Universität Karlsruhe
Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler

Suchalgorithmen auf SIMD-Rechnern
Weitere Ergebnisse zu Polyautomaten

Diplomarbeit
von
Peter Sanders

13. Juni 1994

Betreuer: Prof. R. Vollmar

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Karlruhe, den 13. Juni 1994

Inhaltsverzeichnis

1 Einleitung **4**

2 Einführung in das FSSP **6**
2.1 Das Problem des General Moorehill 6
2.2 Zellulare Automaten 7
2.3 Das Firing Squad Synchronisation Problem 8

3 Sequentielle Suchalgorithmen **10**
3.1 Grundlegende Datenstrukturen 10
3.2 Ein nichtdeterministischer Algorithmus 11
3.3 Eine einfache Suchstrategie 13
3.4 Grundeigenschaften der Suchstrategien 15
3.4.1 Der Suchraum als Graph 15
3.4.2 Reduzierung des Suchraums auf einen Baum 15
3.4.3 Baumtraversierung durch Tiefensuche 16
3.5 Suchstrategie „geordnete Simulation“ 16
3.5.1 Geordnete Simulation macht Regelanwendungen einfach . 16
3.5.2 Der Grundalgorithmus 17
3.5.3 Beschneiden des Suchbaums 19
3.5.4 Vermeidung redundanter symmetrischer Lösungen 20
3.5.5 Eine Variante: Schräge Simulation 21
3.6 Suchstrategie „maximale Simulation“ 22
3.6.1 Bevorzugung deterministischer Regelanwendungen 22
3.6.2 Buchführung über gültige Einträge von C 22
3.6.3 Der Grundalgorithmus 23
3.6.4 Implementierung von *Contour* 25

4 Ergebnisse zum FSSP **27**
4.1 Vier Zustände 27
4.1.1 Vergleich mit Balzers Ergebnis 27
4.1.2 Korrektheit des neuen Beweises 28
4.2 Fünf Zustände 29
4.3 Zusatzannahmen 30

5 Suche nach Trellisautomaten **32**
5.1 Homogene Trellisautomaten 32
5.2 Anpassung des Suchalgorithmus 33
5.3 Ergebnisse 34
5.4 Beweis einer Lösung für ww^R 35

5.4.1 Transformation des Problems 35
5.4.2 Eigenschaften der abstrahierten Übergangsfunktion 37
5.4.3 Vollständigkeit der Lösung 38
5.4.4 Korrektheit der Lösung 38

6 Einführung in parallele Suchalgorithmen 40
6.1 Eingrenzung der Fragestellung 40
6.2 Grundidee für die Parallelisierung 41

7 SIMD-Realisierung des Kontrollflusses 43
7.1 Die Grundsätzliche Realisierung 43
7.1.1 Ein nichtrekursiver Suchalgorithmus 43
7.1.2 Ein endlicher Automat als Kontrollstruktur 43
7.1.3 Die Rolle der problemabhängigen Funktionen 45
7.2 Ansätze zur SIMD-Optimierung 46
7.3 Optimierung der Abfragehäufigkeit 47
7.3.1 Der Ansatz 47
7.3.2 Berechnung der Abfragehäufigkeiten 48
7.3.3 Auswertung des Ergebnisses 50
7.4 Optimierung der Abfragereihenfolge 51
7.4.1 Die Kostenfunktion 51
7.4.2 Ansätze zur Optimierung 52
7.5 Auswahl von Elementaroperationen 53
7.5.1 Schleifen auflösen 53
7.5.2 Elementaroperationen aufspalten 54
7.5.3 Verschmelzen von Elementaroperationen 54
7.5.4 Vereinfachen von Elementaroperationen 55

8 Arbeitsverteilung 56
8.1 Initiale Arbeitsverteilung 56
8.2 Umverteilung 57
8.2.1 Teilen des Suchraums 59
8.2.2 Adaptive Arbeitsverteilung 59
8.2.3 Nachbarschaftskommunikation 60
8.2.4 Zufallspermutationen 63
8.2.5 Sortieren 64

9 Verallgemeinerung der Fragestellung 66
9.1 MIMD-Rechner 66
9.2 Kompliziertere Suchalgorithmen 66
9.2.1 Branch and Bound 66
9.2.2 Abhängigkeiten zwischen Teilbäumen 67

10 Parallele Suche am Beispiel des FSSP 68
10.1 Parallelisierung 68
10.1.1 Der sequentielle Algorithmus 68
10.1.2 Änderungen am sequentiellen Algorithmus 69
10.1.3 Parallele Konstrukte 70
10.2 Optimierungen 70
10.2.1 Arbeitsumverteilung 71

10.2.2 Compiler- und Maschinenaspekte 71
10.2.3 SIMD-Optimierung 72
10.3 Beurteilung der Leistung 74

11 Zusammenfassung und Ausblick 75
11.1 Polyautomaten 75
11.1.1 Das FSSP 75
11.1.2 Trellisautomaten 75
11.1.3 Verallgemeinerung 76
11.2 Parallele Suche 76
11.3 SIMD-Aspekte 77

A Trellisautomaten für $\{ww^R|w \in \{a,b\}^+\}$ I

B C-Implementation von Algorithmus 3.5 III

C Laufzeiten auf sequentiellen Rechnern VI

Kapitel 1

Einleitung

Die vorliegende Arbeit beschäftigt sich mit parallelen Baumsuchalgorithmen für massiv parallele SIMD-Rechner. Als Beispiel werden Algorithmen zur Suche nach Übergangstabellen für Polyautomaten untersucht. Konkret untersuchte Probleme sind das „Firing Squad Synchronisation Problem“ für Zellularautomaten (FSSP) sowie ein offenes Problem über Trellisautomaten.

Da das oben umrissene Themengebiet recht breit aufgefächert ist, soll zunächst dargestellt werden, welche Zusammenhänge zwischen diesen Fragestellungen bestehen und wie es zu dieser Zusammenstellung kam.

Im Sommersemester 92 hörte ich Prof. Vollmars Vorlesung über Zellularautomaten und erfuhr dort, daß es nicht bekannt ist, welches die minimale Zustandszahl ist, die zur Lösung des FSSP nötig ist, obwohl Lösungen mit 6 Zuständen bekannt sind. In Unkenntnis der genauen Ergebnisse von Balzer machte ich mich daran, ein Programm zu schreiben, das hilft diese Frage zu beantworten. Nachdem ich erste Ergebnisse vorzuweisen hatte, ging ich zu Prof. Vollmar, der mich ermunterte, im Rahmen einer Diplomarbeit tiefer in das Gebiet einzudringen. Schnell kam die Idee auf, das Suchprogramm auf der MasPar MP-1 zu implementieren, da Suche auf einem SIMD-Rechner eine interessante Fragestellung ist und die Parallelisierung die Chance erhöhen könnte, zu neuen Ergebnissen zu gelangen.

Die Parallelisierung war zwar erfolgreich, aber die Frage, ob es eine 5-Zustandslösung des FSSP gibt, stellte sich (nicht ganz unerwartet) als so komplex heraus, daß sich auch das parallele Programm als chancenlos erwies. Andererseits waren die bei der Parallelisierung erzielten Erkenntnisse so wenig vom FSSP abhängig, daß eine weitgehend unabhängige Beschreibung sinnvoll war. Dadurch ergab es sich, daß der Hauptteil der Arbeit in zwei fast unabhängige Teile über Suche nach Polyautomaten (Kapitel 2–5) und über parallele Suchalgorithmen (Kapitel 6–9) zerfällt. Erst Kapitel 10 führt die beiden Themen wieder zusammen, indem es die parallele Implementierung des FSSP-Suchalgorithmus beschreibt.

Das zusätzliche Kapitel über Trellisautomaten (Kapitel 5) geht auf die Diskussion zurück, wie sich der Suchalgorithmus für das FSSP auf andere Fragestellungen übertragen ließe. Ursprünglich war nur geplant, kurz die Machbarkeit einer Umstellung des Algorithmus zu demonstrieren. Da sich dann aber recht interessante Ergebnisse einstellten, weitete sich das Thema zu einem eigenen Kapitel aus.

In Kapitel 11 werden die Ergebnisse zusammengefaßt und aufgeworfene Fragen und Ideen dargestellt. Die Anhänge enthalten Material zu den Ergebnissen über

FSSP und Trellisautomaten sowie ein Glossar und das Literaturverzeichnis.

Danksagung

Für ihre uneigennützige Unterstützung möchte ich danken: Prof. R. Vollmar, Thomas Worsch, Thomas Umland, Sebastian Egner, Markus Mock, Roger Buthenuth und Christian von Roques.

Kapitel 2

Einführung in das FSSP

> **firing squad:**
> 1. A military detachment assigned to
> execute a condemned person by shooting.
> 2. A military detachment assigned to fire
> a salute at the burial of a person being honored.
> *(Randomhouse Websters College Dictionary)*

In diesem Kapitel wird die Problemstellung des Firing Squad Synchronisation Problem (FSSP) dargestellt. Nach einer nicht ganz ernstzunehmenden Einstimmung (Abschnitt 2.1) wird zunächst der Begriff des zellularen Automaten eingeführt (Abschnitt 2.2) und dann die formale Definition des FSSP vorgenommen (Abschnitt 2.3).

2.1 Das Problem des General Moorehill

General M. Y. Moorehill — seines Zeichens Chef des Wachbatallions von Gerontokratien — ist ganz schön im Streß. Er und seine Soldaten sind dafür zuständig, bei Staatsbegräbnissen Salut zu schießen. An sich wäre das nicht so schlimm, wenn nicht die gerontokratische Verfassung wäre. Nach alter Sitte (und um überlange Amtszeiten von Politikern zu verhindern) dürfen immer nur die ältesten Mitglieder des Parlaments zu Ministern ernannt werden. So kommt es, daß dauernd Staatsbegräbnisse stattfinden.

Besonders schlimm ist es im momentanen, neblig feuchten Novemberwetter. Fünf Staatsbegräbnisse in den letzten drei Wochen sowie ständige Übungen haben dazu geführt, daß die Soldaten mächtig heiser sind. Dadurch ist es dem General nicht wie sonst möglich, einfach einen zackigen Befehl zu brüllen, woraufhin die Soldaten feuern. (Die Soldaten sind durch die ständige Ballerei auch etwas schwerhörig.) Dazu kommt, daß durch den Nebel die Sichtweite auf etwa einen Meter begrenzt ist. Durch diese Probleme wird es zu einer schwierigen Aufgabe, ein ordnungsgemäßes Salutschießen durchzuführen. Denn jeder Soldat kann sich bei der Entscheidung, wann er feuern soll, nur auf das stützen, was seine unmittelbaren Nachbarn tun. Über einige weitere Randbedingungen denkt der General erst gar nicht nach, da sie tief in der uralten und ehrwürdigen Militärgeschichte seiner Heimat verankert sind und ihm als hohem Offizier natürlich in Fleisch und Blut übergegangen sind.

- Alle Soldaten sind gleich gekleidet und ausgerüstet und verhalten sich nach

den gleichen, genau festgelegten Regeln. Dies gilt auch für den General selbst. Insbesondere sind ihm keine akustischen Hilfsmittel wie Trillerpfeife o.ä. erlaubt.

- Die Regularien enthalten eine feste, recht kleine Zahl von Haltungen, die ein Soldat während der Zeremonie einnehmen darf.
- Ein Soldat darf seine Haltung nur ändern, wenn die ebenfalls anwesende Militärkapelle einen Takt beendet hat.
- Um die intellektuellen Fähigkeiten eines Soldaten nicht zu überfordern, schreiben die Regeln vor, daß die Entscheidung, welche Haltung als nächstes anzunehmen ist, nur von nach außen sichtbaren Dingen abhängen darf. Bei der gegebenen Wettersituation also nur von der aktuellen Haltung eines Soldaten und seiner beiden Nachbarn.
- Die Soldaten stehen in einer Reihe; der General ganz links. Von dort kommt auch der Sarg, so daß es dem General obliegt, das Signal für die Salutschüsse zu geben, wenn das Ehrengeleit aus dem Nebel auftaucht.
- Natürlich müssen die Salutschüsse genau gleichzeitig erfolgen. Alle Soldaten müssen schießen, und kein Schuß darf zu früh oder zu spät fallen.

Als ob der Randbedingungen und Probleme nicht schon genug wären, fallen dem General noch eine Reihe weiterer Dinge ein:

- Am Morgen hat ihn der Premierminister persönlich aufgesucht und ihm gedroht, ihn zu feuern, wenn er das Protokoll auch nur im geringsten verzögere. Der erfahrene General weiß, daß es dem Premierminister darum geht, die Übertragung eines Fußballendspiels am betreffenden Abend nicht zu versäumen. Das bedeutet, daß diese Drohung todernst zu nehmen ist. Die Salutschüsse müssen also unter allen Umständen so schnell wie möglich ausgelöst werden, nachdem das Signal gegeben ist.
- Die Anzahl der Soldaten, die Salut schießen sollen, hängt von äußerst komplizierten tagespolitischen Erwägungen ab. Diese Zahl kann sich in letzter Minute ändern. Deshalb muß eine Lösung unbedingt für jede Anzahl von Schützen funktionieren.
- Die Soldaten müssen die Regeln für die Änderung ihrer Stellung in kürzester Zeit auswendig lernen. Darum soll die Zahl der benutzten Stellungen möglichst klein gehalten werden, damit die Möglichkeit, daß sich Fehler einschleichen, minimal gehalten wird.

Der General, der begeisterter Hobbymathematiker ist, läßt sich von diesen schwierigen Randbedingungen nicht entmutigen und macht sich sofort an die Arbeit...

2.2 Zellulare Automaten

Zellulare Automaten sind ein einfaches, aber dennoch vielseitiges Hilfsmittel zur Beschreibung von Systemen. Für diese Arbeit wird die sehr einfache Variante von eindimensionalen zellularen Automaten mit von Neumann-Nachbarschaft

betrachtet, die im folgenden eingeführt werden. Die hier verwendeten Bezeichnungen richten sich nach Vollmar [19], sind allerdings spezieller gefaßt, um die Darstellung zu vereinfachen.

Definition 1 $(A, 1, H_1, \sigma)$ *charakterisiert einen eindimensionalen zellularen Automaten mit von Neumann-Nachbarschaft. Dabei gelten folgende Bezeichnungen:*

- A *ist eine endliche Menge von* Zuständen.
- *Der Automat operiert auf einer* Retina *von Zellen mit den Nummern* 0 *bis* $s+1$, *von denen jede zu jedem Zeitpunkt in genau einem Zustand aus* A *ist.*
- *Die Zellen 0 und* $s+1$ *befinden sich immer im* Randzustand „#“. *Keine andere Zelle darf je diesen Zustand annehmen.*
- $N = |A| - 1$ *bezeichnet die Anzahl der Zustände des Automaten. Der Randzustand wird dabei wegen seiner sehr speziellen Verwendung nicht mitgezählt.*
- *Die* Konfiguration $C_t^s \in A^s$ *ist das* s*-Tupel der Zustände der Zellen* 1 *bis* s *einer Retina der Größe* s *zum Zeitpunkt* t. $C_t^s(c)$ *steht für den Zustand der* c*-ten Zelle von* C_t^s. C_0^s *wird* Anfangskonfiguration *genannt.*
- *Für die hier benutzte* von Neumann-Nachbarschaft H_1 *wird das Verhalten des Automaten durch die* (lokale) Überführungsfunktion $\sigma : A^3 \to A$ *definiert. Es gilt:* [1]

  $$C_{t+1}^s(c) := \sigma(C_t^s(c-1), C_t^s(c), C_t^s(c+1))$$

  Der Nachfolgezustand einer Zelle wird also jeweils aus dem unmittelbar vorhergehenden Zustand der Zelle selbst und denen ihrer beiden Nachbarzellen bestimmt.

2.3 Das Firing Squad Synchronisation Problem

Definition 2 *Eine Lösung des* Firing Squad Synchronisation Problems *(kurz* FSSP*) ist eine Klasse von Automaten* $(A, 1, H_1, \sigma)$ *mit folgenden Eigenschaften:*

- $\mathtt{G} \in A$ *wird* Generalszustand *genannt.*
- $\mathtt{Z}_0 \in A$ *ist der* Ruhezustand.
- $\mathtt{F} \in A$ *heißt* Feuerzustand.
- $\forall s > 1 : \exists t : \left(C_0^s = \mathtt{G}\mathtt{Z}_0^{s-1} \Rightarrow C_t^s = \mathtt{F}^s \wedge \neg(\exists t' < t \exists c : C_{t'}^s(c) = \mathtt{F}\right))$
  Zu Beginn befindet sich also die Zelle am linken Rand im Generalszustand und alle anderen im Ruhezustand. Für jede mögliche Retinagröße gibt es einen Zeitpunkt, zu dem alle Zellen gleichzeitig (und das erstemal) in den Feuerzustand übergehen. (Die Soldaten feuern.)

[1] In [19] wird das Verhalten in anderer (allgemeinerer) Weise eingeführt. Für die Zwecke dieser Arbeit ist die hier verwendete Definition aber leichter handhabbar.

- $\sigma(\mathtt{Z}_0, \mathtt{Z}_0, \mathtt{Z}_0) = \sigma(\mathtt{Z}_0, \mathtt{Z}_0, \#) = \mathtt{Z}_0$ *Dadurch wird gesichert, daß am Anfang weder von einer Zelle im Ruhezustand (einem gemeinen Soldaten) noch vom rechten Rand eine Initiative ausgeht.*

Synchronisationsprobleme, die dem FSSP sehr ähnlich sind, treten häufig als Teilprobleme in der Formulierung von Algorithmen für Zellularautomaten auf. Deshalb ist es von Interesse, das FSSP nicht nur irgendwie zu lösen, sondern auch möglichst schnell und unter Benutzung möglichst weniger Zustände. Denn oft addiert sich die benötigte Synchronisationszeit zur restlichen Laufzeit, und die Zustandszahl multipliziert sich u.U. mit der Zahl der übrigen benötigten Zustände.

Ein einfacher Widerspruchsbeweis zeigt, daß mindestens $2s-2$ Schritte nötig sind, um das FSSP korrekt zu lösen. Dies ist die Zeit, die ein Signal benötigt, um einmal vom General zum rechten Rand und wieder zurück zu laufen [21]. Von Waksman [21], Balzer [2], Gerken [6] und Mazoyer [10] wurden im Laufe der Zeit Algorithmen gefunden, die tatsächlich mit dieser minimalen Laufzeit auskommen und 16, 8, 7 bzw. 6 Zustände benötigen.

Lösungen mit ein oder zwei Zuständen kann es nicht geben, da zumindest $\mathtt{Z}_0$, $\mathtt{G}$ und $\mathtt{F}$ in der Zustandsmenge vorhanden sein müssen. Ebenso ist durch manuelles Ausprobieren aller Möglichkeiten recht leicht festzustellen, daß auch 3 Zustände nicht ausreichen (siehe auch Abbildung 3.4). Bleibt also die Frage, ob es Lösungen mit 4 oder 5 Zuständen gibt. Ein Ziel dieser Arbeit ist es, Algorithmen auszuarbeiten, mit denen nach einer Lösung mit 4 bzw. 5 Zuständen gesucht werden kann.

Kapitel 3

Sequentielle Suchalgorithmen

In diesem Kapitel werden Suchalgorithmen entwickelt, die zu einer gegebenen Zustandszahl N (und u.U. weiteren Randbedingungen) eine Lösung des FSSP suchen, d.h. eine lokale Überführungsfunktion σ zu bestimmen versuchen, die das FSSP im Sinne von Definition 2 löst.

Zunächst werden einige grundlegende Begriffe und Datenstrukturen eingeführt, die relativ unabhängig von den eigentlichen Suchalgorithmen sind (Abschnitt 3.1). In Abschnitt 3.2 wird dann ein Regelsystem aufgestellt, das Grundlage der weiteren Diskussion ist. Alle weiteren Algorithmen werden sich als Implementierungsstrategie für dieses Regelsystem herausstellen.

Nachdem Abschnitt 3.3 zunächst eine ebenso einfache wie ineffiziente Implementierungsstrategie vorstellt, werden in Abschnitt 3.4 die Grundlagen für effizientere Verfahren gelegt. Abschnitt 3.5 beschreibt eine Klasse von einfachen aber trotzdem effizienten Suchstrategien. Schließlich wird in Abschnitt 3.6 versucht, den Suchraum mit Hilfe einer etwas komplizierteren Strategie noch weiter zu verkleinern.

3.1 Grundlegende Datenstrukturen

Im allgemeinen kann nicht davon ausgegangen werden, daß der Suchalgorithmus die Übergangsfunktion σ auf einen Schlag bestimmen kann. Deshalb ist es zweckmäßig, σ als partielle Funktion aufzufassen, die im Laufe der Zeit um zusätzliche Einträge erweitert werden kann. Je nachdem wie es gerade sinnvoll erscheint, wird über σ mal als partielle Funktion, mal als Menge von Viertupeln der Form „$XYZ \to T$“ und mal als Tabelle (3D-Feld) gesprochen. (Wobei letzteres gleichzeitig der tatsächlichen Implementierung entspricht.)

Ein Suchalgorithmus wird die Korrektheit einer Lösung zwangsläufig nur bis zu einer *maximalen Retinagröße* M überprüfen können. Außerdem spielen für Retinagröße s nur Zeitpunkte $t \leq 2s-2$ eine Rolle. Daraus ergibt sich eine sehr allgemeine Art, über durchgeführte Simulationsschritte Buch zu führen. $C_t^s(c)$ wird als Wert der Funktion $C : \mathrm{POS} \to A$ aufgefaßt, wobei

$$\mathrm{POS} := \{(s,t,c) | s \in \{2,\ldots,M\} \wedge t \in \{0,\ldots,2s-2\} \wedge c \in \{1,\ldots,s\}\}$$

Wegen der Beschränktheit von s, t, c kann C einfach als Tabelle abgespeichert werden. C kann aber auch als Menge von 4-Tupeln aufgefaßt werden. Von C als Ganzem oder von einem einzelnen C^s wird auch als *Raumzeitdiagramm*

```
#G.#G..#G...#G....#G.....#G......#G.......#
#GG#GX.#GX..#GX...#GX....#GX.....#GX......#
#FF#XXX#XXG.#XXG..#XXG...#XXG....#XXG.....#
   #GGG#GX.G#GX.X.#GX.X..#GX.X...#GX.X....#
   #FFF#XXXX#XXG.X#XXG.G.#XXG.G..#XXG.G...#
        #GGGG#GX.GX#GX.XXG#GX.XXX.#GX.XXX..#
        #FFFF#XXXXX#XXG.XX#XXG.G.X#XXG.G.G.#
              #GGGGG#GX.G.G#GX.XXGX#GX.XXXXG#
              #FFFFF#XXXXXX#XXG.XGX#XXG.GGXX#
                     #GGGGGG#GX.G.GX#GX.XXGXG#
                     #FFFFFF#XXXXXXX#XXG.XGXX#
                             #GGGGGGG#GX.G.GXG#
                             #FFFFFFF#XXXXXXXX#
                                      #GGGGGGGG#
                                      #FFFFFFFF#
```

Abbildung 3.1: Raumzeitdiagramm für Lösung bis $s = 8$ mit 4 Zuständen.

gesprochen. Abbildung 3.1 zeigt ein Beispiel für ein solches Raumzeitdiagramm für die Retinagrößen 2 bis 8. („.“ steht für den Grundzustand $\mathtt{Z}_0$.)

Die Information, daß die Endkonfiguration nur Feuerzustände enthalten darf, ließe sich direkt in den Suchalgorithmus kodieren. Um den Algorithmus aber flexibler zu gestalten und zusätzliche Bedingungen an Eigenschaften einer Lösung stellen zu können, wurde eine etwas andere Vorgehensweise gewählt. Eine partielle Funktion *pattern*, die die gleiche Struktur wie C hat, legt für einige Stellen (s, t, c) fest, welcher Zustand an dieser Stelle des Raumzeitdiagramms angenommen werden muß. Durch Initialisieren von *pattern* zu

$$\{(s,\ 2s-2,\ c,\ \mathtt{F}) | s \in \{2, \dots, M\}, c \in \{1, \dots, s\}\}$$

wird gerade festgelegt, daß alle Zellen einer Retina der Größe s zum Zeitpunkt $t = 2s-2$ in den Feuerzustand übergehen müssen, wie es für eine zeitoptimale Lösung des FSSP gefordert ist. Zusätzliche Einträge in *pattern* können dazu dienen, bestimmte Zusatzforderungen an die Lösung zu stellen (siehe auch Abschnitt 4.3).

3.2 Ein nichtdeterministischer Algorithmus

Bevor in den nächsten Abschnitten die konkret implementierten Suchalgorithmen eingeführt werden, soll zunächst ein sehr einfacher, nichtdeterministischer Algorithmus vorgestellt werden, der auf einem Regelsystem beruht. Dadurch ergeben sich die folgenden Vorteile:

- Man erhält einen erheblich leichteren Einstieg in die Materie.
- Alle später vorgestellten Algorithmen lassen sich als deterministische Implementierungen dieses Algorithmus auffassen.

- Die durch die Problemdefinition gegebenen Aspekte erscheinen klar getrennt von der konkreten Kontrollstruktur.
- Die Korrektheit des Suchalgorithmus läßt sich zum großen Teil unabhängig von der konkreten Implementierung — nur anhand des Regelsystems — überprüfen.
- Die Darstellung als Regelsystem ist formalen Beweisen besser zugänglich.

Die Definition des FSSP läßt sich in Form von Ersetzungsregeln ausdrücken, die die in Abschnitt 3.1 eingeführten Datenstrukturen σ und C auffüllen. Diese Regeln werden in der Form

$$l \Longrightarrow r \,|\, \mathit{Bedingung}$$

geschrieben und sind in Definition 3 angegeben.

Regel DEFINE besagt, daß für einen undefinierten Übergang ($\sigma(x, y, z) = \bot$) ein beliebiger Ergebniszustand u eingetragen werden kann. Einträge, die von einem Feuerzustand abhängen, sind allerdings uninteressant, da die Definition des FSSP nichts über sie aussagt ($x, y, z \in \mathrm{A}-\{\mathtt{F}\}$). Außerdem darf der Randzustand „#“ nicht erzeugt werden.

Die Regel SIMULATE definiert, wann ein Zustandsübergang einer Zelle legal ist. Ein Eintrag u in C ergibt sich immer aus dem Wert der lokalen Überführungsfunktion σ für die Nachbarn der betrachteten Zelle zum vorhergehenden Zeitpunkt. Da dieses Tripel der alten Zustände öfter auftritt, wird es durch

$$\nu(s, t, c) := (C^s_{t-1}(c-1), C^s_{t-1}(c), C^s_{t-1}(c+1))$$

mit einem Namen belegt. Daraus ergibt sich die Bedingung $(s, t, c) \in \mathrm{POS} \wedge \sigma(\nu(s, t, c)) = u$ für die Regel SIMULATE. Wenn für die betrachtete Stelle ein Eintrag in *pattern* besteht, so muß der neue Eintrag in C genauso lauten ($\mathrm{pattern}(s, t, c) = u$). In allen anderen Fällen ist ein Übergang in den Feuerzustand illegal ($u \neq \mathtt{F}$), da sonst ein Automat vorzeitig feuern würde.

Wenn im folgenden von der Anwendung einer Regel R *auf* eine Position (s, t, c) die Rede ist, so ist damit gemeint, daß eine Instanz von R angewandt wird, bei der die in R vorkommende Position mit (s, t, c) übereinstimmen muß.

Definition 3

$$\begin{array}{lll} \textbf{DEFINE}: & \sigma \Longrightarrow & \sigma \cup \{xyz \to u\} \,|\, x, y, z \in A-\{\mathtt{F}\} \wedge \\ & & u \in A-\{\#\} \wedge \sigma(x, y, z) = \bot \\ \textbf{SIMULATE}: & C \Longrightarrow & C \cup \{(s, t, c, u)\} \,|\, t > 0 \wedge (s, t, c) \in \mathrm{POS} \wedge \\ & & \sigma(\nu(s, t, c)) = u \wedge \\ & & (\mathrm{pattern}(s, t, c) = \bot \wedge u \neq \mathtt{F} \vee \mathrm{pattern}(s, t, c) = u) \end{array}$$

Auf der Basis dieser Regeln läßt sich nun leicht ein nichtdeterministischer Suchalgorithmus formulieren. Die Pseudocode-Funktion in Abbildung 3.2 beschreibt einen solchen Algorithmus. Eingegeben wird die Zustandszahl N, die maximale Retinagröße M, eine Menge von zusätzlichen Einträgen in *pattern* und eine Menge bereits vorgegebener Einträge der Überführungsfunktion σ (im einfachsten Fall beide leer). Falls eine Lösung des FSSP bis Größe M existiert, die den Randbedingungen genügt, so wird σ ausgegeben.

