Karlsruher Institut für Technologie

Asynchrone Simulation von Schaltkreisen in Zellularautomaten

Masterarbeit
von

Johannes Lorenz

an der Fakultät für Informatik,
Institut für Theoretische Informatik (ITI)

- nachträglich korrigierte Fassung vom 3. Dezember 2014 -

Betreuer:	Prof. Dr.-Ing. Roland Vollmar
Betreuender Mitarbeiter:	Dr. rer. nat. Thomas Worsch

Bearbeitungszeit: 5. Mai 2014 bis 4. November 2014

Dies ist eine **inoffizielle** Fassung der Diplomarbeit. Es handelt sich um die Originalfassung, bei der kleine Fehler korrigiert, jedoch sonst keine Erweiterungen gemacht worden. Die letzte Änderung wurde am 3. Dezember 2014 vorgenommen.

Ich danke Dr. Thomas Worsch für eine exzellente Betreuung und dafür, dass er mir bezüglich der Themenwahl viel Freiheit gelassen hat.

Außerdem danke ich Philipp Lorenz und Mathias Lorenz für das Lesen dieses Dokuments.

Inhaltsverzeichnis

Abstract

This thesis shows multiple algorithms for computing all resulting final configurations, given a cellular automaton and a set of input configurations.

All algorithms use techniques that are designed for cellular automata which can simulate circuits, such as the one by Schneider [Sch12] or by Morrison and Ulidowski [MU14]. The proposed algorithms prove the correctness of some of the proposed modules of the latter cellular automata. Nonetheless, the algorithms work on arbitrary cellular automata.

The main difficulties of the problem are being shown; and multiple techniques for solving them are being provided and evaluated. One algorithm of this thesis is feasible for larger input. Another algorithm shows theoretical results on how a cellular automaton can be simulated in an asymmetric way.

Abstract

Diese Arbeit zeigt mehrere Algorithmen, die nach Eingabe einer Menge initialer Konfigurationen alle resultierenden finalen Konfigurationen berechnen.

Alle Algorithmen bedienen sich Techniken, die für Zellularautomaten, die Schaltkreise simulieren können, so wie die von Schneider [Sch12] oder von Morrison und Ulidowski [MU14], entworfen sind. Die vorgestellten Algorithmen beweisen die Korrektheit einiger der vorgestellten Module zu letzteren Zellularautomaten. Nichtsdestotrotz funktionieren die Algorithmen auch mit beliebigen Zellularautomaten.

Es werden die größten Schwierigkeiten der Aufgabenstellung aufgezeigt; und mehrere Techniken zu deren Lösung werden bereitgestellt und bewertet. Ein Algorithmus aus dieser Arbeit kann mit größeren Eingaben arbeiten. Ein weitere Algorithmus zeigt theoretische Resultate darüber, wie man einen Zellularautomaten auf asymmetrische Weise simulieren kann.

Kapitel 1

Einleitung

Viele naturwissenschaftliche Modelle untersuchen die zeitliche Entwicklung gewisser Abläufe. Man hat eine oder mehrere Anfangskonfigurationen und ist daran interessiert, daraus Endkonfigurationen[1] abzuleiten. Oftmals ist es vielversprechend, solche Abläufe mit dem Computer zu simulieren. Nur eine sehr kleine Menge von Beispielen, für die das denkbar wäre, sind Wettervorhersagen, Erdbebenwarnungen, Waldbrände oder die Evakuierung von Passanten aus einem Gebäude.

Versucht man, so eine Simulation Schritt für Schritt durchzuführen, so kann es in vielen Schritten unterschiedliche Folgekonfigurationen geben. Dadurch steigt die Anzahl der Endkonfigurationen gegebenenfalls exponentiell, wodurch es in der Praxis viel zu lange dauert, alle möglichen Endkonfigurationen zu berechnen. Oftmals ist es dann eine Abhilfe, nur wahrscheinliche Folgekonfigurationen zu berechnen. Manchmal ist man auch nur daran interessiert, ein paar wenige beliebige Endkonfigurationen zu finden, zum Beispiel, wenn man annimmt, dass es nur eine oder wenige Endkonfigurationen gibt. Manchmal reicht auch das nicht, was die Suche nach einem exakteren Verfahren veranlasst.

Eine bekannte Vorgehensweise, natürliche Modelle zu diskretisieren, ist es, einen Zellularautomaten zu konstruieren. Zellularautomaten bestehen aus mehreren Zellen, die gewisse Zustände annehmen können. Die Gesamtheit aller Zustände nennt man eine globale Konfiguration. Zellularautomaten werden für mehrere Iterationen simuliert; und in jedem Schritt kann eine Zelle, abhängig von den Zuständen in einer lokalen Nachbarschaft, einen Folgezustand wählen. Handelt es sich um einen synchronen, deterministischen Zellularautomaten, so ist die Simulation immer ein einzelner Pfad, sofern man die Abfolge der Iterationen als Graph interpretiert. In diesem Fall fallen die Aufgaben, irgendeine Endkonfiguration und alle Endkonfigurationen zu ermitteln, zusammen. Solche Zellularautomaten sind leider oft nicht typisch für naturgegebene Modelle, da sie keinen Zufall erlauben[2].

[1] Manchmal liegen keine eindeutigen Endkonfigurationen vor, sondern strenge Zusammenhangskomponenten von Konfigurationen, aus denen sonst keine weiteren Konfigurationen mehr entstehen. Vereinfachend bezeichnen wir solche Komponenten in diesem Kapitel ebenfalls als „Endkonfigurationen“.

[2] Das gilt jedenfalls für unsere Definition von Zellularautomaten, die weiter unten folgt.

Eine Abhilfe sind asynchrone, oder allgemeiner nichtdeterministische Zellularautomaten. Hier steigt die Anzahl der Endkonfigurationen schnell exponentiell. Eine Reihe solcher Zellularautomaten konvergieren später zwar für bestimmte Eingaben gegen eine kleine Menge von Endkonfigurationen[3], so dass man nach der Berechnung von wenigen gleichen Endkonfigurationen bereits vermuten kann, dass dies alle sind. Sicherheit gewährt dennoch nur ein Beweis, was sich gelegentlich als sehr schwer darstellt, oder ein simulierender Algorithmus.

In dieser Masterarbeit konstruieren wir solche Algorithmen. Obwohl diese generisch in dem Sinne sind, dass sie mit mehr oder weniger beliebigen Zellularautomaten arbeiten kann, so liegt unser Augenmerk vor allem auf der Simulation von Schaltkreisen. Ein Argument für diesen Typ von Zellularautomat ist, dass die Simulation meist leicht nachvollziehbar ist und es oft nur eine eindeutige Endkonfiguration gibt, die Anzahl der möglichen Konfigurationen auf dem Weg dahin jedoch sehr groß werden kann. Außerdem lag uns zu Beginn dieser Arbeit noch kein Beweis für die Module in den Automaten von Schneider[Sch12] beziehungsweise Morrison und Ulidowski [MU14] vor.

Wir werden im Folgenden nach der Einführung einiger Definitionen ein paar grundlegende Dinge über Algorithmen aussagen. Danach erklären wir drei simple Algorithmen, die zwar allein noch nicht ausreichen, aber eine Grundlage für weitere Kapitel sind. Darauf folgen dann zwei Negativbeispiele, und schließlich ein Algorithmus, der sich als praktikabel erwiesen hat. Der Inhalt wird mit Analysen und Implementierungsdetails vertieft. Im Anschluss finden sich noch einige Erklärungen und Simulationen für bestimmte Zellularautomaten, wie die im vorhergehenden Abschnitt genannten.

[3] Beispiele hierfür sind die hier behandelten Zellularautomaten von Schneider [Sch12] und von Morrison und Ulidowski[MU14], sowie einzelne Stabilisierungen des Chip-Firing-Modell, welches von Holroyd et al. [HLM+08] detailliert beschrieben wird.

Kapitel 2

Definitionen

2.1 Grundlegende Definitionen

Wir beginnen mit ein paar geläufigen Definitionen. Im ganzen Kapitel sei $0 \notin \mathbb{N}$ und $\mathbb{N}_0 \coloneqq \mathbb{N} \cup \{0\}$. Für eine Menge M bezeichne $\mathcal{P}(M)$ die Potenzmenge von M.

Für eine Menge M, $n \in \mathbb{N}$ und $M_1, ...M_n \subseteq M$ sei $\dot{\bigcup}_{i=1}^{n} M_i$ als die paarweise disjunkte Vereinigung definiert; genauer gilt:

$$M' = \dot{\bigcup}_{i=1}^{n} M_i \quad \Leftrightarrow \quad M' = \bigcup_{i=1}^{n} M_i \text{ und } M_j \cap M_k = \emptyset \ (1 \leq j < k \leq n)$$

Definition 2.1 (Binäre Relationen)**.** Sei V eine Menge. Eine Menge $\succ$ heißt binäre Relation auf V, falls $\succ \subseteq V \times V$. Seien $v_1, v \in V$. Für $k \in \mathbb{N}$ schreiben wir

$$v_1 \succ^k v$$

falls es k Elemente $v_2, ...v_{k+1} \in V$ gibt mit $v_{k+1} = v$ und $v_i \succ v_{i+1}$ für alle $1 \leq i \leq k$. Darüber hinaus schreiben wir

$$v_1 \succ^0 v \ :\Leftrightarrow \ v_1 = v;$$

und definieren:

$$\begin{aligned} v_1 \succ^* v \ &:\Leftrightarrow \ \exists k \in \mathbb{N}_0 \colon v_1 \succ^k v \quad \text{sowie} \\ v_1 \succ^+ v \ &:\Leftrightarrow \ \exists k \in \mathbb{N} \colon v_1 \succ^k v. \end{aligned}$$

Beispiel 2.1. Für $n \in \mathbb{N}$ definieren wir als eine Ordnungsrelation auf $\mathbb{R}^n$ diejenige, die „zeilenweise“ ordnet. Genauer seien $(p = (p_1, ...p_n), q = (q_1, ...q_n)) \in (\mathbb{R}^n)^2$. Dann gilt:

$$\begin{aligned} p \leq q \quad &:\Leftrightarrow \quad \exists k, 1 \leq k \leq n \colon p_k \leq q_k \\ &\text{und} \quad \forall i, k < i \leq n \colon p_i = q_i. \end{aligned}$$

Definition 2.2. Wir erweitern $\succ$ auf Mengen, wobei die Indexnotation zu $\succ$ ebenfalls übernommen wird und daher hier nicht angeschrieben wird. Seien X, Y Mengen, $\succ \subseteq X \times Y$, $a \in X, b \in Y$, sowie $C \subseteq X$ und $D \subseteq Y$. Wir schreiben

- $C \succ D :\Leftrightarrow \forall d \in D \colon \exists c \in C \colon c \succ d$,
- $a \succ D :\Leftrightarrow \{a\} \succ D$ und
- $C \succ b :\Leftrightarrow C \succ \{b\}$.

Umgangssprachlich würde man über den ersten Stichpunkt sagen, C sei genug, um ganz D zu erreichen.

Definition 2.3 (Maximum und nähester gemeinsamer Vorfahre)**.** Seien V eine Menge und $\succ \subseteq V \times V$ eine Halbordnung, also reflexiv, antisymmetrisch und transitiv. Falls es ein $m \in V$ mit der Eigenschaft

$$\forall m' \in V \colon m' \succ m$$

gibt, so heißt m das **Maximum** von V bezüglich $\succ$. Wir schreiben: $m = \max_{\succ} V$.

Sei nun $(x, y) \in V^2$. Falls es ein $l \in V$ gibt, sodass

$$l = \max_{\succ}\{a \,|a \ \succ^* x, a \succ^* y\},$$

so nennen wir a den **nähesten gemeinsamen Vorfahren** von x und y. Wir schreiben: $l = \mathrm{lca}_{\succ}(x, y)$.

Man beachte, dass das Maximum und somit auch der näheste gemeinsame Vorfahre eindeutig sind, falls sie existieren.

Wir werden Indizes wie $\succ$ bei $\max_{\succ}$ oder $\mathrm{lca}_{\succ}$ häufiger weglassen, wenn sie klar sind, und werden dies nicht für jede Definition extra anmerken.

Definition 2.4 (Graph)**.** Ein Tupel $(V, \succ)$, wobei $\succ \subseteq (V \times V)$ ist, nennen wir einen gerichteten **Graph**. Wir führen die Schreibweisen $V((V, \succ)) \coloneqq V$ und $E((V, \succ)) \coloneqq \succ$ ein.

Bevor wir nun Zellularautomaten definieren, müssen wir noch Nachbarschaften einführen.

Definition 2.5 (Nachbarschaft)**.** Sei R eine Menge, so nennen wir eine Funktion $N : R \to \mathcal{P}(R)$ eine R-**Nachbarschaft**.

Außerdem definieren wir für eine R-Nachbarschaft $N : R \to \mathcal{P}(R)$ eine R-Nachbarschaft $N^- : R \to \mathcal{P}(R)$, so dass für $r \in R$ gilt:

$$N^-(r) = \{r' \in R | N(r') = r\}.$$

Beispiel 2.2 (Von-Neumann-Nachbarschaft)**.** Sei $R \subseteq \mathbb{Z}^d$ für $d \in \mathbb{N}$, $r \in R$ und $\rho \in \mathbb{N}_0$. Wir nennen

$$\mathcal{N}_\rho^{(d)}(r) = r + \{(i_1, ...i_d) \mid \sum_{j=1}^{d} |i_j| \leq \rho\}$$

die **Von-Neumann-$\mathbb{Z}^d$-Nachbarschaft** mit Radius ρ.

Beispiel 2.3 (Moore-Nachbarschaft)**.** Sei $R \subseteq \mathbb{Z}^d$ für $d \in \mathbb{N}$, $r \in R$ und $\rho \in \mathbb{N}_0$. Wir nennen

$$\mathcal{M}_\rho^{(d)} = r + \{(i_1, ...i_d) \mid \max_{\leq}\{|i_j| \mid 1 \leq j \leq d\} \leq \rho\}$$

die **Moore-$\mathbb{Z}^d$-Nachbarschaft** mit Radius ρ.

Definition 2.6 (Automaten)**.** Sei

$$C \coloneqq (R, Q, N_{\text{in}}, N_{\text{out}}, \delta)$$

ein Tupel, wobei R eine abzählbare Menge, Q eine Menge, N_{in} und N_{out} R-Nachbarschaften und δ wie folgt definiert ist:

$$\delta : R \to (Q^{N_{\text{in}}(r_1)} \to Q^{N_{\text{out}}(r_1)}, Q^{N_{\text{in}}(r_2)} \to Q^{N_{\text{out}}(r_2)}, ...), \quad v \mapsto \delta(v).$$

Dann nennen wir C einen **Zellularautomaten**. Im Folgenden reden wir nur noch von „**Automaten**“. Wir nennen auch

- Q die **Zustandsmenge** von C,
- R den **Raum** von C, Elemente aus R **Zellen**, und Elemente aus $\mathcal{P}(R)$ **Zonen**[1],
- Elemente aus Q^R **Konfigurationen** oder **globale Konfigurationen**
- N_{in} die **Eingabe-Nachbarschaft** von C,
- N_{out} die **Ausgabe-Nachbarschaft** von C und
- δ die **lokale Überführungsfunktion** (oder meist nur „Überführungsfunktion“) von C.

Weiter definieren wir $\mathfrak{C}$ als die Menge aller Zellularautomaten.

Unser Modell wird zu dem aus der Literatur bekannten Modell, wenn man $N_{\text{out}} \coloneqq \text{id}_R$ setzt. Wir bevorzugen unsere Definition, da sie viele weitere Modelle, wie zum Beispiel „self-timed cellular automata“ [MU14] zulässt.

Nun besteht die Frage, welche Zelle von welcher Zelle „lesen“ kann. Folgende Definition gibt eine Antwort.

Definition 2.7 (Leser und Schreiber)**.** Sei ein Automat $(\cdot, \cdot, N_{\text{in}}, N_{\text{out}}, \cdot)$ gegeben. Wir definieren:

$$N_{\text{readers}} \coloneqq {N_{\text{in}}}^{-} \circ N_{\text{out}}$$
$$N_{\text{writers}} \coloneqq {N_{\text{out}}}^{-} \circ N_{\text{in}}$$

Diese Variablen seien für jeden Automaten automatisch definiert.

[1] Wir fordern im Allgemeinen nicht, dass eine Zone in einer bestimmten Weise zusammenhängt.

Es gibt ein paar spezielle Automaten, die wir hin und wieder ansprechen werden. Sie sind im Folgenden angegeben.

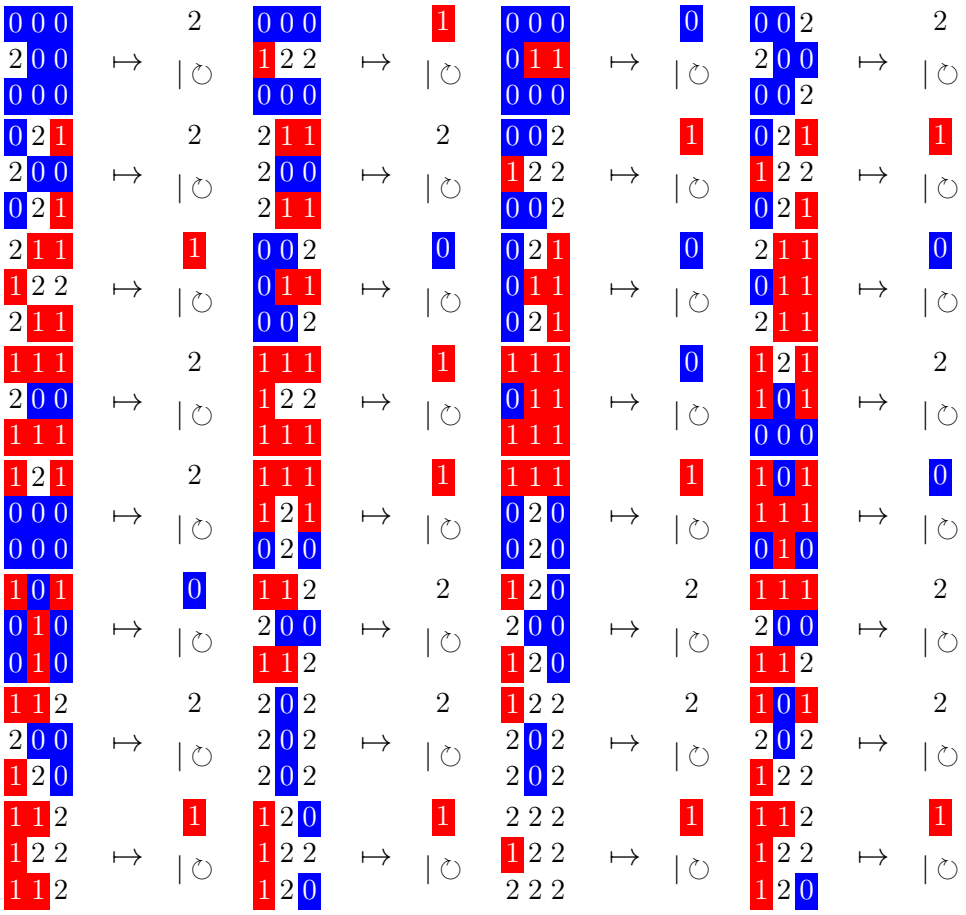

Tabelle 2.1: Überführungsfunktion für den Zellularautomaten „bc3“. ⟳ bedeutet Drehsymmetrie mit $\{0..3\} \cdot 90°$, | bedeutet Spiegelsymmetrie.

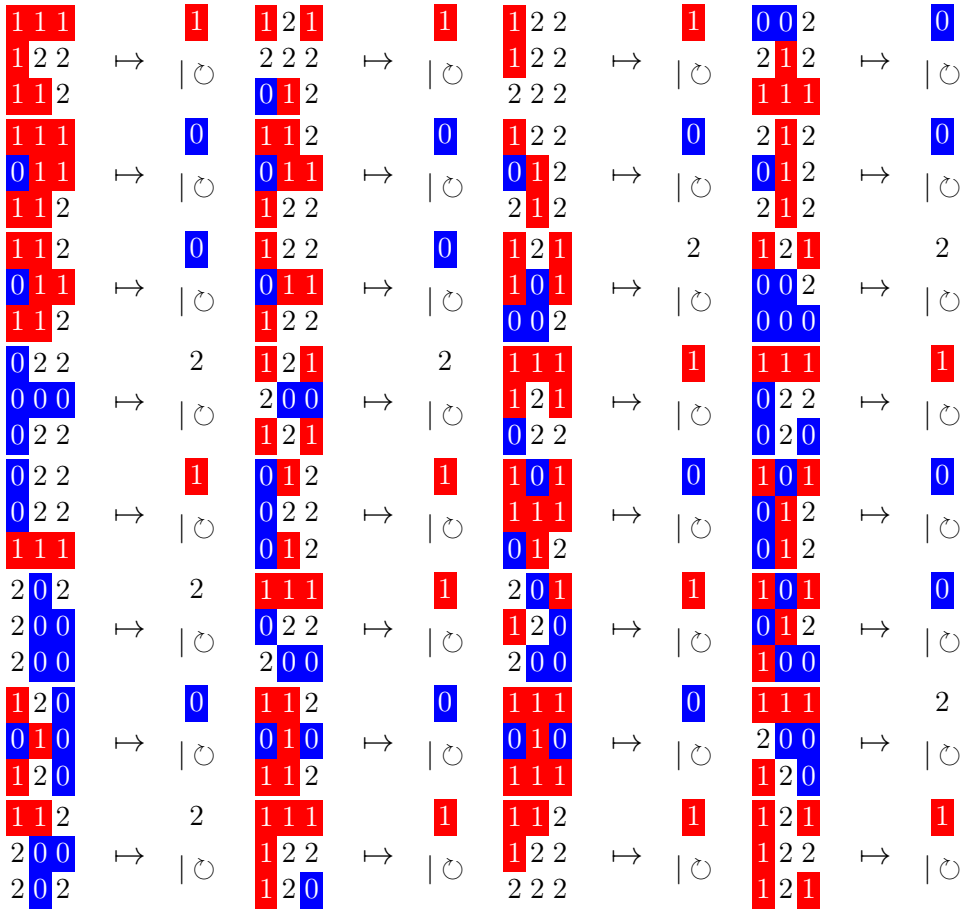

Tabelle 2.2: Fortsetzung von Tabelle 2.1.

