `else
29 clkdata pll1(!clrn, clka, clk, clock_ready);
30 `endif
31
32 wire [7 : 0] tdata;
33 wire [7 : 0] rdata;
34 wire rready, tready;
35 wire tena, tbusy;
36 uart com(clk, clka, !(`reset), 1'b0, rxd, txd, rdata, rready, tdata, tena, tready, tbusy);
37 defparam com.use_tmem = "on", com.use_rmem = "on";
38
39 reg [1 : 0] start = 0;
40 wire starting = start[0] && !start[1];
41 always @(posedge clk or `resetedge)
42     if (`reset)
43         start <= 2'b0;
44     else
45         start <= {start[0], clock_ready};
46
47 reg [amountbits - 1 : 0] generation, amount_tests, amount_success, min_generation,
       amount_min_generation;
48 reg [2 : 0] exec_mode;
49 reg success;
50 wire sim_start, sim_exec, sim_print_result, printing_finished, finished, success_improbable;
51 wire [states * (2 ** inputbits) * (statebits + outputbits) - 1 : 0] automaton [0 : automatons - 1];
52 wire [automatons - 0 : 0] next_automaton;
53 wire [automatons - 1 : 0] overflow, overflow_trigger, automaton_valid, active_automaton;
54 reg [automatons - 1 : 0] overflow_delay;
55 generate genvar e;
56     for (e = 0; e < automatons; e = e + 1) begin: overflow_handle
57         reg overflow_recognized = 1'b0;
58         assign overflow_trigger[e] = overflow[e] && !overflow_recognized;
59         always @(posedge clk or `resetedge)
60             if (`reset)
61                 overflow_recognized <= 1'b0;
62             else if (clock_ready)
63                 overflow_recognized <= overflow[e];
64         automaton #(states, inputs, outputs, statebits, outputbits, inputbits) a(clk, `resetsignal,
               clock_ready, next_automaton[e], overflow[e], automaton[e], automaton_valid[e],
               active_automaton[e]);
65     end
66 endgenerate
67 always @(posedge clk or `resetedge)
68     if (`reset)
69         overflow_delay <= {automatons{1'd0}};
70     else
71         overflow_delay <= overflow_trigger;
72 assign next_automaton[automatons : 1] = overflow_trigger;
73
74 always @(posedge clk or `resetedge)
75     if (`reset)
76         exec_mode <= 3'd0;
77     else case (exec_mode)
78     3'd0: if (starting)
79                 exec_mode <= 3'd2;
80             else if (printing_finished)
81                 exec_mode <= 3'd1;
82     3'd1:  exec_mode <= 3'd2;
```

```
83      3'd2:  if (overflow[automatons - 1])
84                  exec_mode <= 3'd7;
85              else if (&automaton_valid && ~|active_automaton && ~|overflow_trigger && ~|
                    overflow_delay)
86                  exec_mode <= 3'd3;
87      3'd3:  exec_mode <= 3'd4;
88      3'd4:  if (!init)
89                  if (success && |generation)
90                      exec_mode <= 3'd5;
91                  else if ((generation > min_generation + 512) || success_improbable)
92                      exec_mode <= 3'd6;
93      3'd5:  exec_mode <= 3'd0;
94      3'd6:  exec_mode <= 3'd1;
95      3'd7:  if (rready && (rdata == 8'd114))
96                  exec_mode <= 3'd2;
97              else
98                  exec_mode <= 3'd7;
99      default exec_mode <= 3'd0;
100     endcase
101
102 assign next_automaton[0] = exec_mode == 3'd1;
103 assign sim_start = exec_mode == 3'd3;
104 assign sim_exec = exec_mode == 3'd4;
105 assign sim_print_result = exec_mode == 3'd5;
106 assign finished = exec_mode == 3'd7;
107
108 always @(posedge clk or `resetedge)
109     if (`reset)
110         {amount_success, amount_tests} <= 0;
111     else begin
112         if (exec_mode == 3'd5)
113             amount_success <= amount_success + 1;
114         if ((exec_mode == 3'd5) || (exec_mode == 3'd6))
115             amount_tests <= amount_tests + 1;
116     end
117 always @(posedge clk or `resetedge)
118     if (`reset)
119         {min_generation, amount_min_generation} <= {2 * amountbits{1'b0}};
120     else if (starting) begin
121         min_generation <= 100000;
122         amount_min_generation <= {amountbits{1'b0}};
123     end else if (exec_mode == 3'd5)
124         if (generation < min_generation) begin
125             min_generation <= generation;
126             amount_min_generation <= 'd1;
127         end else if (generation == min_generation)
128             amount_min_generation <= amount_min_generation + 'd1;
129
130
131 // bus
132 wire [ld_pos - 1 : 0] pos [0 : I - 1], frontpos [0 : I - 1], bus_pos [areas : 1], bus_frontpos [areas :
        1];
133 reg [ld_pos - 1 : 0] rpos [areas : 1], rfrontpos [areas : 1];
134 wire [0 : I - 1] occupied, col;
135 wire [areas : 1] bus_col;
136 reg [areas : 1] rcol;
137 wire bus_occupied;
138 wire [1 : 0] direction [0 : I - 1], bus_direction;
139 wire [statebits - 1 : 0] state [0 : I - 1], bus_state;
```

```
140 wire [areas : 1] h;
141 wire [0 : I − 1] line_pos [ld_pos − 1 : 0];
142 wire [0 : I − 1] line_frontpos [ld_pos − 1 : 0];
143 wire [0 : I − 1] line_direction [1 : 0], line_state [statebits − 1 : 0];
144 wire [areas : 1] visited;
145
146 reg init = 0, calculating, token_start, ena;
147 reg [1 : 0] delay;
148 reg printing_map, printing_automaton;
149 reg [ld_pos − 1 : 0] mappos, mapcolumn;
150 reg [ld_pos − 5 : 0] init_ra, init_a, init_wa;
151 reg [ld_pos − 5 : 0] watchdog = 0;
152 wire [15 : 0] watch_d;
153 wire [areas : 1] watch_dr [15 : 0];
154 wire init_start, init_end, calc_start, active, last_token, print_after_calculate;
155
156 generate genvar i, j;
157     for (i = 0; i < I; i = i + 1) begin: bus
158         for (j = 0; j < ld_pos; j = j + 1) begin: position
159             assign line_pos[j][i] = pos[i][j];
160             assign line_frontpos[j][i] = frontpos[i][j];
161         end
162         assign line_direction[0][i] = direction[i][0];
163         assign line_direction[1][i] = direction[i][1];
164         for (j = 0; j < statebits; j = j + 1) begin: statebus
165             assign line_state[j][i] = state[i][j];
166         end
167     end
168     assign bus_direction = {|line_direction[1], |line_direction[0]};
169     for (j = 0; j < statebits; j = j + 1) begin: statebus
170         assign bus_state[j] = |line_state[j];
171     end
172     assign bus_occupied = |occupied;
173     for (i = 1; i <= areas; i = i + 1) begin: regional
174         wire [15 : 0] init_d, watch_dmem;
175         wire [ObjectsPerArea : 0] token;
176         assign token[0] = token_start;
177         if (i == areas) begin: last
178             assign last_token = token[ObjectsPerArea];
179         end
180         area5 obstacle(printing_map? mappos : bus_frontpos[i], init_ra, clk,,, 1'b0, 1'b0, h[i],
                init_d);
181         area5 visit(printing_map? mappos : bus_pos[i], init? init_wa : (watchdog − 8'd2), clk, 1'
                b1, init_d, calculating, init, visited[i], watch_dmem);
182         assign bus_col[i] = |col[(i − 1) ∗ ObjectsPerArea +: ObjectsPerArea];
183         for (j = 0; j < ld_pos; j = j + 1) begin: position
184             assign bus_pos[i][j] = |line_pos[j][(i − 1) ∗ ObjectsPerArea +: ObjectsPerArea];
185             assign bus_frontpos[i][j] = |line_frontpos[j][(i − 1) ∗ ObjectsPerArea +:
                    ObjectsPerArea];
186         end
187         always @(posedge clk) begin
188             rpos[i] <= bus_pos[i];
189             rfrontpos[i] <= printing_map? mappos : bus_frontpos[i];
190             rcol[i] <= bus_col[i];
191         end
192         for (j = 0; j <= 15; j = j + 1) begin: bits
193             assign watch_dr[j][i] = watch_dmem[j];
194         end
195         for (j = 0; j < ObjectsPerArea / 4; j = j + 1) begin: creature_connection
```

```
196                 creature #(states, inputs, outputs, statebits, outputbits, inputbits, ld_pos, width, 2'd2, (
                        obj_offset_x + obj_step * j) + (1 * width) + ((i - 1) * (width - 1) * width)) c0(
                        clk, `reset, init, calculating, ena, automaton[0], token[4*j + 0], token[4*j + 1],
                        pos[4*j + 0 + (i - 1) * ObjectsPerArea], frontpos[4*j + 0 + (i - 1) *
                        ObjectsPerArea], h[i], occupied[4*j + 0 + (i - 1) * ObjectsPerArea], state[4*j +
                         0 + (i - 1) * ObjectsPerArea], direction[4*j + 0 + (i - 1) * ObjectsPerArea],
                        rpos[i], rfrontpos[i], rcol[i], col[4*j + 0 + (i - 1) * ObjectsPerArea]);
197                 if (ObjectsPerArea > 1) begin: creatures
198                     creature #(states, inputs, outputs, statebits, outputbits, inputbits, ld_pos, width, 2'
                            d0, (obj_offset_x + obj_step * j) + (33 * width) + ((i - 1) * (width - 1) *
                            width)) c1(clk, `reset, init, calculating, ena, automaton[0], token[4*j + 1],
                            token[4*j + 2], pos[4*j + 1 + (i - 1) * ObjectsPerArea], frontpos[4*j + 1 +
                            (i - 1) * ObjectsPerArea], h[i], occupied[4*j + 1 + (i - 1) *
                            ObjectsPerArea], state[4*j + 1 + (i - 1) * ObjectsPerArea], direction[4*j +
                            1 + (i - 1) * ObjectsPerArea], rpos[i], rfrontpos[i], rcol[i], col[4*j + 1 + (i
                            - 1) * ObjectsPerArea]);
199                 end
200                 if (ObjectsPerArea > 2) begin: multi_creatures
201                     creature #(states, inputs, outputs, statebits, outputbits, inputbits, ld_pos, width, 2'
                            d1, 1 + (obj_offset_y + obj_step * j) * width + ((i - 1) * (width - 1) *
                            width)) c2(clk, `reset, init, calculating, ena, automaton[1], token[4*j + 2],
                            token[4*j + 3], pos[4*j + 2 + (i - 1) * ObjectsPerArea], frontpos[4*j + 2 +
                            (i - 1) * ObjectsPerArea], h[i], occupied[4*j + 2 + (i - 1) *
                            ObjectsPerArea], state[4*j + 2 + (i - 1) * ObjectsPerArea], direction[4*j +
                            2 + (i - 1) * ObjectsPerArea], rpos[i], rfrontpos[i], rcol[i], col[4*j + 2 + (i
                            - 1) * ObjectsPerArea]);
202                     creature #(states, inputs, outputs, statebits, outputbits, inputbits, ld_pos, width, 2'
                            d3, (width - 2) + (obj_offset_y + obj_step * j) * width + ((i - 1) * (width
                            - 1) * width)) c3(clk, `reset, init, calculating, ena, automaton[1], token[4*j
                            + 3], token[4*j + 4], pos[4*j + 3 + (i - 1) * ObjectsPerArea], frontpos[4*j
                            + 3 + (i - 1) * ObjectsPerArea], h[i], occupied[4*j + 3 + (i - 1) *
                            ObjectsPerArea], state[4*j + 3 + (i - 1) * ObjectsPerArea], direction[4*j +
                            3 + (i - 1) * ObjectsPerArea], rpos[i], rfrontpos[i], rcol[i], col[4*j + 3 + (i
                            - 1) * ObjectsPerArea]);
203                 end
204             end
205         end
206         for (i = 0; i <= 15; i = i + 1) begin: bits
207             assign watch_d[i] = |watch_dr[i];
208         end
209     endgenerate
210
211     reg print_request;
212     always @(posedge clk or `resetedge)
213         if (`reset)
214             print_request <= 1'b0;
215         else
216             print_request <= print_request? !print_after_calculate : (starting || (rready && (rdata == 8'
                        d112))); // p
217
218     assign active = init || calculating || printing_map;
219     assign init_start = sim_start;
220     assign init_end = &init_wa;
221     assign calc_start = !active && calc_run;
222     assign calc_run = sim_exec && !success;
223     assign print_after_calculate = ena && (rdata == 8'd109 || rdata == 8'd77 || print_request); // m, M
224     always @(posedge clk or `resetedge)
225         if (`reset)
226             {init, calculating, token_start, delay, ena} <= 5'b00000;
```