FUNCTION searchNonDet(N, M, pattern_0, σ_0)
(*Mark all positions where firing is due*)
pattern := $\text{pattern}_0 \cup \{(s,\ 2s-2,\ c,\ \mathtt{F}) | s \in \{2, \ldots, M\}, c \in \{1, \ldots, s\}\}$
$\sigma := \sigma_0 \cup \{\mathtt{Z}_0\mathtt{Z}_0\# \to \mathtt{Z}_0,\ \mathtt{Z}_0\mathtt{Z}_0\mathtt{Z}_0 \to \mathtt{Z}_0\}$ (*default transitions*)
(*Make initial confugurations*)
$C := \{(s, 0, 1, \mathtt{G}) | s \in \{2, \ldots, M\}\} \cup$
$\{(s, 0, c, \mathtt{Z}_0 | s \in \{2, \ldots, M\}, c \in \{2, \ldots, s\}\}$
WHILE $\exists (s,t,c) \in \text{POS} : C_t^s(c) = \bot$ **DO**
Nondetermistically apply one of the rules SIMULATE or DEFINE
RETURN σ

Abbildung 3.2: Nichtdeterministischer Suchalgorithmus.

Vor Beginn der eigentlichen Suche müssen die verwendeten Datenstrukturen gemäß der Suchparameter und der Definition des FSSP initialisiert werden. Zu den von außen vorgegebenen Einträgen in *pattern* kommen die Einträge aus Gleichung 3.1, die sicherstellen, daß zur rechten Zeit gefeuert wird. Die Definition des FSSP (Def. 2) legt außerdem fest, daß die Übergänge „$\mathtt{Z}_0\mathtt{Z}_0\# \to \mathtt{Z}_0$“ und „$\mathtt{Z}_0\mathtt{Z}_0\mathtt{Z}_0 \to \mathtt{Z}_0$“ von Beginn an in σ vorhanden sind. Schließlich muß C so initialisiert werden, daß für jede Retinagröße die in Definition 2 festgelegte Anfangskonfiguration eingetragen wird.

In der Schleife werden so lange Regeln angewandt, bis C vollständig definiert ist und damit eine Lösung gefunden ist. Zu beachten ist, daß das so berechnete σ i.allg. keine totale Funktion ist. Übergänge, die im Raumzeitdiagramm nirgends auftreten, müssen auch nicht definiert sein.

Falls keine Lösung existiert, wird irgendwann der Fall eintreten, daß keine Regel mehr anwendbar ist, obwohl C noch nicht vollständig belegt ist. Eine solche Situation soll im folgenden kurz *Fehler* genannt werden.

Wenn hier (und im folgenden) von *Lösung* gesprochen wird, so ist immer eine Belegung für σ gemeint, die für die Retinagrößen 2 bis M funktioniert. Dies ist keine große Einschränkung, da für hinreichend große M die Wahrscheinlichkeit immer kleiner wird, ein σ zu erhalten, das für größere Retinae versagt. Insbesondere können die gefundenen Lösungskandidaten zusätzlich einer Simulation für weitere Retinagrößen unterworfen werden. Der Korrektheitsbeweis für eine solchermaßen gefundene Lösung muß allerdings weiterhin manuell durchgeführt werden.

3.3 Eine einfache Suchstrategie

Ein einfacher determinstischer Algorithmus besteht darin, alle möglichen (totalen) Überführungsfunktionen aufzuzählen und dann auf Korrektheit zu prüfen. Übersetzt in die Ausdrucksweise des Regelsystems heißt dies, so oft wie möglich DEFINE anzuwenden und dann nur noch zu simulieren. Führt dies zum Ziel, d.h. werden dadurch alle Zellen von C belegt, so ist die generierte Überführungsfunktion eine Lösung. Sonst muß die nächste Überführungsfunktion ausprobiert werden.

Dieser Algorithmus ist einfach, hat einen geringen Speicherbedarf und läßt sich hervorragend parallelisieren. Allerdings ist er viel zu aufwendig, um sinnvoll einsetzbar zu sein: Sei N die Zahl der Zustände (ohne „#"). Die Einträge in σ, bei denen `F` auf der linken Seite vorkommt, brauchen nicht betrachtet zu werden, da die Simulation beendet ist, sobald irgendeine Zelle feuert. Es bleiben $2(N-1)^2$ Einträge für die Umgebungen am linken und rechten Rand; sowie $(N-1)^3$ Einträge ohne Randzustand. Für jeden Eintrag sind N Belegungen möglich, und es ergibt sich eine Gesamtzahl von $N^{(N-1)^3+2(N-1)^2}$ möglichen lokalen Überführungsfunktionen.

Für den einfachsten interessanten Fall $N = 4$ sind das $4^{27+18} = 2^{90} \approx 1.2 \cdot 10^{27}$ verschiedene Funktionen. Selbst wenn 1 Million Prozessoren verfügbar wären und jeder 1 Million Funktionen pro Sekunde überprüfen könnte, würden dafür $1.2 \cdot 10^{15} s$ oder ungefähr 38 Millionen Jahre benötigt!

Der Hauptfehler dieser einfachen „generiere und teste"-Strategie besteht darin, daß die gesamte Überführungsfunktion festgelegt wird und erst dann überprüft wird, ob sich daraus eine Lösung ergibt. Dabei gibt es große Klassen von Überführungsfunktionen, die einander ähnlich sind und sich bis zu einem gewissen Grade gemeinsam überprüfen lassen. Zum Beispiel sind für Lösungen bis Größe 2 maximal 4 Einträge festzulegen. Schon diese 4 Einträge können nach wenigen Simulationsschritte eine Fehlfunktion herbeiführen. Sämtliche anderen Einträge können beliebig belegt werden, ohne daß die Fehlfunktion aufgehoben wird.

Ein Teil dieser Einsicht kann mit Hilfe des Regelsystems ausgedrückt werden. Es wird eine Variante von DEFINE (Definition 4) eingeführt, die einen neuen Eintrag in σ nur zuläßt, wenn eine Chance besteht, daß er zu einer Lösung beiträgt. Dazu sollte eine zum Eintrag passende Umgebung in C vorkommen ($\exists(s,t,c) \in \mathrm{POS} : \nu(s,t,c) = (x,y,z)$). Da SIMULATE irgendwann auf die betrachtete Stelle angewandt werden muß, können die Bedingungen, die dort einzuhalten sind, schon zum Zeitpunkt der Anwendung von DEFINE' überprüft werden. Falls *pattern* also an der entsprechenden Stelle definiert ist, so ist nur der dort geforderte Zustand als Belegung zulässig, und `F` darf nur eingetragen werden, wenn *pattern* dies erfordert (siehe auch Definition 3).

Definition 4

$$\begin{aligned}\textbf{DEFINE}' : \sigma \Longrightarrow\ & \sigma \cup \{xyz \to u\} \mid x,y,z \in A - \{\mathtt{F}\} \wedge u \in A - \{\#\} \wedge \\ & \sigma(x,y,z) = \bot \wedge \\ & \exists (s,t,c) \in \mathrm{POS} : \nu(s,t,c) = (x,y,z) \wedge \\ & \quad ((\mathrm{pattern}(s,t,c) = \bot \wedge u \neq \mathtt{F}) \vee \mathrm{pattern}(s,t,c) = u)\end{aligned}$$

Diese Regelverschärfung beeinträchtigt die Vollständigkeit[1] des Regelsystems nicht. Belegungen, die nirgends vorkommen, sind ohnehin überflüssig. Belegungen, die *pattern* widersprechen, müssen irgendwann zu einem Fehler führen. Genauso wird es zu einem Fehler führen, wenn ein Eintrag mit `F` belegt wird, ohne daß *pattern* dies erfordert.

[1] Ein Regelsystem (oder Algorithmus) soll vollständig genannt werden, wenn mit ihm alle existierenden Lösungen generiert werden können.

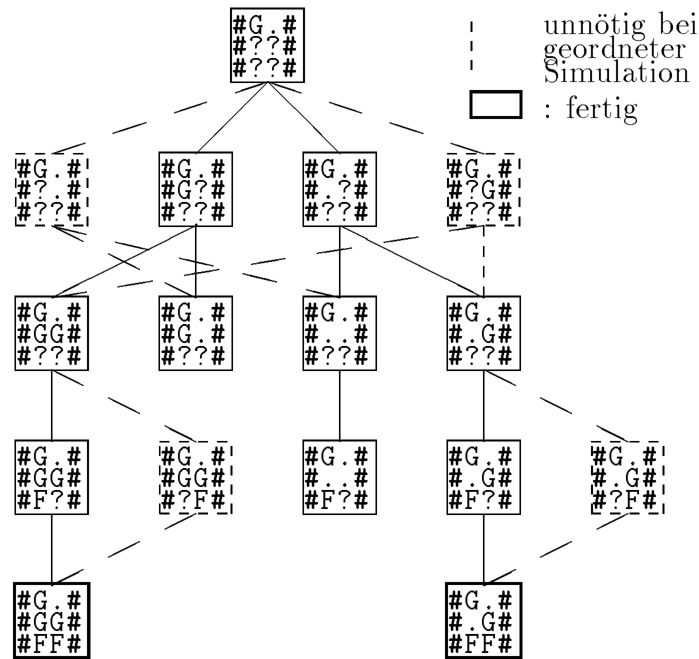

Abbildung 3.3: Suchraum für nichtdeterministischen Algorithmus unter Benutzung von SIMULATE und DEFINE'.

3.4 Grundeigenschaften der Suchstrategien

3.4.1 Der Suchraum als Graph

Algorithmus 3.2 funktioniert auch dann noch, wenn die Regel DEFINE durch die Regel DEFINE' ersetzt wird. Damit ist aber noch nicht die Frage beantwortet, wie am besten der Nichtdeterminismus aufgelöst wird. Diese Frage läßt sich leichter beantworten, indem die Struktur des Suchraums näher betrachtet wird. Der Suchraum läßt sich als gerichteter Graph darstellen, dessen Knoten mit Belegungen von σ und C markiert sind und dessen Kanten Regelanwendungen sind, die einen Knoten in den anderen überführen. Da σ und C durch die Regeln immer nur erweitert werden, handelt es sich offensichtlich um einen gerichteten azyklischen Graphen. Der Graph hat eine eindeutige Wurzel, die durch die in Algorithmus 3.2 beschriebenen Anfangswerte festgelegt wird. Blätter sind Konfigurationen, in denen keine weiteren Regeln mehr anwendbar sind. Dies können sowohl Lösungen als auch Sackgassen sein. Abbildung 3.3 zeigt den Suchraum für $N = 3$, $M = 2$.

3.4.2 Reduzierung des Suchraums auf einen Baum

Für sich allein betrachtet stellt die Sichtweise des Problems als gerichteter Graph noch keinen Fortschritt gegenüber dem vollständigen Aufzählen aller Überführungsfunktionen dar. Im Gegenteil — den Graphen systematisch zu durchsuchen liefe auf einen noch langsameren Algorithmus hinaus.

Bei näherer Betrachtung stellt sich aber heraus, daß der größte Teil des durch SIMULATE und DEFINE' eingeführten Nichtdeterminismus für die Vollständig-

keit der Suche nicht nötig ist. Solange eine Folge von Regelanwendungen zur gleichen Belegung von σ führt, ist die Reihenfolge der Regelanwendungen egal. Daraus ergibt sich, daß eine Suchstrategie aus der Menge aller Stellen im Raumzeitdiagramm, auf die Regeln anwendbar sind, in deterministischer Weise eine einzige Stelle auswählen kann, ohne die Vollständigkeit der Suche zu gefährden. Die verbleibende Quelle von Nichtdeterminismus ist die Auswahl eines Ergebniszustandes bei Anwendung von DEFINE'.

Wird derjenige Teil des Suchgraphen weggelassen, der von der rationalisierten Suchstrategie nicht betrachtet wird, entsteht ein Baum, dessen Wurzel und Lösungsknoten mit denen des ursprünglichen Graphen identisch sind. Da Verzweigungen des Suchbaums nur bei Anwendung von DEFINE' auftreten können, kann der Suchbaum weiter vereinfacht werden, indem alle Knoten in einer Kette von Simulationen zu einem einzigen Knoten zusammengefaßt werden. (Siehe auch Abbildung 3.4.)

3.4.3 Baumtraversierung durch Tiefensuche

Eine weitere Entwurfsentscheidung besteht darin, wie der oben beschriebene Suchbaum zu durchlaufen ist. Es gibt eine ganze Reihe von Gründen, warum Tiefensuche als die beste Grundstrategie erscheint:

- Der Speicherverbrauch ist gering. Bei der vorliegenden Größe (und Breite) des Suchbaums wäre z.B. Breitensuche nicht realisierbar.
- Tiefensuche ist leicht zu implementieren.
- Es gibt keinen offensichtlichen Grund, warum eine andere Traversierungsstrategie Suchaufwand sparen könnte.
- C bzw. σ werden bei jeder Regelanwendung verändert. Beim Backtracking während einer Tiefensuche müssen nur tatsächliche Änderungen rückgängig gemacht werden. Bei anderen Suchbaumtraversierungen müßten dagegen die gesamten (recht umfangreichen) Datenstrukturen ausgetauscht werden.

3.5 Suchstrategie „geordnete Simulation“

3.5.1 Geordnete Simulation macht Regelanwendungen einfach

Nachdem das Regelsystem und die grundsätzliche Suchstrategie geklärt sind, bleibt die Frage, wie die nächste anzuwendende Regelinstanz ausgewählt werden soll. Es wäre ziemlich aufwendig, jedesmal C nach möglichen Stellen für eine Regelanwendung zu durchsuchen. Eine einfache Lösung dieses Problems besteht darin, die Zellen von C der Reihe nach durchzugehen. „Der Reihe nach“ soll hier heißen, mit Retinagröße 2 zu beginnen und Zeile für Zeile das Raumzeitdiagramms von links nach rechts mit zunehmenden Zeiten zu durchlaufen. Nachdem eine Retinagröße komplett abgehandelt ist, wird zur nächstgrößeren Retina übergegangen usw., bis ganz C durchlaufen ist. Dann ist es in jedem Schritt möglich, entweder SIMULATE oder DEFINE' gefolgt von SIMULATE anzuwenden, oder es tritt ein Fehler auf, der Backtracking auslöst. Abbildung 3.4 stellt den für

$N = 3$ entstehenden Suchbaum dar, mit dessen Hilfe bewiesen werden kann, daß keine Lösung des FSSP mit 3 Zuständen existiert.

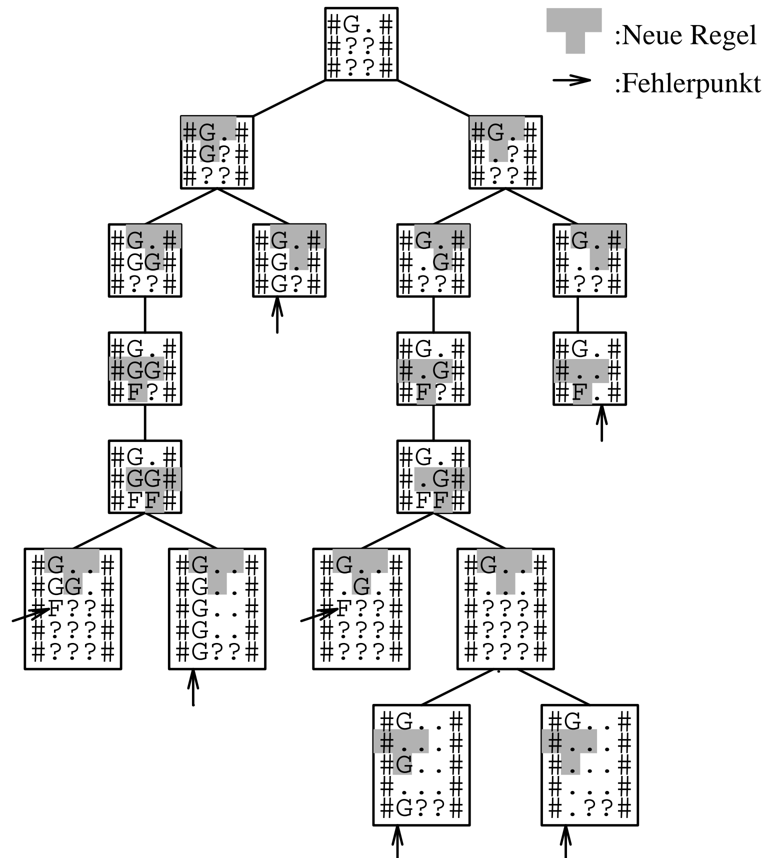

Abbildung 3.4: Suchbaum für geordnete Simulation mit $N = 3$.

Es gibt eine ganze Reihe anderer möglicher Durchlaufreihenfolgen mit ähnlichen Eigenschaften. Die hier gewählte scheint jedoch recht gut zu sein, weil sie einfach ist und weil sehr schnell Stellen in C erreicht werden, in denen gefeuert werden muß. Dadurch lassen sich viele fehlerhafte Überführungsfunktionen frühzeitig aussieben.

Geordnete Simulation hat den zusätzlichen Vorteil, daß Simulationsschritte beim Backtracking quasi automatisch rückgängig gemacht werden. Gültig sind immer genau die Einträge von C, die bezüglich der Simulationsreihenfolge vor der momentanen Position liegen.

3.5.2 Der Grundalgorithmus

Der Algorithmus in Abbildung 3.5 zeigt einen einfachen Suchalgorithmus nach dem Prinzip der geordneten Simulation. Er ist als rekursive boolsche Funktion dargestellt, die überprüft, ob eine Lösung existiert oder nicht. Im positiven Falle findet sich in σ eine Lösung. Das Hauptprogramm searchOrdered macht nur die schon aus Algorithmus 3.2 bekannten Initialisierungen und ruft dann die eigentliche Suchfunktion searchRek mit der Position der ersten zu simulierenden Zelle auf ($s = 2$, $t = 1$, $c = 1$).

Die Funktion searchRek betrachtet immer genau eine Stelle (s, t, c) des Raumzeitdiagramms. Im einfachsten Fall ist SIMULATE auf diese Stelle anwendbar; dann wird die entsprechende neue Eintragung gemacht, und searchRek ruft sich rekursiv für die nächste Position auf. Zu diesem Zweck wird zunächst die Spaltennummer c erhöht. Ist das Ende der Retina erreicht $(c > s)$, so wird die nächste Zeile begonnen, indem c auf 1 zurückgesetzt und t um 1 erhöht wird. Entsprechend wird zur nächsten Retina weitergeschaltet, wenn die Automaten gefeuert haben $(t > 2s-2)$. Ist schließlich die letzte Retina erreicht, so bedeutet dies, daß eine Lösung gefunden wurde, und **TRUE** wird zurückgegeben.

Falls DEFINE' anwendbar[2] ist, so wird nacheinander jede mögliche Belegung des entsprechenden Eintrags von σ ausprobiert und searchRek auf die so geänderte Situation angewandt. Ist einer dieser Aufrufe erfolgreich, so wird **TRUE** zurückgegeben. Hat keiner der Versuche zum Erfolg geführt, so wird der letzte Eintrag in σ wieder entfernt, und das Ergebnis ist **FALSE**. Ist überhaupt keine Regel anwendbar, so ist die Suche in eine Sackgasse gelaufen, und das Ergebnis ist ebenfalls **FALSE**.

```
FUNCTION searchOrdered(N, M, pattern_0, σ_0):BOOLEAN
Initialize σ, C, pattern as in searchNonDet
RETURN seachRek(2, 1, 1)

FUNCTION searchRek(s, t, c):BOOLEAN
IF SIMULATE applies to (s,t,c) THEN
    apply this instance of SIMULATE
    (*Next Step:*)
    c := c + 1 (*next cell*)
    IF t > s        THEN c:=1; t := t + 1     (*next row*)
    IF t > 2s − 2   THEN t:=1; s := s + 1     (*next retina*)
    IF s > M        THEN RETURN TRUE(*finished*)
    ELSE                 RETURN searchRek(s, t, c)
ELSIF DEFINE' applies to (s,t,c) ∧ ν(s,t,c) = (x,y,z) THEN
    FOR each state u allowed by this instance of DEFINE' DO
        σ(x,y,z) := u (*try this choice*)
        IF searchRek(s, t, c) THEN RETURN TRUE
    σ(x,y,z) := ⊥(*Undo changes to σ*)
    RETURN FALSE (*choices exhausted*)
ELSE
    RETURN FALSE (*no rule applicable*)
```

Abbildung 3.5: Grundalgorithmus für geordnete Simulation.

Für eine effiziente Implementierung ist die gezeigte rekursive Formulierung des Algorithmus nicht gut geeignet. Im folgenden wird deshalb von einer nichtrekursiven Variante ausgegangen, bei der nur die Position innerhalb von C und der betroffene Eintrag von σ auf einem explizit verwalteten Keller abgelegt wird und

[2] Man beachte, daß SIMULATE und DEFINE' nie gleichzeitig auf dieselbe Position anwendbar sein können.

auch das nur dann, wenn DEFINE’ zur Anwendung kommt. Ein solcher Punkt wird von nun an *Auswahlpunkt* genannt.

3.5.3 Beschneiden des Suchbaums

Bei genauerer Betrachtung des Suchablaufs in Algorithmus 3.5 stellt sich heraus, daß die folgende Situation häufig auftritt: An einem Auswahlpunkt wird eine Entscheidung getroffen, und nach einigen Simulationsschritten tritt ein Fehler auf. Dann werden nacheinander alle für den Auswahlpunkt möglichen Einträge in σ betrachtet, und jedesmal tritt der Fehler wieder auf. Das Analoge kann auch für ganze Gruppen von Auswahlpunkten auftreten; dann werden alle möglichen Kombinationen von Belegungen betrachtet, ohne daß der Fehler verschwinden kann.

Der Grund für dieses Verhalten ist, daß der Fehler völlig unabhängig von den letzten Auswahlpunkten ist. Selbst wenn die betroffenen Einträge von σ nicht gemacht würden, könnte durch bloße Anwendung der Regel SIMULATE die Fehlersituation herbeigeführt werden. Der rechte Teil von Abbildung 3.6 zeigt ein typisches Beispiel für diesen Fall: Der zuletzt gemachte Eintrag in σ „GX# → X“ hat keinen Einfluß auf das fälschlicherweise ausbleibende Feuern und kann deshalb entfernt werden.

```
#G..#        #G..#
#XG.#        #XG.#
#GGX#        #XGX#
#X..#        #XGX#
#FF.#        #X??#
```

Abbildung 3.6: Beispiele für Entfernbarkeit von Auswahlpunkten.

Aus dieser Beobachtung ergibt sich eine einfache Optimierungsstrategie: Beim Backtracking nach einem Fehler wird zunächst der letzte Eintrag in σ entfernt. Dann wird nochmal (nun ohne den gerade entfernten Eintrag) simuliert und überprüft, ob der gleiche Fehler wieder auftritt. Falls ja, so hängt der Fehler überhaupt nicht mit dem betrachteten Eintrag zusammen, und der Auswahlpunkt kann gänzlich vom Keller entfernt werden. Dann wird der nächste Kellereintrag der gleichen Betrachtung unterzogen.

Naiv angewandt würde die gerade beschriebene Optimierung sehr viele zusätzliche Simulationsschritte nötig machen und damit die erhoffte Zeitersparnis gefährden. Es gibt aber zwei einfache Fälle, in denen sich sofort entscheiden läßt, ob ein Auswahlpunkt (s, t, c) mit Umgebung (x, y, z) einen Einfluß auf einen Fehlerpunkt an der Stelle (s', t', c') mit Umgebung (x', y', z') hat:

1. Wenn (s', t', c') im Ereignishorizont von (s, t, c) liegt, d.h. $s = s' \wedge |c' - c| \leq t' - t$, so liegt in jedem Fall eine Beeinflussung vor (siehe Abbildung 3.7). Der linke Teil von Abbildung 3.6 zeigt ein Beispiel dafür, daß die Umkehrung nicht gilt: Die Einträge von σ „$X.. \to F$“, „$\#X. \to F$“, „$GX\# \to .$“ und „$GGX \to .$“ werden vom Keller entfernt, da sie keine weiteren Alternativen offen[3] lassen. Der Eintrag „$\#GG \to X$“ beeinflußt den Fehler zwar nicht

[3] Der Ruhezustand wird immer als letztes ausprobiert.

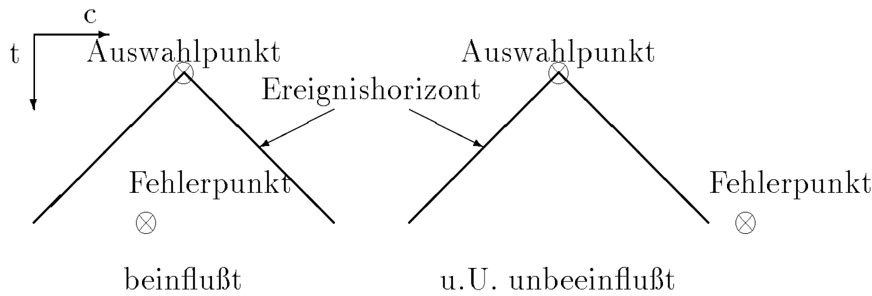

Abbildung 3.7: Fehlerpunkte innerhalb und außerhalb des Ereignishorizonts.

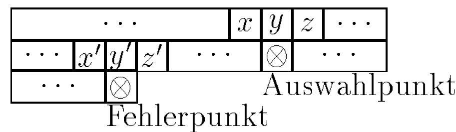

Abbildung 3.8: Backtracking, wenn der Auswahlpunkt unmittelbar vor dem Fehlerpunkt liegt. Beeinflussung gdw. $(x', y', z') = (x, y, z)$.

direkt, aber nachdem für diesen Übergang ein neues Ergebnis gewählt worden ist, macht es auch wieder Sinn, für die vorher entfernten Kellereinträge von neuem alle Alternativen auszuprobieren. Dadurch können sich neue Lösungsmöglichkeiten ergeben.

2. Ist der Fehlerpunkt weniger als $s - 1$ Zellen vom Auswahlpunkt entfernt, d.h. $s' = s \wedge (t = t' \vee t = t' - 1 \wedge c' - 1 > c)$, so liegt eine Beeinflussung vor gdw. $(x, y, z) = (x', y', z')$ (siehe Abbildung 3.8).

3.5.4 Vermeidung redundanter symmetrischer Lösungen

Der Übergang von vollständig spezifizierten Überführungsfunktionen zu solchen, bei denen nur relevante Einträge belegt sind, hat zu einer erheblichen Geschwindigkeitssteigerung beigetragen, indem funktionell äquivalente Lösungen zu Äquivalenzklassen zusammengefaßt wurden. Es ist sogar möglich, eine noch weitergehende Zusammenfassung zu betreiben. Die Definition des FSSP weist nur den Zuständen $\texttt{Z}_0$, `G` und `F` eine spezielle Bedeutung zu. Alle anderen Zustände sind a priori völlig äquivalent zueinander. Daraus ergibt sich, daß sich aus einer gegebenen Überführungsfunktion weitere Überführungsfunktionen durch Umbenennung äquivalenter Zustände gewinnen lassen, die sich bezüglich Ihrer Eignung als Lösung des FSSP in nichts unterscheiden (siehe auch [2]).

Diese äquivalenten Lösungen lassen sich bei der Suche ausblenden, indem über die Menge der in der jeweiligen Situation äquivalenten Zustände Buch geführt wird. Steht eine Entscheidung über einen neuen Eintrag in die Überführungsfunktion an, so werden nur die nichtäquivalenten und einer der äquivalenten Zustände in Betracht gezogen. Fällt die Entscheidung auf einen der äquivalenten Zustände, so findet eine Brechung der vorliegenden Symmetrie statt, da der

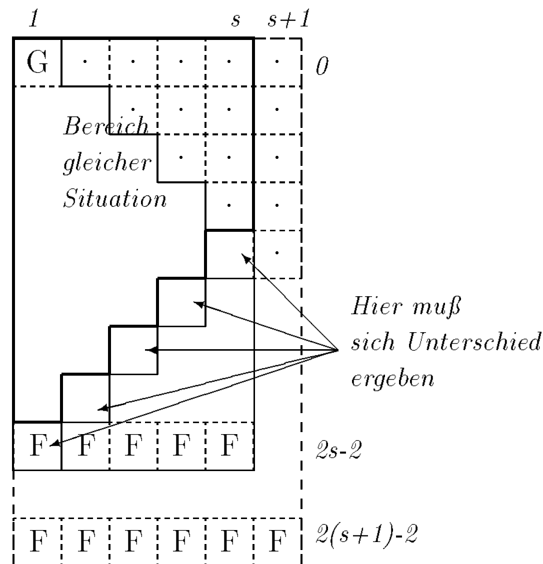

Abbildung 3.9: Satz 1.

neue Überführungsfunktionseintrag den gewählten Zustand von den anderen unterscheidbar macht. Deshalb wird dieser Zustand aus der Menge der äquivalenten Zustände entfernt und die Suche fortgesetzt.

3.5.5 Eine Variante: Schräge Simulation

Balzer weist in [1] auf eine recht vielversprechend klingende zusätzliche Heuristik hin:

Satz 1 $\forall s : \forall c \in \{1, \ldots, s\} : C^{s}_{2s-1-c}(c) \neq C^{s+1}_{2s-1-c}(c)$

Der Grund für Satz 1 läßt sich am besten an Abbildung 3.9 erkennen. Die Information, die das Feuern der ersten Retinazelle auslöst, kann nur auf direktem Wege vom rechten Rand gekommen sein. Denn alle anderen Zellen im Ereignishorizont von $(s, 2s-2, 1)$ verhalten sich genauso wie für kleinere Retinae. Für die nächstgrößere Retina muß in der entsprechenden Diagonale ein anderes Verhalten an den Tag gelegt werden. Würde auch nur an einer Stelle innerhalb der Diagonale der gleiche Zustand angenommen wie für die kleinere Retina, so würde Zelle 1 zu früh feuern. In Abbildung 3.4 tritt diese Situation z.B. gleich zweimal auf.