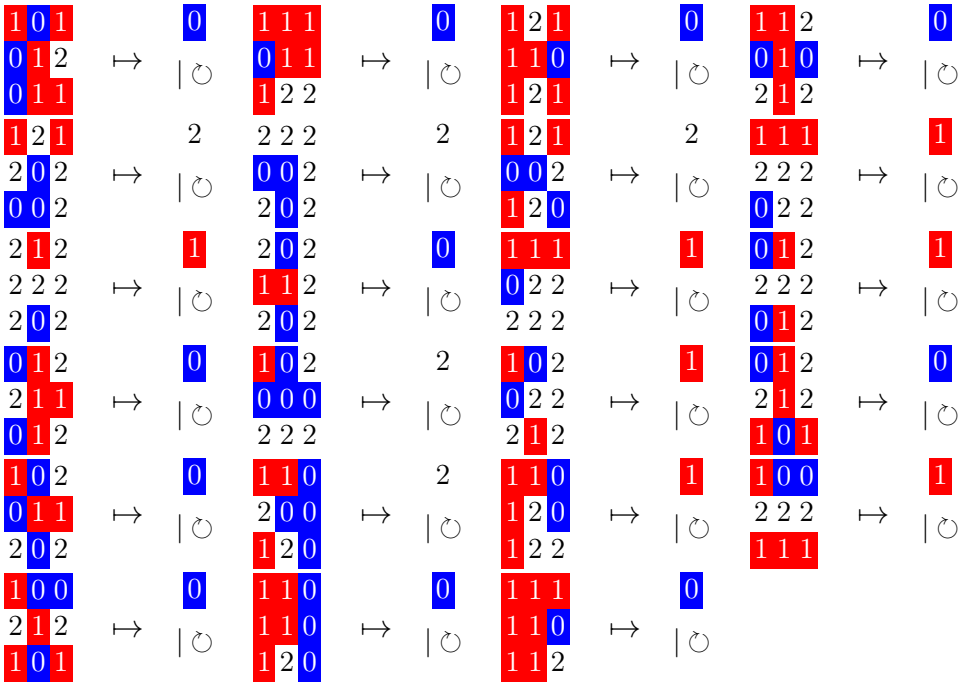

Tabelle 2.3: Fortsetzung von Tabelle 2.2.

Definition 2.8 ($\mathcal{C}_{\text{bc3}}$)**.** Als $\mathcal{C}_{\text{bc3}}$ bezeichnen wir den Automaten von Schneider [Sch12], der nur 3 Zustände benötigt, und mit dem man auch boolesche Schaltkreise (englisch „boolean circuits“) modellieren kann. Es gilt:

$$\mathcal{C}_{\text{bc3}} = (\mathbb{Z}^2, \{0,1,2\}, {M_1}^{(2)}, {M_1}^{(2)}, \delta_{\text{bc3}}),$$

wobei δ_{bc3} wie in Abbildung 2.1 definiert ist.

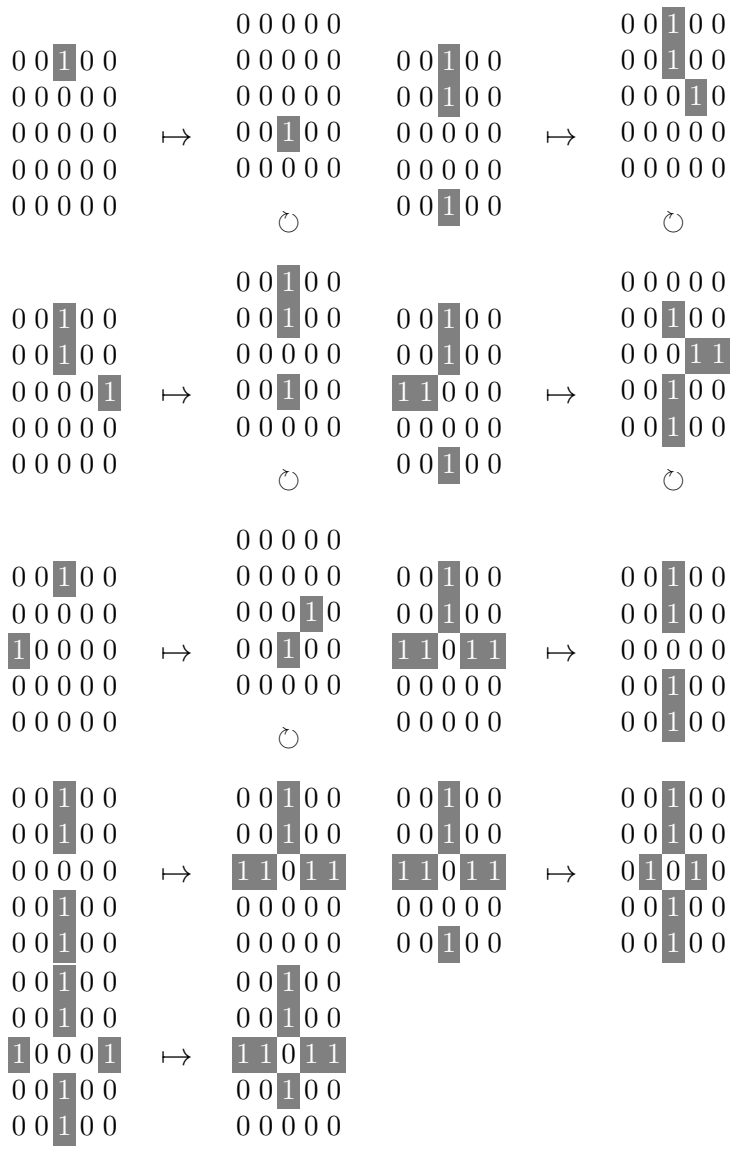

Tabelle 2.4: Überführungsfunktion für den Zellularautomaten „st“. ⟳ bedeutet Drehsymmetrie mit $\{0..3\} \cdot 90°$.

Definition 2.9 ($\mathcal{C}_{\text{st}}$)**.** Als $\mathcal{C}_{\text{st}}$ bezeichnen wir eigentlich mehrere Automaten, nämlich die „self-timed cellular automata“ von Morrison und Ulidowski [MU14]. Exemplarisch beziehen wir uns hier nur auf den dort als dritten Automat angegebenen, bestehend aus den dort definierten Regelsets „RS“, „P“ und „C“. Wir übersetzen diesen Automaten in ein kartesisches Koordinatensystem und erhalten:

$$\mathcal{C}_{\text{st}} = (\mathbb{Z}^2, \{0, 1, 2\}, N_{\text{st}}, N_{\text{st}}, \delta_{\text{st}}),$$

wobei $N_{\text{st}}(r) \coloneqq r + \{(0,-2),(0,-1),(-2,0),(-1,0),(1,0),(2,0),(0,1),(0,2)\}$ für $r \in \mathbb{Z}^2$ gilt, und δ_{bc3} eine der Definitionen aus Abbildung 2.4 annimmt.

Bevor wir Simulatoren definieren, benötigen wir einen Begriff für Zellen, in denen die Anwendung einer Überführungsfunktion einen Effekt hat.

Definition 2.10 (variable Zellen)**.** Sei $C = (R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathfrak{C}$. Eine Zelle $r \in R$ heißt in der Konfiguration $c \in Q^R$ **variable Zelle**, falls $\delta(r)(c(N_{\text{in}}(r))) \neq c(N_{\text{out}}(r))$.

Nachdem wir nun Automaten allgemein definiert haben, so zeigen wir im Folgenden, wie man sie simuliert.

Definition 2.11 (Simulator)**.** Sei $C = (R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathfrak{C}$. Ein **Simulator** S für C ist eine Menge von Daten, die Informationen beinhalten, um auf C deren Übergangsfunktion in bestimmter Weise anzuwenden. Wir definieren eine **Aktivierung** durch S als den Vorgang, dass S eine Menge von Zellen $A \subseteq R$ auswählt, wobei $A \neq \emptyset$ und $\forall (a_1, a_2) \in A^2 : N_{\text{out}}(a_1) \cap N_{\text{out}}(a_2) = \emptyset$. Weiter sei eine **Iteration** von S auf C eine Aktivierung einer Menge von Zellen $A \subseteq R$ durch S, gefolgt von der Veränderung der Zustände in $N_{\text{out}}(A)$ durch Anwendung von δ auf A.

Manchmal schränken wir die Menge der aktivierbaren Zellen auf ein $R_0 \subseteq R$ ein. Auch in diesem Fall muss die Wahl der Zellen $A \neq \emptyset$ sein.

Wir definieren $\mathfrak{S}_{\text{async}}(C)$ als die Menge derjenigen Simulatoren, die, gegeben eine variable Menge $V \subseteq R$, nach irgendeinem Zufallsprinzip eine Menge $A \subseteq R$ für die Aktivierung auswählen. Sie heißen auch **asynchrone Simulatoren**.

Definition 2.12 (Simulation)**.** Sei $(R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathfrak{C}$ und $S \in \mathfrak{S}_{\text{async}}(C)$. Die Berechnung einer Folge von Iterationen von S auf C nennen wir eine **partielle Simulation** von S auf C. Die Berechnung aller möglichen partiellen Simulationen von S auf C nennen wir eine **vollständige Simulation** von S auf C.

Das Augenmerk dieser Arbeit liegt auf vollständigen Simulationen.

Wir suchen eine Visualisierung dessen, welche Konfigurationen man von welchen durch Iterationen erreichen kann. Eine Möglichkeit ist es, diese Beziehung in einen Graphen zu übertragen.

Definition 2.13 (Zustandsgraph)**.** Sei $C = (R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathfrak{C}$ und $S \in \mathfrak{S}_{\text{async}}(C)$.

Seien $v_1, v_2 \in Q^R$. Für $R_0 \subseteq R$ schreiben wir

$$v_1 \vdash_{R_0, C, S} v_2,$$

falls man mit S, ausgehend von v_1, nach einer Iteration auf C zu v_2 gelangen kann, wobei in jeder Iteration ausschließlich Zellen aus R_0 aktiviert werden. Man beachte,

dass, wie in Definition 2.11, keine „leeren“ Iterationen erlaubt sind. Wir definieren zusätzlich

$$\vdash_{C,S} \coloneqq \vdash_{R,C,S}$$

und außerdem eine Funktion

$$\begin{aligned} \mathcal{G}_{C,S}\colon\ \mathcal{P}(Q^R) &\to (\mathcal{P}(Q^R), \mathcal{P}(\vdash_{C,S})), \\ Q_0 &\mapsto (Q \coloneqq \{q \in Q^R \mid Q_0 \vdash^*_{C,S} q\}, \vdash_{C,S} |_{Q\times Q}). \end{aligned}$$

Dabei heißt Q_0 auch Menge der **Anfangskonfigurationen** und $\mathcal{G}_{C,S}(Q_0)$ auch der **Zustandsgraph** von Q_0. Betrachtet man den Graph der strengen Zusammenhangskomponenten aus $\mathcal{G}_{C,S}(Q_0)$, so nennen wir diejenigen mit Ausgangsgrad 0 **Endkomponenten** und ihre Elemente **Endkonfigurationen**. Eine Menge von Repräsentanten der Endkomponenten nennen wir **Endrepräsentanten**.

Man beachte, dass der Zustandsgraph vom Simulator abhängt. Das ganze Skript wird jedoch die Einschränkung machen, dass jeder Simulator jede Zelle mit einer Wahrscheinlichkeit aktiviert, die echt größer 0 ist. Daher ist der Zustandsgraph an sich für jeden Simulator gleich. Allerdings kann man den Graph um Beschriftungen für Kanten erweitern; und dann können die Kanten in Abhängigkeit vom Simulator beschriftet werden.

Eine Möglichkeit, den Ablauf einer vollständigen Simulation festzuhalten, und dabei möglichst gut zu erkennen, in welcher Reihenfolge Zellen aktiviert werden können, bietet folgender Abhängigkeitsgraph.

Definition 2.14 (Abhängigkeitsgraph)**.** Seien $C = (R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathfrak{C}$ und $S \in \mathfrak{S}_{\text{async}}(C)$. Sei weiter $Q_0 \subseteq Q^R$ eine Menge von Anfangskonfigurationen.

Wir definieren einen Abhängigkeitsgraphen $\mathcal{D}_{C,S}(Q_0) \coloneqq (R, \succ)$, wobei $(r_1, r_2) \in \succ$ falls es $(v_1, v_2) \in E(\mathcal{G}_{S,C}(Q_0))$ gibt, so dass

- $r_2 \in N_{\text{readers}}(r_1)$,
- $v_1(N_{\text{out}}(r_1)) \neq v_2(N_{\text{out}}(r_1))$ und
- $\delta(r_2)(v_2(r_2 + N_{\text{in}})) \neq \delta(r_2)(((v_2|_{R\setminus\{r_1\}}) \cup (v_1|_{\{r_1\}}))(r_2 + N_{\text{in}}))$.

2.2 Pseudocode

In den folgenden Kapiteln tritt häufig Pseudocode auf. Dieser wird einige Variablennamen enthalten. Um nicht mit den Bedeutungen der Variablen durcheinander zu kommen, halten wir uns meist an folgende Konventionen.

- $c_{\ldots}$ sind Bedingungen („condition“)

- $n_{...}$ oder $i_{...}$ sind ganze Zahlen („numbers“, „ids“)
- $G_{...}$ sind Graphen („graphs“)
- $V_{...}$ sind Vektoren („vectors“) oder Konfigurationen (die als Vektoren von Zuständen zu lesen sind)
- $P_{...}$ sind Zonen (von „points“, da man in der Praxis oft Koordinaten zu verwalten hat)
- $\Delta_{...}$ sind sogenannte reversible Patches, also Daten, die Information darüber enthalten, wie man eine Konfiguration in eine bestimmte andere und zurück überführen kann. Es gibt einen Konstruktor PATCH, der intuitiv einen Patch für die Argumente erstellt. Das könnten zum Beispiel zwei Konfigurationen sein. Außerdem gibt die Funktion CELLS die Zellen des Patches zurück.

Wie in Pseudocode üblich und entgegen der Konventionen in „C“ erfordern wir im Quellcode keine Deklarationen. Das hat auch zur Folge, dass es globale Variablen gibt: Taucht eine Variable in 2 verschiedenen Bereichen oder gar Funktionen mit gleichem Namen auf, so behält sie ihren Wert zwischenzeitlich und ist implizit als global anzusehen.

Wird eine Variable per Argument einer Funktion übergeben, so muss man sich stets die Übergabe einer Referenz darunter vorstellen. Damit können Funktionen die Parameter ändern. Dies entspricht der Syntax der Programmiersprache „Java“; C++-Programmierer können dies als Übergabe per Referenz sehen.

Außerdem sei zu beachten, dass wir für Zuweisungen „$\leftarrow$“ und für Vergleiche „$=$“ verwenden, so wie es in vielen Pseudocodes gemacht wird. So ist

$$x = y$$

genau dann wahr, wenn x und y den gleichen Wert haben. Ein C-Programmierer könnte denken, dass x zuerst den Wert von y erhält, und dann der Ausdruck den Wert von x annimmt. Dies ist nicht gemeint. Sind also $x = \textbf{false}$ und $y = \textbf{false}$, so ist gemeint, dass der Ausdruck $x = y$ zu **true** auswertet.

2.3 Code-Referenzen

Ab und zu verweisen wir auf C++-Code oder ausführbare Dateien. Dabei werden Dateinamen grau hinterlegt. Ein Beispiel wäre der Dateiname `src/base/split.cpp`. Solche Referenzen verweisen auf die Implementierung zu dieser Arbeit. Man kann den Quellcode entweder von der beiliegenden DVD lesen, oder man lädt ihn aus dem Unterordner „search“ des Projekts „sca-toolsuite“ [Lor14]. Prinzipiell empfehlen wir, den Code herunterzuladen, da er in der „sca-toolsuite“ unter Umständen weiterentwickelt und an neue Compiler und Bibliotheken angepasst werden wird.

Kapitel 3

Theorie

Dieses Kapitel enthält zwei Sätze, die auch ohne einen Algorithmus interessant sein können. Die Sätze sind an sich keine Algorithmen, da sie zwar skizzieren, welche Art von Konfigurationen für eine vollständige Simulation benötigt werden, aber keine Vorgaben machen, wie diese Konfigurationen genau aussehen oder wie sie zu entwickeln sind.

Im ganzen Kapitel wird angenommen, dass $N_{\text{readers}} = N_{\text{writers}}$ ist. Wir setzen $N \coloneqq N_{\text{readers}}$.

Bevor wir mit ihnen beginnen, sind noch zwei Definitionen erforderlich.

Definition 3.1. Seien eine Menge V und eine Relation $\succ$ auf V gegeben. Seien $(v_1, v_2) \in V^2$ und $M \subseteq V$. Wir definieren:

$$v_2 \in \mathfrak{c}(\succ, v_1) :\Leftrightarrow v_1 \succ v_2 \text{ (von englisch „children“).}$$

Darüber hinaus soll $\mathfrak{c}$ auch auf Mengen erlaubt sein:

$$\mathfrak{c}(\succ, M) = \bigcup_{c \in M} \mathfrak{c}(\succ, c).$$

Folgende Definition führt ein Symbol für Zellen ein, die sich nie ändern können, solange man nur eine gegebene Teilmenge $R_0 \subseteq R$ aktiviert.

Definition 3.2. Sei $(R, Q, \cdot, \cdot, \delta) \in \mathfrak{C}$. Sei $R_0 \subseteq R$ und $c \in Q^R$. Wir definieren:

- $c \in \dagger(R_0) :\Leftrightarrow (\forall c' \in Q^R \colon c \vdash_{R_0}^* c' \Rightarrow c' = c)$ und
- $\dagger \coloneqq \dagger(R)$.

Beispiel 3.1. Sei $(R, Q, \cdot) = \mathcal{C}_{\text{bc3}}$, so gilt:

- Falls $a_0 \subseteq R$ die zuletzt aktivierten Zellen sind, so ist $a_0 \in \dagger(a_0)$.
- Ist $c = 0 \in Q^R$, so ist $r \in \dagger$ für alle $r \in R$; und es ist $\mathfrak{c}(\vdash^*, c) = \mathfrak{c}(\vdash^0, c) = \{c\}$ und $\mathfrak{c}(\vdash, c) = \emptyset$.

3.1 N^2-Entwicklung

Wir werden nun folgende Situation haben: Gegeben sind $(R, Q, ...) \in \mathfrak{C}$, $(g_0, g) \in (Q^R)^2$ und $g_0 \vdash^* g$. Gesucht ist eine „praktische“ „Basis“ $\{M_i \subseteq Q^R\}_{1 \leq i \leq n}$, so dass $g_0 \vdash^* \{M_i\}_{1 \leq i \leq n} \vdash^* g$ gilt. Um dies zu erreichen werden wir R in disjunkte Zonen aufteilen und separat auf jeder einzelnen solange iterieren, bis diese entweder auf eine andere trifft, oder sie im Sinne des Symbols „†“ keine neuen Konfigurationen mehr annimmt.

Satz 3.3. *Sei $(R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathfrak{C}$. Sei weiter $R = \dot{\bigcup}_{i=0}^{n} R_i$ für $n \in \mathbb{N}$, wobei stets $N^2(R_i) \subseteq R_i \cup R_0$ für alle $1 \leq i \leq n$. R_0 ist also eine Art Abgrenzung der Dicke „N^2“. Sei nun $g_0 \in \dagger(R_0)$, und sei $g \in Q^R$ mit $g_0 \vdash^* g$ und eine der beiden folgenden Aussagen wahr:*

$$g|_{R_0} \neq g_0|_{R_0} \text{ oder } g \in \dagger(R \setminus R_0) \quad . \tag{3.1}$$

Wir definieren $M_i \subseteq Q^R$ für $1 \leq i \leq n$ wie folgt:

$$M_i \coloneqq \mathfrak{c}(\vdash^*_{R_i}, g_0) \cap ((Q^R \setminus \dagger(N(R_i))) \cup \dagger(R_i)).$$

Dann gilt:

$$\bigcup_{i=1}^{n} M_i \vdash^* g.$$

Beweis. Wegen $g_0 \vdash^* g$ gibt es eine Sequenz A von Aktivierungen auf R, die von g_0 zu g führt. Sei $m \in Q^R$ die letzte Konfiguration der Sequenz, die die Eigenschaft $m|_{R_0} = g_0|_{R_0}$ erfüllt. Nun machen wir eine Fallunterscheidung. Im ersten Fall sei $m = g$. Wegen Voraussetzung 3.1 ist nun $g \in \dagger(R \setminus R_0)$, insgesamt sogar $g \in \dagger$. Wir führen die Aktivierungen von g_0 bis m *zweimal* durch, wobei wir im ersten Durchgang ausschließlich Zellen aus R_1 und im zweiten Durchgang ausschließlich Zellen aus $R \setminus R_1$ aktivieren. Dies zeigt:

$$g_0 \vdash^*_{R_1} m \vdash^*_{R \setminus R_1} g.$$

Da $m \in \dagger(R_i)$ ist gilt auch $m \in M_i$. Also erhalten wir:

$$g_0 \vdash^*_{R_1} M_1 \vdash^*_{R \setminus R_1} g.$$

Nun zum zweiten Fall. Sei dazu $R' \subseteq R_0$ die Teilmenge der nach m aktivierten Zellen, die in R_0 liegt. Sei ohne Einschränkung $R_1 \subseteq N(R')$. Wieder führen wir die Aktivierungen von g_0 bis m *zweimal* durch, wobei wir im ersten Durchgang ausschließlich Zellen aus R_1 und im zweiten Durchgang ausschließlich Zellen aus $R \setminus R_1$ aktivieren. Die anschließenden Aktivierungen von A gehen wir ohne Änderungen zu g weiter. Daran sieht man:

$$g_0 \vdash^*_{R_1} m \vdash^* g.$$

Wegen $m \in Q^R \setminus \dagger(N(R_i))$ ist wieder $m \in M_i$, und es gilt analog:

$$g_0 \vdash_{R_1}^* M_1 \vdash_R^* g.$$

□

Welchen Vorteil bringt dieser Satz? Unsere „Basis“ $\{M_i\}_{1 \leq i \leq n}$ hat den Betrag

$$|\bigcup_{i=1}^{n} M_i| = \sum_{i=1}^{n} |M_i|.$$

Ohne Kenntnis dieses Satzes müsste man alle Kombinationen von Konfigurationen auf den $\{R_i\}_{1 \leq i \leq n}$ berechnen, was zu

$$\prod_{i=1}^{n} |M_i|$$

Konfigurationen führen würde. Man spart also durch Satz 3.3 exponentiell viele Konfigurationen in n.

