5 Sortieren in eindimensionalen Zellularautomaten

5.1 Für alle $x \in A$ und $w \in A^*$ bezeichne im folgenden $N_x(w)$ die Anzahl der Vorkommen des Symboles x in dem Wort w.

5.2 Problem. (Eindimensionales Sortieren von Bits) Es sei $A = \{0, 1\}$. Gesucht ist ein ZA mit $R = \mathbb{Z}$, der jedes Muster $w \in A^+$ in das Muster $0^{N_0(w)}1^{N_1(w)}$ überführt.

Bei dieser Aufgabenstellung handelt es sich um ein einfaches Sortierproblem. Dabei liegt der einfache Fall vor, dass jede Zelle eines der zu sortierenden Daten vollständig speichern kann. Darauf wollen wir uns zunächst auch beschränken.

5.3 Algorithmus. (`0-1-SORTIEREN-1DIM`) Man wähle $R = \mathbb{Z}$, $N = H_1^{(1)}$, $Q = A \cup \{\square\}$ und δ so, dass gilt ($\times$ bedeute wie schon früher, dass der entsprechende Zustand irrelevant ist):

$l(-1)$	$l(0)$	$l(1)$	$\delta(l)$
1	0	$\times$	1
$\times$	1	0	0
und in allen anderen Fällen			
$\times$	s	$\times$	s

5.4 Durch die angegebene Tabelle wird δ offensichtlich vollständig und widerspruchsfrei festgelegt. Salopp gesprochen werden in jedem Schritt bei jedem Paar der Form `10` die beiden Symbole vertauscht.

Das heißt, in jedem Schritt kommen mindestens eine `0` und mindestens eine `1`, die noch nicht ihre „endgültige sortierte Position“ erreicht haben, dieser eine Zelle näher. Der Algorithmus leistet also das Gewünschte.

An dieser Argumentation sieht man leider noch nicht, dass es für ein Muster der Länge n höchstens $n - 1$ Schritte (i.e. Konfigurationsübergänge) dauert, bis es sortiert ist. Man überlege, warum das so ist.

5.5 Problem. (Eindimensionales Sortieren von „Trits“) Es sei $A = \{0, 1, 2\}$. Gesucht ist ein ZA mit $R = \mathbb{Z}$, der jedes Muster $w \in A^+$ in das Muster $0^{N_0(w)}1^{N_1(w)}2^{N_2(w)}$ überführt.

Man kann nun leider *nicht* ganz analog zum obigen Fall des `0-1`-Sortierens vorgehen, denn wohin soll man in der Situation `210` denn die `1` tauschen; nach links oder nach rechts? Da diese Information nicht mehr aus den zu sortierenden Daten, die eine Zelle bei ihren Nachbarn sieht, abgeleitet werden kann, wird sie im folgenden Algorithmus explizit zur Verfügung gestellt:

5.6 Algorithmus. (`ODD-EVEN-TRANSPOSITION-SORT`) Die Idee und der Name für den folgenden Algorithmus stammen von Knuth (1998).

Man wähle $R = \mathbb{Z}$, $N = H_1^{(1)}$, $Q = A \times \{L, R, \_\} \cup \{\square\}$ und δ gemäß der folgenden Tabelle, wobei wir der größeren Übersichtlichkeit wegen die $s \in A$ einfach mit den $(s, \_)$ identifiziert haben:

$l(-1)$	$l(0)$	$l(1)$	$\delta(l)$
für die Initialisierung:			
$\square$	$(s,\_)$	$\times$	(s,L)
(t,L)	$(s,\_)$	$\times$	(s,L)
für das eigentliche Sortieren:			
(s,R)	(t,L)	$\times$	$(\max(s,t),R)$
$\times$	(s,R)	(t,L)	$(\min(s,t),L)$
$\square$	(t,L)	$\times$	(t,R)
$\times$	(s,R)	$\square$	(s,L)
in allen nicht bereits spezifizierten Fällen:			
$\times$	s	$\times$	s

5.7 Beispiel.

	1	0	2	1	0	
	1	0	2	1	0	
	L					
	1	0	2	1	0	
	R	L				
	0	1	2	1	0	
	L	R	L			
	0	1	2	1	0	
	R	L	R	L		
	0	1	1	2	0	
	L	R	L	R	L	
	0	1	1	0	2	
	R	L	R	L	R	
	0	1	0	1	2	
	L	R	L	R	L	
	0	0	1	1	2	
	R	L	R	L	R	
	0	0	1	1	2	
	L	R	L	R	L	

Verglichen und ggf. vertauscht werden immer die Daten zweier Zellen, die ein RL-Paar bilden. Gibt man einem solchen Paar zum Beispiel die Nummer seiner linken Zelle, so finden die Vergleichs-/Vertausch-Operationen immer abwechselnd in ungeraden und in geraden Paaren statt. Daher der Name des Algorithmus.