Um zu testen, wie sich diese Heuristik auswirkt, wurde eine Variante der geordneten Simulation realisiert, die in Diagonalen von rechts oben nach links unten simuliert. (Abbildung 3.10 zeigt die Simulationsreihenfolge für $M = 3$.) Diese *schräge* Simulation hat die Eigenschaft, daß, falls eine gemäß Satz 1 falsche Entscheidung getroffen wird, dies ohne Betrachtung weiterer Auswahlpunkte zu einem Fehler führt.

Entgegen aller Erwartung stellte sich aber heraus, daß diese Durchlaufreihenfolge zur Betrachtung von *mehr* verschiedenen Übergangsfunktionen führt als die einfache horizontale Simulation. Der Hauptgrund dafür dürfte darin liegen, daß die anderen Heuristiken zur Beschneidung des Suchbaums in den meisten Fällen, in denen Satz 1 zur Anwendung kommen könnte, genauso effektiv sind.

```
G..
124
357
689
FFF
```

Abbildung 3.10: Simulationsreihenfolge für schräge Simulation.

Ein expliziter Einbau von Satz 1 könnte zwar immer noch zur Einsparung von Simulationsschritten genutzt werden, aber angesichts der Tatsache, daß der Algorithmus dadurch auch wieder komplizierter wird, dürfte dies zu keiner lohnenden Beschleunigung führen. Deshalb wurde davon abgesehen, diese Idee weiter zu verfolgen.

3.6 Suchstrategie „maximale Simulation“

3.6.1 Bevorzugung deterministischer Regelanwendungen

Die Ineffizienz des „generiere und teste“-Verfahrens aus Abschnitt 3.3 liegt darin begründet, daß die DEFINE-Regel zu früh angewandt wurde. Dies legt die Überlegung nahe, daß umgekehrt ein effizientes Verfahren entstehen könnte, wenn SIMULATE so oft wie möglich angewendet wird.

Diese Idee läßt sich noch durch die Beobachtung verfeinern, daß es auch Instanzen von DEFINE' gibt, die deterministisch sind und keine Verzweigung im Suchbaum hervorrufen. Das sind diejenigen, bei denen der neue Eintrag in σ durch *pattern* vorgegeben ist (z.B. Eintrag von `F` zum Zeitpunkt $2s-2$). Wenn im folgenden also von Simulation die Rede ist, so sind deterministische Anwendungen von DEFINE' einbezogen. Diese Strategie, so oft wie möglich deterministische Regeln anzuwenden, soll *maximale Simulation* genannt werden.

3.6.2 Buchführung über gültige Einträge von C

Eine einfache Implementierung der maximalen Simulation läßt sich aus der geordneten Simulation ableiten. Wann immer an einem Auswahlpunkt (s, t, c) eine Entscheidung ansteht, wird zunächst so oft wie möglich simuliert, um herauszufinden, ob diese Entscheidung einen Fehler provoziert. Tatsächlich wird dadurch die Zahl der zu betrachtenden Auswahlpunkte gegenüber der geordneten Simulation deutlich reduziert. Leider wächst die Zahl der benötigten Simulationsschritte stark an, und deshalb ist die Gesamtbilanz dieser einfachen Implementierung negativ.

Genauer betrachtet ist aber der größte Teil der im obigen Verfahren benötigten Simulationsschritte redundant. Nur weil eine einzige Entscheidung rückgängig gemacht wird, werden noch lange nicht alle gemachten Simulationsschritte ungültig. Gefragt ist also eine flexiblere Buchführung über die gültigen Einträge im Raumzeitdiagramm. Die einfache Regel für die geordnete Simulation ist für diesen Zweck zu primitiv.

Der Schlüssel für diese Buchführung ist eine Datenstruktur *Contour*, die die Menge der momentan möglichen Auswahlpunkte repräsentiert.

$$\text{Contour} := \{(s,t,c)|\text{DEFINE' ist auf } (s,t,c) \text{ anwendbar } \wedge \\ \text{pattern}(s,t,c) = \sigma(s,t,c) = \bot\}$$

Sind alle möglichen deterministischen Regelanwendungen ausgeführt — wie von der zur Diskussion stehenden Strategie verlangt — so lassen sich aus *Contour* alle anderen benötigten Informationen ableiten. Sind nämlich $(s,t,c),(s,t',c') \in$ Contour, $c < c'$ zwei benachbarte Auswahlpunkte, d.h.

$$\neg\exists(s,t'',c'') \in \text{Contour} : c < c'' < c'$$

so sind dazwischenliegende Einträge $C^s_{t''}(c'')$ des Raumzeitdiagramms (mit $c \leq c'' \leq c'$) genau dann bekannt, wenn

$$t'' - t < c'' - c \wedge t'' - t' < c' - c''$$

Abbildung 3.11 veranschaulicht diese Beziehung. Zwischen den beiden Auswahlpunkten gibt es jeweils ein kegelförmiges Gebiet bekannter Einträge. Ganz ähnliche Beziehungen gelten für Zellen zwischen einem Auswahlpunkt und dem Rand der Retina.

3.6.3 Der Grundalgorithmus

Mit Hilfe dieser (zunächst abstrakten) Datenstruktur ist es möglich, einen Suchalgorithmus mit der Strategie der maximalen Simulation zu schreiben, der nur dort Simulationsschritte durchführt, wo durch Änderungen an σ eine neue Situation aufgetreten ist.

Abbildung 3.12 enthält den Pseudocode für diese Strategie. Genau wie bei den vorher vorgestellten Algorithmen werden zuerst σ, C und *pattern* initialisiert. Dann werden alle bereits zu Beginn möglichen Simulationen ausgeführt. (Also insbesondere die Simulationen auf Grund von „$\mathsf{Z_0Z_0\#\rightarrow Z_0}$“ und „$\mathsf{Z_0Z_0Z_0\rightarrow Z_0}$“.) Als Nebeneffekt wird dadurch *Contour* initialisiert, so daß es die offenen Auswahlpunkte enthält. (i. allg. $\{(s,1,1)|s \leq M\}$).

Die Hauptschleife wählt dann jeweils einen Auswahlpunkt aus *Contour* aus, wählt (nichtdeterministisch) eine dazu passende Instanz von DEFINE' und ruft dann `simulateEntry` auf, um alle deterministischen Konsequenzen dieser Entscheidung zu verfolgen.

`simulateEntry` sucht alle Auswahlpunkte, die den gleichen Eintrag in σ betreffen wie ihr Argument und ruft das eigentliche „Arbeitspferd“ `simulateRange` für diese Auswahlpunkte auf.

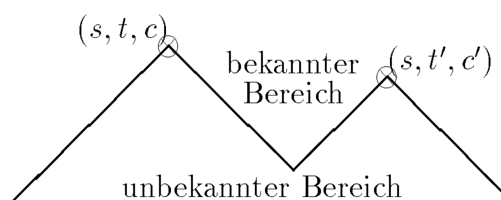

Abbildung 3.11: Struktur von C^s zwischen zwei benachbarten Elementen von *Contour*.

```
PROCEDURE searchMaximalSimulation(N, M, pattern_0, σ_0)
(*Initializations:*)
Initialize σ, C, pattern as in searchNonDet
Contour:= ∅
FOR s := 2 TO M DO simulateRange(s, 1, 1, s)
(*Main loop:*)
WHILE Contour ≠ ∅ DO
    Select Choicepoint (s, t, c) ∈ Contour (*deterministically*)
    Apply DEFINE' to (s, t, c) (*Choice of entry to σ is nondeterministic*)
    simulateEntry(s, t, c)

(*Exploit all deterministic consequences of the preceding definition*)
PROCEDURE simulateEntry(s, t, c)
FOR each Choicepoint (s', t', c') ∈ Contour with ν(s', t', c') = ν(s, t, c) DO
    Contour := Contour − {(s', t', c')}
    simulateRange(s', t', c', c')

(*Perform deterministic rule applications within event horizon of*)
(*(s, t_0, c_left), (s, t_0, c_left + 1),..., (s, t_0, c_right)*)
PROCEDURE simulateRange(s, t_0, c_left, c_right)
FOR t:=t_0 TO 2s − 2 DO
    FOR c:=c_left − (t − t_0) TO c_right + (t − t_0) DO
        IF DEFINE' applies to (s, t, c) in a unique way THEN
            (*deterministic entry to σ due to pattern:*)
            Apply DEFINE' to (s, t, c)
            Contour := Contour − (s, t, c)
            simulateEntry(s, t, c)
        IF SIMULATE applies to (s, t, c) THEN
            Apply SIMULATE to (s, t, c)
        ELSIF DEFINE' applies to (s, t, c) (*nondeterministically*) THEN
            Contour:= Contour ∪ {(s, t, c)}
        ELSIF ν(s, t, c) = (x, y, z) ∧ x ≠ ⊥ ∧ y ≠ ⊥ ∧ z ≠ ⊥THEN
            backtrack
```

Abbildung 3.12: Grundalgorithmus für maximale Simulation.

Abbildung 3.13: Struktur des von simulateRange betrachteten Ausschnitts von C.

SimulateRange betrachtet alle Wirkungen, die von einem Intervall $[c, c']$ in der Retina der Größe s zum Zeitpunkt t ausgehen können. In einer Schleife, die das in Abbildung 3.13 dargestellte wannenförmige Gebiet im Raumzeitdiagramm durchläuft, werden alle Punkte innerhalb des Ereignishorizonts von $[c, c']$ durchlaufen. Dabei ist folgende Fallunterscheidung zu machen:

- Ist eine deterministische Instanz von DEFINE' anwendbar, so wird die Regel angewandt.
- Im einfachsten Fall kann ein Simulationsschritt durchgeführt werden.
- Wird ein neuer Auswahlpunkt gefunden, so wird er in *Contour* aufgenommen.
- Falls keine Regel anwendbar ist, obwohl alle Vorgänger in der Nachbarschaft bekannt sind, so ist ein Fehlerpunkt gefunden, und es muß Backtracking ausgelöst werden.

Da der eben beschriebene Algorithmus der Verständlichkeit halber gegenüber der tatsächlichen Implementierung vereinfacht wurde, sind noch einige Bemerkungen angebracht:

- Die nichtdeterministische Auswahl einer Belegung für σ sei wieder durch Tiefensuche aufgelöst.
- simulateRange wurde hier so formuliert, daß i. allg. ein großer Teil der betrachteten Positionen (s, t, c) keine vollständig bekannte Umgebung $\nu(s, t, c)$ hat und damit keine der Bedingungen in der Doppelschleife anwendbar sein wird. Durch entsprechende Optimierung (die von der konkreten Implementierung der Datenstrukturen abhängt) ergibt sich eine Kontrollstruktur, die diese uninteressanten Positionen ausschließt und die notwendigen Tests in einigermaßen effizienter Weise durchführt.

3.6.4 Implementierung von *Contour*

Es wurden mehrere Varianten ausprobiert, die Datenstruktur für *Contour* zu implementieren.

Die erste Methode legt diejenigen Elemente (s, t, c) von *Contour*, die unterschiedliche Umgebungen $\nu(s, t, c)$ haben, auf einem Stapel ab. Zusätzlich wird zu jedem Eintrag in σ eine Liste der Auswahlpunkte mit passender Umgebung abgespeichert. Die Elemente dieser Listen werden in Komponenten von C abgelegt, so daß keine dynamische Speicherverwaltung erforderlich ist. In der Hauptschleife von searchMaximalSimulation wird dann jeweils das oberste Stapelelement ausgewählt. In simulateEntry muß nur eine Liste abgelaufen werden. Ebenso einfach ist das Einfügen neuer Auswahlpunkte in simulateRange. Die einzige etwas problematische Operation ist das Löschen von Auswahlpunkten bei deterministischen Anwendungen von DEFINE'. Dieses Problem läßt sich lösen, indem der Auswahlpunkt zunächst überhaupt nicht vom Stapel gelöscht wird; bei der Auswahl in der Hauptschleife wird dann jeweils abgefragt, ob $\sigma(\nu(\mathsf{Top}(\text{Contour}))) = \perp$. Im negativen Falle wird der Auswahlpunkt verworfen und der nächste ausprobiert. Da dieser Fall in der Praxis relativ selten auftritt, beschränken sich die zusätzlichen Kosten im wesentlichen auf die zusätzlich nötige Abfrage.

Backtracking wird so implementiert, daß die Elementaroperationen „Simulationsschritt“, „Auswahlpunkt auswählen/einfügen/überspringen“ und „DEFINE' anwenden“ auf einem Stapel gespeichert werden und beim backtracking rückgängig gemacht werden.

Eine einfache Variante verwaltet die verschiedenen Auswahlpunkte als FIFO-Struktur und durchläuft die Listen zusammengehöriger Auswahlpunkte in umgekehrter Reihenfolge. Es stellt sich heraus, daß dadurch der Suchraum nochmal etwas verkleinert wird.

Eine weitere Implementationsmöglichkeit für *Contour* besteht darin, für jedes s und c zu speichern, ob und wann dort ein Auswahlpunkt vorliegt. Dieses Verfahren ist einfacher als das oben betrachtete, und es macht es möglich, in der Hauptschleife jeweils den „frühesten“[4] Auswahlpunkt zu wählen. Wie sich herausstellt, ist die Zahl der betrachteten Auswahlpunkte dadurch etwas geringer als bei den anderen Implementierungen von *Contour*. Nachteil dieser Strategie ist aber, daß in simulateEntry und simulateRange nach Auswahlpunkten gesucht werden muß, was den möglichen Zeitgewinn wieder aufzehrt.

[4] Bezüglich der Simulationsreihenfolge aus der geordneten Simulation

Kapitel 4

Ergebnisse zum FSSP

Dieses Kapitel beschreibt die Ergebnisse, die mit den in Kapitel 3 dargestellten Suchalgorithmen erzielt wurden. Hauptergebnis ist, daß es keine Lösung des FSSP mit vier Zuständen gibt (Abschnitt 4.1). Um die gleiche Frage für fünf Zustände zu beantworten, sind die momentan bekannten Algorithmen nicht schnell genug (Abschnitt 4.2). Da Lösungen mit sechs Zuständen bekannt sind, sind Zustandszahlen größer fünf nicht interessant. Allerdings läßt sich noch fragen, ob es Lösungen mit mehr als vier Zuständen gibt, die bestimmte Zusatzbedingungen erfüllen, welche den Suchraum so weit einschränken, daß die Programme eine Antwort finden (Abschnitt 4.3).

4.1 Vier Zustände

Grundaussage dieses Abschnitts ist der folgende Satz:

Satz 2 *Es gibt keine Lösung des FSSP mit vier Zuständen.*

Der Beweis wird mit Hilfe des C-Programm in Anhang B erbracht, das Algorithmus 3.5 implementiert. Werden diesem Programm vier Zustände und die maximale Retinagröße 9 vorgegeben, so ergibt sich, daß es 27 Lösungen bis Größe 8 aber keine bis Größe 9 gibt und damit auch keine Lösung für beliebige Retinagrößen.
□

4.1.1 Vergleich mit Balzers Ergebnis

Satz 2 findet sich schon in [2]. Allerdings ist der dort aufgeführte Beweis allem Anschein nach nicht richtig. Balzer hat ein Programm geschrieben, das dem Algorithmus zur geordneten Simulation (Abbildung 3.5) plus den Heuristiken aus Abschnitt 3.5.3 sehr ähnlich ist. Allerding hat Balzer die Heuristik zum Abschneiden überflüssiger Suchbaumäste anders formuliert: *"We know that if no occurrence of a production occurs in such a position that it could send a signal to the position where the error occured before this occurrence, then the resultant of this production is irrelevant in eliminating the error."* Diese Aussage stimmt z.B. dann nicht, wenn ein Auswahlpunkt indirekt wirkt, indem er einen Auswahlpunkt beeinflußt, der seinerseits das Auftreten des Fehlers beeinflußt. Auch das Beispiel im linken Teil von Beispiel 3.6 wird von dieser Heuristik nicht richtig

behandelt. Dieses Problem wurde zwar von Balzer bereits erkannt [2] aber offenbar nicht mehr implementiert[1]. Dies läßt sich schon daran erkennen, daß Balzers Programm ca. 60000 verschiedene Übergangsfunktionen betrachtet, während ein Programm mit der korrekten Heuristik fast 16 Millionen Möglichkeiten durchsuchen muß! Eine solche Zahl von Kombinationen wäre übrigens mit den damaligen technischen Mitteln schwer zu bewältigen gewesen.

4.1.2 Korrektheit des neuen Beweises

Nachdem im letzten Abschnitt ein lange akzeptierter (allerdings auch kaum beachteter) Beweis widerlegt wurde, liegt der Einwand nahe, daß auch das in dieser Arbeit benutzte Programm fehlerhaft sein könnte und der Beweis somit immer noch nicht vollständig erbracht sei. Ohne zu tief in die schwierige Frage eindringen zu wollen, was denn nun einen gültigen Beweis ausmacht, soll hier kurz mein persönlicher Standpunkt dargestellt werden:

Nur die wenigsten komplexen Beweise in der theoretischen Informatik werden so rigoros geführt, daß eine lückenlose Beweiskette entsteht, die jeden einzelnen Schluß mit Angabe von benutzten Axiomen, Sätzen und Schlußregeln angibt. Ein Beweis gilt vielmehr dann als richtig, wenn eine eingehende Begutachtung keine Lücken und Fehler offenbart. Faßt man nun ein Programm als einen Beweis auf, so kann es in gleicher Weise begutachtet werden, wie ein Beweis in der üblichen Sprache der Mathematik. Oft ist die Semantik einer Programmiersprache sogar genauer spezifiziert als die Mischung aus Symbolen und natürlicher Sprache, in der andere Beweise formuliert werden. Dient die Ausgabe des Programms nun als Kriterium für die Wahrheit eines Satzes, so ist der einzige neue Aspekt, daß die Korrektheit des Compilers und der Maschine angenommen werden muß, um der Ausgabe des Programms zu glauben. Da Compiler und Maschinen unterschiedlicher Hersteller herangezogen werden können, ist die Wahrscheinlichkeit, daß sich aus dieser Quelle ein Fehler einschleicht, wohl kleiner als die Wahrscheinlichkeit einen unentdeckten Fehler in der Programmierung (und damit der Formulierung des Beweises) zu machen.

Akzeptiert man die grundsätzliche Eignung eines Programms als Beweis, so bleibt die Aufgabe, das eingesetzte Programm so zu formulieren, daß seine Korrektheit einsichtig wird.

Eine wichtige Maßnahme ist das „Abspecken“ des Programms. In Anhang B findet sich ein Listing eines C-Programms für die geordnete Simulation, aus dem alle überflüssigen E/A-Befehle, Debugginghilfen und Heuristiken entfernt wurden. Dadurch läuft es zwar deutlich langsamer, kommt aber immer noch zum Ziel und enthält erheblich weniger Fehlerquellen.

Mindestens genauso wichtig erscheint, daß das Programm überhaupt Bestandteil der Arbeit ist. Nur dadurch wird es möglich, daß Zweifel — wie sie an einem Teil von Balzers Arbeit auftraten — auch nach 30 Jahren noch geklärt werden können. (Selbst in Balzers Doktorarbeit [1] wird die problematische Heuristik nur in natürlicher Sprache beschrieben.)

[1] Dr. Balzer war so freundlich, meine Anfrage diesbezüglich zu beantworten. Da er sich jedoch seit fast 30 Jahren nicht mehr mit dem FSSP beschäftigt und er sich deshalb nicht mehr an die Details erinnert, konnte auch er nur vorschlagen, die Zahl der betrachteten Möglichkeiten für die beiden Algorithmenvarianten zu vergleichen.

Da ein C-Programm nicht das ideale Mittel zur verständlichen Darstellung eines Algorithmus ist, sollte zusätzlich der Algorithmus selbst in einer abstrakteren Sprache eingeführt werden. Aufgabe des „Programminspektors“ ist es dann nur noch, zu überprüfen, daß das Programm den Algorithmus implementiert. Diese Überlegung ist der Grund dafür, daß in Kapitel 3 eine relativ formale Darstellung gewählt wurde.

Auf diesem abstrakten Niveau der Darstellung ist es auch praktikabel, Beweise von Schlüsseleigenschaften des Algorithmus durchzuführen, die dann wieder klassische mathematische Beweise sind. Für Algorithmus 3.5 ergeben sich z.B. folgende Fragen bezüglich der Vollständigkeit und Korrektheit:

- Ist der nichtdeterministische Algorithmus korrekt und vollständig? Insbesondere: Charakterisieren die Regeln SIMULATE und DEFINE sowie die Initialisierung des Algorithmus das FSSP?
- Erhält die verschärfte Regel DEFINE' die Vollständigkeit?
- Bleibt der Algorithmus vollständig, wenn ein Teil der Regelinstanzen deterministisch ausgewählt wird, indem zur geordneten Simulation übergegangen wird?

Aus Zeit- und Platzmangel wurden diese Beweise nicht formal duchgeführt, aber es wurde versucht die Gültigkeit dieser Aussagen zu begründen.

4.2 Fünf Zustände

Nachdem bewiesen ist, daß keine Lösung des FSSP mit vier Zuständen existiert und Lösungen mit sechs Zuständen bekannt sind, wäre es interessant herauszufinden, ob Lösungen mit fünf Zuständen existieren. Dadurch wäre das FSSP-Problem gewissermaßen vollständig geklärt. Aus einer ganzen Reihe von Gründen gelang es aber nicht, der Beantwortung dieser Frage näher zu kommen.

- Es gibt $5^{96} \approx 1.2 \cdot 10^{67}$ verschiedene Übergangsfunktionen. Das ist eine Zahl, die mehr als doppelt so *lang* ist, wie für vier Zustände (siehe Abschnitt 3.3).
- Einfache Heuristiken zur Beschneidung des Suchraums erweisen sich als weniger effektiv als Balzers (fehlerhafte) Ergebnisse erhoffen lassen.
- Alle Versuche intelligentere Algorithmen einzusetzen, um den Suchraum noch weiter einzuschränken, scheinen sich als kontraproduktiv zu erweisen. Der zusätzliche Verwaltungsaufwand ist größer als die Einsparung durch Verkleinerung des Suchraums. Dies gilt nicht nur für die in Abschnitt 3.6 beschriebene maximale Simulation, sondern auch für Balzers Versuche, durch Techniken wie symbolische Ausführung und „constraint propagation“ zum Ziel zu gelangen.

Um deutlich zu machen, wie chancenlos die bekannten Algorithmen sind, soll nun eine grobe Abschätzung der Laufzeit gemacht werden. Es wurde eine Programmversion gewählt, die für jede betrachtete Retinagröße die Anzahl der gefundenen Lösungen angibt. Dieses Programm wurde nun für verschiedene maximale Retinagrößen ($M \in \{3, 4, 5, 6, 7, 8, 15\}$) gestartet. Es gibt 8 Lösungen bis Größe

2 und 18 266 bis Größe 3.[2] Das Programm mit $M = 4$ wurde abgebrochen, nachdem es 91 394 756 Lösungen bis Größe 4 gefunden hatte, die aber nur auf 565 der möglichen Lösungen für $M = 3$ beruhen. Es ist deshalb zu erwarten, daß es ca. $91394756 \cdot 18266/565 \approx 2,9 \cdot 10^9$ Lösungen bis Größe 4 gibt. Die Programmläufe mit $M \in \{5,6,7,8\}$ ergaben in analoger Weise die weiteren Mulitplikationsfaktoren 362243150/503, 262898185/553, 164089505/1, 22694658/5 wobei zu bemerken ist, daß das Programm für $M = 7$ nicht über die erste Lösung für Größe 6 hinauskam, obwohl es ca. 16 Stunden auf einer SPARC-Station lief. Deshalb stellt die folgende Rechnung — bei aller Ungenauigkeit — wohl eher eine Abschätzung nach unten dar. Werden alle diese Faktoren multipliziert, ergibt sich ein Schätzwert von $7,5 \cdot 10^{35}$ Lösungen bis Größe 8. Das Programm mit $M = 15$ fand in 19 h auf einer SPARC 2 nur 532 727 von diesen Lösungen (und keine mit $M > 11$). Demnach würde das Durchsuchen des gesamten Suchraums ca $9,7 \cdot 10^{34}$s oder $3 \cdot 10^{17}$ mal das Alter des Universums benötigen.

4.3 Zusatzannahmen

Die Tatsache allein, daß der Suchraum für fünf Zustände zu groß ist, um ihn komplett zu durchsuchen, heißt noch nicht, daß es unmöglich ist, Suchalgorithmen einzusetzen, um das Problem zu lösen. Gibt es nämlich Lösungen, so reicht es, eine einzige davon zu finden. Sollte es sogar sehr viele Lösungen geben, so würde ein Monte-Carlo-Algorithmus in recht kurzer Zeit auf eine stoßen.

So einfach ist es aber wohl nicht, denn der parallele Algorithmus aus Abschnitt 10 durchsucht viele Zweige des Suchbaums gleichzeitig und hat innerhalb einer Laufzeit von einem Tag nur Lösungen bis Größe 14 gefunden.

Die nächste Möglichkeit besteht darin, Annahmen darüber zu machen, wie eine Lösung aussehen soll und den Suchraum auf solche Übergangsfunktionen zu beschränken, die zu diesen Annahmen konform sind. Auch diese Idee stammt von Balzer und wurde wieder aufgegriffen, weil die damaligen Ergebnisse wohl ebenfalls mit dem unvollständigen Suchalgorithmen erzielt wurden und weil es inzwischen andere Lösungen des FSSP gibt. Interessante Annahmen sind:

Vorfeuerzustand Gilt $\sigma(a,a,a) = \sigma(a,a,\#) = \sigma(\#,a,a) = \mathtt{F}$, und es gibt keine anderen Transitionen in den Feuerzustand, so wirkt a als *Vorfeuerzustand*. Interessant ist, daß alle bekannten Lösungen mit wenig Zuständen so etwas wie einen Vorfeuerzustand besitzen und daß es sich dabei um `G` handelt. (Ausnahmen kann es bei Randzellen und kleinen Retinagrößen geben.) Das gleiche gilt auch für die meisten Teillösungen mit vier Zuständen bis Retinagröße 8. Der Grund dafür liegt wohl darin, daß durch diese Regelung die Zahl der Einträge, die vorzeitiges Feuern auslösen können, minimiert wird.

Stabiler Zustand Oft sehen Algorithmen auch so aus, daß eine Zelle, die einmal den Vorfeuerzustand angenommen hat, ihn nicht mehr verläßt bevor gefeuert wird.

Symmetrie Balzer nennt eine Lösung „*image solution*“, wenn es eine bijektive Abbildung $I : A \to A$ gibt, mit $I \circ I = id$ und der Eigenschaft $\sigma(a,b,c) = d \Rightarrow \sigma(I(c), I(b), I(a)) = I(d)$. Diese Eigenschaft läßt sich

[2] Jede dieser Lösungen ist Spezialfall einer der Lösungen bis Große 3.

so veranschaulichen, daß eine Darstellung des Raumzeitdiagramms spiegelsymmetrisch wird, wenn Symbole für die einzelnen Zustände so gewählt werden, daß a gerade das Spiegelbild von $I(a)$ ist. Es scheint so zu sein, daß Lösungen, die als Grundstrategie die Retina fortlaufend halbieren, sich auf diese Form bringen lassen. Zum Beispiel ist Gerkens Lösung mit sieben Zuständen [6] eine „image solution". Mazoyers Lösung mit sechs Zuständen [10] ist jedoch inhärent asymmetrisch, und Mazoyer argumentiert ausdrücklich so, daß diese Asymmetrie Zustände spart, da nicht jedes Signal in 2 Versionen existieren muß. Was die Symmetrie interessant für Suchalgorithmen macht, ist die Tatsache, daß durch diese Annahme die Anzahl der zu bestimmenden Einträge der Überführungsfunktion beinahe halbiert wird und dadurch der Suchraum viel stärker eingeschränkt wird, als dies die anderen Heuristiken leisten können. Dadurch gelangt man zumindest in die Nähe des mit Suchalgorithmen Machbaren.

Vorgabe von Strategien Statt einfache Eigenschaften der Überführungsfunktion vorzugeben, kann auch versucht werden, die Lösungsstrategie selber vorzugeben. Es wurde z.B. versucht, Mazoyers Lösungsstrategie der rekursiven Teilung im Verhältnis 2:1 mit Hilfe entsprechender Vorgaben in *pattern* zu erzwingen. Dies führte jedoch zu keiner ausreichenden Einschränkung des Suchraums. Außerdem ist es nicht klar, wo genau die Vorgabe einer Grundstrategie aufhört und wo das bloße Kopieren eines spezifischen Algorithmus anfängt. (Zum Beispiel: Wie werden Rundungsprobleme und Ausnahmefälle gehandhabt?)

Die meisten der oben erwähnten Vorgaben konnten durch Vorabeintragung von Werten in σ und durch zusätzliche Einträge in *pattern* gemacht werden.[3] Nur für „image solutions" war eine leichte Modifikation des Algorithmus nötig.