Man beachte übrigens, dass man im Beweis $N^2(R_i)$ nicht mehr durch $N^1(R_i)$ ersetzen kann. Ein Gegenbeispiel ist bereits in einem simplen Automaten mit Raum und Zustandsmenge $\{0, 1, 2\}$ leicht konstruierbar.

Jedoch gilt für den Fall, dass die R_i nur der Bedingung $N(R_i) \subseteq R_i \cup R_0$ für alle $1 \leq i \leq n$ genügen, folgende, schwächere Aussage.

3.2 N^1-Entwicklung

Satz 3.4. *Sei $(R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathfrak{C}$. Sei weiter $R \coloneqq \dot{\bigcup}_{i=0}^{n} R_i$ für $n \in \mathbb{N}$, wobei stets $N(R_i) \subseteq R_i \cup R_0$ für alle $1 \leq i \leq n$. R_0 ist also eine Art Abgrenzung der Dicke „N^1“.*

Sei nun $g_0 \in \dagger(R_0)$. Man definiere ein g mit $g_0 \vdash^+ g$ und außerdem $M_i \subseteq Q^R$ für $1 \leq i \leq n$ wie folgt:

$$M_i \coloneqq \mathfrak{c}(\vdash_{R_i}^1)$$

Dann gilt:

$$\bigcup_{i=1}^{n} M_i \vdash^* g.$$

Beweis. Sei ohne Einschränkung $g_0 \vdash_{R_1}^1 M_1 \vdash_R^* g$. Daraus folgt schon die Aussage. □

Der Satz hat im Vergleich zu seinem Vorgänger den Nachteil, dass man für die $\{M_i\}_{1 \leq i \leq n}$ auf den Regionen $\{R_i\}_{1 \leq i \leq n}$ nur eine Iteration des Automaten ausführen darf. Zwei Iterationen auszuführen ist, wie bereits erwähnt, nicht akzeptabel.

Wieder spart man exponentiell viele Konfigurationen in n. Der Unterschied ist, dass man nur auf einer Iteration exponentiell viele Konfigurationen spart, wohingegen es bei Satz 3.3 möglich war, mehrere Iterationen mit einem Mal zu „erschlagen“.

Kapitel 4

Algorithmen allgemein

Dieses Kapitel wird zunächst einige Begriffe klären und Anforderungen an Algorithmen stellen. Als nächstes wird es eine Einteilung für verschiedene Arten von Algorithmen gegeben. Schließlich stellen wir ein Algorithmen-Grundgerüst und drei simple Algorithmen vor.

4.1 Allgemeines

Es seien in diesem Abschnitt $C = (R, Q, ...) \in \mathfrak{C}$, $S \in \mathfrak{S}_{\text{async}}(C)$ und eine Menge von Anfangskonfigurationen $Q_0 \subseteq Q^R$ gegeben.

4.1.1 Begriffe

Definition 4.1 (Berechnungsgraph)**.** Der **Berechnungsgraph** $\mathcal{B}_{C,S}(Q_0) = (V, \succ)$ sei als derjenige Graph definiert, den ein Algorithmus für die Eingabe (C, S, Q_0) durchläuft. Mit „durchlaufen“ sind Knoten gemeint, die in keinster Weise aussortiert werden und die auch keinen Kreis schließen. Weiter sei M eine Menge und es gelten mindestens folgende Anforderungen:

- $V = V(\mathcal{G}_{C,S}(Q_0)) \times M$
- $(v_1, m_1) \succ_{\mathcal{B}} (v_2, m_2) \Rightarrow v_1 \succ_{\mathcal{G}} v_2$

Man sieht, dass der Berechnungsgraph durch die Menge M auch zusätzliche Informationen speichern, und außerdem die gleiche Konfiguration mehrfach beinhalten kann. Dies führt direkt zur Definition eines Duplikats.

Definition 4.2 (Duplikat)**.** Ein **Duplikat** ist ein Knoten $(v_1, m_1) \in V(\mathcal{B})$, so dass es einen Knoten $(v_2, m_2) \in V(\mathcal{B})$ gibt, so dass für jedes $(v_3, m_3) \in V(\mathcal{B})$ gilt:

$$(v_1, m_1) \succ^* (v_3, m_3) \Leftrightarrow \exists m_4 \in M : (v_2, m_2) \succ^* (v_3, m_4).$$

Beim Erreichen von Duplikaten lohnt es sich üblicherweise nicht, die vollständige Simulation fortzusetzen, da die dort folgenden Konfigurationen schon berechnet sind. Duplikate sind schwer zu erkennen; und, wie man hier sieht, können sie sogar unterschiedliche Konfigurationen haben. Es ist geradezu eine Hauptaufgabe aller Algorithmen, Duplikate frühzeitig zu erkennen.

Für die Visualisierung ist der Berechnungsgraph meist zu groß, daher führen wir nun Resultatgraphen ein.

Definition 4.3 (Resultatgraph)**.** Es sei $Q' \subseteq Q^R$ eine feste Menge von Endrepräsentanten bezüglich Q_0. Diese könnte beispielsweise durch einen Algorithmus gewählt werden. Die Menge der **Resultatgraphen** $\mathcal{R}_{C,S}(Q_0, Q')$ ist dann eine Menge von Graphen, in der jeder Graph $R \in \mathcal{R}_{C,S}(Q_0, Q')$ folgende Eigenschaften hat:

- $V(R) \subseteq V(\mathcal{G}_{C,S}(Q_0))$.
- $E(R) \subseteq E(\mathcal{G}_{C,S}(Q_0))$.
- R besteht aus einer Menge von eindeutigen Pfaden, die von jedem Element aus Q_0 zu jedem jeweils erreichbaren Element aus Q' führt.

Der **kurze Resultatgraph** $\mathcal{R}'_{C,S}(Q_0, Q')$ ist für ein beliebiges $R \in \mathcal{R}_{C,S}(Q_0, Q')$ definiert als

$$\mathcal{R}'_{C,S,A}(Q_0, Q') = (Q_0 \cup Q', E(R)|_{Q_0 \cup Q'}).$$

Während also der eindeutige kurze Resultatgraph eher eine „Funktionstabelle“ liefert, d.h. eine Relation der Art „welche Ausgabe(n) erhält man für welche Eingabe(n)“, so erweitern die Resultatgraphen dies um die Information, auf welche Weise man zu diesen Ausgaben gelangen kann.

4.1.2 Eingabe und Ausgabe

Oftmals sind folgende Ausgaben von Interesse:

1. Der kurze Resultatgraph $\mathcal{R}'_{C,S}(Q_0, Q')$, sofern $Q' \subseteq Q^R$ eine Menge von Endrepräsentanten zu Q_0 ist. Er kann tabellarisch dargestellt werden.

2. Ein Resultatgraph aus $\mathcal{R}_{C,S}(Q_0, Q')$; Q' wie vorher. Hier kann man bestimmte Gütekriterien fordern. Beispielsweise könnte man an einem Repräsentanten mit minimaler oder maximaler Länge interessiert sein. Allerdings kann man einen Pfad oft so umbauen, dass er bezüglich anderer Gütekriterien besser wird. Man könnte etwa einen langen Pfad verkürzen, indem man, soweit möglich, Aktivierungen unabhängiger Zellen gleichzeitig durchführt.

3. Alle Zellen, die sich während der vollständigen Simulation ändern könnten. Dies folgt nicht aus der Ausgabe von Punkt 1.
4. Alle möglichen Endkomponenten in $\mathcal{G}_{C,S}(Q_0)$.
5. Den Abhängigkeitsgraphen $\mathcal{D}_{C,S}(Q_0)$.
6. Der Berechnungsgraph $\mathcal{B}_{C,S}(Q_0)$. Dieser kann sehr groß werden.

Wie wir sehen werden, wurden alle Punkte implementiert. Nur der Berechnungsgraph $\mathcal{B}_{C,S}(Q_0)$ wurde aufgrund seiner Größe nur als Debug-Feature eingebaut.

4.1.3 Beispiel

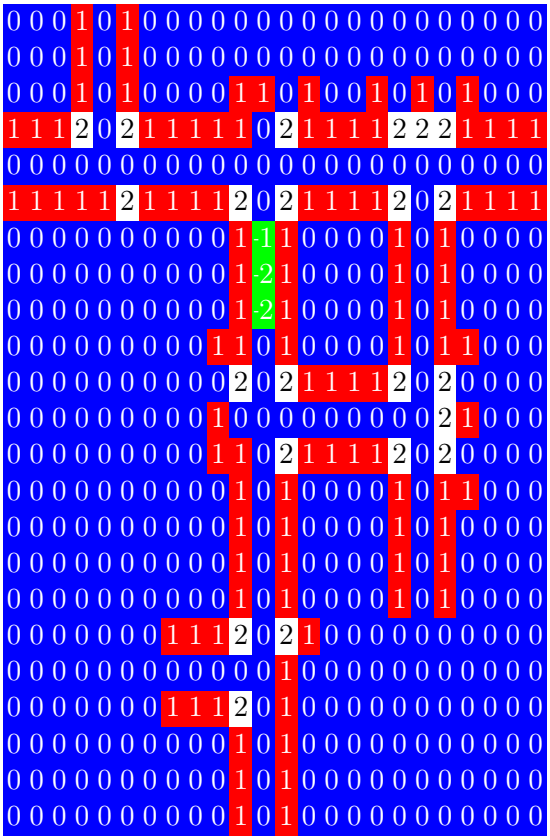

Tabelle 4.1: Gitterbelegung zum Modul „Kreis“

-1
0

Tabelle 4.2: Initiale Ausgabezonen zum Modul „Kreis“

-2 2		-1	scc-size
in		out	
2 1	$\mapsto$	2	≥ 39

Tabelle 4.3: Funktionstabelle zum Modul „Kreis“

Konfigurationen insgesamt:	532
davon nicht bewegbar:	271
davon nicht isolierbar:	137
Laufzeit (s):	<1

Tabelle 4.4: Statistiken zum Modul „Kreis“

Im Folgenden zeigen wir ein paar Beispiele, wie Ein- und Ausgaben aussehen. Abbildungen 4.1, 4.2, 4.3 und 4.4 zeigen die sogenannten „Ein-“ und „Ausgabezonen“, deren „Funktion“ und Statistiken für ein simples Beispiel für $\mathcal{C}_{\text{bc3}}$. Dies ist das Ausgabeformat, welches wir in Kapitel 8 („Ausgaben“) primär verwenden.

Die erste Abbildung gibt an, wo die „Ein-“ und „Ausgabezonen“ liegen. Sie sind mit negativen Zahlen gekennzeichnet und grün gefärbt. Ihre Berechnung geschieht wie folgt.

1. Zunächst werden die Eingabezonen direkt vom Nutzer vorgegeben. Der Nutzer muss dafür sorgen, dass alle variablen Zellen in den Eingabezonen enthalten sind. Grund dafür, die Eingabezonen nicht automatisch zu erkennen ist, dass der Nutzer möglicherweise auch invariable Zellen in die Eingabezonen mit

aufnehmen möchte. Dies könnte vorteilhaft sein, um zwei Eingabezonen absichtlich zusammenzulegen oder zu trennen, oder um irgendetwas anderes zu illustrieren.

2. Nun werden Zonen, die sich in irgendeinem Endrepräsentanten von einer entsprechenden Anfangskonfiguration unterscheiden, zusätzlich als Ausgabezonen markiert. Damit können nun Zellen gleichzeitig in einer Eingabe- und in einer Ausgabezone liegen.

3. Als nächstes werden strenge Zusammenhangskomponenten bezüglich der Nachbarschaft $\mathcal{N}_1^2$ aus den Zellen berechnet, die entweder in Eingabe- oder Ausgabezonen liegen. Dabei werden jedoch speziell Zellen als nicht zusammenhängend gewertet, wenn beide unterschiedlichen Typs sind und eine von ihnen vom Typ „Ausgabe und nicht Eingabe“ ist. Dies hat zur Folge, dass reine Ausgabezonen abgetrennt werden.

4. Eine strenge Zusammenhangskomponente hat nun entweder nur Zellen vom Typ „Ausgabe“ oder eine gewisse Anzahl Zellen vom Typ „Eingabe“ und den Rest vom Typ „Ein-Ausgabe“. Im zweiten Fall wird die strenge Zusammenhangskomponente genau dann zum Typ „Ein-Ausgabe“, falls sie mindestens eine Zelle dieses Typs beinhaltet, ansonsten ist ihr Typ „Eingabe“.

Die Wahl der Nachbarschaft und der Algorithmus, um Ein- und Ausgabezonen festzulegen, ist eher praktisch; man könnte hier viele andere Wahlen treffen. Unsere Wahl für $\mathcal{N}_1^2$ basiert darauf, dass diese Nachbarschaft eine intuitive und leicht darstellbare Form von „Zusammenhang“ im $\mathbb{Z}^2$ darstellt. Der Algorithmus zur Festlegung der Zonen versucht dann, Zellen vom Typ „Eingabe“ und „Ein-Ausgabe“ möglichst zu vereinen, aber dabei letztere beide Typen von Ausgabezonen zu separieren. Den genauen Code kann man in src/base/**eval**.cpp nachlesen.

Abbildung 4.2 gibt nun die Zustände der Ausgabezonen in den Anfangskonfigurationen an, wobei die erste Zeile der Tabelle die jeweilige Zone aus Abbildung 4.1 angibt und die zweite Zeile die Zustände. Die Zustände sind für alle Anfangskonfigurationen gleich und sollten daher eigentlich direkt in Abbildung 4.1 dargestellt werden. Leider wäre dies graphisch schwer zu realisieren, daher erhalten die Ausgabezonen eine separate Tabelle.

Abbildung 4.3 nennen wir **Funktionstabelle**. Sie ist äquivalent zu $\mathcal{R}'_{C,S,A}(Q_0)$ und funktioniert ähnlich wie Abbildung 4.2: Wieder sind Komponenten und entsprechende Zustände angegeben. Zusätzlich gibt die zweite Zeile an, um welchen Typ von Zelle es sich handelt. Dabei sind die Zonen links der Pfeile von den Typen „i/o“ („Ein-Ausgabe“) oder „in“ („Eingabe“), rechts der Pfeile von den Typen „i/o“ oder „out“ („Ausgabe“). Zusätzlich wird ganz rechts die Größe oder Mindestgröße der Endkomponente angegeben. Man liest also hier Folgendes ab: Versetzt man zu

Beginn die eindeutige Eingabezone in die Zustände „2“ und „1“, wobei nach Tabelle 4.2 die darüber liegende Zelle im Zustand „0“ ist, so erhält man eine Endkomponente mit mindestens 66 Konfigurationen, wobei ein Endrepräsentant wie die Anfangskonfiguration aussieht, bei der jedoch die Ausgabezelle im Zustand „2“ ist. Betrachtet man beide Zonen zusammen, so könnte man umgangssprachlich sagen: Aus „0,2,1“ wird „2,2,1“.

Man sieht hier, dass sich der Algorithmus tatsächlich bemüht hat, den Endrepräsentanten ähnlich der Anfangskonfiguration zu wählen. Da aber die Anfangskonfiguration selbst nicht als Repräsentant gewählt wurde, *obwohl* sie Teil der Endkomponente ist[1], so könnte dies den Anschein erwecken, dass die Anfangskonfiguration fälschlicherweise nicht in der Endkomponente enthalten wäre. Dies ist natürlich nicht wahr. Der Algorithmus wählt die Anfangskonfiguration nicht, da er sie hier aufgrund von Abkürzungen in der Berechnung nie wieder erreicht. Würde man die Berechnung noch einmal mit der derzeitigen Endkonfiguration als Anfangskonfiguration starten, so wären Anfangs- und Endkonfiguration identisch.

Abbildung 4.3 gibt schließlich einige Statistiken an. Der benutzte Algorithmus ist immer unser schnellster, nämlich der Greedy-Algorithmus. Nachdem man Kapitel 6 gelesen hat, so wird man die Bezeichnungen für die Konfigurationen verstehen. Die Laufzeiten sind auf einem im Jahr 2013 top-aktuellen Personal Computer berechnet worden. Details liest man im Kapitel 7.2 („Implementierung“).

Eine weitere Art der Ausgabe ist der Abhängigkeitsgraph $\mathcal{D}$, den wir in Abbildung 4.1 sehen. Die Beschriftung der Knoten gibt die entsprechenden Koordinaten im Format „x y“ an, wobei die Zelle links oben im Bild die Koordinaten „0 0“ hat. Man erkennt klar die 4 „Ecken“ und das unterschiedliche Verhalten der variablen Zellen an dieser Stelle. Dieser Graph ist für die meisten Automaten nicht all zu spektakulär. Da er auch viel Platz weg nimmt, vermeiden wir ihn in diesem Dokument.

[1] Man überlege sich, dass die Anfangskonfiguration tatsächlich in dieser Endkomponente liegen muss.

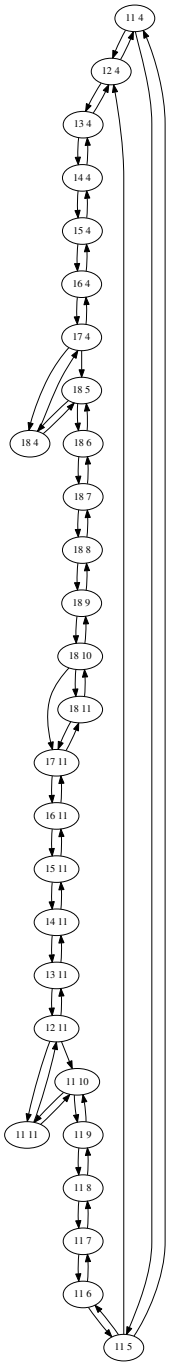

Abbildung 4.1: Der entsprechende Abhängigkeitsgraph.

Man kann sich auch den Teil des Berechnungsgraphen $\mathcal{B}$ ausgeben, der Resultate

liefert. Man sieht ihn in in Abbildung 4.2. Da die entsprechende Endkomponente kein Bestandteil des Graphen ist, besteht dieser nur aus zwei Knoten: Der Anfangskonfiguration und dem Endrepräsentanten.

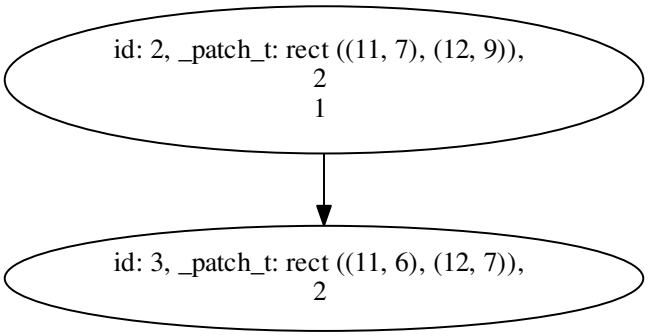

Abbildung 4.2: Der für Ergebnisse relevante Teil des Berechnungsgraphen.

4.1.4 Laufzeit gegen Speicher

Die Verhinderung von Duplikaten ist eine Hauptaufgabe für alle Algorithmen, denn Duplikate können zu exponentieller Laufzeit führen. Wir haben zwei Ansätze gefunden:

1. Ausreichend viele Konfigurationen dauerhaft speichern. Falls eine Konfiguration bereits gespeichert ist, so wurden alle deren Folgekonfigurationen durchlaufen.

2. Eine Technik, die automatisch viele oder alle Duplikate vermeidet, ohne Knoten zu speichern.

Der erste Ansatz mag auf den ersten Blick leichter zu implementieren scheinen. Er hat aber einige Nachteile:

- Man lernt nicht wirklich viel über den Automaten. Meldet das Programm während der Berechnung, dass ein Knoten doppelt vorkommt, so kann der Anwender oft kaum nachvollziehen, was der Grund für dieses Duplikat ist, oder welche weiteren Duplikate noch auftreten werden.

- Der Speicherverbrauch kann enorm ansteigen. Hat man nur 100 Bytes pro Knoten, was in der Praxis eher wenig ist, und nur eine Milliarde Knoten, so werden 100 Gigabyte Speicher belegt. Das ist im Jahre 2014 auf vielen Rechnern

impraktikabel. Im Vergleich dazu kann man eine Milliarde Knoten meist in Bruchteilen von Sekunden abarbeiten[2].

Nichtsdestotrotz müssen wir immer alle Knoten auf dem Pfad von der Anfangskonfiguration zum aktuellen Knoten im Speicher halten, um Kreise im Graphen zu erkennen. Dieser Speicherverbrauch fällt typischerweise nicht ins Gewicht.

4.1.5 Einteilung

Wir benutzen folgende Unterteilung für Algorithmen. In der Beschreibung verwenden wir den Begriff „Komponenten“. Damit ist hier immer eine Menge aktiver Zellen gemeint, die bezüglich einer sinnvollen Nachbarschaft streng zusammenhängen.

Blinde Algorithmen entwickeln Komponenten unabhängig in der Hoffnung, dass sie nie aufeinander treffen. Falls dies dennoch geschieht, so fehlen eventuell Information und die Berechnung muss gegebenenfalls für eine der Komponenten neu gestartet werden.

Vorausschauende Algorithmen sehen voraus, dass Komponenten beliebig aufeinander treffen können.

Asymmetrische Algorithmen entwickeln mehrere Komponenten asymmetrisch in der Hinsicht, dass nur Entwicklungen verfolgt werden, bei denen sich nur eine Komponente ändert, aber keine, bei der dies mehrere in einer Iteration tun.