5.8 Übung. Wie man an der Beispielberechnung sieht, wird keine Endkonfiguration erreicht. Man ändere den Algorithmus so ab, dass dies nach Beendigung des Sortiervorganges geschieht.

Der Zeitbedarf für den Algorithmus ist proportional zur Anzahl zu sortierender Elemente (siehe auch Beweis 5.13). Dies ist asymptotisch optimal. Warum?

Wie und um wieviel kann Algorithmus 5.6 kann noch beschleunigt werden? (Man beachte, dass man nicht für die ganze Berechnung darauf festgelegt ist, dass jede Zelle nur einen Wert enthält.) Wie schnell kann man werden, wenn man annimmt, dass die Zellen vor Beginn der Rechnung schon geeignet mit L und R markiert werden dürfen?

5.9 Man kann sich davon überzeugen, dass Algorithmus 5.6 auch für mehr als drei verschiedene Werte funktioniert. Zur Erzeugung der „Richtungsinformation“ haben wir die Tatsache benutzt, dass sich Zellen am Rand als solchen identifizieren können, und diese Information dann herangezogen, um alle Zellen zu „orientieren“.

5.10 Lemma. Algorithmus 5.6 ist korrekt.

Zum Beweis benutzen wir

5.11 Lemma. (0-1-Sortier-Lemma von Knuth (1998)) Wenn ein Sortieralgorithmus ausschließlich aus Operationen der Art „Vergleich-und-Austausch-wenn-größer“ besteht und wenn von vorneherein (unabhängig von den zu sortierenden Daten) feststeht, an welchen Positionen Werte miteinander verglichen und gegebenenfalls vertauscht werden, dann gilt: Der Algorithmus sortiert genau dann alle Eingabedatensätze, wenn er alle Eingabedatensätze sortiert, die nur aus Nullen und Einsen bestehen.

5.12 Beweis (von Knuths Lemma) Die eine Richtung des Beweises ist trivial.

Zum Beweis der anderen sei $x_1, \ldots, x_n$ ein Eingabedatensatz, der falsch behandelt wird. Wir zeigen, dass dann auch schon ein Eingabedatensatz falsch behandelt wird, der nur aus 0 und 1 besteht.

Nehmen wir an, π und σ wären zwei Permutationen der Zahlen 1 bis n, so dass die Sortierung $x_{\pi(1)} \leqslant x_{\pi(2)} \leqslant \cdots \leqslant x_{\pi(n)}$ die richtige wäre, der Algorithmus die Werte aber in der falschen Reihenfolge $x_{\sigma(1)}, x_{\sigma(2)}, \cdots, x_{\sigma(n)}$ liefert.

Es sei k die erste Stelle, an der die Sortierung verkehrt ist, also:

$$\begin{aligned} x_{\sigma(i)} &= x_{\pi(i)} \quad \text{für } 1 \leqslant i < k \\ x_{\sigma(k)} &> x_{\pi(k)} \end{aligned}$$

Offensichtlich gibt es mindestens k Werte $x_{\pi(1)}, \ldots, x_{\pi(k)}$, die kleiner oder gleich $x_{\pi(k)}$ sind, aber nur $k-1$ von ihnen „landen“ links von $x_{\sigma(k)}$. Also gibt es ein $l \leqslant k$ und ein $r > k$ mit $x_{\sigma(r)} = x_{\pi(l)}$ (und $x_{\pi(l)} \leqslant x_{\pi(k)}$).

Wir haben nun zu zeigen, dass es auch eine 0-1-Folge $x_1^*, \ldots, x_n^*$ gibt, die falsch behandelt wird. Dazu definiere man:

$$x_i^* = \begin{cases} 0 & \text{falls } x_i \leqslant x_{\pi(k)} \\ 1 & \text{falls } x_i > x_{\pi(k)} \end{cases}$$

Dann gilt:

$$\begin{aligned} x_i > x_j \quad &\Longrightarrow \quad \left(x_j > x_{\pi(k)} \Longrightarrow x_i > x_{\pi(k)}\right) \\ &\Longrightarrow \quad \left(x_j^* = 1 \Longrightarrow x_i^* = 1\right) \\ &\Longrightarrow \quad x_i^* \geqslant x_j^* \end{aligned}$$

und analog

$$x_i \leqslant x_j \quad \Longrightarrow \quad x_i^* \leqslant x_j^* \,.$$

Folglich gilt aufgrund der generellen Voraussetzung des Lemmas: Der Algorithmus liefert für die Eingabe $x_1^*, \ldots, x_n^*$ das gleiche Ergebnis, das man erhält, wenn man an $x_1^*, \ldots, x_n^*$ die gleichen Vertauschungen vornimmt, die der Algorithmus an $x_1, \ldots, x_n$ durchführt. Man beachte, dass der Algorithmus selbst für die x^* unter Umständen weniger Vertauschungen vornimmt. Das macht aber nichts, weil das Verhalten nur in den Fällen anders ist, in denen $x_i^* = x_j^*$ ist. Nach Beendigung des Algorithmus ergibt sich daher die Reihenfolge $x_{\sigma(1)}^*, \ldots, x_{\sigma(k)}^*, \ldots, x_{\sigma(r)}^*, \ldots, x_{\sigma(n)}^*$.

