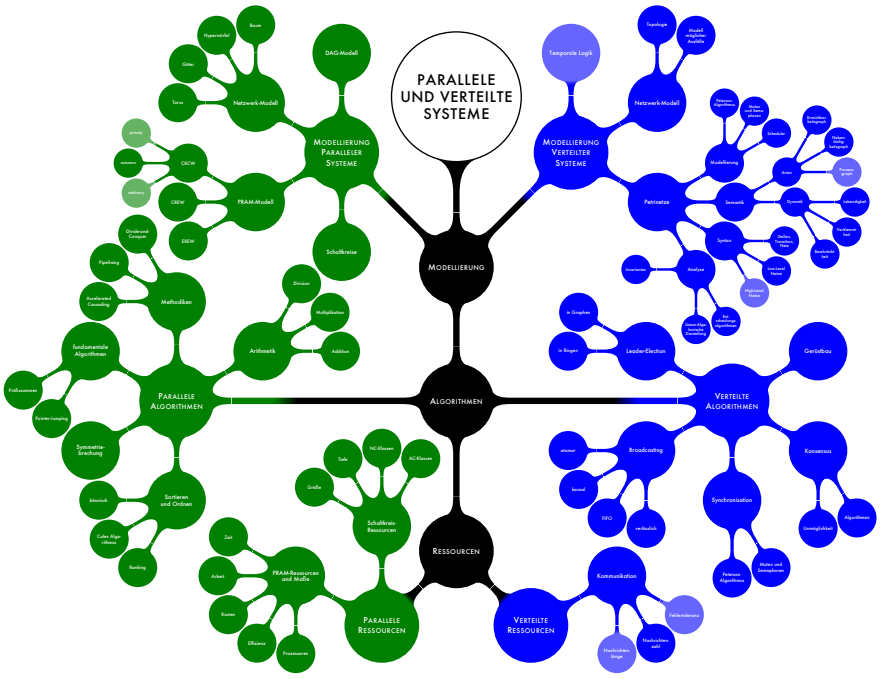
Vorlesungsskript

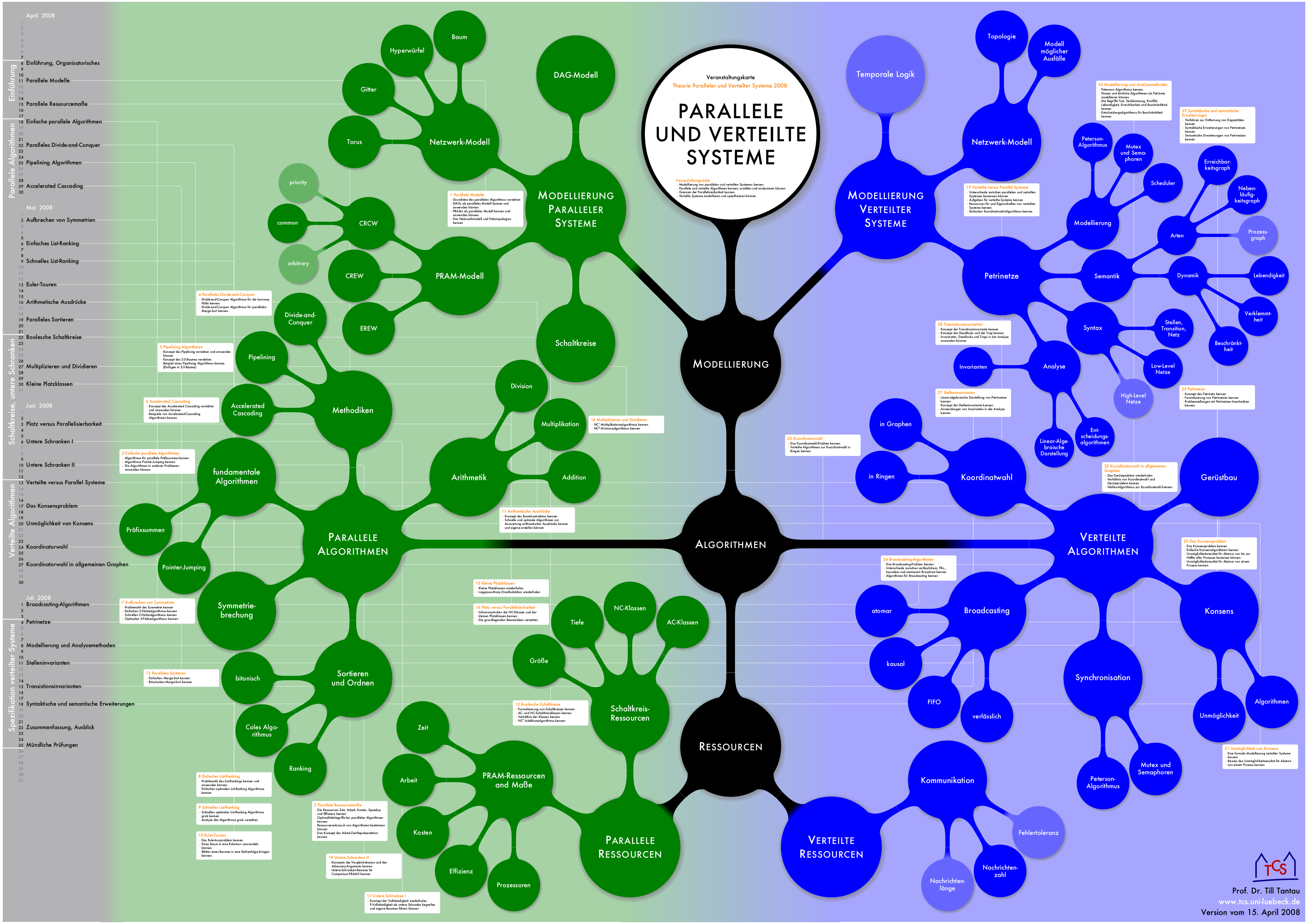
Theorie Paralleler und Verteilter Systeme