Symmetrische Algorithmen entwickeln zusätzlich zu den asymmetrischen auch mehrere Komponenten in einer Iteration.

4.2 Brute-Force-Algorithmus

Im ganzen Kapitel seien ein Automat $C = (R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathfrak{C}$ und Anfangskonfigurationen $Q_0 \subseteq Q^R$ gegeben.

4.2.1 Grundgerüst

Der Brute-Force-Algorithmus ist ein vorausschauender, symmetrischer Algorithmus, der eine Tiefensuche durch den noch zu berechnenden Graphen aller Konfigurationen durchführt. Er ist an sich sehr langsam, da er keine Duplikate erkennt. So kann das

[2] Man kann nicht immer Speicher mit Knoten aufwiegen, da sich durch gefundene Duplikate die Anzahl der noch zu berechnenden Knoten stark verringern kann. Allerdings kann man dies mit einer guten Technik meist wieder ausgleichen.

Simulieren einer kurzen „Röhre“ in $\mathcal{C}_{\text{bc3}}$ schon auf einem im Jahre 2014 modernen Rechner mehrere Minuten dauern. Trotzdem liefert der Algorithmus ein praktisches Gerüst und ein gutes Beispiel zur Einführung.

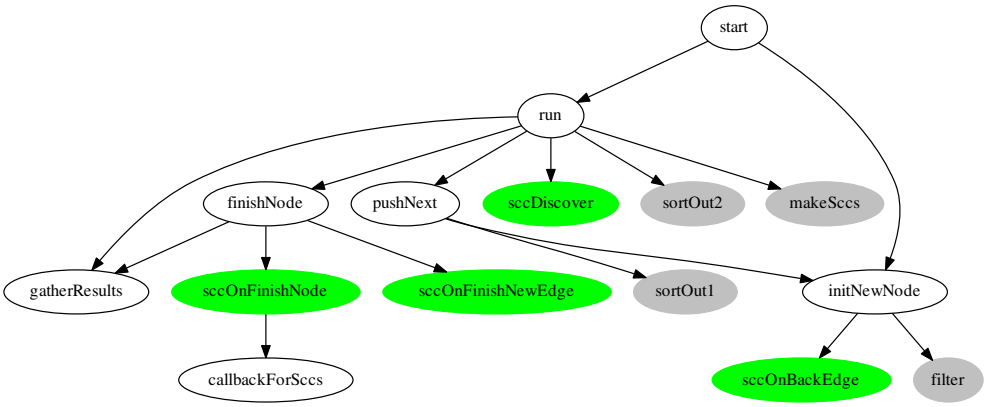

Abbildung 4.3: Abhängigkeiten unter den Funktionen für das Brute-Force-Grundgerüst. Eine gerichtete Kante bedeutet, dass die Quelle das Ziel aufruft.

Algorithmus 1 Standard-Verhalten

```
1:  procedure FILTER(P_variable, P_activated)
2:      return P_variable
3:  end procedure
4:  procedure MAKESCCS(P)
5:      return P
6:  end procedure
7:  procedure SORTOUT1(P_lastactivated, P_activated, V_oldconf)
8:      return false
9:  end procedure
10: procedure SORTOUT2(P_activated, P_unused, i_src)
11:     return false
12: end procedure
```

Wir beginnen mit einer Übersicht. Wie man in Abbildung 4.3 sieht, gibt es 3 Typen von Funktionen:

Transparente Knoten geben Funktionen an, die wir im Folgenden direkt erläutern. Dies ist das Brute-Force-Grundgerüst.

Grau-gefärbte Knoten sind Funktionen, die es noch im zu implementieren gilt. Wir werden später Algorithmen mit verschiedenen Implementierungen dieser Funktionen zeigen. Die Bedeutung der Knoten ist wie folgt:

- SORTOUT1 zum Aussortieren von Kindknoten, bei denen man schon im aktuellen Knoten weiß, dass sie nur zu Duplikaten führen.
- SORTOUT2 zum Aussortieren von Knoten, bei denen man erst zu Beginn deren Bearbeitung weiß, dass sie nur zu Duplikaten führen.
- MAKESCCS zum Unterteilen der aktiven Zellen in strenge Zusammenhangskomponenten.
- FILTER zum Filtern der aktivierten Zellen.

Falls eine dieser Funktionen nicht definiert ist, so nimmt sie das Standard-Verhalten an, das wir in Algorithmus 1 angeben. Speziell für den Brute-Force-Algorithmus definieren wir keine neuen Funktionen. Damit nutzt er genau die Funktionen aus Algorithmus 1.

Grün-gefärbte Knoten, deren Funktionen mit „scc“ beginnen , sind Teile von Tarjans Algorithmus zum Erkennen strenger Zusammenhangskomponenten [Tar72]. Wir lassen ihn neben der eigentlichen Berechnung parallel laufen, um Kreise in $\mathcal{G}(Q_0)$ zu erkennen. Das funktioniert, da die Algorithmen eine Tiefensuche durchführen. Unsere Implementierung zu Tarjans Algorithmus kann man in `src/utils/scc_algo.h` nachlesen.

Grob kann man zu Abbildung 4.3 sagen, dass die Funktion START ein Start-Element auf den Stack legen wird, wofür es sich INITNEWNODE bedienen wird. Dann wird sie RUN aufrufen, wo eine stackbasierte Tiefensuche abläuft. Dort werden ebenfalls Knoten mittels INITNEWNODE erzeugt, oder sie werden mit FINISHNODE aufgeräumt. Die Funktion GATHERRESULTS wird nach der Rückkehr von einem jeden Kindknoten in der Tiefensuche Resultate an den Elternknoten zurückgeben.

Wir gehen nun die Funktionen Stück für Stück durch. Der genaue Code befindet sich vollständig in `src/base/brute_force.h` .

Algorithmus 2 Start

```
1: procedure START(Q_0)
2:     (S, R_complete) ← ∅
3:     for all q_0 ∈ Q_0 do
4:         (V_dict, D, R_last) ← ∅^3
5:         V_sim ← q_0
6:         for all p ∈ D(q_0) do                    ▷ für jede Zelle der Eingabe
7:             V_cur[p] ← V_sim[p + N_out(p)]
8:             δ_p ← δ(p)
9:             if δ_p(V_sim[N_in(p)]) ≠ V_sim[N_out(p)] then
10:                P_variable ← P_variable ∪ {p}
11:            end if
12:        end for
13:        V_prev ← V_cur
14:        i_next ← 2
15:        INITNEWNODE(P_variable, ∅, 1, P_variable, false)
16:        R_complete ← R_complete ∪ RUN
17:    end for
18:    return R_complete
19: end procedure
```

Beginnen wir also mit der Funktion START (Algorithmus 2). Sie erhält Q_0 als Eingabe und soll für Endrepräsentanten $Q' \subseteq Q_R$ einen Resultatgraph aus $\mathcal{R}_{C,S}(q_0, Q')$ ausgeben. Wie man sieht, werden primär globale Variablen initialisiert:

- S ist der Stack für die Rekursion. Er ist zu Beginn leer. S ist eine Menge von Tupeln mit Elementen in folgender Reihenfolge.
  - i_{cur} ist eine ID für den aktuellen Knoten.
  - i_{src} ist die ID des Elternknoten.
  - Δ_{grid} ist immer ein Patch von der Anfangskonfiguration q_0 zur aktuellen Konfiguration.
  - P_{variable} ist die Menge aller variablen Zellen in der aktuellen Konfiguration.
  - P_{unused} ist die Menge aller Zellen, die in der vorherigen Iteration variabel waren, aber nicht aktiviert wurden.
  - Δ_{cur} ist der Patch von V_{cur} zum Elternknoten.
  - V_{sccs} ist ein Vektor von Zonen, der die strengen Zusammenhangskomponenten aus P_{variable} speichert.

  - i_{nextscc} ist die strenge Zusammenhangskomponente, die beim nächsten mal behandelt wird, wenn der aktuelle Knoten wieder $\max(S)$ ist.
  - $i_{\text{nextbitmask}}$ ist die nächste Bitmaske, die die als nächstes zu aktivierenden Knoten in $V_{\text{sccs}}[i_{\text{nextscc}}]$ wählt.
  - $c_{\text{parentedge}}$ wird nur an einer Stelle gebraucht und erst dort erklärt.
  - $\mathcal{R}_{\text{last}} \subseteq V(\mathcal{R}_{C,S}(q_0, Q')) \times \mathcal{P}(V(\mathcal{R}_{C,S}(q_0, Q'))) \times \mathbb{N} \times \{0, 1\}$, wobei Q' die Endkomponenten sind. Das zweite Argument sind die Kindknoten von $\mathcal{R}_{\text{last}}$ in $\mathcal{R}_{C,S}(q_0, Q')$. Die beiden letzten Argumente geben die Größe der strengen Zusammenhangskomponente und deren Exaktheit, wobei 0 für „$\geq$“ und 1 für „$=$“ steht, an. Sie haben nur eine Bedeutung, falls die Konfiguration von $\mathcal{R}_{\text{last}}$ in Q' liegt. Außerdem ist immer $|\mathcal{R}_{\text{last}}| \in \{0, 1\}$, damit wir einen Begriff dafür haben, dass es aktuell ein Resultat gibt oder nicht.

- $\mathcal{R}_{\text{complete}}$ ist ein Vektor von Resultatgraphen des Typs von $\mathcal{R}_{\text{last}}$.
- V_{dict} ist eine bidirektionale Map. Der erste Schlüssel sind Patches, die relativ zu q_0 sind; der zweite Schlüssel sind die Knoten-IDs. Je nachdem, welchen Typs ein Schlüssel k ist, so ist $V_{\text{dict}}[k]$ vom anderen Typ.
- $V_{\text{sim}} \in Q^R$ enthält immer die aktuelle Konfiguration in der Simulation.
- Die Variablen $(V_{\text{cur}}, V_{\text{prev}}) \in (Q^{N_{\text{out}}(r_1)}, Q^{N_{\text{out}}(r_2)}, ...)^2$ enthalten für jede Zelle $r \in R$ die neuen Zustände in $N_{\text{out}}(r)$ durch Aktivierung von r; dies können also durchaus mehrere sein, falls $|N_{\text{out}}(r)| > 1$. Dabei ist $V_{\text{cur}}(r) = \delta(r)(V_{\text{sim}}(N_{\text{in}}(r)))$; V_{prev} bezieht sich hingegen auf den Elternknoten. Dies erklärt auch, wieso V_{cur} *vor* (A. 3, Z. 13) und V_{prev} *nach* (A. 3, Z. 30) dem Erstellen neuer Knoten aktualisiert wird. Dies wird später noch klarer werden. Die Hauptfunktion von V_{cur} und V_{prev} ist es, ein doppeltes Berechnen des nächsten Zustands einer Zelle zu vermeiden, da diese Operation bezüglich der Laufzeit sehr teuer ist.

Wir möchten den ersten Knoten für q_0 möglichst simpel initialisieren. Gleichzeitig benötigen wir einen validen Vorgänger zu q_0 und müssen V_{cur} und V_{prev} passend setzen. Um all dies zu erreichen, initialisieren wir einen Knoten mittels initNewNode für q_0, dem wir vortäuschen, dass er einen Vorgänger mit q_0 hätte und trotzdem alle variablen Zellen aktiviert worden seien. Hätten wir angegeben, dass keine Zelle aktiviert worden sei, so hätten einige Algorithmen, wie man noch sehen wird, die „Kante von q_0 zu q_0“ als ineffizient betrachtet und daher die Berechnung sofort abgebrochen.

Die Details zu initNewNode verschieben wir ans Ende.

Algorithmus 3 Stack-Verhalten

```
1: procedure RUN
2:     while S ≠ ∅ do
3:         (..., V_sccs, i_nextscc, i_nextbitmask, ·, ·) ← max(S)
4:         c_first ← i_nextscc = −1 and i_nextbitmask = 0
5:         if !c_first and i_nextscc ≥ |V_sccs| then
6:             FINISHNODE
7:         else
8:             (·, ·, Δ_grid, ·, ·, Δ_cur, ...) ← max(S)
9:             V_sim ← V_sim + Δ_grid
10:            if c_first then
11:                (i_cur, i_src, ·, P_variable, P_unused, ...) ← max(S)
12:                SCCDISCOVER(i_cur)
13:                V_cur ← V_cur + Δ_cur
14:                if !SORTOUT2(P_variable \ P_unused, P_unused, i_src) then
15:                    if P_variable = ∅ and δ(R)(N_in(R)) = N_out(R) then
16:                        R_cur ← (Δ_grid, ∅, 1, 1)
17:                    else
18:                        V_sccs ← MAKESCCS(P_variable)
19:                        i_nextbitmask ← 2^|V_sccs[0]| − 1
20:                    end if
21:                end if
22:                i_nextscc ← i_nextscc + 1
23:            else
24:                V_prev ← V_prev − Δ_cur
25:                GATHERRESULT
26:            end if
27:            if |V_sccs| then
28:                PUSHNEXT
29:            end if
30:            V_prev ← V_prev + Δ_cur
31:            V_sim ← V_sim − Δ_grid
32:        end if
33:    end while
34:    return R_last
35: end procedure
```

Nun zur Funktion RUN (A. 3). Betrachtet man die äußere Schleife und das enthaltene if-else-Konstrukt, so erkennt man eine typische Rekursion, die durch einen Stack realisiert wurde. Neue Knoten durchlaufen, solange es noch Kindknoten zu

erzeugen gibt, den else-Zweig (Z. 7). Anderenfalls wird der if-Zweig (Z. 5) genommen, um den aktuellen Knoten zu entfernen und um „aufzuräumen“.

Das Erstellen neuer Knoten in RUN ist nun schnell erklärt. Zunächst wird das Resultat des letzten Knotens aufgesammelt, falls der aktuelle Knoten nicht das erste mal vom Stack S geholt wurde (Z. 25). War es doch das erste mal (Z. 10), so werden Initialisierungen durchgeführt. Das beinhaltet eine Prüfung durch die Routine SORTOUT2. Hält der Knoten dieser Prüfung stand, so gibt es nun entweder variable Zellen, oder es gibt sie nicht. Im ersteren Fall führen wir Initialisierungen für die strengen Zusammenhangskomponenten mittels MAKESCCS durch (Z. 18). Im zweiteren Fall fügen wir dem Resultatgraphen einen Knoten hinzu, der keine Kindknoten hat und eine Endkomponente mit genau einer Endkonfiguration darstellt (Z. 16). Schlussendlich rufen wir PUSHNEXT auf, falls es überhaupt einen nächsten Knoten gibt. Den Rest, also die Konfigurationen und Patches, hatten wir schon in der Auflistung der globalen Variablen erklärt.

Algorithmus 4 Vorbereitung des nächsten Knotens

1: **procedure** PUSHNEXT
2: $(i_{\text{cur}}, ..., i_{\text{nextscc}}, i_{\text{nextbitmask}}, c_{\text{parentedge}}, \cdot) \leftarrow \max(S)$
3: $P_{\text{activated}} \overset{i_{\text{nextbitmask}}}{\leftarrow} \mathcal{P}(P_{\text{variable}}{}')$ ▷ siehe Fußnote[3]
4: $V_{\text{sim}}{}' \leftarrow V_{\text{sim}}$
5: $V_{\text{sim}}[N_{\text{out}}(P_{\text{activated}})] \leftarrow V_{\text{cur}}[P_{\text{activated}}]$
6: **if** $\forall (p_1, p_2) \in P_{\text{activated}}{}^2 : N_{\text{out}}(p_1) \cap N_{\text{out}}(p_2) = \emptyset$ **then**
7: $c_{\text{so1}} \leftarrow \text{SORTOUT1}(P_{\text{variable}} \setminus P_{\text{unused}}, P_{\text{activated}}, V_{\text{sim}}{}')$
8: $c_{\text{parentedge}} \leftarrow c_{\text{parentedge}}$ **or** c_{so1}
9: **if** c_{so1} **then**
10: INITNEWNODE$(P_{\text{variable}}{}', \Delta_{\text{grid}}, i_{\text{cur}}, P_{\text{activated}}, \text{true})$
11: **end if**
12: **end if**
13: $V_{\text{sim}} \leftarrow V_{\text{sim}}{}'$
14: $i_{\text{nextbitmask}} \leftarrow i_{\text{nextbitmask}} - 1$
15: **if** $i_{\text{nextbitmask}} = 0$ **then**
16: $i_{\text{nextscc}} \leftarrow i_{\text{nextscc}} + 1$
17: **if** $i_{\text{nextscc}} < |V_{\text{sccs}}|$ **then**
18: $i_{\text{nextbitmask}} \leftarrow |V_{\text{sccs}}[i_{\text{nextscc}}]|$
19: **end if**
20: **end if**
21: **end procedure**

[3] Das i-te ($1 \leq i \leq |P_{\text{variable}}{}'|$) Bit von $i_{\text{nextbitmask}}$ bestimmt, ob Teilmenge Nummer i aus $\mathcal{P}(P_{\text{variable}}{}')$ nach $P_{\text{activated}}$ aufgenommen wird.

Nun also zu PUSHNEXT (A. 4). Zu Beginn werden Zellen aus der Potenzmenge der variablen Zellen mittels der Bitmaske $i_{\text{nextbitmask}}$ aktiviert. Dabei sind aber nur solche Kombinationen erlaubt, bei denen alle aktivierten $(r_1, r_2) \in R^2$ die Eigenschaft $N_{\text{out}}(r_1) \cap N_{\text{out}}(r_2) = \emptyset$ erfüllen (Z. 6), da der Simulator sonst nicht gültig gemäß Definition 2.11 wäre. Zusätzlich kann die Implementierung eines Algorithmus mittels SORTOUT1 die Kombinationen variabler Zellen filtern. Passiert eine Kombination jedoch all diese Barrieren, so wird für sie ein neuer Knoten mittels INITNEWNODE angelegt. Der Rest der Funktion ist weniger interessant; hier werden einfach die nächste passende strenge Zusammenhangskomponente und Bitmaske gewählt.

Man sieht hier auch, welche Bedeutung die Variable $c_{\text{parentedge}}$ hat. Sie wird wahr, wenn irgendein Kindknoten des aktuellen Knotens aussortiert wurde. Dies passiert, wie wir vorgreifen, nur in einem unserer Algorithmen, und zwar genau in dem Fall, in dem dieser Kindknoten eine Konfiguration hat, die schon von einem Eltern des aktuellen Knotens erreicht wird. Schließt nun der aktuelle Knoten eigentlich eine strenge Zusammenhangskomponente, hat aber einen solchen Kindknoten, so heißt dies, dass es sich doch nicht um eine geschlossene Endkomponente handelt[4]. $c_{\text{parentedge}}$ sorgt also gegen falsche Endkomponenten vor.

[4]Es könnte sich um eine noch nicht geschlossene Endkomponente handeln. Der Algorithmus wird dann den ganzen Untergraphen des aktuellen Knotens nicht in die Endkomponente aufnehmen. Damit wird die Endkomponente zwar ungenau abgeschätzt, aber trotzdem korrekt ausgegeben.

Algorithmus 5 Initialisierung eines neuen Knotens

1: **procedure** INITNEWNODE($P_{\text{variable}}, \Delta_{\text{grid}}, i_{\text{cur}}, P_{\text{activated}}, c_{\text{filter}}$)
2: **for all** $p \in N_{\text{readers}}(P_{\text{activated}})$ **do**
3: $\delta_p \leftarrow \delta(p)$
4: $c_{\text{next}} \leftarrow \delta_p(V_{\text{sim}}[N_{\text{in}}(p)])$
5: ${\Delta_{\text{cur}}}' \leftarrow {\Delta_{\text{cur}}}' + \text{PATCH}(p, c_{\text{next}}, V_{\text{cur}}[p])$
6: **end for**
7: $V_{\text{cur}} \leftarrow V_{\text{cur}} + {\Delta_{\text{cur}}}'$ ▷ temporär
8: ${P_{\text{variable}}}' \leftarrow \{p \in N_{\text{readers}}(P_{\text{variable}}) | V_{\text{cur}}[p] \neq V_{\text{sim}}[N_{\text{out}}(p)]\}$
9: **if** c_{filter} **then**
10: ${P_{\text{variable}}}' \leftarrow \text{FILTER}({P_{\text{variable}}}', P_{\text{activated}})$
11: **end if**
12: **for all** $p \in \text{CELLS}({\Delta_{\text{cur}}}')$ **do**
13: **for all** $p' \in P_{\text{activated}} \cap N_{\text{writers}}(p)$ **do**
14: $\delta_p \leftarrow \delta(p)$
15: **if** $\delta_p(V_{\text{sim}} - \Delta_{\text{grid}}|_{p'}) \neq {\Delta_{\text{cur}}}'[p]$ **then**
16: ADDEDGE($\mathcal{D}, p', p, i_{\text{next}}$)
17: **end if**
18: **end for**
19: **end for**
20: $V_{\text{cur}} \leftarrow V_{\text{cur}} - {\Delta_{\text{cur}}}'$
21: $c_{\text{insert}} \leftarrow \text{false}$
22: ${\Delta_{\text{grid}}}' \leftarrow \Delta_{\text{grid}} + \text{PATCH}(N_{\text{out}}(P_{\text{activated}}), {V_{\text{sim}}}', V_{\text{sim}})$
23: **if** $({\Delta_{\text{grid}}}', \cdot) \in V_{\text{dict}}$ **then**
24: SCCCHECKFORBACKEDGE($i_{\text{cur}}, V_{\text{dict}}[{\Delta_{\text{grid}}}']$)
25: **else**
26: $V_{\text{dict}} \leftarrow V_{\text{dict}} \cup ({\Delta_{\text{grid}}}', i_{\text{next}})$
27: ${P_{\text{unused}}}' \leftarrow P_{\text{variable}} \setminus P_{\text{activated}}$
28: $S \leftarrow S \cup \{(i_{\text{next}}, i_{\text{cur}}, {\Delta_{\text{grid}}}', {P_{\text{variable}}}', {P_{\text{unused}}}', {\Delta_{\text{cur}}}', \emptyset, -1, 0, \text{false}, \emptyset)\}$
29: $i_{\text{next}} \leftarrow i_{\text{next}} + 1$
30: **end if**
31: **end procedure**

Die Routine INITNEWNODE (A. 5) aktualisiert die Patches für V_{cur} und V_{prev}. Dies ist die einzige Stelle im Grundgerüst, wo wir die Funktion δ direkt anwenden müssen; und hier wird viel Laufzeit verwendet. Danach wird ${P_{\text{variable}}}'$, also die Menge der neuen variablen Zellen, direkt von V_{cur} gelesen. Nun folgt eine Aktualisierung von $\mathcal{D}_{\text{C}}(Q_0)$, wobei die For-Schleifen das Offensichtliche tun. Anschließend wird der Knoten dem Stack hinzugefügt, sofern seine Konfiguration nicht schon auf dem Stack liegt. Damit ist INITNEWNODE bereits erklärt.