```
227     else if (init) begin
228         init <= !init_end;
229         {calculating, token_start} <= {2{init_end && calc_start}};
230         {ena, delay} <= 2'b00;
231     end else begin
232         init <= init_start;
233         calculating <= !init_start && ((calculating && !ena) || calc_start);
234         token_start <= !init_start && calc_start && (!calculating || ena);
235         {ena, delay} <= {delay, last_token};
236     end
237
238 always @(posedge clk or `resetedge)
239     if (`reset)
240         {init_ra, init_a, init_wa} <= 0;
241     else if (!init)
242         {init_ra, init_a, init_wa} <= 0;
243     else begin
244         init_ra <= init_ra + 1'd1;
245         init_a <= init_ra;
246         init_wa <= init_a;
247     end
248
249 always @(posedge clk or `resetedge)
250     if (`reset)
251         generation <= {amountbits{1'b0}};
252     else if (init)
253         generation <= {amountbits{1'b0}};
254     else if (ena)
255         generation <= generation + {{amountbits - 1{1'b0}}, 1'b1};
256
257
258 // control
259 reg watch = 0;
260 wire reset_watchdog;
261 `ifdef use_success_improbable
262 assign reset_watchdog = init;
263 `else
264 assign reset_watchdog = init || !watch;
265 `endif
266 always @(posedge clk or `resetedge)
267     if (`reset)
268         {watch, watchdog, success} <= {1 + ld_pos - 4 + 1{1'b0}};
269     else begin
270         watch <= ((watchdog == 0) || watch) && &watch_d;
271         if (reset_watchdog)
272             watchdog <= 8'd0;
273         else
274             watchdog <= watchdog + {{ld_pos - 5{1'b0}}, 1'b1};
275         if (init || sim_start)
276             success <= 1'b0;
277         else if (&watchdog && watch && &watch_d)
278             success <= 1'b1;
279     end
280
281 `ifdef use_success_improbable
282 wire [4 : 0] watch_single_amount [0 : 1];
283 reg [amountbits - 1 : 0] watch_step_amount [0 : 1], watch_amount [0 : 1], last_watch_amount,
        last_watch_generation;
284 always @(posedge clk or `resetedge)
```

```
285     if (‘reset)
286         {watch_amount[0], watch_amount[1], last_watch_amount, last_watch_generation} <= 0;
287     else if (init)
288         {watch_amount[0], watch_amount[1], last_watch_amount, last_watch_generation} <= 0;
289     else begin
290         watch_step_amount[0] <= watch_single_amount[0] + (~|watchdog? 20’d0 :
                watch_step_amount[0]);
291         if (&watchdog)
292             watch_amount[0] <= watch_step_amount[0] + watch_single_amount[0];
293         watch_step_amount[1] <= watch_single_amount[1] + (~|watchdog? 20’d0 :
                watch_step_amount[1]);
294         if (&watchdog)
295             watch_amount[1] <= watch_step_amount[1] + watch_single_amount[1];
296         if (last_watch_amount < watch_amount[1]) begin
297             last_watch_generation <= generation;
298             last_watch_amount <= watch_amount[1];
299         end
300     end
301 parallel_add #(.msw_subtract("NO"), .pipeline(0), .representation("UNSIGNED"), .
        result_alignment("LSB"), .shift(0), .size(16), .width(1), .widthr(5)) quersumme0(.data(~
        watch_d), .result(watch_single_amount[0]));
302 parallel_add #(.msw_subtract("NO"), .pipeline(0), .representation("UNSIGNED"), .
        result_alignment("LSB"), .shift(0), .size(16), .width(1), .widthr(5)) quersumme1(.data(
        watch_d), .result(watch_single_amount[1]));
303 assign success_improbable = (((generation − last_watch_generation) >> 14) & 1’d1) == 1’b1;
304 ‘else
305 assign success_improbable = 1’b0;
306 ‘endif
307
308
309 // output map
310 reg [15 : 0] mapchars;
311 reg [2 : 0] mapcharselect;
312 reg [amountbits / 4 : 0] gencharpos;
313 wire [7 : 0] statechar, genchar, adrh, adrl;
314 reg [7 : 0] tadata;
315 digit2char sc(statechar, {{4 − statebits{1’b0}}, bus_state}), gc(genchar, generation[amountbits −
        4 − {gencharpos, 2’b00} +: 4]);
316 wire [7 : 0] home [5 : 0];
317 assign {home[0], home[1], home[2], home[3], home[4], home[5]} = 48’h1b5b313b3166;
318 reg [2 : 0] homechar;
319 reg [1 : 0] mapdelay;
320 always @∗
321     if (mapcharselect[1])
322         mapchars <= {8’d13, 8’d10};
323     else if (!mapcharselect[2])
324         mapchars <= {((rdata == 8’d77)? (home[homechar]) : 8’d12), genchar};
325     else if (bus_occupied) begin
326         case (bus_direction)
327         2’b00: mapchars[15 : 8] <= 8’d094; // ^
328         2’b01: mapchars[15 : 8] <= 8’d062; // >
329         2’b10: mapchars[15 : 8] <= 8’d118; // v
330         2’b11: mapchars[15 : 8] <= 8’d060; // <
331         endcase
332         mapchars[7 : 0] <= statechar; // 0−f
333     end else if (h)
334         mapchars <= {8’d91, 8’d93}; // []
335     else
336         mapchars <= {|visited? 8’d32 : 8’d46, 8’d32}; // .
```

```
337 always @(posedge clk or ‘resetedge)
338     if (‘reset)
339         printing_map <= 1’b0;
340     else if (!printing_map)
341         printing_map <= (!active && rready && (rdata == 8’d112)) || print_after_calculate;
342     else if (mappos == height ∗ width − 1 && &mapcharselect && tready && mapdelay[1])
343         printing_map <= 1’b0;
344 wire maxmappos = mapcolumn == (width − 1);
345 always @(posedge clk or ‘resetedge)
346     if (‘reset)
347         {mappos, mapcolumn, mapcharselect, gencharpos, mapdelay, homechar} <= 0;
348     else if (!printing_map)
349         {mappos, mapcolumn, mapcharselect, gencharpos, mapdelay, homechar} <= 0;
350     else if (tready && mapdelay[1]) begin
351         mapdelay <= 2’d0;
352         if (mapcharselect[2]) begin
353             mapcharselect[0] <= !mapcharselect[0];
354             if (mapcharselect[0]) begin
355                 if (maxmappos && !mapcharselect[1])
356                     mapcharselect[1] <= 1’b1;
357                 else begin
358                     mapcharselect[1] <= 1’b0;
359                     mappos <= mappos + {{ld_pos − 1{1’b0}}, 1’b1};
360                     if (maxmappos)
361                         mapcolumn <= {ld_pos{1’b0}};
362                     else
363                         mapcolumn <= mapcolumn + {{ld_pos − 1{1’b0}}, 1’b1};
364                 end
365             end
366             gencharpos <= 0;
367         end else if (mapcharselect[1]) begin
368             mapcharselect[0] <= !mapcharselect[0];
369             if (mapcharselect[0])
370                 mapcharselect[2 : 1] <= 2’b10;
371         end else if (mapcharselect[0]) begin
372             gencharpos <= gencharpos + 1’d1;
373             if (gencharpos == amountbits / 4 − 1)
374                 mapcharselect <= 3’b010;
375         end else if ((rdata != 8’d77) || (homechar == 5)) // M
376             mapcharselect <= 3’b001;
377         else
378             homechar <= homechar + 3’d1;
379     end else
380         mapdelay <= {mapdelay[0], 1’b1};
381
382 // output automaton
383 initial printing_automaton = 1’b0;
384 reg amount_request = 0, amount_printing = 0;
385 reg [statebits − 1 : 0] print_i = 0;
386 reg [inputbits − 1 : 0] print_j = 0;
387 reg [amountbits − 3 : 0] print_k = 0;
388 reg [1 : 0] print_au = 2’d0;
389 reg [2 : 0] print_c = 3’d0;
390 always @(posedge clk or ‘resetedge)
391     if (‘reset) begin
392         printing_automaton <= 1’b0;
393         amount_printing <= 1’b0;
394         print_c <= 0;
395         print_au <= 0;
```