Insgesamt hat die Idee Zusatzannahmen einzubauen zwar viele interessante Aspekte, führt aber zu keinen greifbaren Ergebnissen

[3] Tatsächlich ist dies der Grund, warum die Behandlung von Feuerzuständen nicht direkt in den Algorithmus eingebaut wurde.

Kapitel 5

Suche nach Trellisautomaten

Da die Suchalgorithmen für das FSSP einige recht spezielle Problemeigenschaften ausnutzen, kann leicht der Eindruck entstehen, daß Suchalgorithmen für andere Automatenmodelle völlig anders aussehen müßten. Es hat sich aber herausgestellt, daß die Grundstruktur der Algorithmen recht vielseitig verwendbar ist. Als Beispiel wurde ein offenes Problem angegangen, daß die Erkennungsmächtigkeit von *homogenen Trellisautomaten* betrifft.

5.1 Homogene Trellisautomaten

Ähnlich wie bei Zellularautomaten und dem FSSP sind auch Trellisautomaten erheblich allgemeiner definiert, als es für das vorliegende Problem nötig ist, und deshalb soll hier nur auf das Allernötigste eingegangen werden. Eine ausführliche Beschreibung findet sich in [20].

Homogene Trellisautomaten bestehen aus einer Menge gleichartiger Funktionselemente (*Knoten*), die entsprechend Abbildung 5.1 angeordnet und miteinander verbunden sind. Ein Knoten nimmt Elemente aus dem *Arbeitsalphabet* A von unten entgegen, verknüpft sie mittels der *Überführungsfunktion* $g : A \times A \to A$ und leitet das Ergebnis über seine beiden Ausgabeleitungen nach oben weiter.

Ein Trellisautomat kann zur Erkennung von Sprachen über einem Eingabealphabet $X \subseteq A$ eingesetzt werden, indem Worte w über X auf Schicht $|w|$ eingegeben werden und die Ausgabe des Funktionselements an der Spitze des Dreiecks betrachtet wird. (Numerierung der Schichten entsprechend Abbildung 5.1.) Das Eingabewort wird genau dann akzeptiert, wenn das Ausgabesymbol in der Menge

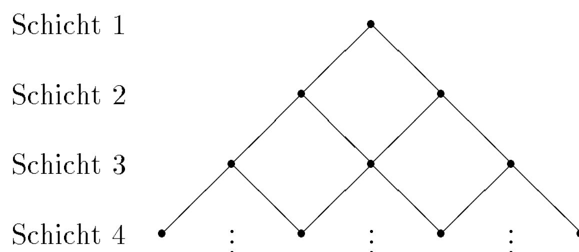

Abbildung 5.1: Trellisstruktur.

der *Aktzeptanzsymbole* $A_0 \subseteq A$ enthalten ist. Die Größe des Automaten wird jeweils der Eingabewortlänge angepaßt. Solchermaßen definierte Trellisautomaten sind äquivalent zu in Realzeit arbeitenden, unidirektionalen, eindimensionalen Zellularautomaten.

In [3] wurde gezeigt, daß Trellisautomaten neben sämtlichen regulären Sprachen auch eine ganze Reihe kontextfreier und sogar kontextsensitiver Sprachen erkennen können. Auffällig ist, daß zwar $\{ww^R|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ und $\{w@w|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ erkennbar sind, es aber ungeklärt ist, ob dies auch für $\{ww|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ gilt. Ziel der folgenden Abschnitte soll es deshalb sein, der Beantwortung dieser Frage mit Computerhilfe näherzukommen. Zumindest sollte es möglich sein, eine untere Schranke für die Größe des Arbeitsalphabets zu bestimmen, die nötig ist, um $\{ww^R|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ bzw. $\{ww|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ zu erkennen.

5.2 Anpassung des Suchalgorithmus

Zur Beantwortung der im letzten Abschnitt aufgeworfenen Frage wurde der in Abschnitt 3.5 beschriebene Algorithmus für geordnete Simulation auf Trellisautomaten umgestellt. Obwohl der Algorithmus nicht explizit auf Änderbarkeit hin geschrieben wurde, ließ sich doch fast die gesamte Struktur übernehmen. Insbesondere erwies es sich, daß die verwendeten Datenstrukturen und Strategien zwar uminterpretiert werden mußten, sich ihre gegenseitigen Beziehungen aber nicht geändert haben. Es war sogar ein großer Teil des Codes direkt wiederverwendbar, da die Datenstrukturen weitgehend als abstrakte Datentypen implementiert sind und deshalb oft nur die Zugriffsmakros geändert werden mußten. Dadurch wurde es möglich, innerhalb eines Tages einen recht effizienten Suchalgorithmus für Trellisautomaten zu konstruieren. Die Hauptarbeit steckte dann in den eigentlichen Experimenten und der Interpretation der Ergebnisse. Eine parallele Implementation wurde nicht durchgeführt, dürfte aber auch keine neuen Probleme aufwerfen.

Folgende Uminterpretationen von Begriffen mußten durchgeführt werden:

- An die Stelle der Zustandsmenge tritt das Arbeitsalphabet. Statt der besonderen Rolle von `F` sind akzeptierende und ablehnende Symbole zu unterscheiden.

- Statt der dreistelligen Überführungsfunktion σ ist die zweistellige Funktion g zu betrachten.

- Statt für jede Retinagröße muß für jedes mögliche Eingabewort simuliert werden. In $C^{|w|}(w)$ wird das Ergebnis der Simulation für das Eingabewort w gespeichert.

- Spalten- und zeilenweise Simulation wie bei Zellularautomaten ist nicht nötig.
  $C^s(xwy)$ ergibt sich nämlich unmittelbar aus $g(C^{s-1}(xw), C^{s-1}(wy))$. Dadurch bietet sich eine Simulationsordnung an, die bei Eingaben der Länge 2 beginnt und sich systematisch durch alle Wörter einer Länge hindurcharbeitet, bevor zur nächsten Eingabewortlänge übergegangen wird. Dadurch

ist es möglich, die komplette Simulation für ein Eingabewort durch 3 Tabellenzugriffe zu erledigen. Es handelt sich um ein typisches Beispiel für dynamische Programmierung.

- In *pattern* wird für jedes Eingabewort gespeichert, ob es akzeptiert werden soll oder nicht. Dadurch ist es im Prinzip möglich, nach Automaten für beliebige Sprachen zu suchen. Besonders einfach und effizient geht es jedoch, wenn nur zwei Eingabesymbole vorhanden sind, denn dann lassen sich Eingabeworte direkt als Bitstrings interpretieren, und Adreßrechnungen für C und *pattern* werden sehr schnell.
- Wie gehabt, wird beim Backtracking versucht, überflüssige Äste abzuschneiden. Allerdings ist der Begriff des Ereignishorizonts nicht mehr anwendbar.
- Beim Ausschluß isomorpher Lösungen müssen akzeptierende und ablehnende Symbole unterschieden werden.

5.3 Ergebnisse

Für die Sprache $\{ww^R|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ ist zwar eine Lösung (d.h. eine Belegung von g) bekannt, die recht einfach zu verstehen ist, aber diese benötigt soviele Arbeitssymbole (64 bei naiver Interpretation), daß es illusorisch wäre, sie mit dem oben beschriebenen Algorithmus finden zu wollen. Deshalb wurde zunächst nach einer Lösung für dieses Problem gesucht, die weniger Symbole benötigt. Während es keine Lösung mit 4 Symbolen gibt, wurden 16 Lösungen gefunden, die jeweils ein akzeptierendes und 4 ablehnende Symbole benutzen. Da der Algorithmus die Korrektheit dieser Lösungen „nur" bis zur Eingabelänge 20 untersucht[1] hat, wird die Allgemeingültigkeit einer dieser Lösungen im nächsten Abschnitt bewiesen. Außerdem sind die Übergangstabellen der Lösungen in Anhang A angegeben.

Anschließend wurde nach einer Lösung für $\{ww|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ gesucht: Nach einigen Stunden Rechenzeit auf einer SPARC-Station war geklärt, daß es keine Lösung mit 6 Symbolen gibt, die mehr als ein akzeptierendes Symbol besitzt. Die Suche nach einer Lösung mit 5 ablehnenden und einem akzeptierenden Symbol wurde nach einiger Zeit abgebrochen, als klar wurde, daß eine solche Lösung ohnehin nicht existieren kann:

Satz 3 *Es gibt keinen homogenen Trellisautomaten mit einem einzigen akzeptierenden Symbol, der* $L = \{ww|w \in \{\mathtt{a},\mathtt{b}\}^+\}$ *erkennt.*

Beweis: Man betrachte die Eingabeworte $w_1 = \mathtt{aaaaaa}$ und $w_2 = \mathtt{ababab}$. Weiterhin sei $\mathtt{x}$ das einzige akzeptierende Symbol. Alle Teilworte der Länge 4 von w_1 und w_2 sind in L. Folglich transformiert der Trellisautomat w_1 und w_2 in drei Schritten in das Wort $\mathtt{xxx}$ (siehe auch Abbildung 5.2). w_1 und w_2 sind also nicht unterscheidbar. Dies steht im Widerspruch zur Tatsache, daß $w_1 \in L$ und $w_2 \notin L$.
□

Die Suche nach Lösungen mit 7 Symbolen ergab, daß es keine Lösung mit mehr als 2 ablehnenden Symbolen gibt. Mit Hilfe von Satz 3 folgt, daß einziger

[1] Dem entsprechen 2 097 150 Eingabeworte.

```
      x              x   Fehler
     ? ?            ? ?
    x x x          x x x   ab hier nicht mehr unterscheidbar
   ? ? ? ?        ? ? ? ?
  x x x x x      ? ? ? ? ?
 a a a a a a    a b a b a b
```

Abbildung 5.2: Behandlung von aaaaaa und ababab wenn x einziges akzeptierendes Symbol ist.

Kandidat für eine Lösung mit 7 Symbolen die Kombination 5 ablehnende und 2 akzeptierende Symbole ist. Drei Monate (!) Rechenzeit auf einer SPARC fanden keine Lösung für diese Variante, konnten den Suchraum aber auch nicht ausschöpfen.

5.4 Beweis einer Lösung für ww^R

Satz 4 *Der homogene Trellisautomat T mit Eingabesymbolen* $\{\mathtt{a}, \mathtt{b}\}$, *dem einzigen akzeptierenden Symbol* x, *dem Arbeitsalphabet* $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{x}\}$ *und einer Übergangsfunktion g entsprechend Abbildung 5.3 akzeptiert genau die Sprache* $L = \{ww^R | w \in \{\mathtt{a}, \mathtt{b}\}^+\}$

	a	b	c	d	x
a	x	c	c	x	
b	c	x	x	c	
c	d	c	c	d	a
d	c	d	d	c	b
x			d	c	b

Abbildung 5.3: Übergangsmatrix einer Lösung für $\{ww^R\}$.

5.4.1 Transformation des Problems

Lemma 5 *Die Ausgabe der Knoten auf einer Schicht, die eine gerade Zahl von Schichten von der Eingabeschicht entfernt ist (kurz* gerade Schicht*), ist ein Wort über* $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, *und die Ausgabe der Knoten auf einer ungeraden Schicht ist ein Wort über* $\{\mathtt{c}, \mathtt{d}, \mathtt{x}\}$.

Beweis: (Induktiv über die Entfernung von der Eingabeschicht n.)

$n = 0$ Die Eingabeschicht gibt per definitionem ein Wort über $\{\mathtt{a}, \mathtt{b}\} \subseteq \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ aus.

$n \longrightarrow n + 1$

```
          x
         b c
        d x d
       c a d c
      c c x c c
     c c b c c b
    x d d x d d x
   a d c a d c a d
  c x c c x c c x c
 a b b a b b a b b a
```

Abbildung 5.4: Erkennung von `abbabbabba`.

1: n **gerade** Durch Inspektion der Übergangsfunktion ergibt sich, daß ein Wort über $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$ auf ein Wort über $\{\mathtt{c},\mathtt{d},\mathtt{x}\}$ abgebildet wird. (Ein `a` oder `b` wird nur ausgegeben, wenn mindestens ein `x` eingegeben wird.)

2: n **ungerade** Durch Inspektion der Übergangsfunktion ergibt sich, daß ein Wort über $\{\mathtt{c},\mathtt{d},\mathtt{x}\}$ auf ein Wort über $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$ abgebildet wird. (Ein `x` wird nur ausgegeben, wenn mindestens ein `a` oder `b` eingegeben wird.)

□

Korollar 6 *Da bei Eingabe eines Wortes ungerader Länge der Knoten an der Spitze des Trellisautomaten eine gerade Zahl von Schichten von der Eingabeschicht entfernt ist, werden Worte ungerader Länge (die nie die Form ww^R haben) nicht akzeptiert.*

Im folgenden kann deshalb immer von einer geraden Eingabewortlänge ausgegangen werden.

Entscheidend für die Fortführung des Beweises ist nun die „Wegrationalisierung“ der geraden Schichten des Trellisautomaten, die das Problem erheblich vereinfacht, da dadurch von bestimmten Details der Informationscodierung abstrahiert werden kann. Zu diesem Zweck wird eine neue Übergangstabelle definiert, die Tripeln über $\{\mathtt{c},\mathtt{d},\mathtt{x}\}$ ein Ergebnissymbol aus $\{\mathtt{c},\mathtt{d},\mathtt{x}\}$ zuordnet, indem der Übergang von einem Tripel auf ein Paar über $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$ und der nachfolgende Übergang auf ein Symbol aus $\{\mathtt{c},\mathtt{d},\mathtt{x}\}$ zu einem Schritt zusammengefaßt werden. Ergebnis ist eine neue Trellisstruktur mit 3 Eingaben (siehe auch Abbildung 5.5) und einer Übergangsfunktion g', die in Abbildung 5.6 dargestellt ist.

Diese Trellisstruktur ist äquivalent zu einem in $\frac{1}{2}$-Realzeit arbeitenden, bidirektionalen, eindimensionalen Zellularautomaten ungerader Retinagröße, bei dem die mittlere Zelle über die Akzeption entscheidet.

Die Eingabe der abstrahierten Trellisstruktur ist ein Wort über $\{\mathtt{c},\mathtt{x}\}$ wobei jeweils ein `c` steht, wenn die Eingaben der ursprünglichen Trellisstruktur ungleich waren (`ab` oder `ba`) und ein `x`, wenn die Eingaben gleich waren. Daraus ergibt sich unmittelbar das folgende Lemma:

Lemma 7 *Genau dann, wenn die ursprüngliche Eingabe die Form ww^R mit $w \in \{\mathtt{a},\mathtt{b}\}^+$ hat, erhält die abstrahierte Trellisstruktur eine Eingabe der Form $w'\mathtt{x}w'^R$ mit $w' \in \{\mathtt{c},\mathtt{x}\}^*$.*

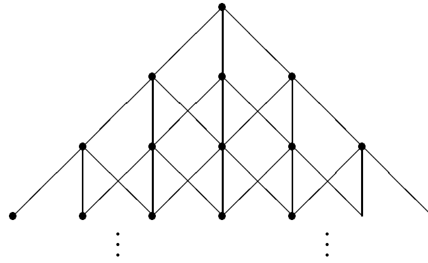

Abbildung 5.5: Abstrahierte Trellisstruktur.

c	c	d	x	d	c	d	x	x	c	d	x
c	c	d	d	c	c	d	d	c	x	c	c
d	d	c	c	d	d	c	c	d	c	x	(x)
x	d	c	c	x	d	c	(c)	x	c	(x)	x

Abbildung 5.6: Übergangsfunktion g' des abstrahierten Trellisautomaten.

Es bleibt also zu zeigen, daß die abstrahierte Trellisstruktur genau die Wörter der Form $w'\mathtt{x}w'^R$ erkennt.

5.4.2 Eigenschaften der abstrahierten Übergangsfunktion

Anhand von Tabelle 5.6 lassen sich einige interessante Details der Übergangsfunktion ablesen:

Lemma 8 *g' ist spiegelsymmetrisch d.h. $g'(u,v,w) = z \Rightarrow g'(w,v,u) = z$.*

Lemma 9 *Ein* `x` *kann nur ausgegeben werden, wenn vorher ein* `x` *in der Mitte eingegeben wurde.*

Mit Hilfe dieser Aussage lassen sich einige Übergänge ausschließen:

Lemma 10 *Die Tripel* `dxx`*,* `xxd` *und* `xdx` *können nicht als Eingabe von g' vorkommen.*

Beweis: (Induktiv über den Abstand n von der Eingabeschicht m)

$n = 0$: Auf der untersten Schicht können per Konstruktion der abstrahierten Trellisstruktur keine `d`'s vorkommen.

$n \longrightarrow n+1$: Angenommen in Schicht $m-(n+1)$ komme `dxx` vor. Mit Hilfe von Lemma 9 und durch Inspektion der Übergangsfunktion ergibt sich, daß dieses Tripel nur entstehen kann, wenn unmittelbar darunter bereits `dxx` stand, was im Widerspruch zur Induktionsvoraussetzung steht. Abbildung 5.7 illustriert die zu dieser Aussage führende Schlußkette. Wegen der Symmetrie der Übergangsfunktion gilt eine analoge Aussage für das Tripel

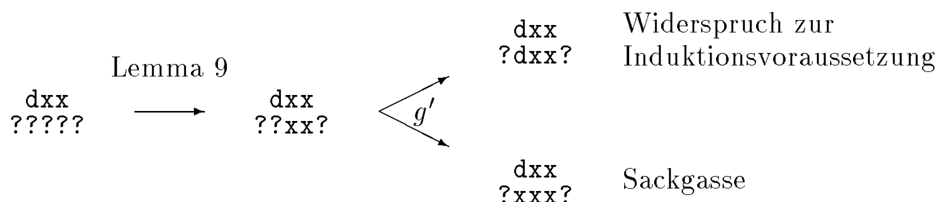

Abbildung 5.7: Schlußkette zum Beweis der Unmöglichkeit von `dxx`.

`xxd`. Noch leichter führt die Annahme, daß `xdx` auf Schicht $m-(n+1)$ vorkommt, zu einem Widerspruch, da überhaupt kein String existiert, aus dem dieses Tripel hervorgehen könnte. (Dies ist wiederum unter Benutzung von Lemma 9 einzusehen.)

□

Aus den Lemmata 9 und 10 sowie durch Inspektion der Übergangsfunktion ergibt sich die folgende wichtige Aussage:

Lemma 11 $g'(u,v,w) = \mathtt{x} \Leftrightarrow v = \mathtt{x} \wedge u = w$

Schließlich kann aus Lemma 11 und der Symmetrie der Übergangsfunktion gefolgert werden:

Lemma 12 *Ein Wort der Form $w\mathtt{x}w^R$ mit $w \in \{c,d,x\}^+$ wird durch den abstrahierten Trellisautomaten in ein Wort der Form $w'\mathtt{x}w'^R$ mit $w' \in \{c,d,x\}^*$ umgeformt.*

5.4.3 Vollständigkeit der Lösung

Aus den im letzten Abschnitt hergeleiteten Aussagen läßt sich nun leicht folgern, daß Wörter der Form $w'\mathtt{x}w'^R$ (und damit auch Wörter der Form ww^R vom ursprünglichen Automaten) akzeptiert werden, da sie nach Lemma 12 in jeder Schicht in ein Wort der gleichen Form transformiert werden. Folglich bleibt im letzten Schritt nur noch das akzeptierende Symbol `x` übrig.
□

5.4.4 Korrektheit der Lösung

Zu zeigen bleibt, daß Wörter, die nicht die Form $w\mathtt{x}w^R$ haben, nicht akzeptiert werden. Entscheidend für die Beantwortung dieser Frage ist das folgende Lemma:

Lemma 13 *Gegeben sei ein Wort $u = yw\mathtt{x}w^Rz$ mit $|w| > 0$ sowie $y,z \in \{\mathtt{c},\mathtt{d},\mathtt{x}\}$, $y \neq z$. Dann wird u im nächsten Schritt auf ein Wort der Form $u' = y'w'\mathtt{x}w'^Rz'$ mit $|w'| = |w| - 1$, $y',z' \in \{\mathtt{c},\mathtt{d},\mathtt{x}\}$, $y' \neq z'$ abgebildet.*

Daß der Mittelteil $w\mathtt{x}w^R$ wie behauptet transformiert wird, folgt aus Lemma 12. Es geht nun noch darum, zu zeigen, daß ein Unterschied zwischen den beiden Randzeichen erhalten bleibt. Zum Beweis wird eine Fallunterscheidung nach den ersten beiden Zeichen von w (die auch die letzten beiden Zeichen von w^R sind) und nach y und z gemacht:

$w = \mathtt{cc}v$

$yz = \mathtt{cd}$: $y' = g'(\mathtt{c},\mathtt{c},\mathtt{c}) = \mathtt{c} \neq \mathtt{d} = g'(\mathtt{c},\mathtt{c},\mathtt{d}) = z' \surd$

$yz = \mathtt{cx}$: $y' = g'(\mathtt{c},\mathtt{c},\mathtt{c}) = \mathtt{c} \neq \mathtt{d} = g'(\mathtt{c},\mathtt{c},\mathtt{x}) = z' \surd$

$yz = \mathtt{dx}$: $y' = g'(\mathtt{d},\mathtt{c},\mathtt{c}) = \mathtt{d} = g'(\mathtt{c},\mathtt{c},\mathtt{x}) = z'$
Hier bliebe der Unterschied nicht erhalten. Deshalb ist zu zeigen, daß diese Situation nicht vorkommen kann.

Restliche Fälle: Symmetrisch zu einem der obigen 3 Fälle.

$w = \mathtt{cd}v$, $w = \mathtt{dd}v$, $w\mathtt{x} = \mathtt{cx}v$, $w\mathtt{x} = \mathtt{dx}v$, $w\mathtt{x} = \mathtt{xx}v$: Analog zum Fall $w = \mathtt{cc}v$ geht der Unterschied für $yz = \mathtt{dx}$ verloren, während alle anderen Fälle unproblematisch sind.

Restliche Fälle: Symmetrisch zu einem der obigen Fälle.

Zum Beweis von Lemma 13 bleibt es also zu zeigen, daß die verbleibenden kritischen Fälle nicht vorkommen können. Allen gemeinsam ist, daß auf einer Seite ein x und auf der anderen ein d außen steht. Da d's nicht in der Ausgangskonfiguration vorkommen, muß es einen Vorgängerschritt gegeben haben. Nach Lemma 9 muß unter dem x ein x eingegeben worden sein. Da auch in der vorhergehenden Schicht Unterschiede nur an den äußersten Buchstaben auftreten können, muß unter dem d ebenfalls ein x gestanden haben.[2]

$$\begin{array}{c}\mathtt{x}\cdots\mathtt{d}\\ \mathtt{??}\cdots\mathtt{??}\end{array} \Rightarrow \begin{array}{c}\mathtt{x}\cdots\mathtt{d}\\ \mathtt{?x}\cdots\mathtt{??}\end{array} \Rightarrow \begin{array}{c}\mathtt{x}\cdots\mathtt{d}\\ \mathtt{?x}\cdots\mathtt{x?}\end{array}$$

Durch Inspektion der Übergangstabelle ergibt sich aber, daß aus einem x nie ein d entstehen kann, was die Annahme, daß einer der problematischen Fälle auftreten kann, zum Widerspruch führt.

□

Der Beweis der Korrektheit der Lösung ist nun relativ einfach:
Annahme: Es gibt ein Eingabewort u des abstrahierten Automaten, das nicht die Form $w\mathtt{x}w^R$ hat und akzeptiert wird.

1. **u hat kein x in der Mitte** Nach Lemma 11 kann dann an der Automatenspitze auch kein x stehen (Widerspruch).

2. **u hat die Form $vyw\mathtt{x}w^Rzv'$ mit $y \neq z$, $|v| = |v'|$** Nach Lemma 11 muß das Teilwort $yw\mathtt{x}w^Rz$ zu einem x reduziert werden — also ebenfalls akzeptiert werden. Induktion über $|w|$ zeigt, daß dies zum Widerspruch führt:

   $|w| = 0$: Da nach Lemma 11 x nur für symmetrische Eingabetripel erhalten bleibt, ergibt sich ein Widerspruch.

   $|w| - 1 \longrightarrow |w|$: Nach Lemma 13 wird $yw\mathtt{x}w^Rz$ zu einem Wort der Form $y'w'\mathtt{x}w'^Rz'$ mit $y' \neq z'$ abgeleitet und dieses dann akzeptiert, was im Widerspruch zur Induktionsvoraussetzung steht.

□

[2] Streng genommen müßte hier wohl ein Induktionsbeweis benutzt werden.

Kapitel 6

Einführung in parallele Suchalgorithmen

6.1 Eingrenzung der Fragestellung

Viele Probleme in der Informatik lassen sich als Traversierung eines Suchbaums formulieren. Da gerade diese Art von Algorithmen sehr zeitaufwendig sein kann, liegt es nahe, die Suche einem Parallelrechner zu übertragen. Allerdings gibt es viele Varianten von Suchalgorithmen, und auch Parallelrechner können sich in vielen Aspekten unterscheiden. Deshalb soll zunächst eine Eingrenzung der Fragestellung stattfinden. In Kapitel 9 wird sich dann herausstellen, daß viele der Ergebnisse sich in viel allgemeinerer Weise anwenden lassen.

- Der Algorithmus soll für wirklich massiv parallele Rechner mit vielen tausend Prozessoren geeignet sein. Damit in dieser Situation keine Flaschenhälse auftreten, muß der sequentielle Anteil des Algorithmus mit der Problemgröße gegen Null gehen.

- Ausgegangen wird von einem nachrichtengekoppelten SIMD-Rechner, dessen Prozessoren ihren Speicher individuell adressieren können. Diese Arbeit wird die für die Rechner der Firma MasPar[1] übliche Terminologie benutzen. Bis auf die oben gemachten Voraussetzungen sind die Ergebnisse aber weitgehend unabhängig von der spezifischen Architektur.

- Alle Blätter des Suchbaums sind voneinander unabhängig. Es interessiert nur, ob ein gegebenes Blatt eine Lösung darstellt. Insbesondere werden damit solche Probleme ausgeschlossen, bei denen ein Wert für innere Knoten aus den Werten seiner Nachfolgeknoten berechnet wird. (Zum Beispiel Suchen des maximalen Blattes oder Minimax-Suche.)

- Es wird davon ausgegangen, daß Tiefensuche eine adäquate Suchstrategie für den sequentiellen Fall darstellt.

Abbildung 6.1 stellt das Grundschema des Algorithmus dar, von dem im folgenden ausgegangen wird. Die Prozedur `search` gibt alle existierenden Lösungen aus, wenn ihr die Wurzel des Suchbaums übergeben wird. Blätter des Suchbaums werden daraufhin überprüft, ob sie eine Lösung darstellen. Handelt es sich um

[1] MasPar MP-1, MasPar MP-2, DEC MPP

```
PROCEDURE search(node)
IF terminalNode(node) THEN
    IF isSolution(node) THEN printSolution(node)
ELSE
    FOR each n ∈ successors(node) DO search(n)
```

Abbildung 6.1: Allgemeiner Suchalgorithmus.

einen inneren Knoten, so werden dessen Nachfolger berechnet, und für jeden dieser Nachfolger wird rekursiv die Suchprozedur aufgerufen.

Aus der „Programmierfolklore“ bekannte Beispiele für diese einfache Art der Suche sind das n-Damenproblem oder eine einfache rekursive Labyrinthsuche. In [15] wird parallele Suche nach Lösungen des 15-Puzzles für verschiedene MIMD-Architekturen untersucht.

Die in dieser Arbeit betrachtete Suche nach Lösungen des FSSP enthält zusätzlich einige Heuristiken zum Abschneiden überflüssiger Suchbaumäste. Außerdem werden gewisse Informationen aus Effizienzgründen nicht in node sondern in globalen Datenstrukturen gespeichert. Trotzdem ist das oben angegebene Schema ein hinreichend genaues Modell für die im folgenden untersuchten Fragestellungen.

6.2 Grundidee für die Parallelisierung

Der hier gewählte Ansatz besteht darin, einen möglichst grobkörnigen Parallelismus zu implementieren. Deshalb wird jedem Einzelprozessor ein eigener Teilbaum des Suchraums zugeteilt, in dem er dann unabhängig von den anderen nach Lösungen sucht. Daraus ergeben sich drei Hauptaufgaben:

1. Es gilt die Suche so zu initialisieren, daß Teilprobleme möglichst gleicher Größe auf die Prozessoren verteilt werden. Um die Arbeitsverteilung nicht zu einem Engpaß werden zu lassen, sollte sie möglichst ebenfalls parallel ablaufen. (Siehe Abschnitt 8.1).

2. Die Prozessoren sollen in die Lage versetzt werden, unabhängig voneinander den ihnen zugeordneten Teilbaum zu durchsuchen. Die Suche innerhalb des Teilbaums entspricht dabei weitgehend dem sequentiellen Algorithmus. Auf einem SIMD-Rechner ist dies ein keineswegs triviales Problem, da ein einziger Befehlsstrom all die asynchron ablaufenden Suchvorgänge steuern muß (Kapitel 7). In [4] wird deshalb parallele Suche mit Hilfe eines sehr feinkörnigen Parallelismus vollständig synchron formuliert. Allerdings entsteht dadurch ein so großer Kommunikationsaufwand, daß der verfolgte Ansatz nicht sehr vielversprechend erscheint.