Was nun noch nicht erklärt wurde, sind die Funktionen zum Zurückkehren (finishNode und callbackForSccs) und zum Sammeln und Weiterreichen der Ergebnisse (gatherResult). Da der Algorithmus auch ohne letztere funktionieren würde[5] folgen zunächst erstere.

Algorithmus 6 Callback-Funktion für jeden Knoten einer strengen Zusammenhangskomponente

```
1: procedure callbackForSccs(i_curnodeofscc, Δ_best)
2:     Δ ← V_dict[i_curnodeofscc]
3:     if |Δ| < |Δ_best| then
4:         Δ_best ← Δ
5:     end if
6:     V_dict ← V_dict \ Δ
7: end procedure
```

Algorithmus 7 Finalisierung des Knotens $\max(S)$

```
1: procedure finishNode
2:     (i_cur, i_src, Δ_grid, ·, ·, Δ_cur, ·, ·, ·, ·, ·, R_cur) ← max(S)
3:     gatherResult
4:     Δ_best ← Δ_cur
5:     n_scc ← sccOnFinishNode(i_cur, Δ_best, callbackForSccs)
6:     if n_scc ≥ 2 and !R_cur and !c_parentedge then
7:         R_cur ← (Δ_best, ∅, n_scc, 0)
8:     end if
9:     if i_src then
10:        sccOnFinishNewEdge(i_src, i_cur)
11:    end if
12:    V_cur ← V_cur − Δ_cur
13:    V_prev ← V_prev − Δ_cur
14:    R_last ← {R_cur}
15:    S ← S \ {max(S)}
16: end procedure
```

Die Funktion finishNode (A. 7) wird also aufgerufen, wenn ein Knoten alle möglichen und sinnvollen Kindknoten bereits entwickelt hat. Nun müssen zunächst, wie bei pushNext, die Ergebnisse des letzten Kindknotens noch aufgesammelt werden. Anschließend wird, falls der Knoten eine strenge Zusammenhangskomponente

[5] Man könnte daran interessiert sein, andere Dinge als $\mathcal{R}_{\text{complete}}$ auszugeben, oder diese Ausgabe zwecks Benchmarking verfallen zu lassen.

schließt, geprüft, wie groß diese ist[6]; dabei wird eine strenge Zusammenhangskomponente gemäß Tarjans Algorithmus [Tar72] geschlossen, wenn der aktuelle Knoten beim Zurückkehren in der Tiefensuche der letzte Knoten der aktuellen strengen Zusammenhangskomponente ist. Die Funktion prüft dies und ruft gegebenenfalls für jeden Knoten einer strengen Zusammenhangskomponente die Funktion CALLBACKFORSCCS auf. Wie wir in Algorithmus 6 sehen, liefert diese den Patch der strengen Zusammenhangskomponente, der am wenigsten von der Anfangskonfiguration q_0 differiert. Danach löscht sie außerdem alle Einträge aus V_{dict} zu Knoten der aktuellen strengen Zusammenhangskomponente, da dies der letzte Zeitpunkt ist, zu dem diese Einträge gebraucht werden.

Falls nun die strenge Zusammenhangskomponente mindestens 2 Konfigurationen beinhaltet, so könnte diese unter zwei zusätzlichen Bedingungen eine Endkomponente bilden:

- R_{cur} darf noch kein Ergebnis beinhalten. Hat R_{cur} ein Ergebnis, was durch GATHERRESULT von Kindknoten gesammelt wurde, so heißt dies, dass die strenge Zusammenhangskomponente keine Endkomponente sein kann.
- $c_{\text{parentedge}}$ muss falsch sein. Das wurde schon in der Beschreibung zu PUSHNEXT (A. 4) besprochen.

Sind aber all diese Bedingungen erfüllt, so wird nach oben weitergereicht, dass eine Endkomponente von Größe n_{scc} mit „ungenauer Abschätzung nach oben“ gefunden wurde, die keine Kindknoten und den Endrepräsentanten Δ_{best} hat (Z. 7).

Algorithmus 8 Sammeln der Ergebnisse

```
1:  procedure GATHERRESULT
2:      (Δ_grid, ..., R_cur) ← max(S)
3:      if |R_last| = 1 then
4:          if R_cur = ∅ then R_cur ← {(∅, ∅, 0, 0)}
5:          end if
6:          (Δ_last, ...) ← R_last[0]
7:          (Δ_res, V_children, ...) ← R_cur
8:          V_children ← V_children ∪ R_last[0]
9:          Δ_last ← −Δ_grid + Δ_last                ▷ Invariante erhalten
10:         Δ_res ← Δ_grid
11:         R_last ← ∅
12:     end if
13: end procedure
```

[6] Jeder Knoten liegt in einer strengen Zusammenhangskomponente, deren Größe im trivialen Fall nur 1 ist.

Nun haben wir also beschrieben, wie die Tiefensuche korrekt abläuft, und wie die Endrepräsentanten erkannt werden. Als letztes muss geklärt werden, wie letztere zur Wurzel weitergereicht werden. Auf diese Weise erhalten wir die Pfade von q_0 zu den Endrepräsentanten.

Die Weiterleitung geschieht in der Funktion GATHERRESULT (A. 8). Zunächst erinnern wir noch einmal daran, dass $\mathcal{R}_{\text{cur}}$ und $\mathcal{R}_{\text{last}}$ beides nicht die Tupel für die Knoten des Resultatgraphen, sondern Vektoren solcher Tupel sind. Sie sind immer leer oder tragen ein Element; das erklärt also den Vergleich mit $\emptyset$. Nun passiert Folgendes: Gibt es kein letztes Resultat in einem Kindknoten, so sammeln wir keines im aktuellen Knoten. Gab es jedoch eines, so wird, falls nicht vorhanden, ein Resultat-Tupel im aktuellen Knoten angelegt (Z. 4), und dann das letzte Resultat als Kind aufgenommen (Z. 8).

Nun müssen noch die Patches aufgeräumt werden: Es ist eine Invariante, dass $\mathcal{R}_{\text{last}}$ zu Beginn von GATHERRESULT noch den vollen Patch zu q_0 hatte. Wir sparen hier heuristisch Speicher, indem wir den Patch dieses Kindknotens nun bezüglich des aktuellen Knotens wählen (Z. 9). Um die Invariante wieder für den aktuellen Knoten zu erfüllen, erhält dieser nun den vollen Patch bezüglich q_0.

4.2.2 Korrektheit

Die Korrektheit des *Grundgerüsts* und damit auch des Brute-Force-Algorithmus sollte klar sein. Das impliziert aber noch keine Korrektheit von Algorithmen, die die fehlenden Funktion nichttrivial implementieren. Wer auch einen formal korrekten Beweis für den Brute-Force-Algorithmus möchte, der sei auf Abschnitt 4.5.1 verwiesen, da die Korrektheit dieses Algorithmus die Korrektheit des Brute-Force-Algorithmus direkt impliziert.

4.2.3 Asynchrone Automaten mit Wahrscheinlichkeiten

Falls Zellen nach irgendeinem Prinzip mit einer bestimmten Wahrscheinlichkeit aktiviert werden, so stellt sich die Frage, mit welcher Wahrscheinlichkeit welche Endkomponenten auftreten.

Für den Fall, dass Knoten mittels SORTOUT1 oder SORTOUT2 aussortiert werden, weil sie anderen Knoten gleichen, so müsste man die Wahrscheinlichkeiten auf diese anderen Knoten aufaddieren. Obwohl logisch klar ist, wie man diese anderen Knoten findet, so ist es doch schwierig, sie zu erreichen. Der Grund dafür ist, dass Knoten auf andere Knoten verweisen können, die wiederum auf andere verweisen. Nun können ganze Ketten von Verweisen entstehen, deren Knoten aber schon oder noch nicht abgearbeitet, und daher nicht gespeichert sind. Man findet also den Knoten, auf den man letztlich verweist, aktuell nicht, und es bleibt vermutlich nur der Ausweg, die Gewichte für die Verweise zu speichern und später aufzuaddieren. Leider wächst

aber die Anzahl der Verweise mit der Anzahl aussortierter Knoten. Daher haben wir keine Möglichkeit gefunden, das Problem zu lösen, ohne Speicher oder Laufzeit asymptotisch zu erhöhen.

4.2.4 Nichtdeterministische Automaten

Aufgrund seiner Einfachheit ist der Algorithmus vermutlich leicht auf nichtdeterministische Automaten erweiterbar. Man müsste dazu lediglich in den Konfigurationen V_{cur} und V_{prev} pro Zelle mehrere Möglichkeiten speichern, die die Zustände der verschiedenen Ausgaben beschreiben.

4.3 Laufzeit und Speicher

Wir definieren folgende Abkürzungen für Variablen, von denen Laufzeit und Speicher abhängen können.

- $N_{\text{all}} \leq |R|$: Anzahl der Zellen, die der Algorithmus als Eingabe erhält. Wir nehmen an, dass diese Größe endlich ist. Akzeptiert die Implementierung beispielsweise nur endliche Konfigurationen aus $\mathbb{Z}^2$ in einem Rechteck der Seitenlängen a und b mit $(a, b) \in \mathbb{Z}^2$ als Eingabe, so wäre $N_{\text{all}} = ab$, auch wenn für den Automaten $R = \mathbb{Z}^2$ gelten würde.
- $N_{\text{n}} = \max_{r \in R}(\max(|N_{\text{in}}(r)|, |N_{\text{out}}(r)|))$. Es ist $N_{\text{n}} \leq N_{\text{all}}$
- $N_{\text{res}} = |V(\mathcal{R}_{C,S}(Q_0, Q'))|$, wobei Q' die Wahl der Endrepräsentanten des Algorithmus ist.
- $N_{\text{path}} \in \mathcal{O}(N_{\text{res}})$: Länge des längsten möglichen Berechnungspfades von einer Anfangs- bis zu einer Endkonfiguration.
- $N_{\text{var}} \leq N_{\text{all}}$: Maximale Anzahl variabler Zellen in einer Iteration. Beinhaltet auch die maximale Abweichung an Zellen der Anfangskonfigurationen voneinander.
- $N_{\text{initial}} \leq |Q|^{N_{\text{all}}}$: Anzahl initialer Konfigurationen.
- T_{ca}: Maximale Laufzeit, um $\delta(r)$ für $r \in R$ zu berechnen. Gilt als teuer. Außerdem gilt: $T_{\text{ca}} \geq N_{\text{n}}$.
- M_{ca}: Größe des Zellularautomaten im Speicher. Dieser kann durch eine kurze Formel realisiert sein, ist aber im schlimmsten Fall eine Tabelle; dann: $M_{\text{ca}} = \mathcal{O}(|Q|^{N_{\text{n}}} \cdot N_{\text{n}})$.

Da es sich nur um ein Grundgerüst handelt, hängen Laufzeit und Speicher von der Implementierung der nicht spezifizierten Funktionen ab. Für deren Laufzeiten seien ebenfalls Variablen wie „$T_{\text{Funktionsname}}$“ definiert. Der gesamte zusätzliche Speicher der Implementierung heißt M_{impl}.

4.3.1 Speicher

Der Speicher ist sehr leicht zu analysieren. Es gibt ein paar globale Variablen, ansonsten liegen alle Variablen auf dem Stack, der nicht größer als N_{path} wird. Initial werden noch $\mathcal{O}(1)$ Konfigurationen erzeugt sowie einige Patches für die initialen Konfigurationen. Wir erhalten:

$$\begin{aligned} M &= \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot N_{\text{initial}} + M_{\text{ca}} + M_{\text{impl}} + N_{\text{var}} \cdot (N_{\text{path}} + N_{\text{res}})) \\ &= \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{res}}) + M_{\text{ca}} + M_{\text{impl}}). \end{aligned}$$

Damit beträgt der Speicher für einen konstanten Automaten

$$M_{\text{const}} \coloneqq \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{res}}) + M_{\text{impl}}).$$

Da aber, wie wir bereits vorgreifen, für alle Automaten in diesem Skript $M_{\text{impl}} \in \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{path}}))$ gilt, so erhalten wir:

$$\begin{aligned} M_{\text{hier}} &\coloneqq \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{res}}) + M_{\text{ca}}) \text{ bzw.} \\ M_{\text{hier,const}} &\coloneqq \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{res}})). \end{aligned}$$

Wie ist das in der Praxis zu interpretieren? Für einen festen Automaten werden initial nur wenige Konfigurationen im Speicher allokiert. Danach beschränkt sich die Speichernutzung immer nur auf ein paar wenige Zellen pro Stack-Element, wobei der Stack nie sehr groß wird. Ein paar Stack-Elemente werden dauerhaft in die Resultate aufgenommen; und dies ist der einzige Anteil des Speichers, der mit der Laufzeit wächst. Ist aber die Anzahl der Endkomponenten wie bei $\mathcal{C}_{\text{bc3}}$ oder $\mathcal{C}_{\text{st}}$ gering, so fällt dies nicht ins Gewicht. Hier ist der Speicherverbrauch also praktisch dauerhaft konstant.

4.3.2 Laufzeit

In diesem Abschnitt werden die Laufzeiten analysiert. Alle Laufzeiten sind im Zweifelsfall entsprechend amortisiert.

Wir unterteilen den Großteil des Codes in 3 aufeinander folgende Abschnitte, deren Laufzeiten wir T_{st1}, T_{st2} und T_{st3} nennen:

T_{st1} Code zum Erstellen eines neuen Knotens bis inklusive SORTOUT1.

T_{st2} initNewNode + Code bis inklusive sortOut2 + Code für das Verlassen des Stacks.

T_{st3} Code bis inklusive makeSccs oder dem Speichern eines Resultats.

Man errechnet Laufzeiten für folgende Abschnitte:

- T_{init}: Zeit für den Aufruf von initNewNode: $\mathcal{O}(N_{\text{n}} \cdot N_{\text{var}}(N_{\text{n}} + T_{\text{ca}}) + T_{\text{filter}} + \log(N_{\text{path}})) = \mathcal{O}(N_{\text{n}} \cdot N_{\text{var}} \cdot T_{\text{ca}} + T_{\text{filter}} + \log(N_{\text{path}}))$
- T_{final}: Behandeln von Endkomponenten mit exakter Größe 1: $\mathcal{O}(\log(N_{\text{res}}))$
- T_{leave}: Verlassen des Stacks: $\mathcal{O}(\log(N_{\text{res}}) + \log(N_{\text{path}}) + N_{\text{var}}) = \mathcal{O}(\log(N_{\text{res}}) + N_{\text{var}})$
- T_{st1}: $\mathcal{O}(N_{\text{n}} \cdot N_{\text{var}} + T_{\text{sortOut1}})$
- T_{st2}: $\mathcal{O}(T_{\text{init}} + N_{\text{var}} + T_{\text{sortOut2}} + \log(N_{\text{path}}) + T_{\text{leave}}) = \mathcal{O}(N_{\text{n}} \cdot N_{\text{var}} \cdot T_{\text{ca}} + T_{\text{filter}} + \log(N_{\text{res}}) + T_{\text{sortOut2}})$
- T_{st3}: $\mathcal{O}(T_{\text{makeSccs}} + T_{\text{final}}) = \mathcal{O}(T_{\text{makeSccs}} + \log(N_{\text{res}}))$

Definiert man nun die Anzahl der Knoten bezüglich der Abschnitte, wo sie aussortiert werden, also $N_{\text{initialized}} \geq N_{\text{passed1}} \geq N_{\text{passed2}}$, so lässt sich die gesamte Laufzeit berechnen:

$$T = \mathcal{O}(N_{\text{all}}) + N_{\text{initialized}} \cdot T_{\text{st1}} + N_{\text{passed1}} \cdot T_{\text{st2}} + N_{\text{passed2}} \cdot T_{\text{st3}}.$$

Speziell für Algorithmen, die nie Knoten aussortieren, ist $N_{\text{initialized}} = N_{\text{passed1}} = N_{\text{passed2}}$, und damit:

$$\begin{aligned} T_{\text{nosortout}} &= \mathcal{O}(N_{\text{all}}) + N_{\text{initialized}}(T_{\text{st1}} + T_{\text{st2}} + T_{\text{st3}}) \\ &= \mathcal{O}(N_{\text{all}} + N_{\text{initialized}}(N_{\text{n}} \cdot N_{\text{var}} \cdot T_{\text{ca}} + \log(N_{\text{res}}) + T_{\text{filter}} + T_{\text{makeSccs}})); \end{aligned}$$

und für einen konstanten solchen Automaten gilt:

$$T_{\text{nosortout,const}} = \mathcal{O}(N_{\text{all}} + N_{\text{initialized}}(N_{\text{var}} + \log(N_{\text{res}}) + T_{\text{filter}} + T_{\text{makeSccs}})).$$

Für den Brute-Force-Algorithmus, und, wie wir bereits ankündigen, alle Algorithmen in diesem Kapitel, sind $T_{\text{filter}}, T_{\text{makeSccs}} \in \mathcal{O}(N_{\text{n}} \cdot N_{\text{var}})$, so dass sie nicht ins Gewicht fallen, und wir erhalten für alle:

$$\begin{aligned} T_{\text{brute}} &= \mathcal{O}(N_{\text{all}} + N_{\text{initialized}}(N_{\text{n}} \cdot N_{\text{var}} \cdot T_{\text{ca}} + \log(N_{\text{res}}))) \text{ sowie} \\ T_{\text{brute,const}} &= \mathcal{O}(N_{\text{all}} + N_{\text{initialized}}(N_{\text{var}} + \log(N_{\text{res}}))). \end{aligned}$$

In der Praxis relevant ist dabei meist nur $N_{\text{initialized}} \cdot T_{\text{ca}}$. Leider ist uns keine Formel bekannt, um $N_{\text{initialized}}$ weiter aufzulösen.

4.4 Einser-Algorithmus

Es sei in diesem Abschnitt $N_{\text{readers}} = N_{\text{writers}}$, und wir setzen $N \coloneqq N_{\text{readers}}$.

Der Einser-Algorithmus funktioniert sehr simpel nach der N^1-Entwicklung (Satz 3.4). Daher auch der Name; es geht um N^1. Der Unterschied zum Brute-Force-Algorithmus ist, dass man nur Zellen aktiviert, die in einer strengen Zusammenhangskomponente liegen, und somit für $n \in \mathbb{N}$ aktive Zellen bestenfalls nur n anstatt 2^n unmittelbare Folgekonfigurationen berechnen muss. Der Pseudocode aus Algorithmus 9 liefert eine mögliche Implementierung, wobei CALCSCCS(P, N) die strengen Zusammenhangskomponenten der Zone P bezüglich der R-Nachbarschaft N liefert.

Algorithmus 9 Einser-Algorithmus

```
1: procedure MAKESCCS(P)
2:     return CALCSCCS(P, N)
3: end procedure
```

Der Beweis, dass dieser Algorithmus korrekt ist, folgt direkt aus dem Beweis des linksseitigen Algorithmus in Abschnitt 4.5.1 im folgenden Kapitel.

Laufzeit und Speicher sind, wie bereits angekündigt, wie beim Brute-Force-Algorithmus:

$$\begin{aligned} M_{\text{one}} &\coloneqq \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{res}}) + M_{\text{ca}}) \text{ bzw.} \\ M_{\text{one,const}} &\coloneqq \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{res}})) \text{ und} \\ T_{\text{one}} &\coloneqq \mathcal{O}(N_{\text{all}} + N_{\text{initialized}}(N_{\text{n}} \cdot N_{\text{var}} \cdot T_{\text{ca}} + \log(N_{\text{res}}))) \text{ bzw.} \\ T_{\text{one,const}} &\coloneqq \mathcal{O}(N_{\text{all}} + N_{\text{initialized}}(N_{\text{var}} + \log(N_{\text{res}}))) \end{aligned}$$

Man beachte, dass $N_{\text{initialized}}$ und N_{res} asymptotisch besser sein können als im Brute-Force-Algorithmus. Daher sind die Ausdrücke unterschiedlich.

Der Einser-Algorithmus hat einen entscheidenden Nachteil: Es gibt, wie man sich leicht überlegen kann, viele Eingaben, für die sehr viele Duplikate entstehen. Es gibt eine Modifikation, die viele davon vermeidet. Der nächste Abschnitt stellt diese Modifikation vor.

4.5 Linksseitiger Algorithmus

Wieder sei in diesem Abschnitt $N_{\text{readers}} = N_{\text{writers}}$ und $N \coloneqq N_{\text{readers}}$.

Der Linksseitige Algorithmus ist ein Hilfsmittel, um das Entstehen von Duplikaten beim Einser-Algorithmus zu vermeiden. Er benötigt eine Vorstellung von „links“.

Gemeint ist damit eine beliebige Ordnungsrelation $\succ$ auf R. Man kann für $\mathbb{Z}^2$ beispielsweise lexikographisch sortieren.

Die Idee ist wie folgt: Wir erlauben nur die Aktivierung bestimmter Zellen, nämlich derer aus der Nachbarschaft der zuletzt aktivierten Zellen und alle Zellen „links“ davon. Algorithmus 10 zeigt alle zu implementierenden Funktionen. Alternativ kann man sich auch den C++-Code unter `src/impl/left.h` anschauen.

Algorithmus 10 Linksseitiger Algorithmus

```
1: procedure FILTER(P_variable, P_activated)
2:     for all p ∈ P_variable do
3:         if p < max(P_activated) or p ∈ N(P_activated) then
4:             P_result ← P_result ∪ {p}
5:         end if
6:     end for
7:     return P_result
8: end procedure
9: procedure MAKESCCS(P)
10:    return CALCSCCS(P, N^2)
11: end procedure
```

4.5.1 Korrektheit

Der Algorithmus liefert alle Endkomponenten, wie folgender Satz zeigt.