```
396     end else if (printing_automaton) begin
397         printing_automaton <= !(print_c == 3'h7 && tready);
398         if (tready)
399             if (!amount_printing && print_c == 3)
400                 if (print_i < states - 1) begin
401                     print_i <= print_i + 1'd1;
402                     print_c <= 3'd0;
403                 end else begin
404                     print_i <= 0;
405                     if (print_j < inputs - 1) begin
406                         print_j <= print_j + 1'd1;
407                         print_c <= 3'd4;
408                     end else if (print_au < automatons - 1) begin
409                         print_au <= print_au + 2'd1;
410                         print_j <= 0;
411                         print_c <= 3'd4;
412                     end else begin
413                         print_j <= 0;
414                         print_au <= 0;
415                         print_c <= 3'd3;
416                         amount_printing <= 1'b1;
417                     end
418                 end
419             else if (!amount_printing && print_c == 5)
420                 print_c <= 3'd0;
421             else if (amount_printing && (print_c == 0 || print_c == 2 || print_c == 4)) begin
422                 if (print_k == 0) begin
423                     print_c <= print_c + 3'd1;
424                     print_k <= amountbits / 4 - 1;
425                 end else
426                     print_k <= print_k - 1'd1;
427             end else
428                 print_c <= print_c + 3'd1;
429     end else begin
430         printing_automaton <= (rready && (rdata == 8'd80)) || amount_request || (!tbusy &&
                sim_print_result); // P
431         amount_printing <= amount_request;
432         print_i <= 0;
433         print_j <= 0;
434         print_k <= amountbits / 4 - 1;
435         print_c <= 0;
436         print_au <= 0;
437     end
438
439 always @(posedge clk or `resetedge)
440     if (`reset)
441         amount_request <= 1'b0;
442     else if (amount_printing)
443         amount_request <= 1'b0;
444     else if (rready)
445         amount_request <= (rdata == 8'd65) || (rdata == 8'd97); // A, a
446
447 function [statebits + inputbits - 1 : 0] index;
448 input [statebits - 1 : 0] i;
449 input [inputbits - 1 : 0] j;
450 begin
451     index = {i, j};
452 end
453 endfunction
```

```
454
455 wire [7 : 0] print_s, print_y, print_a, print_t, print_h, print_m, print_b, print_d;
456 assign print_s = {4'h3, {4 - statebits{1'b0}}, automaton[print_au][statebits * index(states - 1 -
        print_i, inputs - 1 - print_j) + (states * (2 ** inputbits) * outputbits) +: statebits]};
457 assign print_y = {4'h3, {4 - outputbits{1'b0}}, automaton[print_au][outputbits * index(states - 1
        - print_i, inputs - 1 - print_j) +: outputbits]};
458 digit2char aDigit(print_a, min_generation[{print_k, 2'd0} +: 4]);
459 digit2char tDigit(print_t, amount_min_generation[{print_k, 2'd0} +: 4]);
460 digit2char mDigit(print_m, generation[{print_k, 2'd0} +: 4]);
461 digit2char dDigit(print_d, amount_tests[{print_k, 2'd0} +: 4]);
462 digit2char hDigit(print_h, amount_success[{print_k, 2'd0} +: 4]);
463 digit2char bDigit(print_b, generation[{print_k, 2'd0} +: 4]);
464
465 wire second_amounts = rdata == 65;
466 always @*
467     case (print_c)
468     3'd0: tadata = amount_printing? second_amounts? print_d : print_a : print_s;
469     3'd1: tadata = amount_printing? 8'd32 : overflow[print_au]? 8'd92 : 8'd47;
470     3'd2: tadata = amount_printing? second_amounts? print_h : print_t : print_y;
471     3'd3: tadata = 8'd32;
472     3'd4: tadata = amount_printing? second_amounts? print_b : print_m : (print_j == 0)? 8'd43 :
            8'd45;
473     3'd5: tadata = 8'd32;
474     3'd6: tadata = 8'd13;
475     3'd7: tadata = 8'd10;
476     default tadata = 8'd63;
477     endcase
478
479 // output
480 assign tdata = printing_map? (mapcharselect[0]? mapchars[7 : 0] : mapchars[15 : 8]) : tadata;
481 assign tena = (printing_map && mapdelay[1]) || printing_automaton;
482 assign printing_finished = !printing_automaton && !printing_map;
483 assign led = {!finished, !tbusy, |active_automaton, init || success, calculating, exec_mode};
484
485 endmodule
```

Algorithmus D.7: Einzelne Kreatur für Algorithmus D.6

```
1  module creature(clk, reset, init, calculating, ena, automaton, token_in, token_out, pos_out,
       frontpos_out, h, occupied, state, direction, pos_in, frontpos_in, col_in, col_out);
2  parameter states = 4'd8, inputs = 2'd2, outputs = 2'd2;
3  parameter statebits = 3, outputbits = 1, inputbits = 1;
4  parameter ld_pos = 13;
5  parameter [ld_pos - 1 : 0] width = 35;
6  parameter [1 : 0] init_direction = 0;
7  parameter [ld_pos - 1 : 0] init_position = 1'b0;
8  input clk, reset, init, calculating, ena, token_in, h;
9  output reg token_out;
10 output reg [ld_pos - 1 : 0] pos_out, frontpos_out;
11 output reg occupied;
12 output reg [statebits - 1 : 0] state;
13 output reg [1 : 0] direction;
14 input [ld_pos - 1 : 0] pos_in, frontpos_in;
15 input [states * (2 ** inputbits) * (statebits + outputbits) - 1 : 0] automaton;
16 input col_in;
17 output col_out;
18
19 reg [statebits - 1 : 0] s;
20 reg [1 : 0] d;
```

```
21 reg [ld_pos − 1 : 0] p, fp;
22 reg collision = 0;
23 wire move = !collision;
24
25 function [statebits + inputbits − 1 : 0] index;
26 input [statebits − 1 : 0] i;
27 input [inputbits − 1 : 0] j;
28 begin
29     index = {i, j};
30 end
31 endfunction
32 ‘ifdef Xilinx
33 ‘define sij(ij,p) {p[(ij)+2], p[(ij)+1], p[(ij)]}
34 ‘define yij(ij,p) {p[(ij)]}
35 ‘else
36 ‘define sij(ij,p) p[(ij) +: statebits]
37 ‘define yij(ij,p) p[(ij) +: outputbits]
38 ‘endif
39 localparam alloutputbits = (states ∗ (2 ∗∗ inputbits) ∗ outputbits);
40 ‘define size(s,d) ({s{1’b0}} | (d))
41 ‘define times_statebits(i) (statebits ∗ (i))
42 ‘define times_outputbits(i) (i)
43 ‘define s(i,j) ‘sij(‘times_statebits(index(states − 1 − (i), inputs − 1 − (j))) + alloutputbits,
       automaton)
44 ‘define y(i,j) ‘yij(‘times_outputbits(index(states − 1 − (i), inputs − 1 − (j))), automaton)
45
46 always @(posedge clk)
47     if (frontpos_in == p) begin
48         occupied <= 1’b1;
49         state <= s;
50         direction <= d;
51     end else
52         {occupied, state, direction} <= {1 + statebits + 2{1’b0}};
53
54 reg token, token2;
55 always @(posedge clk) begin
56     token_out <= token_in;
57     token <= token_out;
58     token2 <= token;
59     if (token_in) begin
60         pos_out <= p;
61         frontpos_out <= fp;
62     end else
63         {pos_out, frontpos_out} <= {2 ∗ ld_pos{1’b0}};
64 end
65
66 always @∗
67     case (d)
68     2’b00: fp = p − width;
69     2’b01: fp = p + {{ld_pos − 1{1’b0}}, 1’b1};
70     2’b10: fp = p + width;
71     2’b11: fp = p − {{ld_pos − 1{1’b0}}, 1’b1};
72     endcase
73
74 always @(posedge clk or posedge reset)
75     if (reset)
76         {s, d, p, collision} <= {statebits + 2 + ld_pos + 1{1’b0}};
77     else if (init) begin
78         {s, collision} <= {statebits + 1{1’b0}};
```

```
79              d <= init_direction;
80              p <= init_position;
81          end else if (ena) begin
82              s <= ‘s(s, move);
83              case (‘y(s, move))
84              1’b0: d <= d + 2’b01;
85              1’b1: d <= d + 2’b11;
86              endcase
87              if (move)
88                  p <= fp;
89              collision <= 1’b0;
90          end else if (calculating)
91          if (1’b0)
92              collision <= collision || (token2 && (h || col_in));
93          else
94              collision <= collision || (!token && ((fp == frontpos_in) || (fp == pos_in))) || (token2 && h
                    );
95
96  assign col_out = token? 1’b0 : (frontpos_in == fp) || (frontpos_in == p);
97
98  endmodule
99  ‘undef sij
100 ‘undef yij
101 ‘undef size
102 ‘undef times_statebits
103 ‘undef times_outputbits
104 ‘undef s
105 ‘undef y
```

Algorithmus D.8: Kommunikationsschnittstelle

```
1   module uart(clkdata, clkbaud, clrn, none, rxd, txd, rdata, rready, tdata, tena, tready, tbusy,
        rhandshake);
2   parameter maxbaudclk = 5’d26, ldmaxbaudclk = 5;
3   /* values for baudclk:
4    * 50 MHz / 110 Baud / 16 = 28409,09..., ld = 14,8
5    * 50 MHz / 115200 Baud / 16 = 27,12... => maxbaudclk = 26, ld = 4,8
6    * 25,175 MHz / 110 Baud / 16 = 14303,977272..., ld = 13,8
7    * 25,175 MHz / 115200 Baud / 16 = 13,65..., ld = 3,8
8    */
9   parameter use_tmem = "off";
10  parameter use_rmem = "off";
11  localparam dbits = 4’d8;
12  localparam xbits = dbits + 4’d0 + 4’d1; // 8N1
13
14  input clkdata, clkbaud, clrn, none;
15  input rxd;
16  output txd;
17  output [dbits − 1 : 0] rdata;
18  output rready, tbusy, rhandshake;
19  input [dbits − 1 : 0] tdata;
20  input tena;
21  output tready;
22
23  //−−− Baud clock
24  reg [ldmaxbaudclk − 1 : 0] baudcnt;
25  reg baud16a, baud16b;
26  wire baud16;
27  always @(posedge clkbaud) begin
```