3. Im allgemeinen wird die in Punkt 1 beschriebene Arbeitsverteilung nicht perfekt sein und einige Prozessoren werden einen ungleich größeren Teil der Arbeit zugeteilt bekommen als andere. Um einen effizienten Algorithmus zu erhalten, muß deshalb gelegentlich die Arbeit umverteilt werden, indem

vielbeschäftigte Prozessoren einen Teil ihres Suchraums auf bereits fertige Prozessoren „abwälzen“ (Abschnitt 8.2).

Kapitel 7

SIMD-Realisierung des Kontrollflusses

Der in Abbildung 6.1 angegebene rekursive Algorithmus läßt sich (mit heutigen Programmiersprachen) nicht direkt auf SIMD-Rechnern einsetzen. Deshalb wird in Abschnitt 7.1 dargestellt, wie ein auf einem Keller agierender endlicher Automat konstruiert werden kann, der das Suchproblem löst und von einem SIMD-Rechner simuliert werden kann. Da diese Simulation einen erheblichen Overhead darstellt, werden in den Abschnitten 7.2 bis 7.4 verschiedene Ansätze diskutiert, den Overhead zu reduzieren.

7.1 Die Grundsätzliche Realisierung

7.1.1 Ein nichtrekursiver Suchalgorithmus

Da Algorithmus 6.1 rekursiv ist und außerdem die Funktion `successors` i.allg. auf verschiedenen Prozessoren unterschiedlich viele Nachfolgeknoten zurückliefert, ist er nicht direkt für einen SIMD-Rechner geeignet.

Diese Probleme sind in der nichtrekursiven Formulierung in Abbildung 7.1 ausgeräumt. An die Stelle der rekursiven Aufrufe tritt ein explizit verwalteter Keller, der hier mit Hilfe der Funktionen `empty()`, `push(node)`, `top()` und `pop()` verwaltet wird. Oberstes Kellerelement ist immer der aktuelle Suchbaumknoten. Ist ein Blatt abgearbeitet, so werden zunächst Kellereinträge entfernt, die keine weiteren Geschwisterknoten haben. Dann wird das oberste Kellerelement durch den nächsten Geschwisterknoten ersetzt. Ein innerer Knoten wird bearbeitet, indem sein erster Nachfolger auf dem Keller abgelegt wird. Die weiteren Nachfolger werden dann — wie oben beschrieben — im weiteren Verlauf der Suche durch die Funktion `nextSibling` erzeugt.

7.1.2 Ein endlicher Automat als Kontrollstruktur

Die **WHILE**-Schleifen in Algorithmus 7.1 stehen einer effizienten SIMD-Implementierung noch im Wege, da die Anzahl der Schleifendurchläufe von Prozessor zu Prozessor variieren kann. Dieses Problem läßt sich vermeiden, indem die Kontrollstruktur durch den in Abbildung 7.2 angegebenen endlichen Automaten realisiert wird. Eingabe des Automaten ist `top()` — das oberste Kellerelement.

PROCEDURE search(node)
initStack(); push(node)
LOOP
  IF terminalNode(top()) **THEN**
    IF isSolution(top()) **THEN** printSolution(top())
    WHILE noMoreSiblings(top()) **DO**
      pop()
      IF empty() **THEN RETURN**
    top() := nextSibling(top())
  ELSE push(firstSucessor(top()))

Abbildung 7.1: Nichtrekursiver Suchalgorithmus.

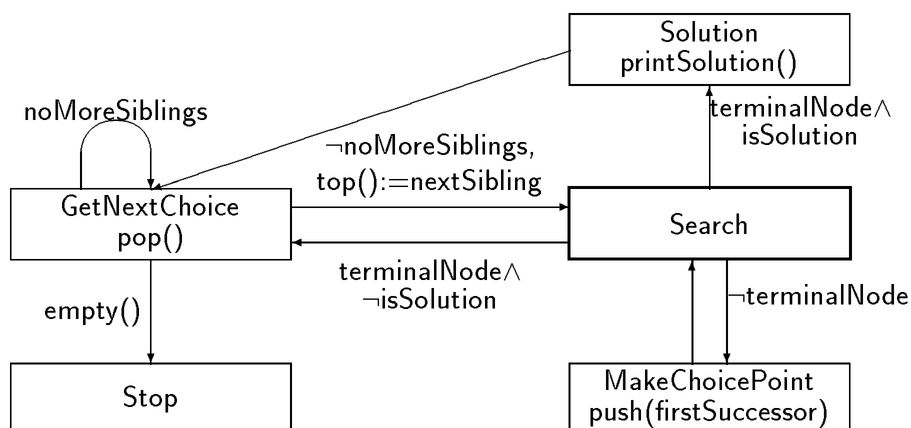

Abbildung 7.2: Suchalgorithmus als Pushdownautomat.

(Das Gesamtsystem könnte auch als Pushdownautomat aufgefaßt werden.) Sowohl mit Zuständen als auch mit Übergängen können Kellermanipulationen assoziiert werden. Der Automat legt das gleiche Verhalten an den Tag wie Algorithmus 7.1. Dabei sind den einzelnen Zuständen die folgenden Rollen zugeordnet:

Search Auf dem Keller liegt ein Suchbaumknoten obenauf, über den a priori nichts bekannt ist. Der Knoten wird untersucht, und abhängig von dessen Eigenschaften wird zum passenden Nachfolgezustand verzweigt.[1]

MakeChoicePoint Ein innerer Knoten liegt vor. Schiebe seinen ersten Nachfolger auf den Keller.

GetNextChoice Entferne alle Knoten vom Keller, die letzter Nachfolger ihres Vorgängers sind. Wird der Keller dabei leer, so ist die Suche beendet. Ersetze schließlich den aktuellen Knoten durch seinen nächsten Geschwisterknoten.

[1] Dieser Zustand ließe sich im Prinzip wegrationalisieren. Das Verhalten der anderen Zustände würde dadurch aber komplizierter. Außerdem gibt es keinen anderen Zustand, der sonst ohne weiteres als Startzustand dienen könnte.

Solution Es wurde eine Lösung gefunden. Gib sie aus, und geh zum nächsten Zweig des Suchbaums.

Stop Die Suche ist beendet.

Der gerade beschriebene Automat kann durch das in Abbildung 7.3 beschriebene Programm simuliert werden.

```
PROCEDURE search(node)
initStack(); push(node)
state := Search
WHILE state ≠ Stop DO
    IF state = Search THEN
        IF terminalNode(top()) THEN
            IF isSolution(top()) THEN state := Solution
            ELSE state := GetNextChoice
        ELSE state := MakeChoicePoint
    IF state = MakeChoicePoint THEN
        push(firstSuccessor(top())); state := Search
    IF state = GetNextChoice THEN
        pop()
        IF empty() THEN state := Stop
        ELSIF noMoreSiblings(top()) THEN state := GetNextChoice
        ELSE top() := nextSibling(top()); state := Search
    IF state = Solution THEN printSolution(top()); state := GetNextChoice
```

Abbildung 7.3: Suchalgorithmus, der Automaten simuliert.

7.1.3 Die Rolle der problemabhängigen Funktionen

Das einzige, was einer SIMD-Realisierung jetzt noch im Wege stehen kann, sind die problemabhängigen Funktionen terminalNode, isSolution, firstSuccessor, nextSibling und printSolution. Im einfachsten Fall enthalten sie keine von lokalen Daten abhängigen inneren Schleifen. Dann läßt sich Algorithmus 7.3 direkt in einer datenparallelen Sprache wie MPL formulieren. Andernfalls müssen die zusätzlichen Schleifen aufgelöst werden. Dabei werden i.allg. zusätzliche Zustände entstehen, aber am Grundprinzip der Implementierung wird sich nichts ändern. Kapitel 10 führt dies exemplarisch für das FSSP vor. Es zeigt sich, daß die Idee eines endlichen Automaten als Kontrollstruktur für komplexere Probleme nicht immer ideal ist. Es ist jedoch immer möglich, den Algorithmus auf eine Form zu bringen, die aus einer einzigen Schleife besteht, die eine Kette bedingter Anweisungen enthält, innerhalb derer keine weiteren Schleifen enthalten sind.

Diese Aussage folgt unmittelbar aus dem Beweis für folgenden Satz:

Satz 14 *Ein SIMD-Rechner kann einen MIMD-Rechner mit konstantem Overhead emulieren.*

Beweisskizze: Das gesamte Programm und der Zustand jedes zu emulierenden Prozessors werden im lokalen Speicher der PE-s abgelegt. Nun wird ein

LOOP
  IF b_1 **THEN** o_1
  IF b_2 **THEN** o_2
  $\vdots$
  IF b_n **THEN** o_n

Abbildung 7.4: Hauptschleife mit n Elementaroperationen.

Interpreter für den Befehlssatz des MIMD-Rechners geschrieben, der die folgende grundsätzliche Form hat:

```
for(;;){...
   if(OpCode(*PC) == JUMP){PC = Operand(*PC);}
   ...
   if(OpCode(*PC) == LOADIM){Accu = Operand(*PC); PC++;}
   ...
}
```

Da der Maschinenbefehlssatz nur endlich viele Opcodes enthält, ergibt sich nur ein konstanter Overhead.
□

Das Interessante an diesem Beweis ist, daß er nicht so praxisfern ist, wie es auf den ersten Blick erscheinen mag. Niemand verlangt nämlich, den Befehlssatz einer realen Maschine zu emulieren. Nichts spricht dagegen, spezifisch für den zu realisierenden Algorithmus einen Befehlssatz zu entwerfen, der möglichst wenige aber trotzdem mächtige Befehle enthält. Algorithmus 7.3 läßt sich zum Beispiel als Interpreter für einen Befehlssatz mit 4 sehr speziellen Befehlen auffassen. Der Ansatz, die Kontrollstruktur als endlichen Automaten darzustellen, läßt sich also als Spezialfall des Interpreteransatzes auffassen.

7.2 Ansätze zur SIMD-Optimierung

Der Rest dieses Kapitels ist nicht von den spezifischen Eigenschaften von Suchalgorithmen abhängig. Vielmehr sind die hier beschriebenen Techniken für alle Algorithmen anwendbar, die die in Abschnitt 7.1 eingeführte Implementierungsstrategie verwenden. Genauer gesagt werden Programme untersucht, die die in Abbildung 7.4 dargestellte Form haben. Die o_i sollen im folgenden *Elementaroperationen* genannt werden. Das Programmstück

$$a_i = \textbf{IF} \;\; b_i \; \textbf{THEN} \;\; o_i$$

wird *Abfrage* der Elementaroperation o_i genannt. Die Ausführungszeit für a_i werde mit c_i bezeichnet.

Es wird davon ausgegangen, daß c_i eine Konstante ist. Auf einem SIMD-Rechner ist diese Annahme im allgemeinen gerechtfertigt, wenn die o_i keine inneren Schleifen mehr enthalten. Allerdings gibt es Ausnahmen. Zum Beispiel wird o_i auf der MasPar überhaupt nicht ausgeführt, wenn b_i auf keinem Prozessor erfüllt ist. Außerdem kann der Zeitaufwand für Bibliotheksaufrufe (wie z.B. komplexe Kommunikationsbefehle) variieren.

Eine weitere wichtige Annahme ist die Unabhängigkeit der Abfragen. Es wird verlangt, daß die Semantik des Programms sich nicht ändert, wenn Abfragen mehrfach gemacht werden oder ihre Reihenfolge permutiert wird. Die *Abfragefolge* $[1, \ldots, n]$ kann also ohne Änderung der Semantik des Programms durch eine Abfragefolge $[l_1, \ldots, l_m]$ ersetzt werden, wenn alle Indizes zwischen 1 und n mindestens einmal in $[l_k]$ vorkommen. Da der im Beweis zu Satz 14 skizzierte Interpreteransatz diese Eigenschaft erfüllt, tut diese Annahme der Allgemeinverwendbarkeit der hier entwickelten Verfahren keinen Abbruch.

Aus der Unabhängigkeit der Abfragen ergeben sich die Grundideen für eine Optimierung der Ausführung:

- Häufig benötigte bzw. billige Elementaroperationen sollten häufiger abgefragt werden als seltene, teure Operationen. Zum Beispiel dürfte es sinnvoll sein, die Operation Solution in Algorithmus 7.3 seltener abzufragen als die anderen Operationen, wenn zu erwarten ist, daß nur wenige Lösungen gefunden werden.

- Operationen, die meist direkt aufeinanderfolgen, sollten möglichst nah beieinander abgefragt werden. Falls der Suchbaum einen hohen Verzweigungsgrad hat, wäre z.B. in Algorithmus 7.3 die Reihenfolge „Search, Solution, GetNextChoice" günstig.

Sollen mit Hilfe der obigen Optimierungsideen gute Ergebnisse erzielt werden, ergeben sich schnell so viele abzuwägende Faktoren, daß eine sehr aufwendige Ausprobiererei droht. Deshalb sollen in den folgenden beiden Abschnitten mathematische Modelle entwickelt werden, mit deren Hilfe sich die Optimierungen zielgerichteter anwenden lassen.

7.3 Optimierung der Abfragehäufigkeit

In diesem Abschnitt sollen Fragen der Reihenfolge von Elementaroperationen zunächst außer Acht gelassen werden. Dadurch wird es möglich, ein analytisches Modell aufzustellen, mit dem sich optimale Abfragehäufigkeiten ausrechnen lassen.

7.3.1 Der Ansatz

Sei p_i die mittlere Wahrscheinlichkeit, daß ein Prozessor als nächste Elementaroperation o_i ausführen möchte. Ferner sei $\sum_{i=1}^{n} p_i = 1$. Es wird also angenommen, daß die Bedingungen b_i disjunkt[2] sind. Dadurch vereinfachen sich die folgenden Berechnungen. Die p_i lassen sich mit Hilfe von Beispieldatensätzen messen oder auf Grund des Programmiererwissens über das Problem abschätzen. Außerdem soll angenommen werden, daß die p_i nicht zeitabhängig[3] sind.

Von der konkreten Abfragereihenfolge wird nun abstrahiert, indem von einer Hauptschleife ausgegangen wird, die die nächste Abfrage auswürfelt (siehe Abbildung 7.5). Die Wahrscheinlichkeit, mit der ein i gewürfelt wird, sei mit f_i

[2] Es ist ohnehin fraglich, ob es sinnvolle Gegenbeispiele gibt, bei denen die Abfragen die Forderung nach Unabhängigkeit trotzdem erfüllen.

[3] Sollten die p_i stark zeitabhängig sein, so ist es z.B. möglich, den Algorithmus (statisch oder dynamisch) in mehrere Phasen aufzuteilen, innerhalb derer die p_i hinreichend konstant sind.

LOOP
randomly determine $i \in \{1..n\}$
IF b_i **THEN** o_i

Abbildung 7.5: Probabilistische Abfragereihenfolge.

bezeichnet. Diese f_i gilt es nun in Abhängigkeit von den p_i und c_i so zu bestimmen, daß die erwartete Ausführungszeit minimal wird.

7.3.2 Berechnung der Abfragehäufigkeiten

Die Berechnung der optimalen f_i wird nun mit Hilfe der folgenden Schritte bewerkstelligt:

1. Aufstellung einer Kostenfunktion $K(f_1, \ldots, f_n)$.
2. Berechnung eines in der Kostenfunktion auftretenden unbekannten Parameters.
3. Lösen des Gleichungssystems $\partial K / \partial f_i = 0$.
4. Nachweis, daß genau eine der so bestimmten Lösungen die Randbedingung $\sum f_i = 1$ erfüllt und daß die so berechnete Lösung ein globales Minimum der Kostenfunktion darstellt.

Kostenfunktion

Als Kostenfunktion sollen die Kosten pro produktivem Schritt dienen. Probabilistisch ausgedrückt ist dies das Verhältnis zwischen dem Erwartungswert der Kosten der nächsten Abfrage $\sum f_i c_i$ und der Wahrscheinlichkeit a, daß die nächste Abfrage produktive Arbeit leistet (also im asynchronen Kontrollfluß gerade benötigt wird).

$$K(f_1, \ldots, f_n) := \frac{\sum_{i=1}^{n} f_i c_i}{a(f_1, \ldots, f_n)} \tag{7.1}$$

Die Aktivitätswahrscheinlichkeit a

Der Zustand des asynchronen Kontrollflusses läßt sich durch eine Markovkette mit den Zuständen A für „aktiv“ und W_j für „wartend auf Abfrage von o_j“ modellieren. Abbildung 7.6 zeigt die Markovkette mit ihren Übergangswahrscheinlichkeiten. Wird auf die Abfrage von o_j gewartet, so wird mit Wahrscheinlichkeit f_j die passende Operation als nächstes abgefragt und die Markovkette geht in den aktiven Zustand über. Sonst muß weiter gewartet werden. Der Automat bleibt genau dann im aktiven Zustand, wenn die als nächstes abgefragte Operation auch tatsächlich benötigt wird. Die Wahrscheinlichkeit, dafür ist $\sum p_i f_i$. Wird die erwartete Operation dagegen nicht abgefragt, so wird in den entsprechenden Wartezustand übergegangen.

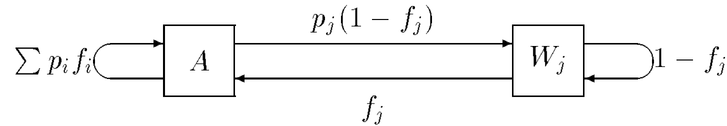

Abbildung 7.6: Markovmodell des asynchronen Kontrollflusses.

Die Wahrscheinlichkeit, daß die Markovkette im Zustand A ist, ist nun gerade der benötigte Parameter a. Zur Bestimmung dieses Wertes gilt es, die folgende Eigenwertgleichung zu lösen:

$$\begin{pmatrix} \sum p_i f_i & f_1 & \cdots & f_n \\ p_1(1-f_1) & 1-f_1 & & \mathbf{0} \\ \vdots & & \ddots & \\ p_n(1-f_n) & \mathbf{0} & & 1-f_n \end{pmatrix} \begin{pmatrix} a \\ w_1 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} a \\ w_1 \\ \vdots \\ w_n \end{pmatrix}$$

Dem entspricht das homogene LGS

$$\begin{pmatrix} (\sum p_i f_i) - 1 & f_1 & \cdots & f_n \\ p_1(1-f_1) & -f_1 & & \mathbf{0} \\ \vdots & & \ddots & \\ p_n(1-f_n) & \mathbf{0} & & -f_n \end{pmatrix} \begin{pmatrix} a \\ w_1 \\ \vdots \\ w_n \end{pmatrix} = 0$$

Durch Addieren aller Zeilen kann die erste Zeile des LGS zu 0 gemacht werden. Nach Dividieren der verbleibenden Zeilen durch f_i ergeben sich die Gleichungen:

$$a\frac{p_i(1-f_i)}{f_i} = w_i$$

Addieren dieser Gleichungen und die Zusatzbedingung $a + \sum w_i = 1$ ergibt:

$$a\sum \frac{p_i(1-f_i)}{f_i} = 1 - a$$

Das ist äquivalent zu

$$a = \frac{1}{\sum_{i=1}^{n} \frac{p_i}{f_i}} \tag{7.2}$$

Lösen der Ableitung der Kostenfunktion

Durch Einsetzen von a in Gleichung 7.1 kann nun die vollständige Kostenfunktion hingeschrieben werden:

$$K(f_1, \ldots, f_n) = \left(\sum_{i=1}^{n} \frac{p_i}{f_i}\right)\left(\sum_{i=1}^{n} f_i c_i\right) \tag{7.3}$$

Partielles Ableiten nach f_k und Null setzen ergibt

$$\frac{\partial K}{\partial f_k} = -\frac{p_k}{f_k^2}\sum_i c_i f_i + \left(\sum_i \frac{p_i}{f_i}\right) c_k \stackrel{!}{=} 0$$

Nun werden die von k abhängigen Werte auf eine Seite gebracht:

$$\frac{c_k f_k^2}{p_k} = \frac{\sum c_i f_i}{\sum \frac{p_i}{f_i}}$$

Durch Gleichsetzen zweier Instanzen dieser Gleichung fallen die Summen weg

$$\frac{c_k f_k^2}{p_k} = \frac{c_j f_j^2}{p_j} \Leftrightarrow \frac{f_k}{f_j} = \frac{\sqrt{\frac{p_k}{c_k}}}{\sqrt{\frac{p_j}{c_j}}}$$

Offenbar legen diese Gleichungen nur die gegenseitigen Proportionen der f_k fest — nicht aber deren Betrag. Dadurch ist es an dieser Stelle möglich, die Nebenbedingung $\sum f_k = 1$ einzubringen.[4] Dadurch ergibt sich:

$$f_j = \frac{\sqrt{\frac{p_j}{c_j}}}{\sum_{i=1}^{n} \sqrt{\frac{p_i}{c_i}}} \tag{7.4}$$

Da diese Belegung für die f_j der einzige Kandidat für ein lokales Extremum ist, und die Kostenfunktion für Randwerte gegen unendlich strebt (geht ein f_j gegen 0, so geht a gegen Null und K gegen unendlich), stellt die Lösung auch das globale Minimum der Kostenfunktion dar.

7.3.3 Auswertung des Ergebnisses

Gleichung 7.4 wird dadurch interessant, daß sie eine gleichzeitig einfache und nichttriviale Beziehung ausdrückt. Intuitiv ist einsichtig, daß eine Elementaroperation um so häufiger abgefragt werden sollte, je öfter sie benötigt wird, daß andererseits aber teure Elementaroperationen seltener abgefragt werden sollten. Gleichung 7.4 erlaubt nun, diesen qualitativen Tradeoff quantitativ auszudrücken. Die Abfragehäufigkeit sollte nämlich proportional zur Wurzel des Verhältnisses von Auftretenswahrscheinlichkeit und Kosten sein. Diese Beziehung scheint auch dann noch zu gelten, wenn deterministische Abfragefolgen verwendet werden (siehe auch Abschnitt 10.2.3).

Durch Einsetzen von Gleichung 7.4 in Gleichung 7.2 ergibt sich ein Maß für die zu erwartenden Kosten:

$$K_{\text{opt}} = \left(\sum_{i=1}^{n} \sqrt{p_i c_i} \right)^2 \tag{7.5}$$

Den erreichbaren Speedup kann man durch Vergleich mit der Kostenfunktion für die „naive“ Lösung $f_i = \frac{1}{n}$ beurteilen.

$$K_{\text{naiv}} = \sum_{i=1}^{n} c_i \tag{7.6}$$

Das sind die Kosten für einmalige Abfrage jeder Operation; also mehr, als die durchschnittlichen Kosten im Falle der einfachen deterministischen Abfragefolge

[4] Der orthodoxere Lösungsweg hätte darin bestanden, die Nebenbedingung von Anfang an (mit Hilfe eines Lagrange-Multiplikators) einzubringen. Da die Lösung ohne Nebenbedingung sich aber mit der Nebenbedingung verträgt, wird dadurch der einfachere Ansatz nachträglich gerechtfertigt.

$[1, \ldots, n]$, bei der im Mittel nur etwa die Hälfte aller Elementaroperationen abgefragt werden muß, um einen produktiven Schritt zu tun. Dadurch wird deutlich, daß die nichtdeterministische Abfragestrategie nur ein mathematisches Hilfsmittel und nicht etwa eine sinnvolle Implementierungsstrategie ist.

Ein einfaches Beispiel

Seit $n = 2$, $c_1 = c_2 = 1$, $p_1 = 4/5$, $p_2 = 1/5$. Dann ergibt Formel 7.4 $f_1 = 2/3$, $f_2 = 1/3$. Die häufigere Elementaroperation sollte also doppelt so oft abgefragt werden wie die seltenere.

7.4 Optimierung der Abfragereihenfolge

Nun soll wieder von deterministischen Abfragefolgen $[l_k]$ ausgegangen werden. Diese sollen unter dem Gesichtspunkt betrachtet werden, wie sich Regelmäßigkeiten im Kontrollfluß auf die Wahl einer günstigen Abfragefolge auswirken.

Diese Regelmäßigkeiten sollen dadurch quantitativ erfaßt werden, daß die Wahrscheinlichkeit p_{ij}, daß Elementaroperation o_i im asynchronen Kontrollfluß auf Operation o_j folgt, gemessen (oder geschätzt) wird. Das zu lösende Problem läßt sich dann so formulieren, daß aus der Matrix der Übergangswahrscheinlichkeiten (p_{ij}) und dem Kostenvektor (c_i) eine Abfragefolge $[l_k]$ mit Länge $\leq m$ zu bestimmen ist, die die erwarteten Ausführungszeiten minimiert. Für die Folge $[l_k]$ gibt es allerdings exponentiell viele Möglichkeiten, und deshalb steht zu befürchten, daß das gegebenen Optimierungsproblem $\mathcal{NP}$-hart ist.

7.4.1 Die Kostenfunktion

Um Abfragefolgen vergleichen zu können, soll nun — analog zu Absatz 7.3 — eine Kostenfunktion entwickelt werden, die aus einer Abfragefolge ein Maß für die aufzuwendenden Kosten berechnet. Wie vorher, soll das Verhältnis der aufgewendeten Kosten zur Anzahl der durchgeführten Schritte als Kostenmaß dienen.

Wieder ist es möglich, das Kostenmaß durch ein Markovkettenmodell zu ermitteln. Ein Zustand z_{ik} in diesem Modell repräsentiert den Zustand eines Prozessors, in dem o_i zur Ausführung ansteht und k die aktuelle Position in der Abfragefolge ist. Allerdings hat diese Markovkette $nm \geq n^2$ Zustände und ist damit nur schwer handhabbar.

Stattdessen soll eine iteratives Verfahren benutzt werden, das schneller und einfacher ist. Es ist äquivalent zur Aufstellung und Auswertung der gerade beschriebenen Markovkette. Abbildung 7.7 zeigt den verwendeten Algorithmus. Parameter der Funktion iterCost sind der Kostenvektor, die Matrix der Übergangswahrscheinlichkeiten, die zu bewertende Abfragefolge und ein Anfangswert für den Vektor (q_i) der Wahrscheinlichkeiten, daß der Prozessor auf Abfrage von a_i wartet. Die Funktion simuliert das Verhalten des Systems, indem sie zyklisch durch die Abfragefolge hindurchläuft. Wird gerade Operation a_l abgefragt, so treten c_l Kosteneinheiten auf, und der Erwartungswert für die Zahl der ausgeführten produktiven Schritte steigt um q_l. Anschließend werden die q_i aktualisiert und dann die nächste Abfrage betrachtet. Dieser Simulationsprozeß wird solange iteriert, bis steps sich stabilisiert. cost/steps gibt dann die Kosten pro produktivem Schritt an.

```
FUNCTION iterCost((c_i), (p_ij), [l_1, ..., l_m], (q_i))
REPEAT
    steps := cost := 0
    FOR each l in [l_k] DO
        cost += c_l (*account for cost of operation*)
        steps += q_l (*increased number of productive steps*)
        FOR i := 1 TO n DO (*Readjust probabilities*)
            IF i ≠ l THEN q_i += q_l p_il
        q_l *= p_ll
UNTIL steps has converged
RETURN (cost/steps)
```

Abbildung 7.7: Iterative Berechnung der Kostenfunktion.

Dieses Iterationsverfahren ist äquivalent zu einer iterativen Lösung der sich aus dem oben beschriebenen Markovkettenmodell ergebenden Eigenwertgleichung. Da die Struktur der Abfragefolge aber direkt interpretiert wird, anstatt in eine große Matrix zu codiert werden, ergibt sich eine erheblich größere Ausführungsgeschwindigkeit. Die für das FSSP durchgeführten Experimente ergaben eine ausreichende Konvergenz des Verfahrens nach 2–3 Iterationen.

7.4.2 Ansätze zur Optimierung

Für kleine n und m ist es nun möglich, die optimale Abfragefolge durch systematisches Ausprobieren zu bestimmen. Bei komplexeren Problemen (z.B. gibt es beim FSSP 8 Elementaroperationen) ergibt sich aber eine hoffnungslose kombinatorische Explosion.

Die Funktion iterCost ließe sich auch mit einer Anzahl von bekannten Verfahren zur kombinatorischen Optimierung (z.B. „hill climbing“, Monte Carlo, „simulated annealing“, genetische Algorithmen) koppeln, um zumindest eine gute Näherung zu bestimmen. Allerdings wird man es in vielen Fällen nicht als der Mühe wert betrachten, einen solchen Aufwand zu treiben und lieber eine Lösung von Hand basteln. Dabei kann Formel 7.4 helfen, die optimalen Abfragehäufigkeiten zu bestimmen, und iterCost kann benutzt werden, um rasch verschiedene Abfragefolgen duchzuspielen, ohne jedesmal den Suchalgorithmus ablaufen zu lassen.

Es wäre denkbar, daß zukünftige parallele Programmiersprachen asynchrone Konstrukte in ansonsten datenparallelen Sprachen zulassen und diese in Abfragefolgen übersetzen. Für diesen Anwendungsfall wäre eine vollautomatische Optimierung der Abfragefolgen nötig, die aber nicht so zeitaufwendig sein darf wie die oben erwähnten Verfahren (u.U. mit Ausnahme von hill climbing). Angebracht wäre vielmehr eine Heuristik, die einfach ist und deutlich bessere Ergebnisse liefert als eine naive Lösung.