Satz 4.4. *Sei* $(R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathcal{C}$ *und* $Q_0 \subseteq Q^R$. *Sei* M *so, dass* $V(\mathcal{B}(Q_0)) \subseteq V(\mathcal{G}_{C,S}(Q_0)) \times M$. *Dann gilt:*

$$\begin{aligned} c_2 \in \mathcal{G}(Q_0) \Rightarrow \exists (q_0, m) \in Q_0 \times M : \\ (c_2, m) \in V(\mathcal{B}(Q_0)) \text{ und} \\ (q_0, \cdot) \vdash_{E(\mathcal{B}(Q_0))} (c_2, m). \end{aligned}$$

Beweis. Wir versuchen, einen Weg von q_0 nach c_2 in $\mathcal{G}(Q_0)$ zu finden, zu dem ein Äquivalent in $\mathcal{B}(Q_0)$ existiert. Dies tun wir in mehreren Iterationen, indem wir, wie es der Algorithmus vorschreibt, pro Iteration genau eine Zellkomponente $S \subseteq R$ aktivieren, wobei damit eine Menge von variablen Zellen gemeint ist, die bezüglich N^1 streng zusammenhängt.

Die Wahl der Zellkomponente S geschieht immer wie folgt. Wir beginnen mit der am weitesten rechts gelegenen Zellkomponente $S' \subseteq R$, die gemäß FILTER genommen werden darf. Wir raten, ob es möglich ist, nach irgendeiner Wahl von Zellen $P \subseteq S'$ und deren Aktivierung noch c_2 erreichen zu können. Dabei raten wir immer richtig,

und sogar so gut, dass keine Kreise von $\mathcal{G}(Q_0)$ durchlaufen werden. Gibt es nun so ein P, so aktivieren wir P und fahren fort. Ansonsten wählen wir S' als die nächstlinke Komponente und suchen weiter.

Eine Invariante ist damit, dass immer alle Zellen, die aktuell aktiviert werden dürfen, um c_2 noch zu erreichen, links von der aktuellen Zellkomponente oder in ihrer Nachbarschaft sind. Genau diese Zellen dürfen aber laut Algorithmus auch aktiviert werden. Die Invariante gilt in jedem Schritt, und so existiert zu jedem Weg in $\mathcal{G}(Q_0)$ zu c_2 immer noch ein entsprechender Weg in $\mathcal{B}(Q_0)$ zu (c_2, m). □

4.5.2 Duplikate

Der Algorithmus findet leider nicht alle Duplikate; und in der Praxis ist er daher nicht einsetzbar. Ein Negativbeispiel kann man schon mit $R = \{0, 1, 2, 3\}$ und $Q = \{0, 1, 2\}$ konstruieren.

4.5.3 Laufzeit und Speicher

Die Formeln für Laufzeit und Speicher sind wieder wie vorher:

$$\begin{aligned} M_{\text{left}} &:= \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{res}}) + M_{\text{ca}}) \text{ bzw.} \\ M_{\text{left,const}} &:= \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{res}})) \text{ und} \\ T_{\text{left}} &:= \mathcal{O}(N_{\text{all}} + N_{\text{initialized}}(N_{\text{n}} \cdot N_{\text{var}} \cdot T_{\text{ca}} + \log(N_{\text{res}}))) \text{ bzw.} \\ T_{\text{left,const}} &:= \mathcal{O}(N_{\text{all}} + N_{\text{initialized}}(N_{\text{var}} + \log(N_{\text{res}}))) \end{aligned}$$

4.6 Schwächen und Stärken

Alle vorgestellten Algorithmen sind mehr oder weniger Brute-Force-Algorithmen mit Aussparung einiger Duplikate. Sie erhalten feste Konfigurationen als Eingabe und erzeugen ein spezifische Ausgabe. Dieses sehr „statische“ Verhalten hat seine Vor- und Nachteile.

4.6.1 Schwächen

Möchte man eine Konfiguration drehen, spiegeln oder verlängern, so müssen die Algorithmen neu ausgeführt werden. Außerdem müssen große Konfigurationen in einem Stück simuliert werden, da keine Aufteilung in kleinere Konfigurationen vorgesehen ist. Bezüglich des Raums, der Konfiguration oder der Überführungsfunktion haben die Algorithmen also keine genauere Vorstellung über Eigenschaften wie Symmetrie oder Zusammensetzung.

Ein weiteres Problem sind Duplikate. Werden sie nicht korrekt erkannt, so steigt die Laufzeit exponentiell. Eine gute Duplikaterkennung ist daher für „große“ Konfigurationen entscheidend.

4.6.2 Stärken

Ein Vorteil ist die Einfachheit der Algorithmen. So kann man unseren Brute-Force-Algorithmus schon in ca. 100 Zeilen implementieren; und er benötigt keine komplizierten Beweise. Andere Algorithmen bauen auf letzterem auf; und zusätzlicher Code kann modular erklärt werden.

Außerdem funktionieren alle hier vorgestellten Algorithmen mit jedem Automaten. Es werden also keine Techniken verwendet, die nur auf bestimmten Arten von Automaten funktionieren würden. So können selbst über Automaten, zu denen keine weiteren strukturellen Eigenschaften bekannt sind, Vermutungen bewiesen oder widerlegt werden.

Ein zusätzlicher Vorteil gegenüber Beweisen, die von Menschen geführt werden ist, dass ein formeller Beweis für eine gewisse Anfangskonfiguration noch nicht beweist, dass eine entsprechende eingegebene Funktion auf einem Rechner auch die Bedingungen für den Beweis erfüllt. Die Eingabe könnte schließlich an irgendeiner Stelle falsch kodiert worden sein. Doch selbst wenn die Eingabe richtig kodiert ist, so könnte der Automat falsch kodiert worden, oder sogar der Quellcode für den Simulator des Automaten fehlerhaft programmiert sein. Daher ist ein Brute-Force-basierter Ansatz, der den programmierten Simulator direkt verwendet, zwingend notwendig, um in der Praxis vollständige Sicherheit zu erlangen.

Kapitel 5

Verworfene Algorithmen

Dieses Kapitel führt zwei Algorithmen auf, die sich im Laufe der Arbeit als nicht zielführend herausgestellt haben. Es sei im ganzen Kapitel $N_{\text{readers}} = N_{\text{writers}}$.

5.1 Superzellen-Algorithmus

Der Superzellen-Algorithmus ist ein blinder Algorithmus. Die Grundidee war, dass man mehrere möglicherweise aktive Zellen iterativ zu einer sogenannten Superzelle zusammenfasst, bis dies nicht mehr möglich ist.

Sei $(R, Q, N_{\text{in}}, ...) \in \mathfrak{C}$ und $Q_0 \subseteq Q^R$ eine Menge an Anfangskonfigurationen. Eine Superzelle ist eine Tupel $(A, P, (V, E))$, wobei

1. $A \subseteq R$ die Zone der Superzelle ist. Die Superzelle kennt implizit alle Konfigurationen $X \subseteq Q^R$ mit $Q_0 \succ_A^* X$.

2. $P \subseteq A$ ist die Menge von Zellen, die Nachbarn in $R \setminus A$ haben, die in X aktiv werden können.

3. (V, E) ist ein Teilgraph aller möglichen Konfigurationen aus X. Genauer ist $V \subseteq X$ so, dass für jede Kombination $k \in X|_P$ eine möglichst kleine Menge V' in V aufgenommen wird, so dass $V'|_P = k$, $V' \succ_{A \setminus P}^* \{x \in X, x|_P = k\}$. Für E gilt, dass $(v_1, v_2) \in E :\Leftrightarrow \exists v_3 \colon v_1 \succ_{A \setminus P}^* v_3 \succ_A v_2$.

Diese etwas spezielle Graphstruktur erlaubt es, nur Repräsentanten aller Kombinationen auf P in V zu speichern. Fügt man eine neue Zelle $n \in R$ an die Superzelle an, so nimmt man alle Konfigurationen aus V, in denen n variabel ist, und kann dann per Brute-Force alle neuen Konfigurationen bestimmen. Das Einsparen von Konfigurationen ist also ungefährlich.

Der genaue Ablauf des Algorithmus beschreibt sich in etwa wie folgt.

- Zu Beginn ist jede strenge Zusammenhangskomponente variabler Zellen[1] eine Superzelle.

[1] Zwei Zellen gehören zusammen, wenn ihr Abstand maximal d beträgt, wobei d mindestens 2 sein muss. Der Wert für d ist ein Detail, auf das wir nicht eingehen.

- Nun beginnt eine Schleife, die aus fünf Schritten besteht, und erst verlassen wird, wenn es keine aktiven Zellen mehr gibt.
  1. Zunächst wird eine beliebige aktive Nachbarzelle $p \in R$ einer beliebigen Superzelle $(A, P, (V, E))$ an diese angehängt.
  2. Gibt es ein $q \in N_{\text{in}}(p), q \notin P$, so muss die komplette Berechnung neu gestartet werden, und es wird ein Zähler für q erhöht, so dass man sich q beim nächsten mal in P gemerkt haben wird.
  3. Ansonsten entstehen durch p neue Konfigurationen, die, ausgehend von V, errechnet werden müssen. Meist geschieht dies durch einen Brute-Force-ähnlichen Algorithmus.
  4. Nun wird jede andere Superzelle, die zu nahe an der aktuellen ist, der aktuellen hinzugefügt. Das Resultat entspricht genau dem kartesischem Produkt.
  5. Schließlich wird der Graph vereinfacht, so dass er wieder den Anforderungen an die Repräsentanten zum neuen P genügt.

Durch die sogenannten „Rückfälle“ von Punkt 2 und auch durch die Behandlung von Kreisen innerhalb des Graphen einer Superzelle wurde der Algorithmus leider anfällig für Fehler und der Code sehr groß[2] und unübersichtlich. Das wohl entscheidendste Problem waren allerdings Speicher und Laufzeit. Zwar sind einzelne Rückfälle für die Asymptotik ohne Bedeutung und schnell genug implementierbar. Leider gab es Fälle, in denen Rückfälle sehr häufig passierten.

0	0	1	0	1	0	0
0	0	1	0	1	0	0
0	1	1	0	2	1	1
0	1	0	0	0	0	0
0	0	2	0	2	1	1
0	1	1	0	1	0	0
0	0	1	0	1	0	0
0	0	1	0	1	0	0

Tabelle 5.1: Ein Beispiel für ein Problem mit Superzellen in $\mathcal{C}_{\text{st}}$. Gehen Signale von oben und von rechts nach unten, so müssen alle Zustände der Zone unterhalb der „Einmündung“ gespeichert werden.

Abbildung 5.1 zeigt so einen Fall für $\mathcal{C}_{\text{st}}$: Nehmen wir an, dass zunächst variable Zellen von oben nach unten und danach von rechts nach unten laufen. Nun müsste der

[2] Der Quellcode war mit circa 2400 Zeilen erstmals funktionsfähig.

Algorithmus nach mehreren „Rückfällen“ schließlich für alle gemeinsam durchlaufenen Zellen, also hier alle unterhalb der Einmündung, alle Kombinationen von Zuständen speichern. Der gemerkte Speicher ist auf solchen Teilen von R so groß wie bei einem Brute-Force-Algorithmus, der sich alle globalen Konfigurationen merkt. Das erklärt die hohe Speichernutzung.

5.2 Split-Algorithmus

Der Split-Algorithmus funktioniert global nach dem Prinzip der N^2-Entwicklung (Satz 3.3). Folgende Stichpunkte geben den groben Ablauf wieder.

1. Man zerlege initial R in Zonen, indem man strenge Zusammenhangskomponenten von R bezüglich N^2 wählt.

2. Nun entwickelt man die Zonen wie in Satz 3.3 separat. Dabei erhält man einen Baum an Konfigurationen.

3. In jeder Iteration können Zonen auch wieder in kleinere Zonen aufgeteilt werden. Einmal getrennte Zonen werden nicht wieder vereinigt.

4. Falls die Entwicklung einer Zone zu aktiven Zellen führt, die in N^2 einer anderen Zone sind, so muss ein „Neustart“ bei Punkt 1 durchgeführt werden.

Es handelt sich also um einen asymmetrischen Algorithmus, da man die Zonen nicht gleichzeitig, sondern separat entwickelt.

Man kann beweisen, dass dieser Algorithmus korrekt funktioniert. Es handelt sich um eine rekursive Anwendung von Satz 3.3 mit der Ausnahme von Punkt 3, bei dem Zonen in kleinere Zonen zerfallen können. Hier wendet man Satz 3.3 rekursiv an, um die Forderung von Satz 3.3, alle Konfigurationen einer Zone zu berechnen, zu erfüllen. Das beweist die Korrektheit.

Zur lokalen Berechnung der Folgezustände auf den einzelnen strengen Zusammenhangskomponenten wird der Linksseitige Algorithmus (Algorithmus 10) verwendet. Dabei muss lediglich die Anpassung gemacht werden, dass nach einem Neustart, unabhängig von den Einschränkungen des Linksseitigen Algorithmus, wieder alle aktiven Zellen aktiviert werden dürfen.

Der Algorithmus wurde schließlich verworfen, da er zu zu vielen Duplikaten führte. Ein Hauptgrund war, dass der verwendete Linksseitige Algorithmus an sich viele Duplikate nicht findet. Ein anderes Problem war, dass wir in Punkt 3 des oben beschriebenen Ablaufs eine Erweiterung hatten, die Zonen teilweise wieder vereinigt. Kurz vor Abgabe der Arbeit wurde jedoch klar, dass dieses Prinzip mindestens zwei logische Fehler hatte. Man kann den Code trotzdem in der Datei `src/split/split_run.cpp` lesen.

Kapitel 6

Greedy-Algorithmus

Im ganzen Kapitel sei $(R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathfrak{C}$ gegeben, und es sei $N \coloneqq N_{\text{readers}}$.

In diesem Kapitel wird angenommen, dass $N_{\text{readers}} = N_{\text{writers}}$ ist, und wir setzen $N \coloneqq N_{\text{readers}}$. Zwar wird die Annahme im ganzen Kapitel nicht gebraucht, aber sie verringert die Menge an Pseudo-Code sehr.

Der Greedy-Algorithmus ist, im Gegensatz zu fast allen vorhergehenden Algorithmen, ein symmetrischer Algorithmus. Grob geht es wieder darum, nach bestimmten Konfigurationen nicht mehr weiter zu simulieren, wenn bereits absehbar ist, dass sie Duplikate von anderen sind. Dazu werden wir diesmal jedoch immer möglichst viele variable Zellen gleichzeitig entwickeln, anstatt sie aufzuteilen. Dies erklärt auch den Namen des Algorithmus; und in einigen praktischen Automaten, wie zum Beispiel $\mathcal{C}_{\text{bc3}}$ oder C_{st}, kommt man so auch am schnellsten zu den finalen Konfigurationen.

Der Greedy-Algorithmus nutzt wieder das Grundgerüst des Brute-Force-Algorithmus. Algorithmus 11 zeigt die Besonderheiten auf.

Algorithmus 11 Greedy-Algorithmus

```
1: procedure sortOut1(P_lastactivated, P_activated, V_oldconf)
2:     return hasOneMovableScc(P_lastactivated, P_activated, V_oldconf)
3: end procedure
4: procedure sortOut2(P_activated, P_unused, i_src)
5:     return hasOneIsolatedScc(P_activated, P_unused, i_src)
6: end procedure
```

Die beiden Methoden, um Untergraphen des Berechnungsgraphen abzuschneiden, werden wir im Folgenden vorstellen. Beide verhindern Duplikate, indem sie zu einem Berechnungspfad einen alternativen finden. Wichtig ist, dass diese „Delegationen“ nicht zyklisch verlaufen. Es darf nicht passieren, dass wir eine Konfiguration auf sich selbst umleiten und aufgrund dessen ignorieren. Allerdings wird dies unmöglich sein, da jedes Mal die „Umleitung“ im ersten unterschiedlichen Schritt des neuen Pfades echt mehr Zellen aktiviert als auf dem alten Pfad. Die Umleitungen sind also azyklisch.

Man kann in der Datei `src/impl/greedy.cpp` beide Funktionen auch als C++-Code finden.

6.1 Bewegbare Zellen

Die Aussparung sogenannter bewegbarer Zellen ist Kern des Greedy-Algorithmus. Mit bewegbar ist durch die Zeit bewegbar gemeint: Wurde eine Zelle durch eine Aktivierung in einen Zustand versetzt, in den sie schon in der vorhergehenden Runde versetzt werden hätte können, so schauen wir, was passiert, wenn man sie schon damals aktiviert hätte. Hat diese Änderung keine Nebeneffekte auf äußere Zellen, so sagen wir, dass die entsprechende Zelle (durch die Zeit) „bewegbar“ ist.

Etwas genauer bewegen wir nicht einzelne Zellen, sondern ganze Zusammenhangskomponenten auf einmal. Codesegment 12 gibt im Detail wieder, wie man die Bedingung zum Aussortieren gewinnt.

Algorithmus 12 Suche nach bewegbaren Zellen

```
1: procedure HASONEMOVABLESCC(P_lastactivated, P_activated, V_oldconf)
2:     c_res ← false
3:     V_overlap ← V_sim − V_sim                                  ▷ Null-Konfiguration
4:     V_overlap[N_out(P_lastactivated)] ← 1
5:     for all V_scc ∈ CALCSCCS(P_activated, N) do
6:         c_movable ← true
7:         for all p ∈ V_scc do
8:             if V_sim[N_out(p)] ≠ V_prev[p] then
9:                 Δ_2 ← Δ_2+ PATCH(p, V_oldconf[p], V_sim[p])
10:            end if
11:        end for
12:        if |Δ_2| = |V_scc| then
13:            c_movable ← false                                  ▷ Keine bewegbare Zelle.
14:        else
15:            V_sim' ← V_sim + Δ_2
16:            P_Δ ← CELLS(Δ_2)
17:            if δ(V_sim'[N_in(P_Δ)]) ≠ V_sim[N_out(P_Δ)]  or  V_overlap[N_out(P_Δ)] ≠ 0 then

18:                c_movable ← false
19:            end if
20:        end if
21:        if c_movable then
22:            c_res ← true
23:        end if
24:    end for
25:    return c_res
26: end procedure
```

Was macht dieser Code? Die Eingabe der Funktion besteht aus den Menge der zum vorherigen beziehungsweise aktuellen Knoten veränderten Zellen $P_{\text{lastactivated}}$, $P_{\text{activated}}$ und der Konfiguration V_{oldconf} im Elternknoten. Die äußere For-Schleife iteriert über strenge Zusammenhangskomponenten der aktivierten Zellen. Die innere For-Schleife in Zeile 7 findet nun die Zellen, die man geändert hat, und die nicht bewegbar wären, da sie in der Runde zuvor aktiviert einen anderen Zustand eingenommen hätten. Falls die ganze strenge Zusammenhangskomponente aus solchen Zellen besteht, kann man zwar die leere Menge bewegen, was aber keinen alternativen Pfad bringt. Anderenfalls setzt man diese Zellen temporär in die vorherige Konfiguration V_{oldconf} zurück und prüft (Z. 17), ob sie sich in der aktuellen Konfiguration wie üblich verhalten, wenn wir nur die bewegbaren Zellen bewegen.

Erwähnenswert ist noch die Konfiguration V_{overlap}. Sie dient dazu, dass sich die Ausgabenachbarschaften der bewegbaren Zellen nicht mit denen der in der vorigen Runde aktivierten überlagern. Natürlich müssen die bewegbaren Zellen nicht gegen andere bewegbare Zellen und Zellen der vorhergehenden Runde nicht gegen andere solche geprüft werden.

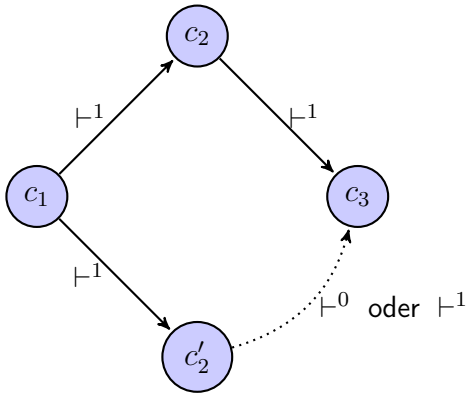

Abbildung 6.1: Die Konfigurationen im Überblick.

Wieso ist der Algorithmus korrekt? Sei c_1 die letzte Konfiguration, c_2 die aktuelle und c_3 die nächste (implizit via $P_{\text{activated}}$). Angenommen, c_{movable} wird wahr, so heißt dies, dass es eine Menge M von Zellen gibt, die schon eine Iteration früher hätten aktiviert werden können, also statt von c_2 zu c_3 schon von c_1 zu einer anderen Konfiguration c_2'. Abbildung 6.1 zeigt die Anordnung der Konfigurationen. Wir zeigen, dass $c_2' \vdash^1 c_3$ oder $c_2' \vdash^0 c_3$ gilt:

- Ist $r \notin P_{\text{activated}}$, so wurde r nicht zu c_3 hin aktiviert, also $c_2'(N_{\text{out}}(r)) = \delta(r)(c_1(N_{\text{in}}(r))) = c_2(N_{\text{out}}(r)) = c_3(N_{\text{out}}(r))$.
- Anderenfalls ist entweder $r \notin M$, also hat r den if-Zweig in Zeile 17 passiert, und damit ist direkt $\delta(r)(c_2'(N_{\text{in}}(r))) = c_3(N_{\text{out}}(r))$;
- oder es ist $r \in M$, also r bewegbar, also

$$c_2'(N_{\text{out}}(r)) = \delta(r)(c_1(N_{\text{in}}(r))) = \delta(r)(c_2(N_{\text{in}}(r))) = c_3(N_{\text{out}}(r)).$$

Fassen wir also zusammen:

$$c_3(N_{\text{out}}(r)) = \begin{cases} \delta(r)(c_2'(N_{\text{in}}(r))) & \text{falls } r \in P_{\text{activated}} \setminus M \\ c_2'(N_{\text{out}}(r)) & \text{sonst.} \end{cases}$$

Das heißt, man gelangt von c_2' nach c_3 genau dann, wenn man alle Zellen aus $P_{\text{activated}} \setminus M$ aktiviert. Ist $P_{\text{activated}} = M$, so gilt $c_2' \vdash^0 c_3$, sonst $c_2' \vdash^1 c_3$. Es ist

in jedem Fall unnötig, c_3 von c_2 aus zu entwickeln, falls der Weg über c_2' bereits überprüft wird.