```
28      baud16a <= (baudcnt == 0);
29      baud16b <= baud16a;
30  end
31  assign baud16 = baud16a && !baud16b;
32
33  always @(posedge clkbaud)
34      if (baud16)
35          baudcnt <= maxbaudclk;
36      else
37          baudcnt <= baudcnt - 1'b1;
38
39
40  //--- receive
41  reg rvalid, rready, tbusy, rhandshake;
42  reg [1 : 0] rmode;
43  reg [3 : 0] rbaudcnt, rcnt;
44  reg [xbits - 1 : 0] rxdata;
45  reg [dbits - 1 : 0] rdata;
46  wire rbaud = baud16 && (rbaudcnt == 4'd15);
47
48  always @(posedge clkbaud or negedge clrn)
49      if (!clrn)
50          rmode <= 2'd0;
51      else case (rmode)
52      2'd0: if (!rxd) // wait for start bit
53                  rmode <= 2'd1;
54      2'd1: if (rxd) // check start bit (duration is a half bit)
55                  rmode <= 2'd0;
56              else if (rbaud)
57                  rmode <= 2'd2;
58      2'd2: if (rcnt == 4'd0)
59                  rmode <= 2'd0;
60      default rmode <= 2'd0;
61      endcase
62
63  always @(posedge clkbaud or negedge clrn)
64      if (!clrn)
65          rbaudcnt <= 4'd0;
66      else if (rmode == 2'd0)
67          rbaudcnt <= 4'd7;
68      else if (baud16)
69          rbaudcnt <= rbaudcnt + 4'd1;
70
71  always @(posedge clkbaud or negedge clrn)
72      if (!clrn) begin
73          rxdata <= {xbits{1'b0}};
74          rcnt <= 4'd0;
75      end else if (!rmode[1])
76          rcnt <= xbits;
77      else if (rbaud) begin
78          rxdata <= {rxd, rxdata[xbits - 1 : 1]};
79          rcnt <= rcnt - 4'd1;
80      end
81
82  wire rxvalid = (rmode == 2'd0) && rxdata[xbits - 1];
83  wire rxtrigger = rxvalid && !rvalid;
84  always @(posedge clkbaud or negedge clrn)
85      if (!clrn)
86          rvalid <= 1'b0;
```

```
87      else
88          rvalid <= rxvalid;
89
90  wire [dbits − 1 : 0] rmdata;
91  wire rtrigger;
92  always @(posedge clkdata or negedge clrn)
93      if (!clrn) begin
94          rdata <= 8'd0;
95          rready <= 1'b0;
96          {tbusy, rhandshake} <= 2'b00;
97      end else if (rtrigger) begin
98          rdata <= rmdata;
99          rready <= 1'b1;
100         if (rmdata == 8'd17) begin // xon
101             tbusy <= 1'b0;
102             rhandshake <= 1'b1;
103         end else if (rmdata == 8'd19) begin // xoff
104             tbusy <= 1'b1;
105             rhandshake <= 1'b1;
106         end else
107             rhandshake <= 1'b0;
108     end else
109         rready <= 1'b0;
110
111 generate
112     if (use_rmem == "on") begin: fifo_in
113         wire empty;
114         uartrbuf rmem(!clrn, rxdata[7 : 0], clkdata, rtrigger, clkbaud, rxtrigger, rmdata, empty);
115         assign rtrigger = !empty;
116     end else begin: single_register_in
117         reg [7 : 0] rcdata;
118         reg rctrigger;
119         always @(posedge clkdata or negedge clrn)
120             if (!clrn)
121                 {rcdata, rctrigger} <= 9'h000;
122             else begin
123                 rcdata <= rxdata[7 : 0];
124                 rctrigger <= rxtrigger;
125             end
126         assign rmdata = rcdata;
127         assign rtrigger = rctrigger;
128     end
129 endgenerate
130
131
132 //−−− transmit
133 reg [xbits − 1 : 0] txdata;
134 wire [dbits − 1 : 0] tmdata;
135 reg [3 : 0] tbaudcnt, tcnt;
136 reg tmode;
137 wire tstart;
138
139 assign txd = txdata[0] | !tmode;
140
141 always @(posedge clkbaud or negedge clrn)
142     if (!clrn) begin
143         tbaudcnt <= 4'd0;
144         tcnt <= 4'd0;
145         tmode <= 1'b0;
```

```
146             txdata <= {xbits{1'b0}};
147         end else if (!tmode) begin
148             tbaudcnt <= 4'd0;
149             if (tstart)
150                 txdata <= none? 9'h1f : {tmdata, 1'b0};
151             tcnt <= xbits + 4'd1;
152             tmode <= tstart;
153         end else if (baud16) begin
154             tbaudcnt <= tbaudcnt + 4'd1;
155             if (&tbaudcnt) begin
156                 txdata <= {1'b1, txdata[xbits - 1 : 1]};
157                 tcnt <= tcnt - 4'd1;
158             end
159             tmode <= |tcnt;
160         end
161
162     generate
163         if (use_tmem == "on") begin: fifo_out
164             wire rdreq, rdempty, wrfull;
165             reg tmode2;
166             uarttbuf tmem(!clrn, tdata, clkbaud, rdreq, clkdata, tena && !wrfull, tmdata, rdempty,
                        wrfull);
167             assign tready = !wrfull;
168             assign tstart = !rdempty;
169             assign rdreq = tmode && !tmode2;
170             always @(posedge clkbaud or negedge clrn)
171                 if (!clrn)
172                     tmode2 <= 1'b0;
173                 else
174                     tmode2 <= tmode;
175         end else begin: single_register_out
176             reg tvalid, tvalid2, tcena, tcvalid;
177             reg [7 : 0] tcdata;
178             always @(posedge clkbaud or negedge clrn)
179                 if (!clrn)
180                     {tcdata, tcena, tcvalid} <= 10'd000;
181                 else begin
182                     tcdata <= tdata;
183                     tcena <= tena;
184                     tcvalid <= ~|tcnt;
185                 end
186             assign tstart = tcena;
187             assign tmdata = tcdata;
188             assign tready = tvalid && !tvalid2;
189             always @(posedge clkdata or negedge clrn)
190                 if (!clrn)
191                     {tvalid, tvalid2} <= 2'b00;
192                 else
193                     {tvalid, tvalid2} <= {tcvalid, tvalid};
194         end
195     endgenerate
196
197     endmodule
```

Algorithmus D.9: Umwandlung Hexadezimalziffer zu ASCII-Zeichen

```
1 module digit2char(output reg [7 : 0] out, input [3 : 0] digit);
2
3 always @(digit)
```

```
    case (digit)
    4'd0, 4'd1, 4'd2, 4'd3, 4'd4, 4'd5, 4'd6, 4'd7, 4'd8, 4'd9:
        out <= {4'h3, digit};
    4'ha, 4'hb, 4'hc, 4'hd, 4'he, 4'hf:
        out <= {5'b01100, digit[2 : 0] - 3'd1};
    endcase

endmodule
```

Algorithmus D.10: Zustandsdiagrammgenerator

```
#!/usr/bin/perl
print "%\\newsavebox{\\state}%\n%\\newsavebox{\\blocked}%\n\\unitlength.1in%\n%\\savebox
    {\\blocked}(0,0){\\blacken\\path(-.1,.1)(.1,.1)(.1,-.1)(-.1,-.1)(-.1,.1)}%\n";
while ($line = <>) {
    $line =~ s/[rR]/\/0 /g; $line =~ s/[lL]/\/1 /g; $line =~ s/--/ /g; $line =~ s/ / /g; $line =~ s/[\r\n
        ]+//;
    unless (defined $states) {
        @_ = split /-/, $line; $inputs = int(1 + $#_);
        @_ = split /\//, $line; $states = int($#_ / $inputs);
        $states_inputs = $states * $inputs;
        print STDERR "$states states with $inputs inputs.\n";
        print "\\savebox{\\state}(2," . ($states_inputs + 2) . ")[b]{" . "\\path(0,.6)(0,".
            $states_inputs .")" . "\\spline(0," . $states_inputs . ")(0," . ($states_inputs + 1.4) . ")
            (1," . ($states_inputs + 2) . ")(2," . ($states_inputs + 1.4) . ")(2," . $states_inputs . ")\\
            path(2," . $states_inputs . ")(2,.6)\\spline(2,.6)(2,0)(0,0)(0,.6)}%\n";
    } else {
        print "\\hspace{.1in}\\hfil\n";
    }
    $i = 0;
    foreach $input (split / - /, $line) {
        foreach (split / /, $input) {
            ($state[$i], $out[$i]) = /(\d)\/(\d)/;
            $i ++;
        }
    }

    print "{\\unitlength.1in\\begin{picture}(" . (4 * $states - 2) . "," . ($states_inputs + 2) . ")\n" .
        "\\multiput(0,0)(4,0){$states}{\\usebox{\\state}}\n";
    for ($s = 0; $s < $states; $s ++) {
        print "\\put(" . (4 * $s + 1) . "," . $states_inputs . ".4)" . "{\\makebox(0,0)[b]{$s}}\n";
    }
    for ($s = 0; $s < $states; $s ++) {
        for ($i = 0; $i < $inputs; $i ++) {
            $a = $s + $i * $states;
            $y = ($states - $s) * $inputs - $i;
            $x = 4 * $s;
            $x += 2 if (($s == 0) || ($s < $state[$a]));
            print "{";
            &printcolor($out[$a]);
            if ($s == $state[$a]) { # cycle
                $t = $x;
                if ($s == 0) { $f = $x + 1.5; } else { $f = $x - 1.5; }
                print "\\spline($x,$y)($f,$y)($f," . ($y - .4) . ")($x," . ($y - .4) . ")";
                $y -= .4;
                &printarrow($s == 0);
                $y += .4;
            } else {
                $t = 4 * $state[$a];
```