Unter diesem Gesichtspunkt wurden einige Greedy-Algorithmen ausprobiert, die eine unendliche Folge von Abfragen generieren. Das grundsätzliche Vorgehen entspricht dabei demjenigen der inneren Schleife von iterCost. Nur wird die nächste Operation nicht aus einer vorgegebenen Folge entnommen, sondern auf

Grund einer einfachen Heuristik bestimmt. Aus der so generierten Folge läßt sich dann ein passendes Teilstück auswählen und als Abfragefolge verwenden. Folgende Heuristiken wurden untersucht.

- Wähle jeweils die Operation o_i mit momentan maximaler Wahrscheinlichkeit q_i. Dieser Ansatz war insofern nützlich, als daß er Abfragefolgen generiert, aus denen sich Versatzstücke für eine manuelle Konstruktion guter Abfragefolgen gewinnen lassen. Für sich genommen waren die generierten Folgen aber kaum besser als beim naiven Ansatz. Das Hauptproblem besteht darin, daß seltene Operationen vernachlässigt werden.

- Wähle o_i so, daß c_i/q_i minimiert wird. Erstaunlicherweise waren die Ergebnisse noch schlechter als der erste Ansatz, da teure, seltene Operationen sträflich vernachlässigt werden.

- Maximiere q_i multipliziert mit einem Gewichtungsfaktor, der Operationen bevorzugt, die bislang seltener abgefragt wurden als Formel 7.4 dies vorschlägt. Auch dieses Verfahren funktionierte nicht gut, da zeitweise vernachlässigte Operationen völlig erratisch irgendwo eingestreut werden, wo sie nicht hinpassen.

7.5 Auswahl von Elementaroperationen

In den letzten Abschnitten ist immer davon ausgegangen worden, daß ein Satz von Elementaroperationen fest vorgegeben ist. Hier sollen nun einige einfache Regeln beschrieben werden, die helfen können, einen Algorithmus durch Elementaroperationen zu beschreiben und vorhandene Sätze von Elementaroperationen zu optimieren. Der Einfachheit halber soll davon ausgegangen werden, daß die Kontrollstruktur — wie in Algorithmus 7.3 — durch ein Automatenmodell mit Zustandsvariabler state realisiert wird.

7.5.1 Schleifen auflösen

Ein Programm der Form

α
WHILE b **DO** β
γ

wobei b eine lokale Bedingung ist, kann durch die drei Elementaroperationen

o_α:α; state := o_β
o_β:**IF** b **THEN** β **ELSE** state := o_γ
o_γ:γ

ersetzt werden. Im Endeffekt läuft es darauf hinaus, Schleifen (oder allgemeiner Rückwärtssprünge im Kontrollflußgraphen) mit Hilfe eines verklausulierten **GOTO** zu implementieren. Entsprechend groß ist die Gefährdung der Lesbarkeit der entstehenden Programme[5]. Deshalb sollte man diese Regel nicht immer rein

[5] Deshalb könnte es sich als gute Idee herausstellen, die Transformation mit Hilfe eines Compilers durchzuführen.

mechanisch anwenden, sondern den Kontrollfluß so umgestalten, daß die entstehenden Elementaroperationen eine klare Semantik haben.

Sollen die für Elementaroperationen gemachten Annahmen erfüllt werden, so müssen im Prinzip alle lokalen Schleifen auf die angegebene Weise aufgelöst werden. In einigen Fällen mag es jedoch möglich (und sinnvoll) sein, von dieser Regel abzuweichen; insbesondere dann, wenn sich eine obere Grenze für die Anzahl der Schleifendurchläufe angeben läßt, die so klein ist, daß die Wartezeit von Prozessoren mit weniger Schleifendurchläufen tolerierbar ist.

7.5.2 Elementaroperationen aufspalten

Eine Elementaroperation der Form

α
IF b **THEN** β
γ

wobei b eine lokale Bedingung ist, kann durch die Elementaroperationen

o_α:α; state := (**IF** b **THEN** o_β **ELSE** o_γ)
o_β:β; state := o_γ
o_γ:γ

ersetzt werden.

Diese Transformation ist insbesondere dann nützlich, wenn das Codestück β teuer ist, aber nur selten ausgeführt wird, weil die Bedingung b nur selten erfüllt ist. Da die Operationen o_α und o_γ entsprechend Formel 7.4 öfter abgefragt werden können als die Ursprungsoperation, ergibt sich dadurch eine mögliche Einsparung. Ein typisches Beispiel für diese Art Optimierung ist die Operation MakeChoicePoint in Algorithmus 7.3, die durch Abspaltung aus einem Vorläufer der Operation Search entstanden sein könnte.

7.5.3 Verschmelzen von Elementaroperationen

Führen zwei Elementaroperationen ähnliche Aktionen aus, so ist es u.U. möglich, sie zu einer einzigen Operation zu verschmelzen. Gelingt dies, so ist mit einer deutlichen Reduzierung des SIMD-Overheads zu rechnen, da die für diese Operationen aufgewandte Ausführungszeit halbiert wird.

Ein typisches Beispiel sind zwei Operationen der Form

o_1:α; state := o'_1
o_2:α; state := o'_2

Solch ein Operationenpaar entsteht z.B. wenn α gemeinsamer Teilausdruck von zwei Operationen war und per Operationsaufspaltung (Abschnitt 7.5.2) aus diesen herausfaktorisiert wurde. Die beiden Operationen lassen sich zu folgender Operation verschmelzen:

o_{12}: α; state = follow

Zusätzlich muß in den Operationen, die zu o_1 (o_2) verzweigen, die Anweisung $\mathsf{state} := o_1$ (o_2) zu $\mathsf{follow} := o_1(o_2)$; $\mathsf{state} := o_{12}$ abgeändert werden.

Dieser Mechanismus läßt sich als asynchroner Unterprogrammaufruf deuten. Das Konzept der Kontrollstruktur als endlicher Automat wird an dieser Stelle schon sehr strapaziert. Treten solche und kompliziertere Fälle von Operationsverschmelzung öfter auf, wäre der Interpreteransatz aus Satz 14 wohl eleganter. Dort ließe sich α als abstrakter Maschinenbefehl implementieren.

Der beschriebene Mechanismus erinnert an die aus dem Compilerbau bekannte Technik der „*global common subexpression elimination*“. Dies weist darauf hin, daß Techniken der Datenflußanalyse helfen können, die hier beschriebenen Optimierungstechniken zu automatisieren.

7.5.4 Vereinfachen von Elementaroperationen

In sequentiellen Algorithmen kann oft Zeit gespart werden, indem Sonderfälle einer speziellen Behandlung unterzogen werden. Für SIMD-Algorithmen sind solche Optimierungen ein zweischneidiges Schwert, denn die Abfrage des Sonderfalles muß auch von denjenigen Prozessoren abgewartet werden, für die der Sonderfall nicht eingetreten ist. Beim FSSP-Algorithmus brachte das Weglassen einer Heuristik, die im sequentiellen Fall Zeit einsparte, einen Speedup von 17 %. Außerdem kann das Entfernen von Spezialbehandlungen helfen, Kandidaten für das Verschmelzen von Elementaroperationen zu finden.

Ein ähnlicher Effekt ergibt sich bei Operationen, die zwar noch optimiert werden können, die aber so selten ausgeführt werden, daß sich im sequentiellen Fall der Aufwand nicht lohnt. Selbst mit Optimierung der Abfragehäufigkeit nach Formel 7.4 werden diese Operationen aber erheblich öfter abgefragt, als sie ausgeführt werden. Deshalb kann sich im SIMD-Algorithmus eine Optimierung auch solcher Fälle lohnen.

Kapitel 8

Arbeitsverteilung

8.1 Initiale Arbeitsverteilung

Zu Beginn der Suche ist nur der Wurzelknoten gegeben. Um die Parallelität des Rechners auszunutzen, werden aber nproc Wurzeln disjunkter Teilbäume benötigt, die an die einzelnen Prozessoren zu verteilen sind. Eine Reihe verschiedener Ansätze zur Lösung des Problems bieten sich an:

Sequentiell Ein sequentielles Programm kann die benötigten Wurzelknoten berechnen (z.B. durch Breitensuche) und nacheinander an die Prozessoren verteilen. Aus praktischen Erwägungen mag sich diese Lösung manchmal anbieten, aber für große Prozessorzahlen kann sie leicht zu einem Engpaß führen.

Saatkorn-Methode Nur ein einziger Prozessor wird mit dem Wurzelknoten initialisiert. Der Rest wird der Arbeitsumverteilung überlassen. Diese Methode bietet sich wegen ihrer Einfachheit dann an, wenn die Arbeitumverteilung gut funktioniert und die Teilaufgaben dabei gleichmäßig verteilt werden. Will man aber z.B. Nachbarschaftskommunikation für die Umverteilung anwenden, wird das in Abschnitt 8.2.3 beschriebene Problem der Clusterbildung noch verstärkt.

Baumförmige Verteilung Wie bei der Saatkorn-Methode wird ein Knoten mit der Baumwurzel initialisiert und beginnt eine Suche. Anstatt aber Baumverzweigungen auf den Keller zu legen, verteilt der Prozessor die Nachfolger an unbeschäftigte Prozessoren. Diese Methode dürfte funktionieren, ist aber etwas umständlich. Insbesondere ist es nicht trivial, unbeschäftigte Prozessoren aufzuspüren, wenn der Baumverzweigungsgrad nicht konstant ist.

Verteilung ohne Kommunikation Alle bisher vorgestellten Verfahren haben gemeinsam, daß Teilprobleme übertragen werden müssen. Dies ist aber nicht zwingend nötig, wenn vorausgesetzt werden kann, daß jeder Prozessor mit einer eindeutigen Nummer iproc versehen ist. Mit Hilfe dieser Nummer kann der Prozessor in verblüffend einfacher Weise ein Teilproblem berechnen, derart, daß der gesamte Suchbaum erfaßt ist und jeder Prozessor ein anderes Teilproblem bearbeitet. Dieses Verfahren soll im folgenden näher beschrieben werden.

Das Verfahren funktioniert so, daß der Wurzelknoten zu Beginn an alle Prozessoren gesendet wird.[1] Dann beginnen alle Prozessoren mit der Suche. Zu diesem Zeitpunkt herrscht völlige Symmetrie, da alle das gleiche tun. Geraten die Prozessoren nun an eine Suchbaumverzweigung, so wird diese Symmetrie gebrochen, indem die Menge der symmetrischen Prozessoren in gleich große Äquivalenzklassen geteilt wird — eine Äquivalenzklasse für jeden Nachfolger des betrachteten Knotens. Dieser Prozeß der Symmetriebrechung wird so lange wiederholt, bis nur noch ein Prozessor in jeder Äquivalenzklasse ist.

Der eigentliche Trick ist, daß die Einteilung in Äquivalenzklassen völlig ohne Kommunikation — nur auf der Basis der Prozessornummer — funktioniert. Zu diesem Zweck speichert jeder Prozessor die Anzahl der Prozessoren in seiner Äquivalenzklasse und die eigene Nummer innerhalb der Äquivalenzklasse. Mit Hilfe dieser Information und der Anzahl der zu bildenden Tochteräquivalenzklassen kann jeder Prozessor ausrechnen, wo er in welcher Tochteräquivalenzklasse landet.

Abbildung 8.1 zeigt eine mögliche Realisierung dieser Idee. Der Algorithmus ist im Detail etwas kompliziert, da er den Fall, daß die Größe einer Äquivalenzklasse nicht durch die Anzahl der Nachfolgeknoten teilbar ist, korrekt behandeln muß. Für den einfachen Fall, daß nproc z.B. eine Zweierpotenz und der Suchbaum ein vollständiger Binärbaum ist, reduziert der Algorithmus sich darauf, die Binärziffern der Prozessornummer heranzuziehen, um zu entscheiden, welcher Teilbaum welchem Prozessor zugeordnet wird. Abbildung 8.2 zeigt ein komplizierteres Beispiel, in dem ein unregelmäßig aufgebauter Baum auf 8 Prozessoren verteilt wird. Die Knoten des Baums sind mit den für den jeweiligen Teilbaum zuständigen Äquivalenzklassen markiert. Der Einfachheit halber wurde angenommen, daß der Initialisierungsprozess auf keine Blätter des Suchbaums trifft. Eine entsprechende Erweiterung von searchInit ist recht einfach. Allerdings werden in diesem Fall nicht unbedingt alle Knoten mit Arbeit versorgt.

8.2 Umverteilung

Im allgemeinen kann nicht davon ausgegangen werden, daß die anfängliche Arbeitsverteilung so gleichmäßig ist, daß eine dauerhaft gute Auslastung der Prozessoren erreicht wird. Deshalb müssen vielbeschäftigte Prozessoren von Zeit zu Zeit einen Teil ihres Suchraums an arbeitslos gewordene Prozessoren abgeben. Gleichzeitig muß der dazu benötigte Kommunikationsaufwand so klein gehalten werden, daß er nicht die eigentliche Arbeit dominiert.

In Abschnitt 8.2.1 wird eine einfache Möglichkeit vorgestellt, den Suchraum aufzuteilen. Abschnitt 8.2.2 geht dann darauf ein, wie das Wechselspiel zwischen eigentlicher Suche und Arbeitsumverteilung geregelt werden kann. Die Abschnitte 8.2.3 bis 8.2.5 führen dann drei verschiedene Kommunikationsmuster für die Arbeitsumverteilung ein.

[1] Auf SIMD-Rechnern (und einigen MIMD-Rechnern) dauert dies nicht länger, als die Übertragung zu einem einzelnen Prozessor.

```
PROCEDURE searchInit(node)
n := nproc; k := iproc
LOOP
    s := |successors(node)| (*number of choices*)
    IF s ≥ n THEN EXIT (*Less Processors than choices*)
    i := k MOD s (*Which branch is taken?*)
    (*How many symmetric PE's remain?*)
    n := n DIV s + (IF i < n MOD s THEN 1 ELSE 0)
    k := k DIV s (*New ordinal number in symmetry class*)
    node := successors(node)[i + 1] (*commit choice*)
(*Distribute successors to remaining equivalent nodes*)
(*Number of successors to be investigated:*)
s' := s DIV n + (IF k < s MOD n THEN 1 ELSE 0)
(*Index of first successor to be investigated*)
i := ((s DIV n) + 1) · min(k, s MOD n)+
    (s DIV n) · max(0, k − (s MOD n))
searchMultipleNodes(successors(node)[i + 1..i + s'])
```

Abbildung 8.1: Arbeitsverteilung durch Symmetriebrechung.

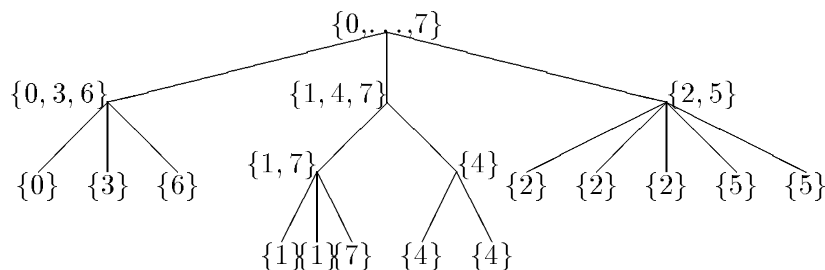

Abbildung 8.2: Beipiel für initiale Arbeitsverteilung.

```
WHILE noMoreSiblings(bottom(oldStack)) DO
    removeBottom(oldStack)
newStack := Push(initStack(), nextSibling(bottom(oldStack)))
removeBottom(oldStack)
```

Abbildung 8.3: Aufteilung des Suchraums.

8.2.1 Teilen des Suchraums

In [15] werden eine Reihe von Strategien diskutiert, den Suchraum in Gegenwart von Tiefensuche aufzuteilen. All diese Verfahren haben gemeinsam, daß der Keller manipuliert wird. Ein besonders einfacher Ansatz, der sich für das FSSP sehr gut bewährt hat, ist in Abbildung 8.3 dargestellt. Es benutzt ähnliche abstrakte Datenstrukturen wie Algorithmus 7.1.[2] Die erste noch nicht ausgeschöpfte Entscheidung des momentan laufenden Suchprozesses wird fixiert, und alle dadurch „abgeklemmten" Entscheidungen werden auf den Keller des neuen Suchprozesses[3] gebracht. Diese Strategie hat die folgenden positiven Eigenschaften:

- Tief unten im Keller finden sich die größten noch unberührten Teile des Suchraums, die am ehesten den Aufwand einer Übertragung auf andere Prozessoren lohnen.
- Der Umfang der zu übertragenden Daten ist geringer als bei anderen denkbaren Verfahren.
- Das Verfahren ist sehr einfach.
- Die Aufteilung des Suchraums erscheint zwar asymmetrisch, aber bei geringem Baumverzweigungsgrad und durch Iterieren des Aufteilungsprozesses (insbesondere in Verbindung mit der Arbeitsverteilungsroutine aus Abschnitt 8.2.5) ist das nicht unbedingt ein Problem.
- Der FSSP-Suchalgorithmus benutzt Suchraumbeschneidungsheuristiken, die dazu führen können, daß parallel gestartete Teilaufgaben sich später als überflüssig herausstellen. Da diese Heuristiken eher lokal wirksam werden, wird die Wahrscheinlichkeit, überflüssige Arbeit zu tun, durch die gewählte Aufteilungsstrategie recht klein.

8.2.2 Adaptive Arbeitsverteilung

Ein SIMD-Rechner kann nicht gleichzeitig Arbeit umverteilen und die eigentliche Suche vorantreiben. Deshalb gilt es, zwischen Phasen der Suche und Phasen der Arbeitsverteilung Balance zu halten. Der hier gewählte Ansatz besteht darin, die Suchprozesse solange unabhängig voneinander laufen zu lassen, bis der Anteil der

[2] Unterschiede: Der Keller, auf den eine Operation sich bezieht, wird explizit angegeben; das unterste Kellerelement kann entfernt und eine einmal gemachte Entscheidung damit festgeschrieben werden.

[3] Zu beachten ist noch, daß in praktischen Anwendungen neben dem Kellerinhalt noch zusätzliche Information zu übertragen ist. Dies ändert aber nichts an der grundlegenden Vorgehensweise.

```
initialize Search
LOOP
    WHILE reduceAdd(IF state = Stop THEN 1 ELSE 0)/nproc < α DO
        FOR i:=1 to k DO
            make transition of search automaton (*see algorithm 7.3*)
    redistribute work
```

Abbildung 8.4: Adaptive Arbeitsverteilung.

arbeitslosen Prozessoren eine gewisse Schwelle überschritten hat. Dann werden alle Prozesse angehalten und die Umverteilung findet durch Kooperation aller Prozessoren unter Vermeidung von Bottlenecks statt. In Abschnitt 8.2.5 wird sich heraustellen, daß ein solches phasenweises Vorgehen auch für MIMD-Rechner vorteilhaft sein kann.

Abbildung 8.4 zeigt das Schema dieser *adaptiven* Arbeitsverteilungsstrategie. Das aus Algorithmus 7.3 bekannte Automatenmodell des Suchalgorithmus führt k Übergänge aus, und dann wird überprüft ob die Anzahl der arbeitslosen Prozessoren einen Anteil α überschreitet. Ist dies der Fall, so wird der Arbeitumverteilungsalgorithmus aufgerufen, bevor wieder zur Suchphase übergegangen wird. Der Parameter k muß so eingestellt werden, daß der Aufwand für die Reduktionsoperation nicht den Zeitaufwand für die Suche dominiert. α regelt den Tradeoff zwischen Prozessorauslastung und Kommunikationsaufwand. Außerdem sollte k nicht so groß werden, daß ein Überschreiten einer „Arbeitslosenquote“ größer α zu lange unbemerkt bleibt.

Da der Zeitaufwand für die Reduktionsoperation mit der Prozessorzahl steigt, ist der vorgestellte Algorithmus streng genommen nur skalierbar, wenn die Problemgröße stärker vergrößert wird als die Prozessorzahl. In der Praxis dürfte sich daraus aber kaum ein Problem ergeben, da man ja gerade am Durchsuchen sehr großer Suchräume interessiert ist. Der FSSP-Algorithmus lastet z.B. 16384 Prozessoren zu ca. 90 % aus, und die Reduktionsoperation benötigt weniger als 1 % der Laufzeit von ca. 12s.

8.2.3 Nachbarschaftskommunikation

Eine naheliegende Strategie für Arbeitsumverteilung besteht darin, daß arbeitslose Prozessoren versuchen, Arbeit von den Prozessoren zu erhalten, mit denen sie unmittelbar verbunden sind. Diese Strategie ist einfach, und die Kommunikation selber geht sehr schnell. Allerdings ist bekannt (siehe z.B. [16], [18]), daß z.B Ringnetzwerke mit mehr als ca. 16 Prozessoren in der Praxis nicht in der Lage sind, die Arbeit gleichmäßig zu verteilen.

In der vorliegenden Arbeit wurde versucht, das 2D-Gitter der MasPar zur Arbeitsverteilung einzusetzen. Da Nachbarschaftskommunikation auf der MasPar viel schneller ist als Routerkommunikation, weil eine recht gute initiale Arbeitsverteilung zur Verfügung stand (Abschnitt 8.1) und weil ein zweidimensionales Verbindungsnetzwerk viel mehr Wege für den Informationsfluß bietet als ein Ring, wurde davon ausgegangen, daß dies ein sinnvoller Ansatz ist.

Es stellte sich aber heraus, daß es mit diesem Ansatz nicht möglich war, auch

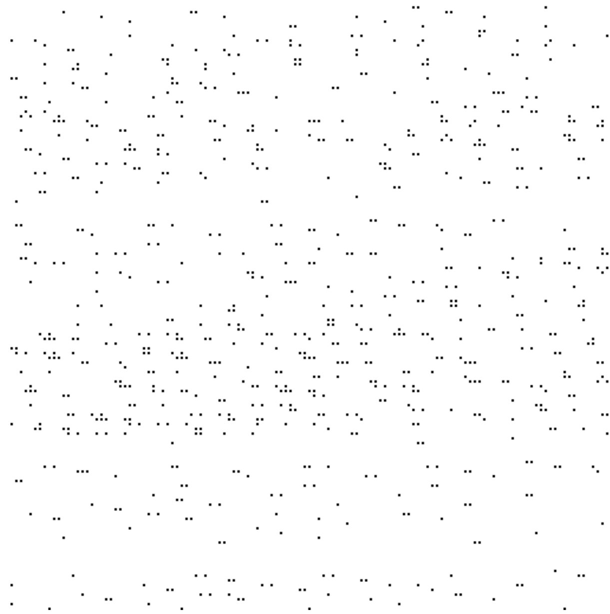

Abbildung 8.5: Aktive PEs nach initialer Arbeitsverteilung.

Abbildung 8.6: Gute PE-Auslastung zu Beginn der Suche.

Abbildung 8.7: Clusterbildung.

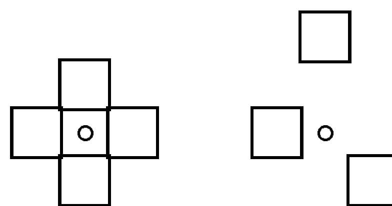

Abbildung 8.8: Raster für Nachbarschaftskommunikation.

nur eine durchschnittliche Prozessorauslastung von 20 % aufrecht zu erhalten. Interessant ist vor allem, wie dieser Fehlschlag zustande kommt. Abbildung 8.5 zeigt die Ausgangssituation in der 590 aktive Prozessoren recht gleichmäßig über das Prozessorgitter verstreut sind. Nach einer Anzahl von Umverteilungsschritten ist die Prozessorauslastung gut (Abbildung 8.6). Es ist erkennbar, daß wegen kleiner Unregelmäßigkeiten in der initialen Arbeitsverteilung kleinere Gebiete arbeitslos sind. Schon nach kurzer Zeit aber werden die inaktiven Gebiete immer größer, und es bilden sich kompakte Cluster von beschäftigten Prozessoren (Abbildung 8.7). Eine nähere Betrachtung zeigt, daß in der Mitte dieser Cluster Prozessoren sind, die viel Arbeit abzugeben hätten, dies aber nicht können, da alle Nachbarn beschäftigt sind. Die Prozessoren am Rand der Cluster haben dagegen nur sehr wenig Arbeit abzugeben. Deshalb werden unbeschäftigte Prozessoren, die von den Randprozessoren Arbeit erhalten, schneller wieder beschäftigungslos, als Arbeit aus der Mitte der Cluster herausdiffundieren kann.

Für das FSSP-Problem hat es sich dann herausgestellt, daß Routerkommunikation gute Ergebnisse liefert. Deshalb wurde der Ansatz der Nachbarschaftskommunikation nicht weiter verfolgt. Da viele aktuelle Architekturen aber eine langsamere Routerkommunikation (bzw. deren Emulation) haben als die MasPar, soll hier trotzdem diskutiert werden, wie sich Nachbarschaftskommunikation verbessern ließe:

- Wenn möglich, sollte eine Netzwerktopologie mit möglichst geringem Durchmesser und ohne Bottlenecks gewählt werden. In [15] wurden z.B. gute Ergebnisse mit einem Hypercube erzielt.

- Ein Raster mit größerem Durchmesser führt dazu, daß die Cluster sich weiter verteilen. Zum Beispiel bewirkte die Benutzung des Kommunikations-

rasters aus Abbildung 8.8 rechts eine um 50 % bessere Prozessorauslastung als die normale NEWS-Kommunikation aus Abbildung 8.8 links. Bei noch größeren Rasterdurchmessern wäre es wahrscheinlich günstiger, jeweils nur eine Kommunikationsrichtung zu würfeln, um eine gleichmäßige Informationsverteilung zu erreichen.

- Eine verbesserte initiale Arbeitsverteilung könnte dafür sorgen, daß mehr und dafür kleinere Cluster entstehen. Zumindest würde die anfängliche Phase guter Prozessorauslastung verlängert.

- Ein entscheidender Grund für Entstehung und Stabilität der Cluster besteht wohl darin, daß Arbeit nur an gänzlich unbeschäftigte Prozessoren weitergegeben wird. Eine Alternative bestände darin, Daten immer dann zu übertragen, wenn ein Prozessor mehr Arbeit hat als der andere. (Prozessor A habe Arbeit für a s, Prozessor B habe Arbeit für b s, dann sollte idealerweise Arbeit für $\frac{a-b}{2}$ s übertragen werden.) Setzt man voraus, daß ein tatsächlicher Ausgleich des Suchraumumfangs möglich ist, und die Zahl der Prozessoren hinreichend groß ist, so kann man dieses Arbeitsverteilungsschema annähernd durch die Diffusionsgleichung $\dot{a} = D\Delta a$ beschreiben. Dabei bezeichnet $a(\vec{x}, t)$ die Arbeitsdichte und die Konstante D ist ein Maß für die Diffusionsgeschwindigkeit. Unter diesen Bedingungen können Cluster nicht stabil sein; sie laufen auseinander. Es müßte sich ausrechnen lassen, wie groß D werden muß, damit Cluster sich auflösen, bevor der Suchraum ohnehin ausgeschöpft ist. Für Rechner mit kleinem lokalen Speicher wie der MasPar gibt es allerdings ein Problem mit diesem Ansatz, da mehrere Teilprobleme gleichzeitig im Speicher gehalten werden müssen.

- Nachbarschaftskommunikation könnte mit effektiveren aber teureren Umverteilungsstrategien kombiniert werden. Zum Beispiel könnte die im nächsten Abschnitt beschriebene Zufallspermutation immer dann angewandt werden, wenn die Auslastung der Prozessoren zu stark gesunken ist.

8.2.4 Zufallspermutationen

Probleme mit der oben beschriebenen Clusterbildung lassen sich vermeiden, wenn beschäftigungslose Prozessoren einen potentiellen Arbeitgeber aus allen Prozessoren auswürfeln. Um dabei Probleme mit Zugriffskonflikten zu vermeiden, sollte es sich beim entstehenden Kommunikationsmuster um eine Permutation[4] handeln.

Eine einfache und wirkungsvolle Methode zur Berechnung einer Zufallspermutation besteht darin, eine globale Zufallszahl $r \in \{0, \ldots, \mathsf{nproc}-1\}$ zu bestimmen, und Prozessor i mit Prozessor i **XOR** r kommunizieren zu lassen. Zwar lassen sich auf diese Weise nicht alle möglichen Permutationen erzeugen, aber für die Zwecke der Arbeitsumverteilung erwies sich dieser Ansatz als sehr wirkungsvoll. Der Spezialfall, daß nur Zweierpotenzen für r gewählt werden, entspricht übrigens gerade der Nachbarschaftskommunikation im Hypercube, die ja auch schon recht gute Ergebnisse liefert.

[4]Genauer: Um eine Einschränkung einer Permutation auf die Menge der arbeitslosen Prozessoren und deren Kommunikationspartner.

Beim FSSP-Algorithmus ließ sich mit diesem Verfahren eine Prozessorauslastung von 75 % erreichen wobei ca. 1/10 der Rechenzeit für Kommunikation aufgewendet wurden.