Nun muss man nur noch aufpassen, dass der Weg über c_2' auch wirklich überprüft wird. Das kann nur dann nicht passieren, wenn sich ein Kreis von „Delegationen“ bildet. Wir haben das aber schon im letzten Kapitel ausgeschlossen und die entsprechende Aussage trifft auch hier zu.

6.2 Isolierte Zellen

Nun zu den isolierten Zellen. Der Name kommt daher, dass einige Zellen vom Geschehen außerhalb gewissermaßen isoliert sind. Damit ist gemeint, dass sie zwar eventuell Einfluss nach außen haben, aber nie von außen beeinflusst werden. Wir zeigen, dass man sich oft eine der beiden Konfigurationen sparen kann, wenn sich zwei Konfigurationen ausschließlich nahe an einer isolierten Zelle unterscheiden.

Algorithmus 13 zeigt genau, was gemeint ist.

Algorithmus 13 Suche nach isolierten Zellen

```
1: procedure hasOneIsolatedScc(P_activated, P_unused, i_src)
2:     c_res ← false
3:     for all U ∈ calcSccs(P_unused, N) do
4:         c_cur ← true
5:         U' ← N(U) \ U
6:         if V_recent[U'] = V_sim[N_out(U')] then
7:             for all p ∈ ((N(N(U))) \ N(U)) do
8:                 if D(p) then                          ▷ Liegt p in V_sim?
9:                     for all e ∈ outEdges(D(p)) do
10:                        if D(target(e)) ∈ U' and time(e) ≥ i_src then
11:                            c_cur ← false
12:                        end if
13:                    end for
14:                end if
15:            end for
16:        else
17:            c_cur ← false
18:        end if
19:        if c_cur and ∀(p1, p2) ∈ (U ∪ P_activated)^2 : N_out(p1) ∩ N_out(p2) = ∅ then
20:            c_res ← true
21:        end if
22:    end for
23:    return c_res
24: end procedure
```

Was tut dieser Algorithmus? Er betrachtet alle strengen Zusammenhangskomponenten von Zellen, die in der vorherigen Konfiguration c_1 variabel, aber beim Übergang zur aktuellen Konfiguration c_2 nicht aktiviert wurden. Für solche strengen Zusammenhangskomponenten U wird geprüft, ob alle Nachbarn inaktiv sind. Außerdem darf bei allen bisherigen Iterationen, die zeitlich nach c_1 entstanden, der Abhängigkeitsgraph $\mathcal{D}$ nur Kanten von $N(N(U))$ nach $N(U)$ enthalten. Ist all dies erfüllt, so müssen noch Überlappungen in $U \cup P_{\text{activated}}$ ausgeschlossen werden.

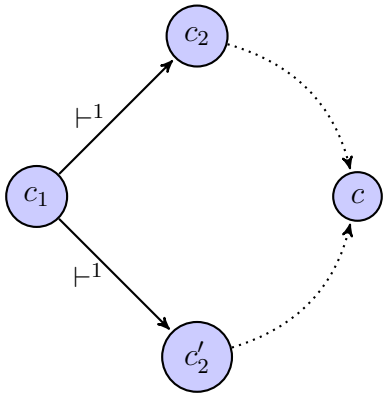

Abbildung 6.2: Die Konfigurationen im Überblick.

Wieso darf bei einer solchen isolierten Zelle U abgebrochen werden? Wir erinnern uns an das Grundgerüst. Dort war abgesichert, dass jeder Unterbaum einer Folgekonfiguration c_2', die sich ausschließlich durch die zusätzliche Aktivierung eines $U' \subseteq U$ in c_1 von c_2 unterscheidet, bereits genommen wurde und in $\mathcal{D}$ die entsprechenden Abhängigkeiten vermerkt wurden. Nun müssen wir zeigen, dass jede Endkonfiguration, die durch c_2 erreicht wird, auch schon aus so einem c_2' folgt. Abbildung 6.2 deutet das an.

Zunächst wissen wir, dass $N(U) \setminus U$ in c_2 ausschließlich aktiviert werden kann, wenn Zellen aus U aktiviert wurden. Da U noch immer aktiv ist, muss bis zum Erreichen einer Endkonfiguration mindestens eine Teilmenge $U' \subseteq U$ irgendwann einmal aktiviert werden. Wurde U' aktiviert, so sei c diese Konfiguration. Nun haben wir von c_1 zu c die Zonen U' und $R \setminus N(U)$ völlig unabhängig simuliert. Genauer gilt Folgendes:

$$\begin{cases} c_1|_{R\setminus N(U)} \vdash^*_{R\setminus N(U)} c|_{R\setminus N(U)} \\ c_1|_{U'} \vdash^1_{U'} c|_{U'} \\ c_1|_{\text{Rest}} = c|_{\text{Rest}} \end{cases}$$

Der Simulator erreicht also c auch schon von einer Folgekonfiguration c_2' von c_1, indem er einfach alles Notwendige auf $(R \setminus N(U))$ tut.

Wieder muss man sich am Ende fragen, ob es passieren kann, dass man im Kreis delegiert. Die Antwort ist, wie bereits zuvor, nein.

Es sei angemerkt, dass wir den Abhängigkeitsgraphen nicht richtig ausgenutzt haben. Es könnten Abhängigkeiten existieren, die eine Zelle in c_2 nicht isoliert machen, aber von einem irrelevanten Nachfolger von c_1 stammen. Wir kennen leider keine Möglichkeit, dieses Problem zu beheben, da wir aus Gründen begrenzten Speichers nicht jedem Knoten einen eigenen Abhängigkeitsgraphen spendieren können.

6.3 Laufzeit

Für den Speicher gilt, wie schon erwähnt:

$$M_{\text{greedy}} \coloneqq \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{res}}) + M_{\text{ca}}) \text{ bzw.}$$
$$M_{\text{greedy,const}} \coloneqq \mathcal{O}(N_{\text{all}} + N_{\text{var}} \cdot (N_{\text{initial}} + N_{\text{res}})).$$

Für die vollständige Laufzeit errechnen wir Laufzeiten für die beiden Funktionen. Die Unterroutine CALCSCCS läuft in $\mathcal{O}(N_{\text{n}} \cdot N_{\text{var}})$. Für die bewegbaren Zellen erhält man daher den Aufwand $N_{\text{var}} \cdot (N_{\text{n}} + T_{\text{ca}}) = N_{\text{var}} \cdot T_{\text{ca}}$. Die Berechnung der isolierten Zellen hat den maximalen Aufwand von ${N_{\text{n}}}^3 \cdot N_{\text{var}}$ bei der Abfrage an $\mathcal{D}$. Folgende Auflistung fasst die Laufzeiten zusammen.

$$T_{\text{filter}} \in \mathcal{O}(1) \ ,$$
$$T_{\text{makeSccs}} \in \mathcal{O}(1)$$
$$T_{\text{sortOut1}} = N_{\text{var}} \cdot T_{\text{ca}}$$
$$T_{\text{sortOut2}} = {N_{\text{n}}}^3 \cdot N_{\text{var}}$$

Einsetzen ergibt:

$$\begin{aligned} T_{\text{greedy}} &= \mathcal{O}(N_{\text{all}}) + N_{\text{initialized}} \cdot \mathcal{O}(N_{\text{var}} \cdot T_{\text{ca}}) \\ &\quad + N_{\text{passed1}} \cdot \mathcal{O}(N_{\text{n}} \cdot (N_{\text{var}} + {N_{\text{n}}}^2) \cdot T_{\text{ca}} + \log(N_{\text{res}})) \\ &\quad + N_{\text{passed2}} \cdot \mathcal{O}(\log(N_{\text{res}})). \end{aligned}$$

Da $N_{\text{passed2}} \leq N_{\text{passed1}}$ ist, entfällt der dritte Summand

$$\begin{aligned} T_{\text{greedy}} &= \mathcal{O}(N_{\text{all}} + N_{\text{initialized}} \cdot N_{\text{var}} \cdot T_{\text{ca}} \\ &\quad + N_{\text{passed1}} \cdot (N_{\text{n}} \cdot (N_{\text{var}} + {N_{\text{n}}}^2) \cdot T_{\text{ca}} + \log(N_{\text{res}}))). \end{aligned}$$

Der Term vereinfacht sich stark für einen konstanten Automaten, da $N_{\text{passed1}} \leq N_{\text{initialized}}$:

$$\begin{aligned} T_{\text{greedy,const}} &= \mathcal{O}(N_{\text{all}} + N_{\text{initialized}} \cdot N_{\text{var}} \\ &\quad + N_{\text{passed1}} \cdot (N_{\text{var}} + \log(N_{\text{res}}))) \\ &= \mathcal{O}(N_{\text{all}} + N_{\text{initialized}} \cdot N_{\text{var}} + N_{\text{passed1}} \cdot \log(N_{\text{res}})). \end{aligned}$$

Man sollte allerdings beachten, dass T_{ca} ein hoher konstanter Faktor sein kann.

6.4 Fazit

Der Greedy-Algorithmus ist ein etwas modifizierter Brute-Force-Algorithmus, der an zwei Stellen die Berechnung nicht weiter fortsetzt: Einmal während der Initialisierung

eines neuen Knotens, und einmal, wenn schon ähnliche Knoten durchlaufen wurden. Die Laufzeit ist in der Praxis oft linear in der Anzahl der besuchten Knoten, und diese Anzahl ist „hoffentlich“ gering. Der Speicher ist praktisch linear in der Laufzeit.

Wir haben auch gesehen, dass man auf einige Feinheiten wie den Ausschluss gleichzeitiger Aktivierung benachbarter Zellen oder die beschriebenen zyklischen Delegationen achten muss, wenn man so einen Algorithmus konstruiert.

Man beachte übrigens, dass der Greedy-Algorithmus noch lange nicht alle Duplikate erkennt. Das kann schon passieren, wenn $R = \{0, 1\}$ ist: Zelle 0 und 1 sind zu Beginn aktiv, dann ändert sich entweder Zelle 0 und danach Zelle 1, oder zuerst 2 mal Zelle 1 und anschließend Zelle 0 einmal. Sind am Ende beide Konfigurationen gleich, so wurde ein Duplikat auf zwei Wegen erreicht, auf denen unterschiedlich viele Aktivierungen stattgefunden haben. Mögen solche Automaten eher „konstruiert“ erscheinen, so zeigen sie doch sehr klar die Grenzen des Greedy-Algorithmus auf.

Kapitel 7

Implementierung

7.1 Grenzen

Die Implementierung ist durch die zugrundeliegende Hard- und Software beschränkt und macht im Gegensatz zum Pseudocode einige besondere Annahmen über den Automaten $(R, Q, N_{\text{in}}, N_{\text{out}}, \delta) \in \mathcal{C}$:

Die Menge aller aktivierbaren Zellen aus R ist endlich. Zwar darf R unendlich sein, doch die Menge aller aktivierbaren Zellen ist endlich. Grund dafür ist die Endlichkeit des Speichers.

Q ist endlich. Dabei ist $|Q|$ durch die Größe eines Integers beschränkt.

Es sind $N_{\text{in}} = \text{const}, N_{\text{out}} = \text{const}$ auf R.

Es ist $R = \mathbb{Z}^2$. Dies ist zur Ein- und Ausgabe sowie zur Verwaltung des Speichers praktisch. Man kann jedoch oft zu gegebenem R' eine sinnvolle Einbettung $R' \hookrightarrow \mathbb{Z}^2$ finden. Elemente aus $\mathbb{Z}^2$ ohne Urbild sollten dabei am besten mit Zuständen belegt werden, die sich nie ändern. Ein Beispiel für so eine Prozedur ist unser Umbau von $\mathcal{C}_{\text{st}}$.

7.2 Hardware

Die Implementierung wurde auf einem „Intel® Core™ i5-3570 CPU @ 3.40GHz“ kompiliert und ausgeführt. Diese CPU der Serie „Ivy-Bridge“ hat eine „Turbo-Taktfrequenz“ von 3.80GHz, 4×256kB L2-Cache und 6MB geteilten L3-Cache. Das genaue Datenblatt findet man auf der Website von Intel [ark13]. Für den RAM haben wir zwei „KHX1600C9D3/4G“ im Dual-Channel-Modus verbaut. Diese RAM-Riegel sind aus der Serie „Kingston HyperX®“ und bieten je 4GB DDR3-1600 CL9 SDRAM [kin13].

Alles in allem handelt es sich um einen im Jahr 2013 top-aktuellen Personal Computer.

Alle Laufzeiten sind auf diesem System gemessen worden.

7.3 Programmiersprache

Als Programmiersprache setzen wir C++ ein. C++ ist standardisiert und kombiniert sehr hohe Performanz mit Konstrukten einer objektorientierten Hochsprache.

An vielen Stellen nutzen wir Konstrukte („Features“), die erst mit C++11 in den Standard aufgenommen wurden. Die Compiler clang und gcc unterstützen C++11 in den Versionen 3.3 [cla14] bzw 4.8.1 [gcc14] nahezu vollständig.

7.4 Compiler

Unser Code kompiliert und läuft gleichwertig sowohl mit clang, als auch mit gcc. Wir empfehlen jedoch clang, da er schneller und mit weniger Arbeitsspeicher kompiliert.

7.5 Bibliotheken

Wir beschränken uns auf zwei Bibliotheken:

1. boost [www14b]
2. sca-toolsuite [Lor14]

sca-toolsuite enthält eine Bibliothek mit vielen Werkzeugen für Zellularautomaten. Besonders wichtig für uns sind die Simulatoren.

Boost wird benötigt, um Graphen zu zeichnen. Außerdem wird es von sca-toolsuite für den Parser des Simulators genutzt. Wir halten boost für eine gute Wahl, da es „eine der höchst angesehenen und meisterhaftesten C++-Bibliotheks-Projekten weltweit“ [SA04] ist und außerdem für sehr viele Plattformen frei zur Verfügung steht.

Allerdings gibt es einen Bug in der aktuellen Version von boost::graph (1.56), der auf einer Inkompatibilität mit C++11 beruht [www14a]. Daher nutzen wir boost::graph nur an wenigen Stellen beim Evaluieren der Ergebnisse und zum Erstellen von Debug-Graphen. Falls Probleme beim Kompilieren auftreten, so kann boost::graph mittels `-DNO_BOOST_GRAPH` ausgeschlossen werden.

7.6 Benutzung

Dieser Abschnitt soll nur kurz erklären, wie man die Implementierung benutzt. Detaillierte Instruktionen, die insbesondere das Kompilieren betreffen, sind dem Quellcode beigelegt.

Wir haben den Algorithmus von der Ausgabe getrennt. Der Algorithmus gibt am Ende serialisierte Dateien aus, die man dann mit einem anderen Programm evaluieren kann. Die erzeugten Dateien sind typischerweise nur wenige Kilobyte groß. Der Vorteil dieser Trennung ist, dass man den Algorithmus für verschiedene Arten von Ausgaben nur einmal laufen lassen muss. Außerdem kann man so die Ausgaben auf „starken“ Computern berechnen und dann auf „schwachen“ Computern visualisieren.

Den Algorithmus kann man zum Beispiel wie folgt starten:

```
./search circuit.tbl 3 nodump greedy < input/crossing.txt
```

Dabei ist `circuit.tbl` die Tabellen-Datei, in der der Zellularautomat gespeichert ist[1], `3` der Wert, den die Randzellen einnehmen sollen, damit sie nie variabel werden, `nodump` eine Angabe, die die Menge der Debug-Ausgaben beschränkt, und `greedy` der Name des Algorithmus. Man kann danach Ausgaben mittels

```
./eval <Modus> < results.dat
```

erzeugen. So generiert die Eingabe

```
cat input/crossing.txt |
./search circuit.tbl 3 nodump greedy pipe |
./eval tex |
../io/tik complete
```

eigenständige LaTeX-Dokumente, die die Funktionstabelle für eine Kreuzung enthalten. Es gibt mit

```
./eval help
```

noch viele andere Modi zu entdecken.

7.7 Verfügbarkeit

Der Quellcode wir nach der Veröffentlichung dieser Arbeit zum Projekt „sca-toolsuite“ hinzugefügt. Er kann dann von dort [Lor14] heruntergeladen werden.

[1] Die Tabelle muss Paare von Eingabe- und Ausgabezuständen enthalten. Das genaue Format ist das von sca-toolsuite vorgegebene, welches dort durch den Konvertierer für Automaten, `ca/converter`, erzeugt werden kann.

Kapitel 8

Ausgaben

Dieses Kapitel enthält einige Ausgaben, die unser Algorithmus für ausgewählte Eingaben geliefert hat. Es werden die Zellularautomaten von Schneider [Sch12] und von Morrison und Ulidowski [MU14] aufgeführt, und es wird sich zeigen, dass sie sich tatsächlich so verhalten, wie vermutet wurde.

Zu jedem Modul wird es wieder drei Abbildungen und eine Statistik geben. Hierbei gilt, wie bereits in Kapitel 4 besprochen, dass die erste Abbildung die Aufteilung der Eingabe in Komponenten, die zweite die initialen Ausgabebelegungen und die dritte die „Funktionstabelle“ zeigen. Die Statistiken sind jeweils von unserem schnellsten Algorithmus, dem Greedy-Algorithmus, auf unserem Testsystem (siehe Kapitel 7) erzeugt worden. Sie zeigen, wie Effizient der Greedy-Algorithmus in der Praxis ist.

Wir haben bei der Eingabe auf triviale Eingaben, bei denen R invariabel ist, verzichtet.

8.1 $\mathcal{C}_{\text{bc3}}$

Wir werden hier die Module von $\mathcal{C}_{\text{bc3}}$ auflisten. Fachausdrücke aus dieser Literatur sind in Anführungszeichen geschrieben.

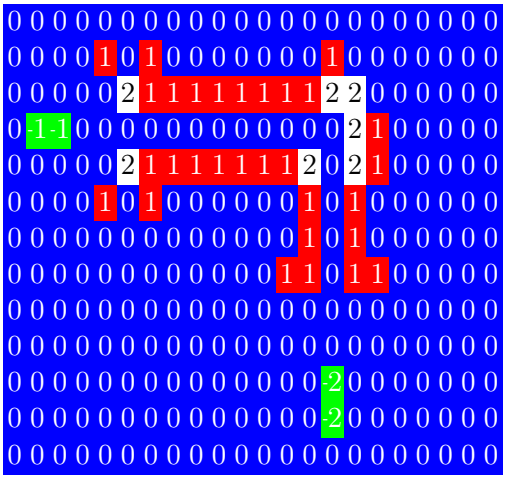

Tabelle 8.1: Gitterbelegung zum Modul „Röhre“

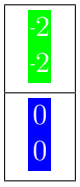

Tabelle 8.2: Initiale Ausgabezonen zum Modul „Röhre“

-1 -1		-1 -1	-2 -2	scc-size
i/o		i/o	out	
1 2	$\mapsto$	0 0	1 2	$= 1$

Tabelle 8.3: Funktionstabelle zum Modul „Röhre“

Konfigurationen insgesamt:	335
davon nicht bewegbar:	172
davon nicht isolierbar:	91
Laufzeit (s):	<1

Tabelle 8.4: Statistiken zum Modul „Röhre“

Tabellen 8.1, 8.2, und 8.3 und 8.4 zeigen die Resultate für das „Röhren-Modul“. Das Ergebnis ist wenig überraschend: Die „Röhre“ leitet das „Signal“[1] wie gewünscht. Wir zeigen damit sowohl ein funktionierendes Beispiel für die Fortbewegung in der Röhre, in der auch eine „Umleitung“ enthalten ist, als auch außerhalb. „Ein-“ und „Ausgänge“ sind ebenfalls abgedeckt.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	2	-1	2	0	0	0	0	0	0	0	0	0	0
0	1	1	-1	1	1	0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0	1	0	0	0	
0	0	2	0	2	1	1	1	1	1	1	0	0	0	
0	1	2	0	0	0	0	-2	-2	0	0	0	0	0	
0	0	2	0	2	1	1	1	1	1	1	0	0	0	
0	1	1	0	1	0	0	0	0	0	1	0	0	0	
0	0	1	0	1	0	0	0	0	0	0	0	0	0	
0	0	1	0	1	0	0	0	0	0	0	0	0	0	
0	1	1	-3	1	1	0	0	0	0	0	0	0	0	
0	0	2	-3	2	0	0	0	0	0	0	0	0	0	
0	1	0	0	0	1	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Tabelle 8.5: Gitterbelegung zum Modul „Abzweigung“

[1] Signal gemäß [Sch12]. Gemeint ist eine variable Zelle im Zustand „2“, „gefolgt“ von einer im Zustand „1“.

-1 -1	-3 -3
0 0	0 0

Tabelle 8.6: Initiale Ausgabezonen zum Modul „Abzweigung“

-2-2		-2-2	-1 -1	-3 -3	scc-size
i/o		i/o	out	out	
2 1	$\mapsto$	0 0	2 1	1 2	$= 1$

Tabelle 8.7: Funktionstabelle zum Modul „Abzweigung“

Konfigurationen insgesamt:	3355
davon nicht bewegbar:	723
davon nicht isolierbar:	175
Laufzeit (s):	<1

Tabelle 8.8: Statistiken zum Modul „Abzweigung“

In Tabellen 8.5, 8.6, und 8.7 sieht man eine „Abzweigung“. Ein von rechts eingehendes Signal wird tatsächlich nach oben und unten weitergeleitet. Man liest in Tabelle 8.8 nach, dass Module dieser Größe keine große Herausforderung an den Algorithmus sind, da die Anzahl der Knoten kaum in die Tausender geht. Hingegen beträgt die Laufzeit für einen reinen Brute-Force-Algorithmus ohne Speichern von Duplikaten bereits mehrere Minuten auf einem modernen Rechner.