```
43                      $t += 2 if ($state[$a] < $s);
44                      print "\\path($x,$y)($t,$y)";
45                      &printarrow($x > $t);
46                  }
47                  print "}%\n";
48                  print "\\put($x,$y){\\usebox{\\blocked}}%\n" if ($i == 0);
49                  print "\\put($x,$y){\\circle*{.3}}%\n" if ($i == 2);
50                  print "\\put($x,$y){\\circle{.3}}%\n" if ($i == 3);
51              }
52      }
53      print "\\end{picture}}% $line\n";
54  }
55
56  sub printcolor
57  {
58      print("\\color{green}"), return if ($_[0] == 0);
59      print("\\color{red}"), return if ($_[0] == 1);
60      print("\\color{blue}"), return if ($_[0] == 2);
61      print("\\color{cyan}"), return if ($_[0] == 3);
62      print("\\color{magenta}"), return if ($_[0] == 4);
63      print("\\color[cmyk]{0,.75,.75,.35}"), return if ($_[0] == 5);
64      print("\\color[cmyk]{0,.2,1,0}"), return if ($_[0] == 6);
65      print("\\color[cmyk]{.32,0,.82,0}"), return if ($_[0] == 7);
66      print("\\color[cmyk]{0,.34,1,0}"), return if ($_[0] == 8);
67  }
68
69  sub printarrow
70  {
71      $f = $t + ($_[0]? +.6 : -.6);
72      if (($out[$a] % 3 == 0) && ($out[$a] != 9)) {
73          print "\\path($f," . ($y - .2) . ")($t,$y)($f," . ($y + .2) . ")";
74      } elsif ($out[$a] % 3 == 1) {
75          print "{\\allinethickness{0cm}\\blacken";
76          &printcolor($out[$a]);
77          print "\\path($f," . ($y + .2) . ")($t,$y)($f," . ($y - .2) . ")($f," . ($y + .2) . ")}";
78          print "\\path($f," . ($y + .2) . ")($t,$y)($f," . ($y - .2) . ")($f," . ($y + .2) . ")";
79      } else { # ($out[$a] % 3 == 2)
80          print "\\path($f," . ($y + .2) . ")($t,$y)($f," . ($y - .2) . ")($f," . ($y + .2) . ")";
81      }
82      return if ($out[$a] == 9);
83      for ($fi = int($out[$a] / 3); $fi > 0; $fi --) {
84          $f += ($_[0]? +.2 : -.2);
85          print "\\path($f," . ($y + .2) . ")($f," . ($y - .2) . ")";
86      }
87  }
```

Anhang E

Anmerkungen zur Automatenaufzählung

Im Folgenden sind veranschaulichende Darstellungen zu den im Kapitel 3 behandelten Algorithmen aufgeführt, die einen weiteren Einblick in deren Funktionsweise bieten.

Die Funktionsparameter und -rückgabewerte der einzelnen Algorithmen zur Aufzählung von Automaten sind:

InitiateAutomaton (Algorithmus 3.2, Seite 26)
Veränderte Variablen: s', y.

NextAutomaton (Algorithmus 3.3, Seite 27)
Voraussetzung: s', y.
Veränderte Variablen: s', y (nächster Automat).
Rückgabe: falsch, wenn es einen Überlauf gab, die Aufzählung also von vorne beginnt.

PreviouslyEnumerated (Algorithmus 3.4, Seite 28)
Voraussetzung: $\dot{s}'$, $\dot{y}$, $\ddot{s}'$, $\ddot{y}$.
Rückgabe: wahr, wenn $\dot{s}'$,$\dot{y}$ nach $\ddot{s}'$,$\ddot{y}$ aufgezählt wurde.

OrdinalNumber (Algorithmus 3.5, Seite 28)
Voraussetzung: s', y.
Rückgabe: Ordinalzahl.

MapToNormalized (Algorithmus 3.6, Seite 33)
Voraussetzung: s', y.
Veränderte Variablen: $\mathbf{s}'$, **y** (normierter Automat).

Reducable (Algorithmus 3.7, Seite 37)
Voraussetzung: s', y.
Veränderte Variablen: s', y (reduziert).
Rückgabe: wahr, wenn Automat ist reduzierbar.

PrefixFree (Algorithmus 3.8, Seite 38)
Voraussetzung: s', y.
Veränderte Variable: reflexive transitive Hülle A.
Rückgabe: wahr, wenn Automat ist präfixfrei.

IsomorphismTest (Algorithmus 3.9, Seite 40)
Voraussetzung: s', y.
Veränderte Variablen: $\mathbf{s}'$, $\mathbf{y}$ (ein eher aufgezählter Automat, isomorph zu s', y).
Rückgabe: wahr, wenn keine Automatenpermutation eher aufgezählt.

NextValidAutomaton (Algorithmus 3.12, Seite 47)
Voraussetzung: s', y.
Veränderte Variable: s', y (nächster relevanter Automat), h (verwendeter Sprung).
Rückgabe: falsch, wenn es einen Überlauf gab, die Aufzählung also von vorne beginnt.

Validate (Algorithmus 3.13, Seite 48)
Voraussetzung: s', y.
Veränderte Variable: h (durch aufgerufene Funktionen).
Rückgabe: wahr, wenn Automat relevant ist.

Normalized (Algorithmus 3.15, Seite 51)
Voraussetzung: s', y.
Veränderte Variable: h.
Rückgabe: normiert.

Reachable_CA (Algorithmus 3.16, Seite 53)
Voraussetzung: s'.
Rückgabe: wahr, wenn alle Zustände des Automaten erreichbar.

PartlyConnected (Algorithmus 3.19, Seite 56)
Voraussetzung: s', y.
Veränderte Variable: h.
Rückgabe: teilzusammenhängender Automat.

SortedStates1 (Algorithmus 3.22, Seite 58)
Voraussetzung: s', y.
Veränderte Variable: h.
Rückgabe: Zustände nicht permutiert (arrangiert)

SortedStates (Algorithmus 3.23, Seite 60)
Parameter: η (h veränderbar).
Voraussetzung: s', y.
Veränderte Variable: h.
Rückgabe: Zustände unter Beachtung von Start- und Terminalzuständen nicht permutiert (arrangiert).

SetScopePermutation (Algorithmus 3.24, Seite 61)
Voraussetzung: Zustand i, Zählerarray k.
Veränderte Variablen: Mengenzugehörigkeit g, veränderte Zustandspermutation σ_i, k.

ImprovableFirstState (Algorithmus 3.25, Seite 62)
Voraussetzung: s', y.
Rückgabe: anderer Startzustand als erster Zustand besser.

PermutationTest (3.26, 63)
Voraussetzung: s', σ^{-1}.
Veränderte Variable: h.
Rückgabe: wahr, wenn Permutation von s' eher aufgezählt.

ImprovableTerminalStartState (Algorithmus 3.27, Seite 64)
Voraussetzung: s', y.
Rückgabe: wahr, wenn der übergebene Automat mit Terminalzuständen einen anderen Repräsentanten hat.

ImprovableWithInputPermutation (Algorithmus 3.28, Seite 66)
Voraussetzung: s', y.
Veränderte Variable: h.
Rückgabe: wahr, wenn ein Automat mit anderer Eingangswertpermutation eher aufgezählt wurde; eine andere Permutation wird nicht überprüft.

OutputPermutationTest (Algorithmus 3.29, Seite 66)
Voraussetzung: s', y, σ, χ.
Veränderte Variable: υ.
Rückgabe: wahr, wenn ein Automat mit anderer Ausgabewertpermutation eher aufgezählt wurde.

PermutationRepresentative (Algorithmus 3.30, Seite 67)
Voraussetzung: s', y.
Rückgabe: wahr, wenn Automat bezüglich Permutation von Ein-, Ausgabewerte und Zustände ein Repräsentant ist.

OutputSemantic0 (Algorithmus 3.33, Seite 69)
Voraussetzung: s', y.
Rückgabe: wahr, wenn der Automat bei $x = 0$ einen gleich bleibenden Ausgabezyklus aufweist.

OutputSemantic1 (Algorithmus 3.34, Seite 70)
Voraussetzung: s', y.
Veränderte Variable: O.
Rückgabe: wahr, wenn der Automat bei $x = 1$ einen veränderlichen Ausgabezyklus aufweist.

InputSemantic (Algorithmus 3.35, Seite 70)
Voraussetzung: s', y.
Rückgabe: wahr, wenn der Automat alle Eingangswerte nutzt.

Quellenverzeichnis

[Art91] Michael Artin: *Algebra*. Birkhäuser, 1991.

[Ash59] Robert L. Ashenhurst: *The Decomposition of Switching Functions*. Annals of the Harvard Computation Laboratory, 29:74–116, 1959.

[ASU88] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman: *Compilerbau*, Seite 172f. Addison-Wesley, 1988.

[BA00] Victor J. Blue and J. L. Adler: *Cellular Automata Model Of Emergent Collective Bi-Directional Pedestrian Dynamics*. In Mark A. Bedau, John S. McCaskill, Norman H. Packard, and Steen Rasmussen (editors): *Artificial Life VII: The Seventh International Conference on the Simulation and Synthesis of Living Systems*, Reed College, Portland, Oregon, August 2000.

[Bal67] Robert Balzer: *An 8-State Minimal Time Solution to the Firing Squad Synchronization Problem*. In Murray Eden (editor): *Information and Control*, volume 10(1), pages 22 – 42. Academic Press, January 1967.

[Bas87] Pierre Basieux: *Roulette: Die Zähmung des Zufalls*. printul, 1987.

[Bas05] Frédérique Bassino: *Automates, Énumération et Algorithmes*. Mémoire d'habilitation à diriger des recherches, Spécialité Informatique, Université de Marne-la-Vallée, décembre 2005.

[BBW99] Wolfgang Bauer, Walter Benenson und Gary Westfall: *cliXX Physik*, Kapitel 9. Verlag Harri Deutsch, 1999. `http://www.harri-deutsch.de/cgi-bin/start?1593`.

[Ber66] A. J. Bernstein: *Analysis of Programs for Parallel Processing*. IEEE Transactions on Electronic Computers, EC-15(5):757–763, October 1966.

[BHM85] Robert K. Brayton, Gary D. Hachtel, and Curtis T. McMullen: *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1985.

[BHMSV84] Robert Brayton, G.D. Hachtel, C.T. McMullen, and Alberto Sangiovanni-Vincentelli: *EXPRESSO-II: A New Logic Minimizer for Programmable Logic Arrays*. In Wayne C. Luplow (editor): *IEEE Transactions on Consumer Electronics*, volume CE-30(2), pages 370 – 376, May 1984.

[BK99] Matthias Böge and Andreas Koch: *A Processor for Artificial Life Simulation*. In Patrick Lysaght, James Irvine, and Reiner W. Hartenstein (editors): *Field Programmable Logic and Applications: 9th International Workshop, FPL'99, Glasgow, UK, August 30 - September 1, 1999. Proceedings*, volume 1673 of *LNCS*, pages 495 – 500, Glasgow (Scotland), September 1999. Springer.

[BLW76] Norman L. Biggs, E. Keith Lloyd, and Robin J. Willson: *Graph Theory 1736-1936*. Clarendon Press, Oxford, 1976.

[BN06a] Frédérique Bassino and Cyril Nicaud: *Accessible and Deterministic Automata: Enumeration and Boltzmann Samplers*. In *DMTCS Proceedings: Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities*, pages 151 – 160, September 2006.

[BN06b] Frédérique Bassino and Cyril Nicaud: *Enumeration and random generation of accessible automata*. Internet, April 2006. `http://www-igm.univ-mlv.fr/~bassino/publications/tcs06.ps`.

[Bot06] Lars Both: *Entwurf eines Framework zur Zellularverarbeitung und die Erzeugung von Zustandsalgorithmen für intelligente Roboter*. Diplomarbeit, TU Darmstadt, April 2006.