8.2.5 Sortieren

Der Algorithmus zur Arbeitsverteilung mit Hilfe von Zufallspermutationen hat noch zwei Schwächen:

1. Es läßt sich nicht ausschließen, daß ein arbeitsloser Prozessor einen ebenfalls arbeitslosen Kommunikationspartner erhält. Deshalb läßt sich nie eine Prozessorauslastung von 100 % erreichen.

2. Selbst wenn ein arbeitsloser Prozessor einen Arbeitgeber findet, wird der abzugebende Suchraum des Arbeitgebers oft so klein sein, daß schon nach kurzer Zeit beide Kommunikationspartner arbeitslos sind. Deshalb muß die Arbeitsumverteilung recht oft laufen.

Diese beiden Probleme lassen sich beseitigen, indem die Prozessoren nach dem Umfang ihres verbleibenden Suchraums sortiert werden und arbeitlose Prozessoren sich ihre Arbeit von den meistbeschäftigten Prozessoren holen. Da Sortieren im allgemeinen eine recht teure Operation ist, bleibt nun noch zu klären, wie sich diese Idee in eine effiziente Arbeitsumverteilungsprozedur umsetzen läßt.

Für das FSSP-Problem wurde eine sehr einfache Abschätzung für den Umfang des verbleibenden Suchraums gemacht. Je mehr Entscheidungen im unteren Ende des Kellers nicht mehr rückgängig gemacht werden müssen (weil alle anderen Alternativen bereits berücksichtigt sind oder von anderen Prozessoren verfolgt werden), desto kleiner ist der Suchraum. Die Anzahl dieser Entscheidungen läßt sich sehr leicht feststellen.

Außerdem kann diese Anzahl nur wenige verschiedene Werte annehmen. Dadurch kann die Sortierung durch einen ebenso einfachen wie schnellen verteilten „bucket-sort"-Algorithmus vorgenommen werden (Abbildung 8.9). Zunächst werden die Adressen der arbeitslosen Prozessoren zusammengeschoben. Prozessoren, in denen keine gültige Empfängeradresse steht, werden mit -1 markiert. Die Adressen der gestoppten Prozessoren werden dann in der **FOR**-Schleife von den aktiven Prozessoren mit der jeweils höchsten Arbeitslast abgeholt. Abbildung 8.10 illustriert den Umverteilungsprozeß.

Die router-Operationen lassen sich auf der MasPar noch etwas beschleunigen, indem statt router[adr] router[Map(adr)] geschrieben wird, wobei Map eine bijektive Funktion von Prozessornummern ist, die die Zieladressen gleichmäßiger verteilt. Da jeweils 16 Prozessoren (der MasPar MP-1) sich eine Routereinheit teilen müssen, kommt es sonst zu Stauungen an den für kleine Prozessornummern zuständigen Routereinheiten.

Es stellte sich heraus, daß die Ausführungszeit für den Sortierprozeß gegenüber der eigentliche Datenübertragung klein ist. Die erreichbare Prozessorauslastung wurde auf ca. 90 % erhöht

```
(*Put adresses of jobless PEs into PE 0, PE 1, ...*)
potentialReceiver := -1; (*Not a potential Receiver of work*)
IF state = Stop THEN router[enumerate()].potentialReceiver := iproc
receiversFound := 0; receiver := -1
IF state ≠ Stop THEN
    FOR d := reduceMin(fixedDecisions) TO reduceMax(fixedDecisions)DO
        IF fixedDecisions=d THEN
            receiver := router[enumerate() + receiversFound].potentialReceiver
            receiversFound += reduceAdd(1)
            IF globalor(receiver = -1) THEN EXIT (*no receivers left*)
    IF receiver ≠ -1 THEN shareWorkWith(receiver)
```

Abbildung 8.9: Arbeitsumverteilung durch bucket sort.

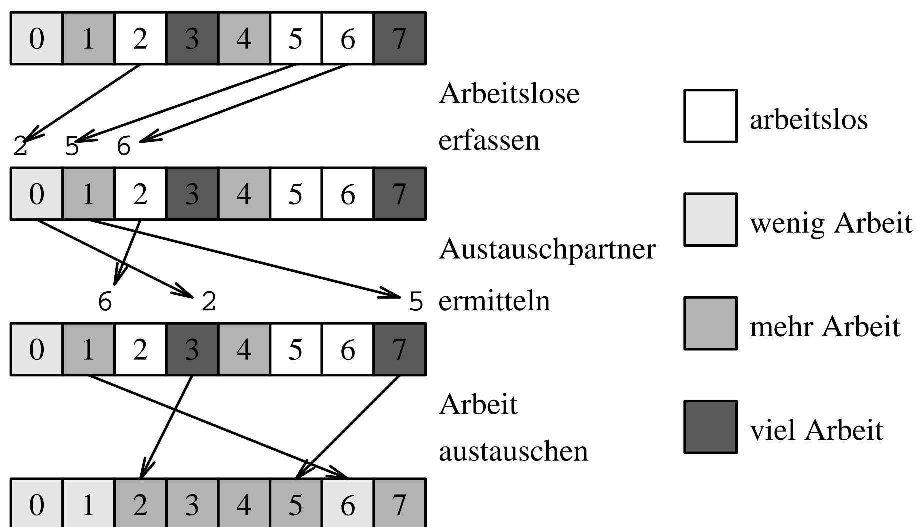

Abbildung 8.10: Arbeitsumverteilung durch Sortieren.

Kapitel 9

Verallgemeinerung der Fragestellung

Um die Darstellung klarer und kompakter zu machen, wurden viele der in den letzten Abschnitten diskutierten Aspekte unter spezielleren Voraussetzungen diskutiert als nötig. Zum Beispiel stellte es sich heraus, daß die Methoden zur Ausführung von asynchronem Code auf SIMD-Rechnern nicht von den speziellen Charakteristiken von Suchalgorithmen abhängen. Deshalb soll hier auf einige weitere, möglicherweise interessante Verallgemeinerungen der Ergebnisse eingegangen werden.

9.1 MIMD-Rechner

Alle in Kapitel 6 und 8 beschriebenen Ideen sind auch auf MIMD-Rechnern einsetzbar. Durch den meist größeren Speicher und den individuellen Kontrollfluß sind MIMD-Rechner eigentlich sogar besser für den vorgestellten grobkörnigen Ansatz zur Parallelisierung geeignet.

Die Umverteilungsstrategien „Nachbarschaftskommunikation" (Abschnitt 8.2.3) und „Zufallspermutation" (Abschnitt 8.2.4) lassen sich auch asynchron — d.h. ohne Einschalten von gemeinsamen Umverteilungsphasen — realisieren. Dadurch wird einerseits Synchronisations-Overhead eingespart, außerdem wird das Kommunikationsnetzwerk gleichmäßiger belastet.

9.2 Kompliziertere Suchalgorithmen

Der hier betrachtete Ansatz zur Parallelisierung von Suchalgorithmen durch Aufteilen des Suchbaums ist nicht auf den oben beschriebenen einfachen Fall beschränkt. Durch entsprechende Erweiterung lassen sich auch andere Baumsuchalgorithmen parallelisieren.

9.2.1 Branch and Bound

Viele wichtige Probleme lassen sich als die Suche eines Suchbaumblattes mit maximaler Bewertung beschreiben, wobei zusätzlich eine „bounding"-Funktion b zur Verfügung steht, die eine obere Schranke für die Bewertung der Nachfolger eines inneren Knotens berechnet. Wichtige sequentielle Algorithmen zur Suche

nach dem Optimum sind eine Variante der Tiefensuche, die zuerst den vielversprechendsten Nachfolger betrachtet und „best-first“, wo jeweils der global beste Knoten expandiert wird.

Eine Parallelisierung bietet sich an, bei der einzelne Prozessoren eine Tiefensuche durchführen und Arbeitsverteilung durch Sortieren eingesetzt wird, um nach dem „best-first“ Prinzip Prozessoren mit möglichst erfolgversprechender neuer Arbeit zu versorgen. Zusätzlich kann durch „max“-Reduktion der von den Einzelprozessoren erreichten Ergebnisse entschieden werden, welche Prozessoren an überflüssigen Suchbaumzweigen arbeiten.

9.2.2 Abhängigkeiten zwischen Teilbäumen

Im bisher Gesagten war immer davon ausgegangen worden, daß das Suchergebnis nicht von der Struktur des Suchbaums abhängt. Zum Beispiel bei Und-Oder-Bäumen bzw. Minimax-Bäumen ist dies aber nicht mehr der Fall. Wird ein Suchbaum aufgespalten und auf mehrere Prozessoren verteilt, so müssen die Ergebnisse wieder zusammengeführt werden. Dadurch wird neben der Arbeitsverteilung zusätzliche Kommunikation nötig. Ein prinzipielles Problem ergibt sich dadurch aber nicht.

Viel einschneidender ist, daß oft starke Heuristiken existieren (z.B. Alpha-Beta), die die Auswertung ganzer Teilbäume überflüssig machen. Welche Teilbäume dies sind, kann aber erst mit Sicherheit entschieden werden, wenn andere Teilbäume ausgewertet sind. Um trotzdem eine hinreichende Parallelisierung zu erreichen, muß das Risiko eingegangen werden, überflüssige Arbeit zu tun. Allerdings sollte es möglich sein, überflüssige Arbeit vorzeitig zu beenden, sobald die nötige Information eintrifft.

Kapitel 10

Parallele Suche am Beispiel des FSSP

In diesem Kapitel sollen die allgemeinen Ergebnisse zu parallelen Suchalgorithmen aus den letzten Kapiteln auf das FSSP angewandt werden. Insbesondere geht es darum, zu demonstrieren, wie die z.T. abstrakten Ergebnisse und Ideen sich in der Praxis auswirken. Ausserdem werden die Erfahrungen dargestellt, die während der Implementierung mit paralleler Programmiermethodik im allgemeinen und der MasPar und MPL im besonderen gemacht wurden.

Zunächst beschreibt Abschnitt 10.1, wie aus dem anfangs vorliegenden sequentiellen Algorithmus ein paralleler Algorithmus entwickelt wurde. Das so entstandene Programm war dann noch so langsam, daß eine lange Reihe von Optimierungen nötig war, um es effizient zu machen (Abschnitt 10.2). Schließlich wird versucht, zu beurteilen, wie erfolgreich die Parallelisierung war (Abschnitt 10.3).

10.1 Parallelisierung

10.1.1 Der sequentielle Algorithmus

Zu Beginn der Arbeit lag ein rekursiver Suchalgorithmus vor, der im wesentlichen dem in Anhang B entsprach. Bevor zur parallelen Implementierung übergegangen wurde, sollten die in Abschnitt 3.5.3 beschriebenen Heuristiken zur Verkleinerung des Suchraums eingebaut werden. Dadurch wurde die Formulierung immer komplexer, weil Funktionen plötzlich komplizierte Rückgabewerte erhielten und weil immer mehr Abfragen nötig wurden, warum eine Schleife verlassen wurde. Deshalb wurde bereits für die sequentielle Implementierung eine nichtrekursive Formulierung und eine Realisierung des Kontrollflußes durch einen endlichen Automaten gewählt. Der Code sah dadurch zwar etwas merkwürdig aus, erwies sich aber als leichter handhabbar als die vermeintlich strukturierte Version. Durch den Wegfall der mehrfachen Abfrage von Bedingungen wurde außerdem die Effizienz verbessert.

Es entstand der in Abbildung 10.1 dargestellte Automat, der eine Erweiterung des Automaten aus Abbildung 7.2 ist:

advanceCol Führe Simulationsschritte innerhalb der momentanen Zeile von C aus. Ist die Zeile beendet, gehe zu advanceTime. Liegt ein Auswahlpunkt

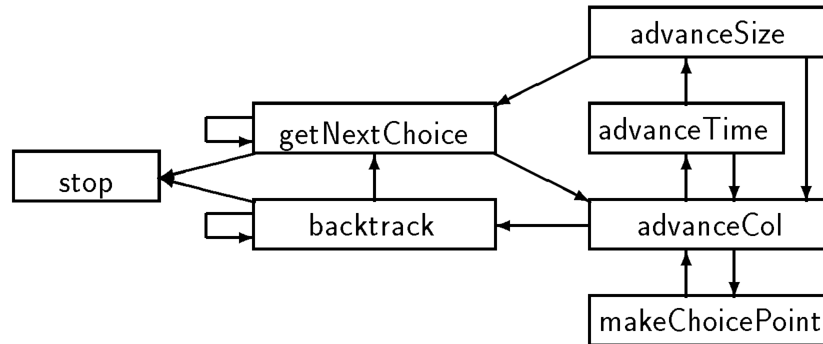

Abbildung 10.1: Automatendarstellung des sequentiellen FSSP-Algorithmus.

vor, verzweige zu makeChoicePoint und im Falle eines Fehler gehe zu backtrack.

advanceTime Neue Position wird die erste Zelle der nächsten Zeile von C. Ist die letzte Zeile fertig ($t > 2s - 2$), gehe zu advanceSize — sonst zurück zu advanceCol.

advanceSize Übergang zur nächsten Retinagröße. Ist die maximale Größe erreicht, gib die Lösung aus und verweige zu getNextChoice. Sonst zurück zu advanceCol.

makeChoicePoint Bringe Auswahlpunkt auf den Keller und wende DEFINE' an. Zurück zu advanceCol.

backtrack Entferne alle Auswahlpunkte vom Keller, die den aufgetretenen Fehler nicht beeinflussen, und mache die entsprechenden Anwendungen von DEFINE' rückgängig. Stop, wenn der Keller dabei leer wird. Sonst weiter mit getNextChoice.

getNextChoice Entferne alle Auswahlpunkte vom Keller, die keine Alternativen mehr offen lassen, und mache die entsprechenden Anwendungen von DEFINE' rückgängig. Stop, wenn der Keller dabei leer wird. Sonst versuche nächste Alternative und weiter mit advanceCol.

10.1.2 Änderungen am sequentiellen Algorithmus

Der in Abschnitt 10.1.1 beschriebene Algorithmus mußte an zwei Stellen geändert werden, um für einen SIMD-Rechner geeignet zu sein.

- Der Code für den Zustand backtrack stellte noch keine Elementaroperation dar, da er u.U. vom gerade betrachteten Auswahlpunkt bis zum Fehlerpunkt simulieren muß (Abschitt 3.5.3). Diese Schleife wurde aufgelöst, indem ein zusätzlicher Zustand simulateError eingeführt wurde, der solange einen Simulationsschritt durchführt, bis der Fehlerpunkt erreicht ist und je nach Ergebnis zurück zu backtrack oder direkt zu getNextChoice verzweigt. Dies ist ein Beispiel für die in Abschnitt 7.5.1 beschriebene Technik des Auflösens von Schleifen.

- Während der sequentielle Algorithmus den Automaten mit Hilfe von **GOTO** implementieren kann, muß der SIMD-Algorithmus auf die in Abbildung 7.3 eingeführte Implementierung durch eine Folge von **IF**-Abfragen zurückgreifen.

10.1.3 Parallele Konstrukte

Wie im MPL-User-Guide [12] empfohlen, wurde die Parallelisierung des Codes in kleinen Schritten durchgeführt. Der im User-Guide ebenfalls zu findende Rat, dies zu bewerkstelligen, indem der Code Stück für Stück vom Host auf die MasPar verschoben wird, erscheint allerdings in diesem Fall als nicht sinnvoll, da dadurch zwischendurch ein großer Aufwand an DPU-Host Kommunikation entstünde.

Stattdessen wurde ausgenutzt, daß MPL ANSI-C kompatibel ist, um in einem Schritt komplett auf MPL umzustellen (allerdings noch ohne Parallelisierung). Tatsächlich reichte es aus, das sequentielle ANSI-C Programm von `xxx.c` in `xxx.m` umzubenennen um ein auf der ACU ausführbares Programm zu erhalten.

In einem nächsten, fast ebenso trivialen Schritt wurden alle Variablen mit dem Attribut plural versehen, womit ein Programm entstand, das auf allen PE's parallel abläuft. (Ein/Ausgabe-Befehle müssen gesondert behandelt werden, spielen hier aber keine große Rolle.)

Dann wurde zusätzlicher Code eingeführt, der mit Hilfe des Symmetriebrechungsverfahrens aus Abschnitt 8.1 dafür sorgt, daß nicht alle PEs das gleiche tun. Dazu genügt es, den Zustand makeChoicePoint so zu modifizieren, daß er die in Algorithmus 8.1 beschriebenen Operationen an Stelle des Ablegens eines Auswahlpunktes auf dem Keller durchführt. Ist der Initialisierungsprozeß abgeschlossen, wird wieder die normale Version von makeChoicePoint verwendet.

Nach Hinzufügen einer Komponente, die die Ergebnisse einsammelt (im einfachsten Fall, wenn nur die Existenz einer Lösung interessiert, genügt ein globalor), ist damit bereits eine minimale parallele Version des Programms vorhanden.

10.2 Optimierungen

Die in Abschnitt 10.1 hergeleitete Version des parallelen Algorithmus ist noch so langsam, daß jeder Homecomputer schneller wäre. Eine lange Reihe von Verbesserungen war nötig, um das zu ändern. In Angriff genommen wurde jeweils die Optimierung, die mit dem geringsten Aufwand die größte Verbesserung versprach. Alle vorgenommenen Optimierungen würden für die sequentielle Version keine oder nur marginale Verbesserungen erbringen. Folglich bleibt der direkte Laufzeitvergleich zwischen sequentieller und paralleler Version sinnvoll.

Um die Darstellung konsistenter zu machen, sind die Optimierungen nach den Themen Arbeitsverteilung, Compileraspekte und SIMD-Optimierung geordnet. Einige Verbesserungen, die weniger interessant sind und sich keiner der anderen Optimierungen zuordnen lassen, werden nicht erwähnt. Die angegebenen Zahlen beziehen sich immer auf die Fragestellung, die Anzahl der Lösungen mit 4 Zuständen bis Größe 8 zu bestimmen.

10.2.1 Arbeitsumverteilung

Die erste Version der Arbeitsumverteilung bestand darin, mit Hilfe der adaptiven Arbeitsverteilung aus Abschnitt 8.2.2 Austauschphasen einzuleiten, in denen arbeitslose Prozessoren versuchen, Arbeit von ihren unmittelbaren Nachbarn im Norden, Osten, Süden oder Westen zu erhalten. Die Ausführungszeit betrug ca. 7 Minuten und war damit ungefähr mit der Zeit für eine SPARC 2 zu vergleichen.

Durch Benutzung des dreieckigen Rasters aus Abbildung 8.8 konnte diese Zeit auf $4\frac{1}{2}$ Minuten gedrückt werden. Aber immer noch war es wegen des in Abschnitt 8.2.3 beschriebenen Problems der Clusterbildung nicht möglich, eine hinreichende Zahl von Prozessoren langfristig an der Arbeit zu halten. Außerdem wurde ein großer Teil der Zeit für die Kommunikation selbst vertan.

Das wurde schlagartig anders, nachdem die Arbeitsverteilungsstrategie aus Abschnitt 8.2.4 eingesetzt wurde, die eine Routerverbindung auswürfelt. Nun war es möglich, etwa 3/4 der Przessoren gleichzeitig aktiv zu halten, ohne daß mehr als ca. 1/10 der Ausführungszeit für Kommunikation aufgewandt wurde.

Durch Sortieren der Arbeitslast entsprechend Abschnitt 8.2.5 wurde eine weitere Beschleunigung um ca. 12 % erreicht. Ca. 1/3 dieser Beschleunigung stellt sich erst ein, wenn der Parameter α für die adaptive Arbeitsverteilung (Abbildung 8.4) angepaßt wird. Als Idealwert für α ergab sich ca. 13 %.

Da das Sortieren wider Erwarten weniger Zeit beanspruchte, als die Kommunikation selbst, wurde diese noch optimiert, indem die zu übertragenden Daten in einem Paket zusammengefaßt werden, das dann in einem Rutsch übertragen werden kann. Dadurch war eine Beschleunigung um 6,7 % möglich. Selbst nach dieser Optimierung benötigt die eigentliche Datenübertragung mit 10 % der Gesamtrechenzeit noch doppelt soviel Zeit wie der Sortierprozeß. Die Umverteilung findet insgesamt 94 mal statt und dabei werden insgesamt 222510 Arbeitsübertragungen durchgeführt. Jeder Prozessor erhält also ca. 14 mal neue Arbeit. Das ist wenig genug, um die teure aber leistungsfähige Routerkommunikation tragbar zu machen, wäre aber zuviel für eine zentrale Arbeitsverteilungstechnik. Auf jede Umverteilungsphase kommen etwa 1500 Abfragen von Elementaroperationen.

10.2.2 Compiler- und Maschinenaspekte

Die Messungen auf sequentiellen Maschinen wurden mit hochoptimierenden C-Compilern gemacht. Der MPL-Compiler hat dagegen eine Reihe von Schwächen, die das Ergebnis zuungunsten des parallelen Programms beeinflussen. Trotzdem ist ein direkter Laufzeitvergleich fair, da es typisch für Nischenmärkte wie den Markt für massiv parallele Rechner ist, daß die zur Verfügung stehende Software nicht dem vollen Stand der Technik entspricht.

An einigen Stellen war es allerdings möglich, durch Handoptimierung Versäumnisse des Compilers nachzuholen. So konnte — durch konsequente Deklarierung *aller* lokalen Variablen als `register`, sowie durch Ersetzung der logischen Operatoren `&&` und `||` durch `&` bzw. `|` und duch Wahl von Datentypen möglichst geringer Länge — eine Beschleunigung von 91 % erreicht werden!

Weitere 5,5 % Verbesserung waren möglich, indem die Speicherabbildungsfunktion für das 3D-Array σ von Hand kodiert wurde. Der Compiler hatte wohl an Stelle einfacher Konstanten-Multiplikation mit 5, die sich ohne echte Multiplikation und komplett mit 8-bit Arithmetik durchführen läßt, vollwertige 16- oder

32-bit Multiplikationen produziert, die auf der MasPar recht teuer sind.

10.2.3 SIMD-Optimierung

Ändern der Elementaroperationen

Jede der Elementaroperationen advanceCol, advanceTime, advanceSize und simulateError müssen die Umgebung $\nu(s,t,c)$ bestimmen und dann auf den entsprechenden Eintrag der Übergangsfunktion σ zugreifen. Dabei entstehen recht hohe Kosten, da jeder Prozessor über vier individuell berechnete Adressen auf seinen lokalen Speicher zugreifen muß. Um das mehrfache Vorkommen dieser teuren Operation zu vermeiden, wurde deshalb eine neue Elementaroperation recomputeEntry geschaffen, die allein für diese Zugriffsoperation zuständig ist. Dabei wurde die in Abschnitt 7.5.3 beschriebene Technik zu Verschmelzung von Elementaroperationen ausprobiert. Durch diese Maßnahme ergab sich eine 19%-ige Beschleunigung.

Eine weitere Ähnlichkeit bestand zwischen den beiden Operationen backtrack und getNextChoice. Beide müssen Auswahlpunkte, die keine Wahlmöglichkeiten mehr offen lassen, vom Keller entfernen. Deshalb wurde eine neue Elementaroperation cleanStack eingeführt, die diese Aufgabe übernimmt. In diesem Fall wurde der Geschwindigkeitsgewinn durch den zusätzlichen Kontrollaufwand beinahe aufgezehrt.

Eine deutliche Beschleunigung (17 %) ergab sich allerdings durch Weglassen einer Heuristik aus backTrack. Die in Abbildung 3.8 dargestellte Heuristik, die Simulationsschritte spart, wenn Auswahlpunkt und Fehlerpunkt nah beeinanderliegen, bringt im sequentiellen Fall eine deutliche Beschleunigung. Da die dazu nötige Abfrage aber recht kompliziert ist und selten gebraucht wird, ist sie im SIMD-Fall nicht der Mühe wert.

Um (auf der MasPar knappen) lokalen Speicher zu sparen, wird das Raumzeitdiagramm C in sehr kompakter Form abgespeichert. Die dadurch relativ komplizierte Speicherabbildungsfunktion wird im sequentiellen Algorithmus an einer Stelle innerhalb der Hauptschleife benutzt, dort aber so selten aufgerufen, daß die entstehenden Kosten kaum ins Gewicht fallen. In der SIMD-Version wurde diese Operation aber doch so oft abgefragt, daß sich 6 % Beschleunigung erreichen ließen, indem der dort vorliegende Aufruf (der von recht spezieller Art ist) durch einen Tabellenzugriff ersetzt wurde.

Die Elementaroperation advanceTime wird zwar seltener gebraucht als advanceCol, ist dafür aber auch billiger. Formel 7.4 ergab nun, daß beide Operationen ungefähr gleich oft abgefragt werden sollten. Daraufhin bot es sich an, advanceTime ganz wegzulassen und als zusätzliche Bedingung in advanceCol einzubauen. Das Wegfallen des Kontrollaufwandes und die einfachere Operationsanordnung ergaben insgesamt eine 27%-ige Verbesserung.

Anordnung der Abfragen

Um die Anordnung der Abfragen zu optimieren, wurden die in Abschnitt 7.3 und 7.4 diskutierten Methoden eingesetzt. Die Kosten c_i wurden mit Hilfe des MP-PE Profilers [11] und die Auftretenswahrscheinlichkeiten p_i durch entsprechende Zähler im Programm gemessen. Tabelle 10.2 zeigt die Messergebnisse, und die sich daraus ergebenden Abfragehäufigkeiten gemäß Formel 7.4.

Operation	p_i [%]	c_i [ticks]	f_i [%]
advanceCol	28,8	343	20,7
recomputeEntry	37,4	215	29,7
makeChoicePoint	3,5	256	8,3
backtrack	3,7	700	5,2
getNextChoice	2,3	352	5,8
cleanTos	8,0	399	10,1
simulateError	16,1	241	18,5
advanceSize	0,2	372	1,7

Abbildung 10.2: Abfragefäufigkeiten für FSSP-Operationen.

Mit Hilfe der Greedy-Algorithmen aus Abschnitt 7.4.2 wurden Versatzstücke für mögliche Abfragefolgen generiert. Aus diesen wurde dann manuell eine Abfragefolge der Länge 18 konstruiert:

[advanceCol, makeChoicePoint, simulateError, recomputeEntry,
advanceCol, simulateError, cleanTos, getNextChoice,
recomputeEntry, advanceCol, simulateError, recomputeEntry,
advanceCol, simulateError, advanceSize, cleanTos,
backtrack, recomputeEntry]

Die Abfragefolge weicht nur an einer Stelle von den Häufigkeiten ab, die das analytische Modell vorschlägt. Da recomputEntry grundsätzlich Nachfolgezustand von getNextChoice und simulateError ist, ist es sinnvoll, diese drei Operationen gemeinsam abzufragen. Deshalb wurde die für simulateError errechnete Abfragehäufigkeit von $3\frac{1}{3}$ zu 4 aufgerundet, und recomputeEntry wird nur viermal statt fünfmal abgefragt. Alle anderen Versuche, durch Abweichen von der errechneten Häufigkeit Verbesserungen zu erzielen, führten zu nichts. Dies spricht dafür, daß Formel 7.4 auch für nichtprobabilistische Abfragefolgen sinnvoll ist, solange Reihenfolgeaspekte keine zu große Rolle spielen.

Gegenüber einer „naiven“ Abfragefolge, bei der jede Operation nur einmal vorkommt, ergibt sich eine Beschleunigung von ca. 63 %. Gegenüber einer reihenfolgeoptimierten Abfragefolge minimaler Länge ist immerhin noch eine Beschleunigung von ca. 50 % zu verzeichnen. Einfache Optimierungsansätze, wie verdoppeln der wichtigsten Abfragen, bloße Benutzung von Formel 7.4 ohne Benutzung von Reihenfolgeinformation, oder Benutzung eines der „greedy“-Algorithmen aus Abschnitt 7.4.2, führten nur zu Beschleunigungen von 10–20 %.

Die durch die iterative Kostenabschätzung aus Abschnitt 7.4.1 vorausgesagte Beschleunigung lag jeweils um ca. zwei Prozentpunkte neben den Messungen. Die Modellierung des asynchronen Kontrollflusses durch Übergangswahrscheinlichkeiten scheint also angemessen zu sein. Die Kostenfunktion für die probabilistische Abfragestrategie hat eine Beschleunigung von ca. 59 % vorausgesagt, was recht nah am tatsächlich Erreichten liegt. Wie erwartet, waren die für probabilistische Abfrage errechneten absoluten Kosten ungefähr doppelt so hoch wie die für deterministische Abfrage beobachteten Werte.

Alle SIMD-bezogenene Optimierungen zusammen haben eine Beschleunigung um mehr als das 3 fache bewirkt.

10.3 Beurteilung der Leistung

Alle Optimierungen zusammen haben eine Beschleunigung um das 38 fache der ursprünglichen Leistung erbracht. Interessant ist dabei, daß nur eine der Einzeloptimierungen (der Umstieg auf Routerkommunikation) eine Beschleunigung von mehr als 100 % erbracht hat. Ansonsten waren es viele kleine Verbesserungen, die sich zu einer recht beachtlichen Zahl multiplizieren.

Die sich daraus ergebende Gesamtleistung ist wohl höher als alles, was gegenwärtig mit sequentiellen Rechnern erreichbar scheint.[1] Damit ist zumindest ein Kriterium für den Erfolg einer parallelen Implementierung erfüllt. Vergleicht man den Preis einer MasPar mit dem einer entsprechenden Zahl von Workstations (z.B. 38x SPARC 2) kommt man sogar zu einem akzeptablen Preis-Leistungsverhältnis.