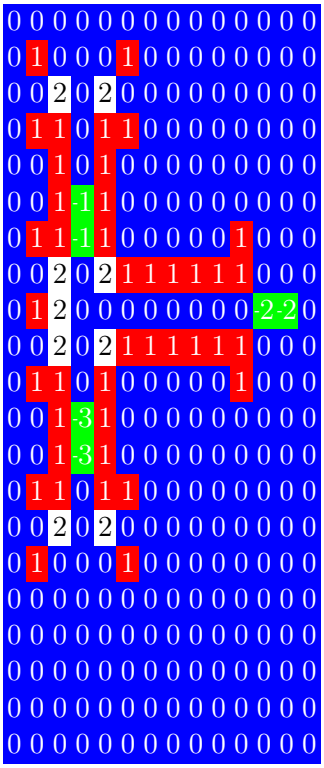

Tabelle 8.9: Gitterbelegung zum Modul „Zusammenführung“

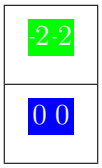

Tabelle 8.10: Initiale Ausgabezonen zum Modul „Zusammenführung“

-1 -1	-3 -3		-1 -1	-3 -3	-2 -2	scc-size
i/o	i/o		i/o	i/o	out	
1 2	0 0	$\mapsto$	0 0	0 0	1 2	$= 1$
0 0	2 1	$\mapsto$	0 0	0 0	1 2	$= 1$

Tabelle 8.11: Funktionstabelle zum Modul „Zusammenführung“

Konfigurationen insgesamt:	355
davon nicht bewegbar:	196
davon nicht isolierbar:	97
Laufzeit (s):	<1

Tabelle 8.12: Statistiken zum Modul „Zusammenführung“

Tabellen 8.9, 8.10 und 8.11 zeigen eine „Zusammenführung“. Der Aufbau ist identisch mit dem der Abzweigung. Es darf hier jedoch nur ein Signal oben oder unten anliegen, und dieses wird dann zum „Ausgang“ transportiert.

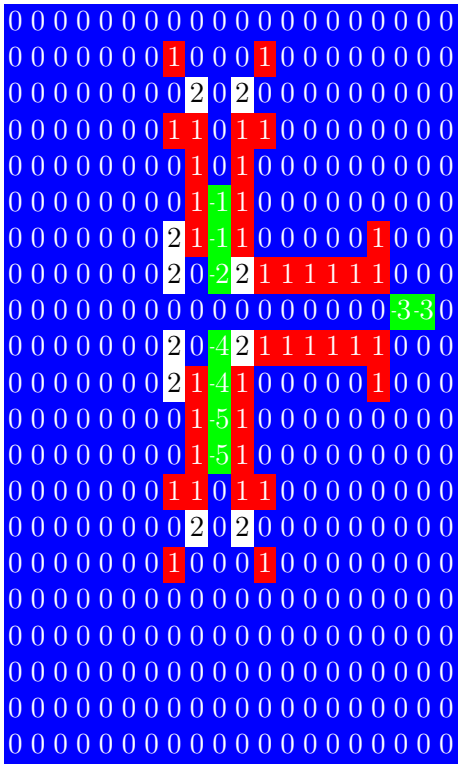

Tabelle 8.13: Gitterbelegung zum Modul „Synchronisierung“

-2	-3 -3	-4 -4
0	0 0	0 0

Tabelle 8.14: Initiale Ausgabezonen zum Modul „Synchronisierung“

-1 -1	-5 -5		-1 -1	-5 -5	-2	-3 -3	-4 -4	scc-size
i/o	i/o		i/o	i/o	out	out	out	
1 2	0 0	$\mapsto$	0 1	0 0	2	0 0	0 0	$= 1$
0 0	2 1	$\mapsto$	0 0	0 0	0	0 0	2 1	$= 1$
1 2	2 1	$\mapsto$	0 0	0 0	0	1 2	0 0	$= 1$

Tabelle 8.15: Funktionstabelle zum Modul „Synchronisierung“

Konfigurationen insgesamt:	559
davon nicht bewegbar:	212
davon nicht isolierbar:	107
Laufzeit (s):	<1

Tabelle 8.16: Statistiken zum Modul „Synchronisierung“

Tabellen 8.13, 8.14, und 8.15 zeigen in gewisser Weise das Pendant zur Abzweigung, die „Synchronisierung“. Liegen zwei Signale an, also oben und unten, so vereinigen sie sich und bewegen sich nach rechts fort. Falls nur ein Signal anliegt, so bewegt es sich in die Mitte und wartet dort.

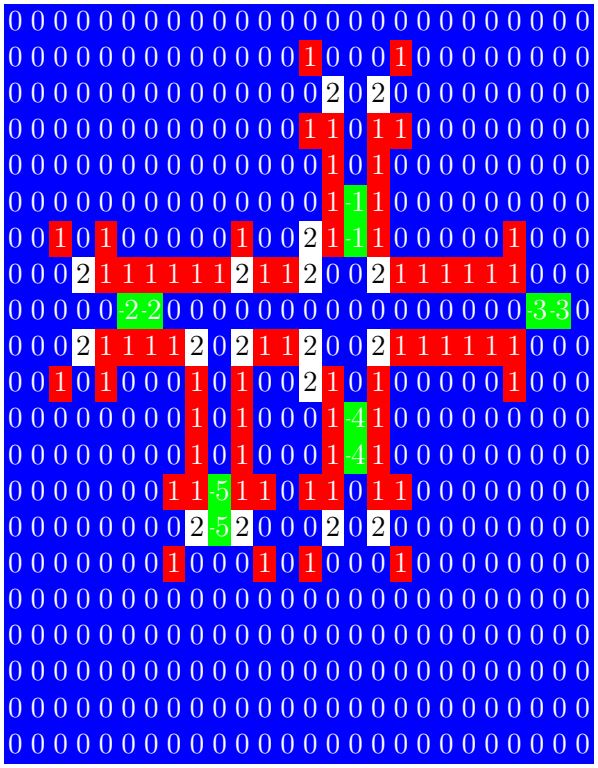

Tabelle 8.17: Gitterbelegung zum Modul „Synchronisierung mit Clearline“

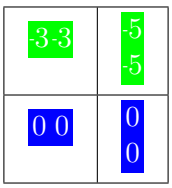

Tabelle 8.18: Initiale Ausgabezonen zum Modul „Synchronisierung mit Clearline“

-1 -1	-2-2	-4 -4		-1 -1	-2-2	-4 -4	-3-3	-5 -5	scc-size
i/o	i/o	i/o		i/o	i/o	i/o	out	out	
1 2	0 0	2 1	$\mapsto$	0 0	0 0	0 0	1 2	0 0	$= 1$
1 2	1 2	0 0	$\mapsto$	0 0	0 0	0 0	0 0	1 2	$= 1$
0 0	1 2	2 1	$\mapsto$	0 0	0 0	0 0	0 0	1 2	$= 1$

Tabelle 8.19: Funktionstabelle zum Modul „Synchronisierung mit Clearline“

Konfigurationen insgesamt:	2912
davon nicht bewegbar:	908
davon nicht isolierbar:	350
Laufzeit (s):	<1

Tabelle 8.20: Statistiken zum Modul „Synchronisierung mit Clearline“

Tabellen 8.17, 8.18, und 8.19 zeigen eine „Synchronisierung mit Clearline“: Es handelt sich dabei um eine Erweiterung der Synchronisierung. Die Synchronisierung funktioniert dabei wie gehabt. Allerdings gilt, dass genau dann, wenn einer ihrer Eingänge belegt ist, dies auch für die Clearline gelten muss. Diese löscht dann das unbewegliche Signal des entsprechenden Eingangs und gibt ein separates Signal in Zone -5 aus. Die Statistiken in Tabelle 8.20 zeigen: Auch dieses Modul war noch keine Herausforderung an den Greedy-Algorithmus.

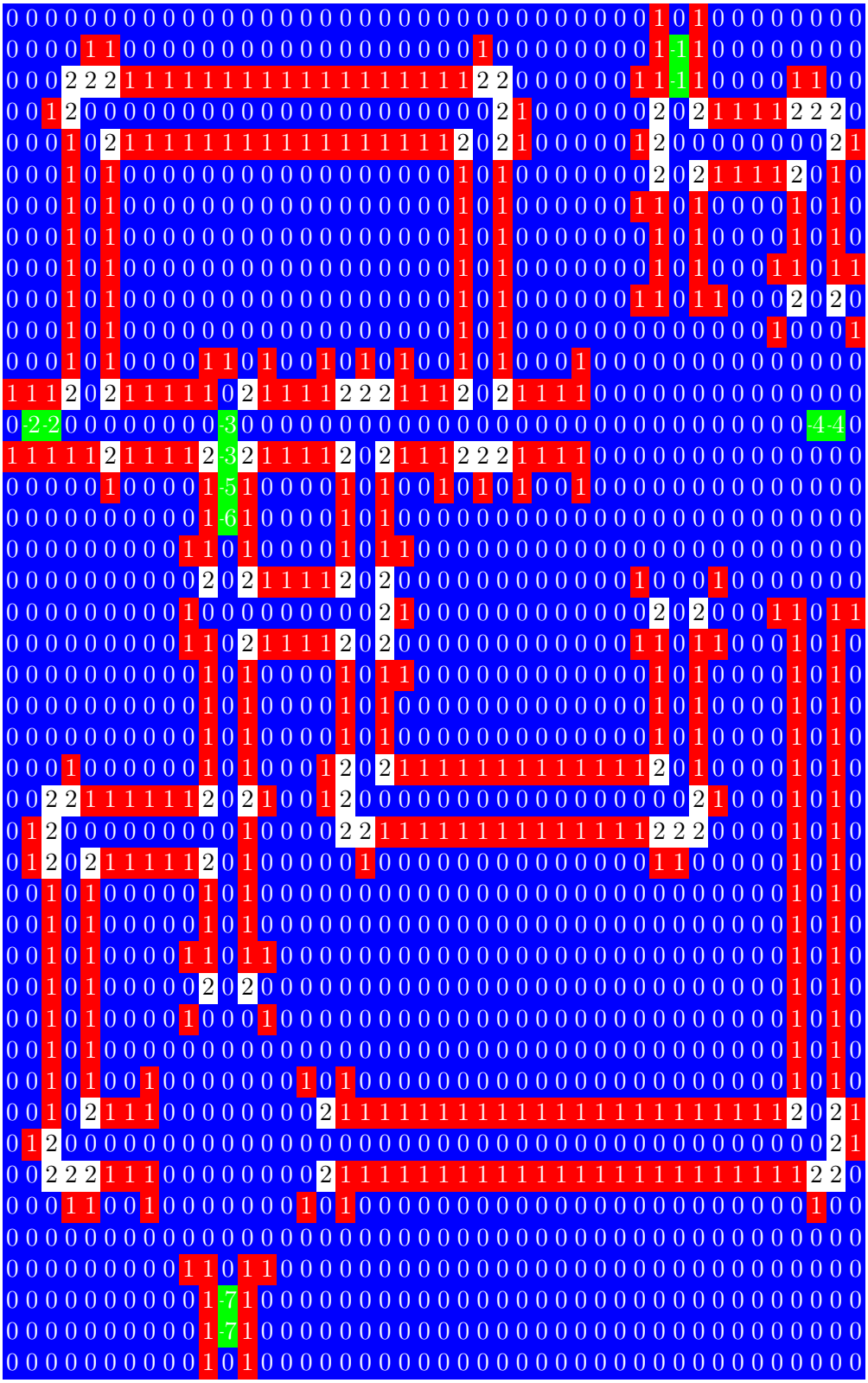

Tabelle 8.21: Gitterbelegung zum Modul „Sichere Drahtkreuzung“

-1 -1	3 3	-4-4
0 0	0 0	0 0

Tabelle 8.22: Initiale Ausgabezonen zum Modul „Sichere Drahtkreuzung“

-2-2	-5	-7 -7	-6		-2-2	-5	-7 -7	-1 -1	-3 -3	-4-4	scc-size
i/o	i/o	i/o	in		i/o	i/o	i/o	out	out	out	
1 2	2	2 1	1	$\mapsto$	0 0	1	0 0	2 1	2 2	1 2	≥ 34

Tabelle 8.23: Funktionstabelle zum Modul „Sichere Drahtkreuzung“

Konfigurationen insgesamt:	1402721
davon nicht bewegbar:	114838
davon nicht isolierbar:	6735
Laufzeit (s):	<11

Tabelle 8.24: Statistiken zum Modul „Sichere Drahtkreuzung“

In Tabellen 8.21, 8.22, und 8.23 sieht man eine „Sichere Drahtkreuzung“. Tabelle 8.24 zeigt, dass dieser Fall eine mittelmäßige Anforderung darstellt. Wie korrekt angezeigt wird, haben am Ende in jedem Fall beide Signale die Kreuzung passiert, und das mittlere Signal läuft weiter im Kreis und bildet dabei mindestens 34 Konfigurationen.

8.2 $\mathcal{C}_{\text{st}}$

Wir beschränken uns für $\mathcal{C}_{\text{st}}$ auf die interessanteste Eingabe, das „Fork and Join“-Modul.

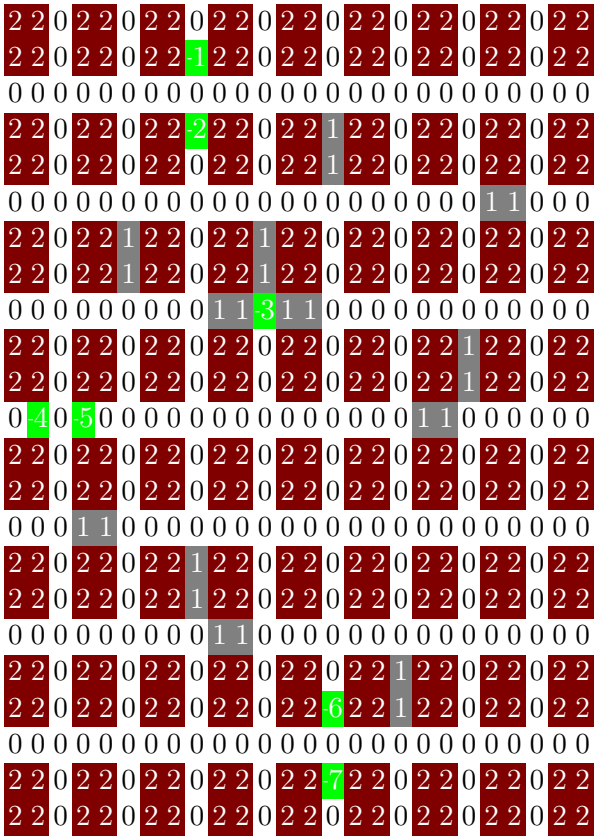

Tabelle 8.25: Gitterbelegung zum Modul „Fork and Join“

-1	-4	-7
0	0	0

Tabelle 8.26: Initiale Ausgabezonen zum Modul „Fork and Join“

-2	-5	-6	-3		-2	-5	-6	-1	-4	-7	scc-size
i/o	i/o	i/o	in		i/o	i/o	i/o	out	out	out	
0	1	0	0	$\mapsto$	0	0	0	1	0	1	≥ 2
1	0	1	0	$\mapsto$	0	0	0	0	1	0	≥ 2

Tabelle 8.27: Funktionstabelle zum Modul „Fork and Join“

Konfigurationen insgesamt:	10924
davon nicht bewegbar:	8495
davon nicht isolierbar:	5228
Laufzeit (s):	<6

Tabelle 8.28: Statistiken zum Modul „Fork and Join“

Tabellen 8.25, 8.26, und 8.27 zeigen das „Fork and Join“-Modul. Liegt links eine variable Zelle an, so teilt sie sich auf und „bewegt“ sich nach oben und unten. Umgekehrt entsteht links eine variable Zelle, wenn oben und unten initial variable Zellen anliegen. Das Modul funktioniert also wie erwartet.

Das Modul ist laut Tabelle 8.28 eine mittelmäßige Anforderung an den Algorithmus. An der hohen Knotenzahl kann man ablesen, dass es einige unerkannte Duplikate gab. Grund dafür ist die Schwierigkeit mit Eingabezone -5, deren Nachbarschaft sich fast dauernd ändert, und die andere variable Zellen „blockieren“ kann. Das Modul ist übrigens auch das schwierigste aus $\mathcal{C}_{\text{st}}$.

Kapitel 9

Fazit

Wir haben uns in dieser Arbeit mit dem Problem der vollständigen Simulation von Zellularautomaten beschäftigt.

Es wurden mehrere Algorithmen vorgestellt, die für einen Automaten und zugehörige Anfangskonfigurationen die resultierenden Endkomponenten berechnen. Die meisten Algorithmen konnten zu wenige Duplikate verhindern. Der Greedy-Algorithmus erwies sich jedoch als praktikabel für einige Eingaben. Mit ihm konnten wir die ausstehenden Beweise für die Richtigkeit einiger Automaten und deren „Module“ liefern. Außerdem konnten wir mit dem Split-Algorithmus Ergebnisse liefern, die auch für theoretische Betrachtungen sinnvoll sein können.

Die Laufzeiten der Algorithmen hing primär von den Duplikaten ab. Der Greedy-Algorithmus konnte selbst Ausgaben für Eingaben mittlerer Größe mit moderner Hardware in wenigen Sekunden berechnen. Die Speicherauslastung war stets sehr gering und stieg nur unwesentlich mit der Zeit. So konnte man bei großen Eingaben minutenlang keinen Anstieg der Speichernutzung beobachten.

Alle vorgestellten Algorithmen waren unabhängig vom zugrundeliegenden Automaten. Die Algorithmen funktionieren sogar mit einer abgeschwächten Definition eines Zellularautomaten, so dass zu hoffen ist, dass man sie vielleicht sogar in anderen diskreten Modellen verwenden kann.

Darüber hinaus hat diese Arbeit auch hilfreiche Mittel zur Visualisierung eingeführt. Dazu gehören neben der Funktionstabelle auch der Konfigurations- und der Abhängigkeitsgraph.

Diese Arbeit lässt mehrere Fragen offen, besonders beim Greedy-Algorithmus. Bei der Überprüfung, ob Zellen bewegbar sind, haben wir, wie schon beschrieben, viele Möglichkeiten übersprungen. Hinzu kommt, dass, wenn Zellen bewegbar sind, diese Information nicht genutzt wurde um zu folgern, dass andere Mengen von Zellen bewegbar sind. Dies könnte das kostspielige Überprüfen deutlich abkürzen. Auch die Technik der isolierten Zellen wurde noch nicht vollständig genutzt, da hier ein unpassender Abhängigkeitsgraph benutzt wird, der zu viele Abhängigkeiten hat und daher einige Zellen als nicht isoliert darstellt.

Darüber hinaus gibt es die üblichen Fragen, die für Algorithmen oft im Raum

stehen. Allen voran die Frage, ob man die Algorithmen sinnvoll parallelisieren kann. Da für den Fall, dass die Eingabe mehrere Anfangskonfigurationen enthält, diese immer unabhängig nacheinander abgearbeitet werden, scheint eine Parallelisierung hier unkompliziert. Doch auch der Berechnungsgraph für eine einzelne Eingabe kann vermutlich leicht aufgeteilt werden, da die Knoten, die für die Ausgabe benötigt werden, meist nur einen sehr geringen Bruchteil ausmachen. Praktisch gesehen wäre eine Parallelisierung auch sehr vorteilhaft, da die Laufzeit für manche Eingaben sehr hoch sein kann.

Auch Erweiterungen sind denkbar, zum Beispiel für asynchrone Zellularautomaten mit Wahrscheinlichkeiten oder nichtdeterministische Zellularautomaten.

Insgesamt ist diese Arbeit also nur eine unvollständige Einführung in ein komplexes Thema, das eine breite Anwendung hat und noch viele Herausforderungen bieten kann.

Index

Literaturverzeichnis

[ark13] ark.intel.com. Ark | intel® core™ i5-3570k processor (6m cache, up to 3.80 ghz). http://ark.intel.com/de/products/65520, 2013.

[cla14] clang.llvm.org. C++ Support in Clang. http://clang.llvm.org/cxx_status.html, 2014.

[gcc14] gcc.gnu.org. C++0x/c++11 Support in GCC. https://gcc.gnu.org/projects/cxx0x.html, 2014.

[HLM+08] Alexander E. Holroyd, Lionel Levine, Karola Mészáros, Yuyal Peres, James Propp, and David B. Wilson. Chip-Firing and Rotor-Routing on Directed Graphs. In Vladas Sidoravicius and Maria Eulália Vares, editors, *In and Out of Equilibrium 2*, volume 60 of *Progress in Probability*, pages 331--364. Birkhäuser Basel, 2008.

[kin13] kingston.com. Khx1600c9d3/4g. http://www.kingston.com/datasheets/KHX1600C9D3_4G.pdf, 2013.

[Lor14] Johannes Lorenz. sca-toolsuite. https://github.com/JohannesLorenz/sca-toolsuite, 2014.

[MU14] Daniel Morrison and Irek Ulidowski. Direction-Reversible Self-Timed Cellular Automata for Delay-Insensitive Circuits. In Jarosław Wąs, GeorgiosCh. Sirakoulis, and Stefania Bandini, editors, *Cellular Automata*, volume 8751 of *Lecture Notes in Computer Science*, pages 367--377. Springer International Publishing, 2014.

[SA04] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[Sch12] Oliver Schneider. Schaltkreise in einem Zellularautomat mit 3 Zuständen, 2012.

[Tar72] Robert Tarjan. Depth-First Search and linear Graph Algorithms. *SIAM journal on computing*, 1(2):146--160, 1972.

[www14a] www.boost.org. #10382 (1.56.0 Graph adjacency_list has compile errors on g++ 4.6.4) – Boost C++ Libraries. https://svn.boost.org/trac/boost/ticket/10382, 2014.

[www14b] www.boost.org. Boost C++ Libraries. http://www.boost.org/, 2014.

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorstehende Diplomarbeit selbstständig angefertigt und die benutzten Hilfsmittel, Quellen sowie die befragten Personen und Institutionen vollständig angegeben habe. Darüber hinaus versichere ich, dass ich die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) in der gültigen Fassung beachtet habe.

Ort, Datum Unterschrift