[BS68] Robert G. Busacker und Thomas L. Saaty: *Endliche Graphen und Netzwerke: Eine Einführung mit Anwendungen*. R. Oldenbourg Verlag, 1968.

[BSMM00] Ilja N. Bronstein, Konstantin A. Semendjajew, Gerhard Musiol und Heiner Mühlig: *Taschenbuch der Mathematik*. Verlag Harri Deutsch AG, Dresden, 5. Auflage, 2000.

[BZ06] Michael Bader and Christoph Zenger: *A Cache Oblivious Algorithm for Matrix Multiplication Based on Peano's Space Filling Curve*. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wásniewski (editors): *Parallel Processing and Applied Mathematics*, volume 3911/2006 of *Lecture Notes in Computer Science*, pages 1042 – 1049. Springer, 2006.

[Cay75] E. Cayley: *Ueber die analytischen Figuren, welche in der Mathematik Bäume genannt werden und ihre Anwendung auf die Theorie chemischer Verbindungen*. In: H. Wichelhaus (Herausgeber): *Berichte der Deutschen Chemischen Gesellschaft zu Berlin*, Band 8, Seiten 1056 – 1059. Verlag Chemie, Weinheim/Bergstraße, 1875.

[CBN05] Zdenek Cerman, Wilhelm Barthlott und Jürgen Nieder: *Erfindungen der Natur: Bionik – Was wir von Pflanzen und Tieren lernen können*. Rowohlt Taschenbuch Verlag, 2005.

[CKFL05] Iain D. Couzin, Jens Krause, Nigel R. Franks, and Simon A. Levin: *Effective leadership and decision-making in animal groups on the move*. Nature, 433:513–516, February 2005.

[Con71] John Horton Conway: *Regular Algebra and Finite Machines*. Chapman and Hall Ltd., London, 1971.

[Deo74] Narsingh Deo: *Graph Theory with Applications to Engineering and Computer Science*, chapter 10. Prentice-Hall, 1974.

[DH96] Philip J. Davis und Reuben Hersh: *Erfahrung Mathematik*. Birkhäuser Verlag, 1996.

[DKS02] M. Domaratzki, D. Kisman, and J. Shallit: *On The Number Of Distinct Languages Accepted By Finite Automata With n States*. Journal of Automata, Languages and Combinatorics, 7(4):496–486, September 2002.

[Dom03] Michael Domaratzki: *On Enumeration of Müller Automata*. LNCS, 2710:254–265, 2003.

[Dro93] Alexis Drogoul: *De la simulation multi-agents à la résolution collective de problèmes*. Thèse de doctorat, Université Paris VI, novembre 1993. Une Étude De l'Émergence De Structures D'Organisation Dans Les Systèmes Multi-Agents.

[DS04] Marco Dorigo and Thomas Stützle: *Ant Colony Optimization*. MIT Press, 2004.

[DSL06] Bruno N. Di Stefano and Anna T. Lawniczak: *Autonomous Roving Object's Coverage of its Universe*. In *Canadian Conference on Electrical and Computer Engineering*, pages 1591 – 1594, May 2006.

[ECS88] Hermann Engesser, Volker Claus und Andreas Schwill (Herausgeber): *Duden Informatik*. Bibliograpisches Institut Wissenschaftsverlag, 1. Auflage, 1988.

[EHH08a] Patrick Ediger, Rolf Hoffmann, and Mathias Halbach: *How efficient are creatures with time-shuffled behaviors?* In *9th Workshop on Parallel Systems and Algorithms (PASA 2008) in Conjunction with 21st International Conference on Architecture of Computing Systems (ARCS)*, February 2008.

[EHH08b] Patrick Ediger, Rolf Hoffmann, and Mathias Halbach: *Is a non-uniform system of creatures more efficient than a uniform one?* In *The 11th International Workshop on Nature Inspired Distributed Computing (NIDISC'08) held in conjunction with The 22th IEEE/ACM International Parallel and Distributed Processing*. To appear, April 2008.

[EM92] Horst Elschner und Albrecht Möschwitzer: *Einführung in die Elektrotechnik – Elektronik*. Verlag Technik GmbH Berlin, 3. Auflage, 1992.

[Fli01] Thomas Flik: *Mikroprozessortechnik: CISC, RISC, Systemaufbau, Assembler und C*. Springer, 6. Auflage, 2001.

[Foc06] Axel Focke: *Regionale Leistungs- und Krankenhausplanung: Ein Simulationsmodell auf Basis eines Ameisenalgorithmus*, Kapitel 6, Seiten 151 – 211. Deutscher Universitäts-Verlag, 2006. Dissertation, Universität Duisburg-Essen.

[Fri03] FriendlyRobotics, www.friendlyrobotics.com: *Betriebs- & Sicherheitshandbuch Robomow*, 2003. DOC0063A.

[FS05a] Dietmar Fey and Daniel Schmidt: *Marching-Pixels: A New Organic Computing Paradigm for Smart Sensor Processor Arrays*. In *Proceedings ACM International Conference on Computing Frontiers 2005*, pages 1 – 7, Ischia, Italy, May 2005. ACM.

[FS05b] Dietmar Fey and Daniel Schmidt: *Marching Pixels: A new organic computing principle for smart CMOS camera chips*. In *Workshop on Self-Organization and Emergence — Organic Computing and its Neighboring Disciplines*, pages 123 – 130, Innsbruck, Austria, March 2005.

[Gar70] Martin Gardner: *The Fantastic Combinations of John Conway's New Solitaire Game "Life"*. Scientific American, 223(4):120–123, 1970.

[GB89] B. Gurunath and Nripendra N. Biswas: *An algorithm for multiple output minimization*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 8(9):1007–1013, September 1989.

[Gin62] Seymour Ginsburg: *An Introduction to Mathematical Machine Theory*. Addison-Wesley, 1962.

[GJ79] Michael R. Garey and David S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co Ltd., April 1979.

[GL73] Wolfgang K. Giloi und Hans Liebig: *Logischer Entwurf digitaler Systeme*, Kapitel 2.6. Springer, 1973.

[Glu61] В. М. Глушков: *Некоторые проблемы синтеза цифровых автоматов*. Вычисл. Матем. и матем. физики, 1(3):371–411, 1961.

[Glu63] Viktor Michailowitsch Gluschkow: *Die Theorie der abstrakten Automaten*. VEB Deutscher Verlag der Wissenschaft, Berlin, 1963. Übersetzung von [Glu61].

[Goo98] Gerhard Goos: *Vorlesung über Informatik, Band 4: Paralleles Rechnen und nicht-analytische Lösungsverfahren*. Springer, 1998.

[HAB+02] Tien Ruey Hsiang, Esther M. Arkin, Michael A. Bender, Sándor P. Fekete, and Joseph S. B. Mitchell: *Algorithms for Rapidly Dispersing Robot Swarms in Unknown Environments*, December 2002. In 5th International Workshop on Algorithmic Foundations of Robotics.

[Har57] Frank Harary: *The Number of Oriented Graphs*. Michigan Mathematical Journal, 4(3):221–224, 1957.

[Har60] Frank Harary: *Unsolved problems in the enumeration of graphs*. A Magyar Tudományos Akadámia Matematikai Kutató Intézetének közleményei (Publications of the Mathematical Institute of the Hungarian Academy of Science), 5:63–95, 1960. Budapest.

[Har65] M. A. Harrison: *A census of finite automata*. Canadian journal of mathematics, 17:100–113, 1965.

[Har69] Frank Harary: *Graph Theory*. Mathematics. Addison-Wesley, 1969.

[Har06] Reiner Hartenstein: *Reconfigurable Supercomputing: Hurdles and Chances*. In *International Supercomputer Conference (ICS 2006)*, June 2006.

[HCS79] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate: *Computing Connected Components on Parallel Computers*. Communications of the ACM, 22(8):461–464, August 1979.

[Hee07] Wolfgang Heenes: *Entwurf und Realisierung von massivparallelen Architekturen für Globale Zellulare Automaten*. Dissertation, TU Darmstadt, 2007.

[HEW05] Jan Haase, Frank Eschmann und Klaus Waldschmidt: *Die selbstverteilende virtuelle Maschine SDVM*. it – Information Technology, 47(3):132–139, 2005.

[HFV00] Dirk Helbing, Illés Farkas, and Tamas Vicsek: *Simulating dynamical features of escape panic*. Nature, 407:487–490, 2000.

[HH68] M. H. Harrison and R. G. High: *On the cycle index of a product of permutation groups*. Journal of combinatorial theory, 4:277–299, 1968.

[HH04] Mathias Halbach and Rolf Hoffmann: *Implementing Cellular Automata in FPGA Logic*. In *International Parallel & Distributed Processing Symposium (IPDPS), Workshop on Massively Parallel Processing (WMPP)*, page 258. IEEE Computer Society, April 2004.

[HH05a] Mathias Halbach and Rolf Hoffmann: *Optimal Behavior of a Moving Creature in the Cellular Automata Model*. In Victor Malyshkin (editor): *Parallel Computing Technologies*, number 3606 in *LNCS*, pages 129 – 140, Krasnoyarsk, September 2005. Springer.

[HH05b] Mathias Halbach and Rolf Hoffmann: *Optimizing the Behaviour of a Moving Creature in a CA Field*. In *11th Workshop on Cellular Automata*, Gdansk, Poland, September 2005.

[HH06a] Mathias Halbach and Rolf Hoffmann: *Minimising the Hardware Resources for a Cellular Automaton with Moving Creatures*. In Wolfgang Karl, Jürgen Becker, Karl Erwin Großpietsch, Christian Hochberger, and

Erik Maehle (editors): *ARCS'06. 19th International Conference on Architecture of Computing Systems, Workshop Proceedings. Lecture Notes in Informatics (LNI)*, volume P-81, pages 323 – 332, Frankfurt, Germany, March 2006.

[HH06b] Rolf Hoffmann and Mathias Halbach: *Are several creatures more efficient than a single one?* In S. El Yacoubi, B. Chopard, and S. Bandini (editors): *Seventh International Conference on Cellular Automata for Research and Industry (ACRI 2006)*, number 4173 in *LNCS*, pages 707 – 711, 2006. Workshop Crowds and Cellular Automata (C&CA) at ACRI 2006.

[HH07a] Mathias Halbach and Rolf Hoffmann: *Parallel Hardware Architecture to Simulate Movable Creatures in the CA Model*. In *9th International Conference on Parallel Computing Technologies (PaCT-2007)*, volume 4671 of *LNCS*. Springer, September 2007.

[HH07b] Mathias Halbach and Rolf Hoffmann: *Solving the exploration's problem with several creatures more efficiently*. In *Eleventh International Conference on Computer Aided Systems Theory*, Instituto Universitario de la Ciencia y la Tecnología Cibernéticas, E-35017 Las Palmas de Gran Canaria, Spain, February 2007.

[HH07c] Mathias Halbach and Rolf Hoffmann: *Solving the exploration's problem with several creatures more efficiently*. In R. Moreno-Díaz *et al.* (editors): *EUROCAST 2007*, volume 4739 of *LNCS*, pages 596 – 603. Springer, 2007.