Speedupmessungen sind nicht ganz einfach, da Routerkommunikation und Reduktionsoperationen auf einer 16K-MasPar wahrscheinlich langsamer als auf kleineren Maschinen sind, selbst wenn nur eine geringe Prozessorzahl verwendet wird. Außerdem hat sich herausgestellt, daß der Parameter α der adaptiven Arbeitsverteilung (Abschnitt 8.2.2) bei geringerer Prozessorzahl etwas größer gewählt werden sollte. Tabelle 10.3 zeigt die Laufzeiten für verschiedene Prozessorzahlen zusammen mit dem Wert für α mit dem die Messungen durchgeführt wurden. Trotz der erwähnten Ungenauigkeiten wird deutlich, daß es einen fast linearen Speedup gibt. Der Effizienzvorteil der kleineren Prozessorzahlen kann nämlich im wesentlichen nur von der weniger häufig nötigen Arbeitsverteilung zehren.

nproc	T [s]	α [%]
16384	12.7	87
8192	22.1	90
4096	40.7	93
2048	76.8	97
1024	148.4	97

Abbildung 10.3: Laufzeitmessungen für verschiedene Prozessorzahlen.

[1] Zwei der schnellsten sequentiellen Rechner, die auf dem Campus ausfindig gemacht werden konnten (DEC-Alpha und SGI Indigo R4000), haben beide fast 12x länger gebraucht als die MasPar (siehe auch Anhang C).

Kapitel 11

Zusammenfassung und Ausblick

In den letzten Kapiteln wurde eine große Zahl positiver wie negativer Ergebnisse beschrieben, von denen hier noch einmal die wichtigsten herausgehoben werden sollen. Außerdem wird versucht, auf einige offene Fragen hinzuweisen, die vielleicht eine nähere Untersuchung wert sind.

11.1 Polyautomaten

11.1.1 Das FSSP

Greifbares Ergebnis in Bezug auf das FSSP ist, daß keine Lösung mit 4 Zuständen existiert. Im gleichlautenden Ergebnis von Balzer wurde ein Fehler entdeckt und behoben.

Die Hoffnung, durch systematische Suche die Frage nach Existenz einer 5-Zustandslösung zu klären, ist eher in weitere Ferne gerückt als vorher, da der Fehler in Balzers Algorithmus die Leistungen des Suchalgorithmus besser aussehen läßt als sie sind. Außerdem ist ein weiterer Versuch gescheitert, bessere und intelligentere Suchalgorithmen zu bauen. Auch die durch Parallelisierung erreichbare Leistungssteigerung ist nur ein Tropfen auf den heißen Stein. Es gibt zwar einige plausible Annahmen darüber, wie eine mögliche Lösung aussehen könnte, aber selbst die Hinzunahme weniger plausibler Annahmen reichte bei weitem nicht aus, den Suchraum hinreichend einzuschränken.

11.1.2 Trellisautomaten

Mit Hilfe einer Variante des Suchalgorithmus für das FSSP gelang es, die 16 einfachsten Übergangstabellen für homogene Trellisautomaten zu finden, die die Sprache $\{ww^R | w \in \{\mathtt{a}, \mathtt{b}\}^+\}$ akzeptieren. Jede dieser Lösungen benutzt ein akzeptierendes und 4 ablehnende Symbole. Die Korrektheit einer dieser Lösungen wurde bewiesen und die der anderen für alle Eingabeworte mit Länge ≤ 20 überprüft.

Die Akzeption von $\{ww | w \in \{\mathtt{a}, \mathtt{b}\}^+\}$ ist zumindest insoweit schwieriger, als daß keine Lösung existiert, die weniger als 7 Symbole benutzt. Lösungen mit 7 Symbolen müßten genau zwei ablehnende Symbole vorsehen.

Suche nach Lösungen mit mehr als 7 Symbolen scheitert wieder an der schieren Größe des Suchraums. Trotzdem könnte ein Suchalgorithmus helfen, weitere Ergebnisse zu erzielen. So könnte man beobachten, welche Wörter besonders

oft zum Scheitern von Teillösungen führen oder besonders tiefes Backtracking auslösen. Analyse dieser Wörter könnte helfen, Beweise ähnlich dem von Satz 3 zu führen, die z.B. zeigen könnten, daß keine Lösung mit weniger als 3 akzeptierenden Symbolen existiert. Im günstigsten Fall könnte sich dieser Beweis für eine beliebige feste Zahl von akzeptierenden Zuständen verallgemeinern lassen, womit bewiesen wäre, daß $\{ww|w \in \{\mathtt{a}, \mathtt{b}\}^+\}$ von keinem homogenen Trellisautomaten akzeptiert werden kann.

11.1.3 Verallgemeinerung

Im Grunde genommen ist die Suche nach Übergangstabellen für Polyautomaten, die eine gegebene Problemspezifikation erfüllen, nur ein Spezialfall des alten Wunschtraums rein deklarativ programmieren zu können. Diese Idee hat in der KI und im Programmiersprachendesign immer mal wieder eine Rolle gespielt, ist aber auch immer wieder gescheitert, da der resultierende Suchraum für nichttriviale Probleme viel zu groß ist. Außerdem ist das Problem, einen Lösungskandidaten zu überprüfen, i.allg. unentscheidbar. (So war es ja z.B. im Fall der gefundenen Trellisautomaten nötig, einen Beweis der Korrektheit der Lösungskandidaten nachzuschieben.)

Ein Beitrag, den die Erfahrungen mit dem FSSP zu dieser alten Diskussion leisten können, ist die Erfahrung, daß einfache, schnelle Algorithmen oft bessere Ergebnisse erzielen, als ausgefeilte Systeme mit KI-Einfärbung.

Dennoch kann Suche nach Programmen bei sorgfältiger Auswahl der Problemstellung ein nützlicher Ansatz sein. Eine bekannte Anwendung ist z.B. der GNU-Superoptimizer. Dabei handelt es sich um ein System, das Maschinenbefehlssequenzen sucht, die eine gegebene Operation möglichst schnell durchführen. Er kann eingesetzt werden, um die Versatzstücke zu optimieren, mit deren Hilfe der Codegenerator eines Compilers den Code zusammensetzt. Es ist nicht auszuschließen, daß die Ergebnisse dieser Arbeit (insbesondere die Erfahrungen mit der Parallelisierung) helfen können, solche Superoptimizer zu verbessern.

11.2 Parallele Suche

Der Ansatz, Suchbäume in möglichst große Stücke zu unterteilen und diese parallel zu durchsuchen, hat sich — auch für massiv parallele Rechner — bewährt. Falls die Äste des Suchbaums hinreichend unabhängig sind, läßt sich ein annähernd linearer Speedup erreichen. Eine Schlüsselrolle spielt dabei die Arbeitsverteilung.

- Einfache Nachbarschaftskommunikation ist auf Verbindungsnetzen mit großem Durchmesser u.U. nicht ausreichend, um die Prozessoren auszulasten.

- Auswürfeln von Zufallsverbindungsmustern ist ein einfaches und robustes Verfahren.

- Je nach Problemstellung lassen sich gute Ergebnisse erzielen, indem Kandidaten für die Arbeitsumverteilung nach einer Bewertungsfunktion sortiert werden. Wird diese Sortierung parallel — während einer für alle Prozessoren gemeinsamen Umverteilungsphase — durchgeführt, kann der durch

einen zentralen Arbeitsverteilprozessor drohende Flaschenhals vermieden werden.

- Indem die Arbeitsverteilung nur angeworfen wird, wenn hinreichend viele Prozessoren arbeitslos sind, kann die Balance zwischen Suche und Umverteilung eingehalten werden.

- Initiale Arbeitsverteilung kommt im Prinzip ohne Kommunikation aus.

Da der FSSP-Algorithmus von recht einfachem Typus ist, wäre es interessant, kompliziertere Algorithmen wie „branch and bound“, „alpha-beta“,... zu untersuchen. Insbesondere ist zu klären, wie stark der Suchraum gegenüber dem sequentiellen Algorithmus wächst und wieviel zusätzliche Kommunikation durch Abhängigkeiten zwischen Suchbaumästen nötig wird.

Der eigentliche Suchalgorithmus und die parallelen Komponenten lassen sich weitgehend trennen, deshalb bestände eine weitere interessante Fragestellung darin, ein „Framework“ für parallele Suchalgorithmen zu erstellen, dessen Benutzer nur noch die problemabhängigen Aspekte spezifizieren muß. Da die Erstellung effizienter paralleler Programme sich als recht aufwendig herausgestellt hat, könnte das Framework helfen, solche Anwendungen auch durch Personal mit weniger Spezialkenntnissen erstellen zu lassen.

Das Beispiel des FSSP zeigt aber auch, daß auch eine erfolgreiche Parallelisierung ein Grundproblem von Suchalgorithmen kaum mildert. Da die Algorithmen im allgemeinen exponentiell sind, wird selbst eine vieltausendfache Beschleunigung oft nur geringe Auswirkungen auf die erreichbare Suchtiefe haben. Parallelisierung wird sich deshalb vor allem in folgenden Fällen anbieten:

- Wenn auch kleine Verbesserungen den Aufwand wert erscheinen lassen. Beispiele wären Produktionsplanungssysteme, bei denen Maschinenstandzeiten u.U. erhebliche Kosten verursachen oder Schachprogramme, deren Leistung eine Art Prestigeobjekt sind.

- Wenn die Problemgröße festliegt, aber kürzere Antwortzeiten gewünscht werden, um z.B. Echtzeitanforderungen gerecht zu werden.

- Bei Problemen mit relativ geringer Zahl von Knoten aber hohen Kosten pro Knoten. In diesem Fall kann die Parallelisierung zu einer relativ deutlichen Vergrößerung der handhabbaren Problemgröße führen.

11.3 SIMD-Aspekte

Das Verhalten eines MIMD-Rechners läßt sich auf einem SIMD-Rechner mit Hilfe einer Folge von Abfragen von Elementaroperationen emulieren. Die Elementaroperationen lassen sich auf recht systematischem Wege finden. Die wichtigsten Maßnahmen sind die Elimination von Rekursion und das Auflösen von Schleifen. Dieser Prozeß läßt sich als Auswahl eines speziell auf das gegebene Problem zugeschnittenen abstrakten Maschinenbefehlsatzes deuten, der vom SIMD-Rechner interpretiert wird. Der durch die Emulation entstehende Overhead kann deutlich vermindert werden:

- Sind durchschnittliche Auftretenswahrscheinlichkeit und Kosten der Elementaroperationen bekannt, so läßt sich die optimale Abfragehäufigkeit analytisch ausrechnen. Sie ist proportional zur Wurzel des Verhältnisses von Auftretenswahrscheinlichkeit zu Kosten.
- Durch sorgfältiges Anordnen der Abfragen kann der Durchsatz erhöht werden. Sind die Übergangswahrscheinlichkeiten zwischen Elementaroperationen bekannt, so kann die Effektivität von Abfragefolgen mit Hilfe eines einfachen iterativen Verfahrens beurteilt werden.
- Durch Umformen des Satzes von Elementaroperationen kann die Effizienz gesteigert werden. Folgende Umformungen haben sich bewährt:
  - Zusammenfassung ähnlicher Operationen.
  - Wegrationalisieren selten benötigter Spezialfälle.
  - Optimierung von Code, auch wenn er im asynchronen Kontrollfluß selten benötigt wird.
  - Aufnehmen billiger Elementaroperationen in aufrufende Operationen.
  - Abspalten und Zusammenfassen gemeinsamer Teilausdrücke von Elementaroperationen.
  - Abspalten selten benötigter, teurer Zweige in **IF**-Konstrukten.

Obwohl es sich herausgestellt hat, daß die Darstellung eines Algorithmus mit Hilfe von Elementaroperationen auch zur Verständlichkeit beitragen kann, wäre es ein interessantes Thema, Compiler zu entwickeln, die asynchronen Code für SIMD-Rechner übersetzen, indem sie die Auswahl der Elementaroperationen automatisch durchführen.

Ein anderer Bereich, in dem Computerunterstützung sinnvoll wäre, ist die Bestimmung effizienter Abfragefolgen. Ideal wäre eine einfache Heuristik, die gute Abfragefolgen produziert (unter Zuhilfenahme von Profiling Information). Für sehr zeitkritische Anwendungen könnten sich auch Verfahren zur kombinatorischen Optimierung lohnen. Von (zumindest theoretischem) Interesse wäre auch, ob die Optimierung von Abfragefolgen tatsächlich $\mathcal{NP}$-vollständig ist.

Anhang A

Trellisautomaten für $\{ww^R | w \in \{a, b\}^+\}$

Einziger akzeptierender Zustand ist jeweils **x**. Fragezeichen weisen darauf hin, daß eine Produktion nicht vorkommt. Die Lösungen wurden für alle Eingabeworte mit Länge bis 20 getestet. Die Korrektheit von Lösung 1 wurde bewiesen.

```
Solution: 1
AA->x AB->C AC->C AD->x Ax->? BA->C BB->x BC->x BD->C Bx->? CA->D
CB->C CC->C CD->D Cx->A DA->C DB->D DC->D DD->C Dx->B xA->? xB->?
xC->D xD->C xx->B

Solution: 2
AA->x AB->C AC->x AD->? Ax->? BA->C BB->x BC->C BD->D Bx->? CA->D
CB->? CC->D CD->C Cx->A DA->C DB->x DC->C DD->D Dx->D xA->? xB->?
xC->C xD->B xx->B

Solution: 3
AA->x AB->C AC->x AD->C Ax->? BA->C BB->x BC->C BD->x Bx->? CA->D
CB->C CC->D CD->C Cx->A DA->C DB->D DC->C DD->D Dx->B xA->? xB->?
xC->C xD->D xx->B

Solution: 4
AA->x AB->C AC->x AD->C Ax->? BA->C BB->x BC->C BD->x Bx->B CA->B
CB->C CC->D CD->C Cx->A DA->C DB->D DC->C DD->D Dx->B xA->? xB->D
xC->C xD->D xx->B

Solution: 5
AA->x AB->C AC->D AD->C Ax->? BA->C BB->x BC->C BD->D Bx->? CA->x
CB->C CC->D CD->C Cx->C DA->C DB->x DC->C DD->D Dx->D xA->? xB->?
xC->A xD->B xx->B

Solution: 6
AA->x AB->C AC->D AD->C Ax->? BA->C BB->x BC->? BD->x Bx->? CA->x
CB->C CC->D CD->C Cx->C DA->? DB->D DC->C DD->D Dx->B xA->? xB->?
xC->A xD->D xx->B

Solution: 7
```

```
AA->x AB->C AC->B AD->C Ax->? BA->C BB->x BC->C BD->D Bx->D CA->x
CB->C CC->D CD->C Cx->C DA->C DB->x DC->C DD->D Dx->D xA->? xB->B
xC->A xD->B xx->B

Solution: 8
AA->x AB->C AC->D AD->C Ax->? BA->C BB->x BC->C BD->D Bx->? CA->C
CB->x CC->C CD->D Cx->D DA->x DB->C DC->D DD->C Dx->C xA->? xB->?
xC->A xD->B xx->B

Solution: 9
AA->x AB->C AC->x AD->C Ax->? BA->C BB->x BC->C BD->x Bx->? CA->C
CB->D CC->C CD->D Cx->B DA->D DB->C DC->D DD->C Dx->A xA->? xB->?
xC->D xD->C xx->A

Solution: 10
AA->x AB->C AC->C AD->D Ax->? BA->C BB->x BC->x BD->? Bx->? CA->?
CB->D CC->D CD->C Cx->B DA->x DB->C DC->C DD->D Dx->D xA->? xB->?
xC->C xD->A xx->A

Solution: 11
AA->x AB->C AC->C AD->x Ax->? BA->C BB->x BC->x BD->C Bx->? CA->C
CB->D CC->D CD->C Cx->B DA->D DB->C DC->C DD->D Dx->A xA->? xB->?
xC->C xD->D xx->A

Solution: 12
AA->x AB->C AC->C AD->x Ax->A BA->C BB->x BC->x BD->C Bx->? CA->C
CB->A CC->D CD->C Cx->B DA->D DB->C DC->C DD->D Dx->A xA->D xB->?
xC->C xD->D xx->A

Solution: 13
AA->x AB->C AC->C AD->D Ax->? BA->C BB->x BC->D BD->C Bx->? CA->C
CB->x CC->D CD->C Cx->C DA->x DB->C DC->C DD->D Dx->D xA->? xB->?
xC->B xD->A xx->A

Solution: 14
AA->x AB->C AC->? AD->x Ax->? BA->C BB->x BC->D BD->C Bx->? CA->C
CB->x CC->D CD->C Cx->C DA->D DB->? DC->C DD->D Dx->A xA->? xB->?
xC->B xD->D xx->A

Solution: 15
AA->x AB->C AC->C AD->D Ax->D BA->C BB->x BC->A BD->C Bx->? CA->C
CB->x CC->D CD->C Cx->C DA->x DB->C DC->C DD->D Dx->D xA->A xB->?
xC->B xD->A xx->A

Solution: 16
AA->x AB->C AC->C AD->D Ax->? BA->C BB->x BC->D BD->C Bx->? CA->x
CB->C CC->C CD->D Cx->D DA->C DB->x DC->D DD->C Dx->C xA->? xB->?
xC->B xD->A xx->A
```

Anhang B

C-Implementation von Algorithmus 3.5

Das nachstehende Programm sucht die Anzahl der Lösungen zum FSSP mit N Zuständen bis Größe M. Es ist Grundlage des Beweises für den Fall $N = 4$. Das Programm lehnt sich eng an Algorithmus 3.5 an. C, *pattern* und σ werden mit Hilfe dreidimensionaler Felder implementiert. Die Regeln SIMULATE und DEFINE' tauchen nicht mehr direkt auf, sondern sind direkt als Datenstrukturmanipulationen implementiert. Eine zusätzliche Variable `entry` speichert einen Zeiger auf $\sigma(\nu(s,t,c))$. Außerdem werden aus Effizienzgründen keine rekursiven Aufrufe gemacht, solange nur simuliert wird. Statt die Existenz einer Lösung zu überprüfen, wird die Anzahl der Lösungen berechnet.

```
/**********************************************************/
/* Search for a n-state Solution of the                   */
/* Firing Squad Synchronisation Problem                   */
/* minimal version                                        */
/**********************************************************/

#include <stdio.h>

/* Number of States */
#define N 4

/* Maximum size of retina to be considered */
#define M 9

/* States */
typedef char State;
#define ANY      0
#define Z0       1
#define GENERAL  2
#define BORDER   N
#define FIRE     (N+1)

/* pattern and C */
typedef struct {State pattern, history;} Trace;
Trace trace[M+1][2*M][M+2];
```

```
/* Transition function */
State next[N+1][N+1][N+1];
#define Next(i, j, k) (next[(int)(i)][(int)(j)][(int)(k)])

/* number of solutions */
long solutionCount;


/* Initialize C and Pattern */
void initTrace(void)
{  int s, t, c;

   for(s=2;  s<=M;  s++){
      /* initial condition */
      trace[s][0][1].history = GENERAL;
      for(c=2;  c<=s;  c++) trace[s][0][c].history=Z0;
      /* mark borders */
      for(t=0;  t<=2*s-2;  t++)
         trace[s][t][0].history = trace[s][t][s+1].history = BORDER;
      /* mark firing time */
      for(c=1;  c<=s;  c++) trace[s][2*s-2][c].pattern = FIRE;
   }
}

/* Find number of solutions given current position (s,t,c) and nu(s,t,c) */
void search(int s, int t, int c, State *entry)
{  while(*entry != ANY){                          /* transition already known */
      trace[s][t][c].history = *entry;                  /* perform simulation */
      if((trace[s][t][c].pattern != ANY || *entry == FIRE) &&
          trace[s][t][c].pattern != *entry){return;}     /* inconsistent */
      else{                      c++;                          /* next column */
         if(c > s){       c=1; t++;                         /* next time step */
            if(t > 2*s-2){t=1; s++;                             /* next size */
               if(s > M){solutionCount++; return;}}}                /* done */
         entry = &Next(trace[s][t-1][c-1].history,              /* next entry */
                       trace[s][t-1][c  ].history,
                       trace[s][t-1][c+1].history);
      }
   }
   if(trace[s][t][c].pattern != ANY){            /* forced new transition */
      *entry = trace[s][t][c].pattern;
      search(s, t, c, entry);
   }else{                                          /* free new transition */
      for(*entry = N-1;  *entry != ANY;  (*entry)--){
         search(s, t, c, entry);
      }
   }
   *entry = ANY; /* backtrack */
}


void main(void)
```

```
{  Next(Z0, Z0, Z0    ) = Next(Z0, Z0, BORDER) = Z0;
   initTrace();
   search(2, 1, 1, &(Next(BORDER, GENERAL, Z0)));
   printf("%ld solutions found\n", solutionCount);
}
```

Anhang C

Laufzeiten auf sequentiellen Rechnern

Die folgenden Messungen wurden mit einer optimierten Version des FSSP-Suchalgorithmus durchgeführt. Gesucht wurde jeweils nach Lösungen mit 4 Zuständen bis Größe 8. Gemessen wurde die verbrauchte „user time“. Für die Durchführung der Messungen möchte ich Christian von Roques, Roger Buthenuth und Markus Mock danken.

Maschine	Zeit [s]
T800 30 MHz	2637
Sun3	2093.4
DECstation 2100	904.5
DECstation 3100	671.8
Sun4/20 sparc I-SLC	617.3
Sun4/40 sparc I-IPC	499.1
Iris indigo R3000	362.612
Sun4/75 sparc II	286.8
i486/50 Linux	255.51
Sun10 cc -O	186.9
Sun10 gcc -O6 ...	181.7
hp 9000/710	169.57
hp 9000/720	168.66
Iris indigo R4000	127.050
DEC alpha, 150 MHz	126.77

Glossar

$\perp$ Nichtdefinierter Funktionswert. Hier undefinierter Zustand

$\#$ Randzustand

$\nu(s,t,c)$ Das Zustandstupel (a,b,c), das den Wert der Zelle $C_t^s(c)$ bestimmt

σ Lokale Übergangsfunktion eines zellularen Automaten

$abc \rightarrow d$ Eintrag in σ ($\sigma(a,b,c) = d$)

$l \Longrightarrow r \mid b$ Regel, die l in r transformieren kann, wenn Bedingung b erfüllt ist

A Menge der Zustände eines zellularen Automaten

a_i Abfrage der Elementaroperation o_i

Abfrage Codestück der Form **IF** b **THEN** Elementaroperation

Abfragefolge Liste $[l_1, \ldots, l_m]$ von Indizes, die das Codestück $a_{l_1}; \ldots; a_{l_m}$ charakterisiert

ACU Array Control Unit — Steuereinheit der MasPar

Anfangskonfiguration Von außen vorgegebene Konfiguration eines zellularen Automaten zum Zeitpunkt 0

Auswahlpunkt Position (s,t,c) in C an der mehrere Instanzen von DEFINE' anwendbar sind

c Nummer einer Zelle eines zellularen Automaten

c_i Kosten pro Ausführung von Elementaroperation i

C Stand der Simulation für Retinae der Größen 2–M

C^s Raumzeitdiagramm für eine Retina der Größe s

C_t^s Konfiguration eines zellularen Automaten mit Retinagröße s zum Zeitpunkt t

$C_t^s(c)$ Zustand der c-ten Zelle in Retina der Größe s zum Zeitpunkt t

Contour Menge aller aktuellen Auswahlpunkte

DEFINE Regel, die definiert, wann ein Eintrag in σ möglich ist

DEFINE' Verschärfte Version von DEFINE

Elementaroperation Codestück, das keine von pluralen Bedingungen abhängigen Schleifen enthält

enumerate MPL Bibliotheksfunktion, die aktive PE's abzählt

f_i Häufigkeit der Abfrage von Elementaroperation i

Fehlerpunkt Position (s, t, c) in C auf die keine Regel anwendbar ist, obwohl $\nu(s, t, c)$ bekannt ist

`F` Feuerzustand

FSSP Firing Squad Synchronisation Problem

`G` Generalszustand

globalor MPL-Konstrukt, das globale Oder-Verknüpfung durchführt

iproc Prozessornummer $(0 \ldots \mathsf{nproc} - 1)$ in MPL

Konfiguration Zustandsvektor der Zellen einer Retina

M Maximale betrachtete Retinagröße

MasPar Hersteller der MasPar MP-1, MP-2 Rechner

MPL MasPar Programming Language, ANSI-C mit datenparallelen Erweiterungen

N Zahl der Zustände eines zellularen Automaten ohne Randzustand $(M = |A| - 1)$

nproc Gesamtzahl verfügbarer PEs

o_i Elementaroperation i

p_i Wahrscheinlichkeit des Auftretens von Elementaroperation i

p_{ij} Wahrscheinlichkeit, daß Elementaroperation o_i auf o_j folgt

pattern A priori vorgegebene Einträge in C

PE Processing Element

plural Attribut eines MPL-Objekts, das auf jeder PE angelegt wird

POS Menge aller für C und *pattern* sinnvollen Positionen (s, t, c)

Raumzeitdiagramm Darstellung eines Simulationsablaufs durch Auftragen der Konfigurationen eines zellularen Automaten über die Zeit

reduceAdd/Max/Min MPL Bibliotheksfunktion, die Summe, Maximum bzw. Minimum eines plural-Objekts bestimmt

Regelanwendung Regel $l \Longrightarrow r \mid b$ kann auf Position (s, t, c) *angewandt* werden, wenn (s, t, c) in b vorkommt und b erfüllt ist

Retina Menge der Zellen eines zellularen Automaten

s Retinagröße

Router Komponente eines Parallelrechners, die Kommunikation zwischen beliebigen Partnern ermöglicht

router[a].x MPL Konstrukt, das Zugriff auf Datenelement x von Prozessor a ermöglicht

SIMULATE Regel, die definiert, wann ein Simulationsschritt möglich ist

singular Attribut eines MPL-Objekts, das nur auf der ACU angelegt wird

t Zeitpunkt

z_0 Ruhezustand eines zellularen Automaten

Literaturverzeichnis

[1] R. Balzer. *Studies Concerning Minimal Time Solutions to the Firing Squad Synchronisation Problem.* Doctoral Thesis, Carnegie Institute of Technology, 1966.

[2] R. Balzer. *An 8-state minimal time solution to the firing squad synchronisation problem.* Information and Control 10, 22–42, 1967.

[3] K. Culik II, J. Gruska, A. Salomaa. *Systolic trellis automata II.* International Journal of Computer Mathematics 16, 3–22, 1984.

[4] F. Dehne, A. Rau-Chaplin. *Parallel branch and bound on finegrained hypercube multiprocessors.* Parallel Computing 15, 201–209, 1990.

[5] H. Eldermann. *Betrachtungen zu Erkennbarkeit der Sprache $L = \{ww|w \in \{0,1\}^+\}$ mittels eines speziellen Zellularautomaten in Realzeit.* Studienarbeit, TU Braunschweig, November 1989.

[6] H.D. Gerken. *Über Synchronisationsprobleme bei Zellularautomaten.* Diplomarbeit, TU Braunschweig, April 1987.

[7] C. Gerthsen, Kneser, Vogel. *Physik.* 15. Auflage, Springer Verlag, 1986.

[8] B.W. Kernighan, D.M. Ritchie. *Programmieren in C, 2. Ausgabe.* Hanser Verlag, 1990.

[9] M. S. Littmann, C.D. Metcalf. *An Exploration of Asynchronous Data-Parallelism.* Technical report, /pub/milan.tex.Z über ftp-site cag.lcs.mit.edu, 1990

[10] J. Mazoyer. *A six-state minimal time solution to the firing squad synchronisation problem.* Theoretical Computer Science 50, 183-238, 1987.

[11] *MPPE User Guide.* Maspar Corporation, 1992.

[12] *MPL User Guide.* Maspar Corporation, 1992.

[13] *MPL Reference Manual.* Maspar Corporation, 1992.

[14] H.J. Raasch, M. Kutrib. *Betrachtungen zu Erkennbarkeit der Sprache $L = \{ww|w \in \{0,1\}^+\}$ mittels eines speziellen Zellularautomaten in Realzeit.* Studienarbeit, TU Braunschweig, November 1989.

[15] V.N. Rao, V. Kumar. *Parallel depth first search. Part I. Implementation.* International Journal of Parallel Programming, 16(6), 501–519, 1987.

[16] V.N. Rao, V. Kumar. *Parallel depth first search. Part II. Analysis*. International Journal of Parallel Programming, 16(6), 501–519, 1987.

[17] D. M. Sharnoff. *Catalog of Compilers, Interpreters, and other Language Tools*. News group comp.compilers, Mai 1993.

[18] T. Umland. Persönliche Mitteilung, Karlsruhe 1993.

[19] R. Vollmar. *Algorithmen in Zellularautomaten*. Teubner, 1979.

[20] R. Vollmar, Th. Worsch. *Einführung in die Parallelverarbeitung — Modelle und Maschinen —*. Vorlesungsskriptum, Universität Karlsruhe, WS 1992/93.

[21] A. Waksman. *An optimum solution to the firing squad synchronisation problem*. Information and Control 9, 66–78, 1966.