Nach Definition ist aber $x^*_{\sigma(k)} = 1$ (da $x_{\sigma(k)} > x_{\pi(k)}$) und $x^*_{\sigma(r)} = x^*_{\pi(l)} = 0$ (da $x_{\pi(l)} \leqslant x_{\pi(k)}$). Es ergibt sich also:

$$\begin{array}{rccccccc} & x^*_{\sigma(1)} & \cdots & x^*_{\sigma(k)} & \cdots & x^*_{\sigma(r)} & \cdots & x^*_{\sigma(n)} \\ & \cdots & \cdots & || & \cdots & || & \cdots & \cdots \\ \text{i.e.} & \cdots & \cdots & 1 & \cdots & 0 & \cdots & \cdots \end{array}$$

und das ist eine *nicht sortierte* Folge. ∎

5.13 Beweis (von Lemma 5.10) Wegen des Lemmas von Knuth können wir uns darauf beschränken, nachzuweisen, dass Algorithmus 5.6 alle 0-1-Folgen korrekt sortiert. Es sei e die Anzahl der Einsen in einer Eingabefolge.

Fasst man den Algorithmus so auf, dass von der linken Randzelle nach rechts R-Signale laufen, so gilt für diese: Das erste sorgt dafür, dass die in der Eingabe am weitesten rechts stehende 1 am Ende, d. h. nach spätestens $2 + (n-1) = n+1$ Schritten, an ihrer Endposition steht. Das zweite Signal sorgt dafür, dass die in der Eingabe zweite 1 von rechts nach spätestens $4 + (n-2) = n+2$ Schritten an ihrer Endposition steht, usw.. Das e-te Signal schließlich sorgt dafür, dass die in der Eingabe e-te 1 von rechts nach spätestens $2e + (n-e) = n+e$ Schritten an ihrer Endposition steht. Dann sind aber offensichtlich alle Einsen am Ziel, und die 0-1-Folge ist korrekt sortiert. ∎

5.14 Übung. Überlegen Sie sich, ob man aus der eben gemachten Abschätzung für den maximalen Zeitbedarf beim Sortieren von 0-1-Folgen auch etwas für den Zeitbedarf im allgemeinen folgern kann.

5.15 Es gibt nun verschiedene Verallgemeinerungen der Aufgabenstellung 5.5, die man betrachten kann. Eine Möglichkeit besteht darin, Sortieren im Zwei- und Höherdimensionalen zu betrachten. Dabei muss man sich als erstes darüber klar werden, wann denn etwa zum Beispiel ein Quadrat als sortiert anzusehen sei. Da schnelle Algorithmen für dieses Problem aber doch schon etwas anspruchsvoller sind, kommen wir darauf erst in einem späteren Kapitel zurück, nachdem in ausreichendem Maße „Standard-Techniken“ und „Algorithmen-Bausteine“ eingeführt wurden.

Für den eindimensionalen Fall besteht eine andere Aufgabenstellung darin, dass die („vielen“) zu sortierenden Werte nicht als Elemente der Zustandsmenge dargestellt werden, sondern etwa als Dualzahlen, deren Bits in nebeneinander liegenden Zellen gespeichert werden. Als interessierende Eingaben könnten z. B. solche der Form

```
#0011#0101#0110#1011#0010#0011#1001#0000#
```

betrachtet werden, und ein geeigneter Zellularautomat sollte z. B. diese in

```
#0000#0010#0011#0011#0101#0110#1001#1011#
```

sortieren. Außerdem sollte er (i. e. immer der gleiche Zellularautomat!) für Eingaben mit beliebig langen Dualzahlen funktionieren, so dass man dieses Problem nicht durch die Wahl einer hinreichend großen Nachbarschaft auf weiter das vorne behandelte zurückführen kann.

Noch etwas komplizierter wird es, wenn in einer Eingabe unterschiedlich lange Dualzahlen zugelassen werden. In beiden Fällen ist die Kenntnis einiger grundlegender Techniken von Nutzen, auf die wir als nächstes eingehen werden.

Zusammenfassung

Knuths 0-1-Sortierlemma sichert zu, dass bei gewissen Sortieralgorithmen für den Nachweis der Korrektheit die Betrachtung von 0-1-Eingaben genügt.

Sortieren in eindimensionalen Zellularautomaten mit einem Datum je Zelle ist in Linearzeit möglich.

Literatur

Knuth, Donald E. (1998). *The Art of Computer Programming*. Bd. 3. Addison Wesley.