[HHB06] Mathias Halbach, Rolf Hoffmann, and Lars Both: *Optimal 6-State Algorithms for the Behavior of Several Moving Creatures*. In S. El Yacoubi, B. Chopard, and S. Bandini (editors): *Seventh International conference on Cellular Automata for Research and Industry (ACRI 2006)*, number 4173 in *LNCS*, pages 571 – 581. ACRI, 2006.

[HHH04] Rolf Hoffmann, Wolfgang Heenes, and Mathias Halbach: *Implementation of the Massively Parallel Model GCA*. In *Parallel Computing in Electrical Engineering (PARELEC), Parallel System Architectures*, pages 135 – 139, September 2004.

[HHHT04] Mathias Halbach, Wolfgang Heenes, Rolf Hoffmann, and Jan Tisje: *Optimizing the Behavior of a Moving Creature in Software and in Hardware*. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI 2004)*, number 3305 in *LNCS*, pages 841 – 850, October 2004.

[HHR04] Mathias Halbach, Rolf Hoffmann, and Patrick Röder: *FPGA Implementation of Cellular Automata Compared to Software Implementation*. In Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, and other (editors): *ARCS 2004, Organic and Pervasive Computing, Workshop Proceedings*, volume P-41 of *Lecture Notes in Informatics*, pages 309 – 317, Augsburg, Germany, March 2004. GI-Edition.

[Hie73] Carl Hierholzer: *Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren*. Mathematische Annalen, 6:30–32, 1873.

[Hil91] David Hilbert: *Ueber die stetige Abbildung einer Linie auf ein Flächenstück*. In: *Mathematische Annalen*, Band 38(3), Seiten 459 – 460. Springer, September 1891.

[HKT03] Christian Haubelt, Dirk Koch, and Jürgen Teich: *ReCoNet: Modeling and Implementation of Fault Tolerant Distributed Reconfigurable Hardware*. In *Proceedings of 16th Symposium on Integrated Circuits and Systems Design (SBCCI2003)*, pages 343 – 348, São Paulo, Brazil, September 2003.

[HMU02] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium, München, 2. Auflage, 2002.

[Hoc98] Christian Hochberger: *CDL – Eine Sprache für die Zellularverarbeitung auf verschiedenen Zielplattformen*. Dissertation, TU Darmstadt, 1998.

[Hof74] Rolf Hoffmann: *Algorithmen mit booleschen Matrizen in der Schaltwerkstheorie*. Dissertation, Technische Universität Berlin, 1974.

[Hof93] Rolf Hoffmann: *Rechnerentwurf: Rechenwerke, Mikroprogrammierung, RISC*. Oldenbourg, 3. Auflage, 1993.

[Hom04] Prof. Dr. Günter Hommel: *An Autonomously Operating Flying Robot*. Internet, July 2004. `http://pdv.cs.tu-berlin.de/MARVIN/`.

[Hop71] John Hopcroft: *An n log n algorithm for minimizing states in a finite automaton*. Technical Report STAN-CS-71-190, Computer Science Department, School of Humanities and Sciences, Stanford University, January 1971.

[HP67] Frank Harary and Ed Palmer: *Enumeration of finite automata*. Information and Control, 10(5):499–508, May 1967.

[HP73] Frank Harary and Edgar M. Palmer: *Graphical Enumeration*. Academic Press, New York, 1973.

[HR99] Matthew M. Huntbach and Gream A. Ringwood: *Agent-Oriented Programming: From Prolog to Guarded Definite Clauses*. Number 1630 in *Lecture Notes in Artificial Intelligence*. Springer, 1999.

[HS03] Rolf Hoffmann and Smitha Sambharaju: *Optimal Algorithm for Checking the Environment by a Moving Creature*. Report RA-1-2003, Technische Universität Darmstadt, March 2003. `http://www.ra.informatik.tu-darmstadt.de/publikationen/Report1-03.pdf`.

[HU79] John E. Hopcroft and Jeffrey D. Ullman: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Huf54] D. A. Huffman: *The synthesis of sequential switching circuits*. Journal of the Franklin Institute, 257:(3)161–190, (4)275–303, 1954.

[HUVW00] Rolf Hoffmann, Bernd Ulmann, Klaus Peter Völkmann, and Stefan Waldschmidt: *A Stream Processor Architecture Based on the Configurable CEPRA-S*. In R. W. Hartenstein and H. Grünbacher (editors): *Field-programmable Logic: The Roadmap to Reconfigurable Systems (FPL 2000)*, number 1896 in *LNCS*, pages 822 – 825, Villach, Austria, August 2000. Springer Verlag.

[HUWV01] Rolf Hoffmann, Bernd Ulmann, Stefan Waldschmidt, and Klaus Peter Völkmann: *The Machine CEPRA-S Configured for Stream Processing*. In *9th International Symposium on Field Programmable Gate Arrays (FPGA 2001)*, page 226, Monterey, California, February 2001.

[HVS94] Rolf Hoffmann, Klaus Peter Völkmann, and Marek Sobolewski: *The Cellular Processing Machine CEPRA-8L*. In Chris Jesshope, Jossifov, and Wilhelmi (editors): *Parcella '94: Proceedings of the VI. International Workshop on Parallel Processing by Cellular Automata and Arrays, held in Potsdam, September, 21-23, 1994*, volume 81 of *Mathematical Research*, pages 179 – 188. Akademie Verlag, September 1994.

[Ins01] The Institute of Electrical and Electronics Engineers, Inc.: *IEEE Standard Verilog® Hardware Description Language*, IEEE Std 1364-2001 Version C edition, March 2001.

[Ive62] Kenneth E. Iverson: *A Programming Language*. John Wiley and Sons, Inc., 1962.

[JK81] Gordon James and Adalbert Kerber: *The Representation Theory of the Symmetric Group*, volume 16 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, 1981.

[Jon03] Willie D. Jones: *Hot-Wired Housekeeper*. IEEE Spectrum, 40(11):45, November 2003.

[Kam98] Peter Kammerer: *Von Pascal zu Assembler*. Vieweg, 1998.

[Kap65a] Yu.V. Kapitonova: *On an isomorphism of abstract automata I*. Cybernetics and Systems Analysis, 1(3):28–31, May 1965. Springer New York.

[Kap65b] Yu.V. Kapitonova: *On an isomorphism of abstract automata II*. Cybernetics and Systems Analysis, 1(5):10–13, September 1965. Springer New York.

[Kar53] Maurice Karnaugh: *The Map Method for Synthesis of Combinational Logic Circuits*. In *Transactions of the American Institute of Electrical Engineers on Communications and Electronics*, volume 72(9), part I, pages 593 – 599, November 1953.

[Kar72] Richard M. Karp: *Reducibility among combinatorial problems*. In Raymond E. Miller and James W. Thatcher (editors): *Proceedings of a Symposium on the Complexity of Computer Computations (IBM Thomas J. Watson Research Center, Yorktown Heights, New York)*, pages 85 – 103, New York, March 1972. The IBM Research Symposia Series, Plenum Press.

[KF07] Marcus Komann and Dietmar Fey: *Realising emergent image preprocessing tasks in cellular-automaton-alike massively parallel hardware*. International Journal on Parallel, Emergent and Distributed Systems, 22:79–89, April 2007.

[Kla06] Erica Klarreich: *The Mind of the Swarm. Math explains how group behavoir is more than the sum of its parts*. Science News, 170(22):347, November 2006.

[Kle56] S.C. Kleene: *Representation of events in nerve-nets and finite automata*. In Claude Elwood Shannon and John McCarthy (editors): *Automata Studies*, pages 3 – 44, Princeton, 1956.

[Kön36] Dénes König: *Theorie der endlichen und unendlichen Graphen*. Akademischer Verlag mbH, Leipzig, 1936.

[Kok93] Claus Koken: *Roulette: Computersimulation & Wahrscheinlichkeitsanalyse von Spiel und Strategien*. R. Oldenbourg Verlag, 3. Auflage, 1993.

[Kor78] Алексей Дмитриевич Коршунов: *О Перечислении Конечных Автоматов (Über die Aufzählung endlicher Automaten)*. Проблемы Кибернетики, 34:5–82, 1978.

[Koz92] John R. Koza: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[KR88] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*. Prentice Hall, 2nd edition, April 1988.

[KVBSV97] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli: *Theory and algorithms for state minimization of nondeterministic FSMs*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 16(11):1311–1322, November 1997.

[Leh76] Wolfgang Lehmann: *Ein vereinheitlichender Ansatz für die Redfield-Pólya-de Bruijnsche Abzähltheorie*. Dissertation, Mathematisch-Naturwissenschaftliche Fakultät der Rheinisch-Westfälischen Technischen Hochschule Aachen, 1976.

[Mea55] George H. Mealy: *A Method for Synthesizing Sequential Circuits*. Bell System Technical Journal, 34(5):1045–1079, September 1955.

[Med56] Ю. Т. Медведев: *О классе событий, допускающих предсавление в конечном автомате.* In: *Автоматы*, Seiten 385 – 401, 1956. Übersetzung siehe [Med64].

[Med64] Yu. T. Medwedew: *On the class of events representable in a finite automaton*. In *Sequential Machines: Selected Papers*. Addison-Wesley, 1964.

[Mes01] Dieter Meschede (Herausgeber): *Gerthsen Physik*. Springer, 21. Auflage, 2001.

[Mil80] Robin Milner: *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Moo56] Edward F. Moore: *Gedanken-experiments on sequential machines*. In Claude E. Shannon and John McCarthy (editors): *Automata Studies*, volume 34 of *Annals of Mathematics*, pages 129 – 153. Princeton University Press, 1956.

[MS94] Pierre Marchal and Eduardo Sanchez: *CAFCA: (Compact Accelerator for Cellular Automata) the Metamorphosable Machine*. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 66 – 71, Napa Valley, USA, April 1994.

[MSPPU02] Bertrand Mesot, Eduardo Sanchez, Carlos Andres Peña, and Andres Perez-Uribe: *SOS++: Finding Smart Behaviors Using Learning and Evolution*. In Standish, Abbass, and Bedau (editors): *Artificial Life VIII*, page 264ff. MIT Press, 2002.

[Nuu95] Esko Nuutila: *Efficient Transitive Closure Computation in Large Digraphs*. PhD thesis, Acta Polytechnica Scandinavica, Helsinki, June 1995. Mathematics and Computing in Engineering Series No. 74.

[Obe70] W. Oberschelp: *Isomorphe Automaten*. In: Johannes Dörr und Günther Hotz (Herausgeber): *Automatentheorie und Formale Sprachen*, Seiten 27 – 38. Bibliographisches Institut, 1970.

[OW90] Thomas Ottmann und Peter Widmayer: *Algorithmen und Datenstrukturen*, Band 70 der Reihe *Informatik*. Bibliographisches Institut & F. A. Brockhaus AG, Mannheim, 1990.

[Par81] David Park: *Concurrency and automata on infinite sequences*. In Peter Deussen (editor): *Theoretical Computer Science*, volume 104, pages 167 – 183, March 1981.

[Pas07] Daniel Paschka: *Entwicklung eines FPGA-Servers und Software zur Anbindung an einen User-PC*. Diplomarbeit, TU Darmstadt, Januar 2007.

[Pea90] Giuseppe Peano: *Sur une courbe, qui remplit une aire plane*. Dans *Mathematische Annalen*, tome 36(1), pages 157 – 160. Springer, mars 1890.

[PH98] David A. Patterson and John L. Hennessy: *Computer Organization & Design*. Morgan Kaufmann, 2nd edition, 1998.

[Phi60] Montgomery Phister: *Logical design of digital computers*. John Wiley & Sons Inc., New York, 5th edition, 1960.

[Pól37] G. Pólya: *Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen*. Acta Mathematica, 68(1):145–254, Dezember 1937.

[Poh00] Hartmut Pohlheim: *Genetik: Evolutionäre Algorithmen*. Springer, 2000.

[PR90] Martha E. Pollack and Marc Ringuette: *Introducing the Tileworld: Experimentally Evaluating Agent Architectures*. In *Proceedings Eighth National Conference on Artificial Intelligence*, pages 183 – 189. American Association for Artificial Intelligence, MIT Press, July 1990.

[Pur70] Paul Purdom: *A transitive closure algorithm*. In *BIT Numerical Mathematics*, volume 10(1), pages 76 – 94. Springer Netherlands, March 1970.

[Röd03] Patrick Röder: *Effiziente Programmierung des Game of Life*. Studienarbeit, TU Darmstadt, 2003.

[RN03] Stuart Russell and Peter Norvig: *Artificial Intelligence: a modern approach*. Prentice-Hall, 2nd edition, 2003.

[SB97] George Spencer-Brown: *Laws of Form (Gesetze der Form)*. Bohmeier Verlag, 1997.

[Sch75] Hugo Schiff: *Zur Statistik chemischer Verbindungen*. In: H. Wichelhaus (Herausgeber): *Berichte der Deutschen Chemischen Gesellschaft zu Berlin*, Band 8, Seiten 1542 – 1547. Verlag Chemie, Weinheim/Bergstraße, 1875.

[Sch98] Ralf Schneider: *Verfahren zur effizienten Zellularen Verarbeitung*. Dissertation, TU Darmstadt, September 1998. Verlag Dr. Kovač.

[Sök92] Wilfried Söker: *PostScript Level 2 griffbereit*. Vieweg, 1992.

[SMM59] Sundaram Seshu, R. E. Miller, and G. Metze: *Transition Matrices of Sequential Machines*. IRE Transactions on Circuit Theory, 6(1):5–12, March 1959.

[SSO+00] Werner Scholze-Stubenrecht, Ralf Oserwinter *et al.* (Herausgeber): *Die deutsche Rechtschreibung*, Band 1 der Reihe *Duden*. Bibliograpisches Institut & F. A. Brockhaus AG, 22. Auflage, 2000.

[Ste88] Franz Stetter: *Grundbegriffe der Theoretischen Informatik*. Springer-Verlag, 1988.

[Str92] Bjarne Stroustrup: *Die C++-Programmiersprache*. Addison-Wesley, 2. Auflage, 1992.

[Tis04] Jan Tisje: *Optimization of the Moving Creatures Problem in Software and Hardware*. Diplomarbeit, TU Darmstadt, März 2004.

[TM87] Tommaso Toffoli and Norman Margolus: *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, May 1987.

[Tur37] Alan M. Turing: *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, s2-42(1):230–265, 1937.

[Tur50] A. M. Turing: *COMPUTING MACHINERY AND INTELLIGENCE*. Mind, LIX(236):433–460, October 1950.

[US06] Andres Upegui and Eduardo Sanchez: *On-chip and on-line self-reconfigurable adaptable platform: the non-uniform cellular automata case*. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006.* IEEE, April 2006.

[VC97] Johan Vromans and Squirrel Consultancy: *Programming Perl 5.004*, 1997. `http://www.squirrel.nl/pub/perlref-5.004.1.pdf`.

[Wal04] Klaus Waldschmidt: *Technische Informatik 2*, Kapitel 6.5.6. Johann-Wolfgang-Goethe-Universität Frankfurt am Main, 2004.

[War62] Stephen Warshall: *A Theorem on Boolean Matrices*. Journal of the ACM (JACM), 9(1):11–12, January 1962.

[Web77] Wolfgang Weber: *Einführung in die Methoden der Digitaltechnik*, Band 6 der Reihe *AEG-Telefunken-Handbücher*. Elitera-Verlag, 5. Auflage, 1977.

[Weg93] Ingo Wegener: *Theoretische Informatik*. B. G. Teubner Stuttgart, 1993.

[Wol02] Stephen Wolfram: *A New Kind of Science*. Wolfram Media, Champaign, IL, 2002. `http://www.wolframscience.com`.

[Yua06] Xuesong Yuan: *FPGA-basiertes Simulationssystem für intelligente Roboter*. Diplomarbeit, TU Darmstadt, Dezember 2006.

[Zha96] Weimin Zhang: *Finite State Systems in Mobile Communications*. PhD thesis, School of Electronic Engineering, University of South Australia, February 1996.

[Zus69] Konrad Zuse: *Rechnender Raum*. Vieweg Verlag, Braunschweig, 1969.

Glossar

ASCII 93
American Standard Code for Information Interchange. Zuordnung von Zeichen zu einem Zahlenwert für Übertragung zwischen verschiedenen Computersystemen, z. B. einem großem „A“ ist der Wert 65 zugeordnet, 66 steht für ein großes „B“.

CSV 94
Comma Separated Values. Durch Semikolon und Zeilenwechseln voneinander getrennte Werte, die in Tabellenkalkulationsprogrammen herstellerunabhängig, in Spalten und Zeilen unterteilt, eingelesen werden können.

Density-Problem 12
Viele Automaten sind in Reihe angeordnet, so dass jeder Automat einen rechten und einen linken Nachbarn hat, von dem der aktuelle Zustand gelesen werden kann. Mögliche Zustände sind lediglich 0 und 1. Am Anfang ist der initiale Zustand eines jeden Automaten zufällig festgelegt. Ziel ist es, mit einer möglichst einfachen Regel möglichst schnell zu erreichen, dass jeder Automat nur dann den Zustand 1 erreicht, wenn dieser am Anfang mehrheitlich vorkam. Andernfalls sollen alle Automaten den Zustand 0 aufweisen.

FIFO 77
First In First Out. Das zuerst eintreffende Byte wird auch zuerst aus dem Speicher ausgelesen.

FPGA 23
Field Programmable Gate Array, programmierbarer Logikschaltkreis. Neben einer blockförmig angeordneten Logikstruktur (z. B. Gatter für OR, NAND) werden auch Flip-Flops, Speicher und komplexe Funktionen, z. B. Multiplikatoren, zur Verfügung gestellt. Die Funktion wird mittels einer Hardware-Beschreibungssprache (Hardware Description Language) – wie z. B. Verilog HDL [Ins01] – beschrieben und nach Compilierung, Synthese und weiteren Schritten auf den Baustein übertragen. Bekannte Hersteller sind Altera und Xilinx.

HDL 237
Hardware Description Language. Siehe auch FPGA, im Glossar Seite 235.

Heuristik 42
Schlussfolgerung ziehen und daraus neues Wissen gewinnen, z. B. eine Regel aufstellen.

Karnaugh-Veitch-Plan 15
Findet Verwendung in der Logikminimierung per Hand für bis zu vier Variablen.

LED 80
Light Emitting Diode, Leuchtdiode.

Natürliche Zahlen 85
Die Menge der natürlichen Zahlen $\mathbb{N}$ umfasst alle positiven ganzen Zahlen beginnend bei 1. Soll zusätzlich die Null enthalten sein, so wird die Mengenbezeichnung um den Index 0 erweitert, also $\mathbb{N}_0 = \{0,1,2,3,4,5,\ldots\}$.

Read Only Memory (ROM) 99
Speicher, aus dem nur gelesen werden kann. Der Zugriff auf die Daten erfolgt wahlfrei, d. h. nach Anlegen einer Adresse erfolgt mit kurzer Verzögerung die Datenausgabe.

Turing-Maschine 6
Eine Turing-Maschine (TM) besteht aus einem Zustandsautomaten und einem endlosen Band für die Ein- und Ausgabe. Ist der Zustandsautomat deterministisch, so handelt es sich dabei dann um eine deterministische Turing-Maschine (DTM). Generell ist der Zugriff nur auf eine Stelle des Bandes zum Lesen und Schreiben möglich. Ein Verschieben dieser Stelle ist danach nur um eine Position nach links oder rechts möglich. Formal wird eine TM beschrieben durch $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ mit den Bedeutungen

- Q: endliche Zustandsmenge (analog zu $\mathbb{S}$),
- Σ: endliches Eingabealphabet, mit dem das Band initialisiert ist,
- Γ: endliches Bandalphabet, $\Sigma \subset \Gamma$,
- δ: Übergangsfunktion $Q \times \Gamma \rightarrow Q \times \Gamma \times D$ mit $D = \{R, L, N\}$ für die Richtung der nächsten Position zum Lesen und Schreiben (Rechts, Links, Unverändert),
- q_0: Anfangszustand, $q_0 \in Q$,
- B: Leerzeichen („blank“), das auf den unbeschriebenen Stellen des Bandes steht, $B \in \Gamma \backslash \Sigma$,
- F: Menge der akzeptierten Endzustände, $F \subseteq Q$.

Wenn die Position auf dem Band unverändert bleibt, evtl. auch nur aufgrund des aktuell gelesenen Zeichens, ist die Maschine gestoppt und die Problemberechnung abgeschlossen. [Weg93, Seite 9f.]

Stichwortverzeichnis

Werdegang

Mathias Halbach, geboren am 28. Oktober 1971 in Berlin-Tempelhof.

Sept. 1978 bis Juni 1982	Otto-Hahn-Schule, Rüsselsheim
Aug. 1982 bis Juli 1984	Friedrich-Ebert-Schule, Rüsselsheim
Aug. 1984 bis Juni 1991	Immanuel-Kant-Schule, Gymnasium, Rüsselsheim mit Abschluss Abitur
Okt. 1991 bis Sept. 1997	Studium der Informatik an der Technischen Hochschule Darmstadt mit Abschluss als Diplom-Informatiker, Thema der Diplomarbeit: „Entwicklung eines CDL-Compilers zur Generierung von Zellular-Automaten-Simulatoren als Java-Applet mit Experimentverwaltung und Bedienung über Internet-Web-Browser“
Okt. 1997 bis Juni 2001	Mitarbeiter bei der Telenet GmbH Kommunikationssysteme in Darmstadt, Teil der Alcatel SEL AG
Juli 2001 bis Mai 2003	Mitarbeiter bei der Telenet AG Rhein-Main in Darmstadt
Juni 2003 bis Mai 2008	Wissenschaftlicher Mitarbeiter an der Technischen Universität Darmstadt im Fachbereich Informatik, Fachgebiet Rechnerarchitektur