Sommersemester 2008

23. April 2008

Prof. Dr. Till Tantau
Insitut für Theoretische Informatik
Universität zu Lübeck




Veranstaltungskarte
Theorie Paralleler und Verteilter Systeme 2008
PARALLELE UND VERTEILTE SYSTEME
April 2008
8 Einführung, Organisatorisches
11 Parallele Modelle
15 Parallele Ressourcemaße
18 Einfache parallele Algorithmen
22 Paralleles Divide-and-Conquer
25 Pipelining Algorithmen
29 Accelerated Cascading
Mai 2008
2 Aufbrechen von Symmetrien
6 Einfaches List-Ranking
9 Schnelles List-Ranking
13 Euler-Touren
16 Arithmetische Ausdrücke
19 Paralleles Sortieren
22 Boolesche Schaltkreise
27 Multiplizieren und Dividieren
30 Kleine Platzklassen
Juni 2008
3 Platz versus Parallelisierbarkeit
6 Untere Schranken I
10 Untere Schranken II
13 Verteilte versus Parallel Systeme
17 Das Konsensproblem
20 Unmöglichkeit von Konsens
24 Koordinatorwahl
27 Koordinatorwahl in allgemeinen Graphen
Juli 2008
1 Broadcasting-Algorithmen
4 Petrinetze
8 Modellierung und Analysemethoden
11 Stelleninvarianten
15 Transitionsinvarianten
18 Syntaktische und semantische Erweiterungen
22 Zusammenfassung, Ausblick
25 Mündliche Prüfungen
Einführung
Parallele Algorithmen
Schaltkreise, untere Schranken
Verteilte Algorithmen
Spezifikation verteilter Systeme
Modellierung
Algorithmen
Ressourcen
Modellierung paralleler Systeme
DAG-Modell
Netzwerk-Modell
Baum
Hyperwürfel
Gitter
Torus
PRAM-Modell
CRCW
priority
common
arbitrary
CREW
EREW
Schaltkreise
Parallele Algorithmen
Methodiken
Divide-and-Conquer
Pipelining
Accelerated Cascading
Arithmetik
Division
Multiplikation
Addition
fundamentale Algorithmen
Präfixsummen
Pointer-Jumping
Symmetrie-brechung
Sortieren und Ordnen
bitonisch
Coles Algo-rithmus
Ranking
Parallele Ressourcen
PRAM-Ressourcen and Maße
Zeit
Arbeit
Kosten
Effizienz
Prozessoren
Schaltkreis-Ressourcen
Größe
Tiefe
NC-Klassen
AC-Klassen
Modellierung verteilter Systeme
Temporale Logik
Netzwerk-Modell
Topologie
Modell möglicher Ausfälle
Petrinetze
Modellierung
Peterson-Algorithmus
Mutex und Sema-phoren
Scheduler
Semantik
Arten
Erreichbar-keitsgraph
Neben-läufig-keitsgraph
Prozess-graph
Dynamik
Lebendigkeit
Verklemmt-heit
Beschränkt-heit
Syntax
Stellen, Transition, Netz
Low-Level Netze
High-Level Netze
Analyse
Invarianten
Linear-Alge-braische Darstellung
Ent-scheidungs-algorithmen
Verteilte Algorithmen
Koordinatwahl
in Graphen
in Ringen
Gerüstbau
Konsens
Unmöglichkeit
Algorithmen
Broadcasting
atomar
kausal
FIFO
verlässlich
Synchronisation
Peterson-Algorithmus
Mutex und Semaphoren
Verteilte Ressourcen
Kommunikation
Fehlertoleranz
Nachrichten-zahl
Nachrichten-länge
Prof. Dr. Till Tantau
www.tcs.uni-luebeck.de
Version vom 15. April 2008

Inhaltsverzeichnis

Teil II
Parallele Algorithmen

3 Einfache parallele Algorithmen

4 Paralleles Teilen und Herrschen

5 Pipelining Algorithmen

6 Accelerated-Cascading

7 Aufbrechen von Symmetrien

8 Einfache List-Ranking-Algorithmen

9 Schneller optimaler List-Ranking-Algorithmus

Teil III
Schaltkreise und untere Schranken

13 Boole'sche Schaltkreise

14 Multiplizieren und Dividieren

15 Kleine Platzklassen

16 Verhältnis von Platz und Parallelisierbarkeit

17 Untere Schranken I

18 Untere Schranken II

Teil IV
Verteilte Algorithmen

19 Verteilte versus parallele Systeme

20 Das Konsensproblem

21 Unmöglichkeit von Konsens

22 Koordinatorwahl

23 Koordinatorwahl in allgemeinen Graphen

24 Broadcasting-Algorithmen

Teil V
Spezifikation verteilter Systeme

25 Petrinetze

26 Modellierung und Analysemethoden

27 Stelleninvarianten

Vorwort

Liebe Leserin, lieber Leser, willkommen in einer Welt, in der alles gleichzeitig passiert. Da die Veranstaltung *Theorie der* Parallelen und Verteilten Systeme heißt, geht es eher (wenn auch nicht ausschließlich) um die *Theorie* hinter zum einen parallelen Systemen und zum anderen verteilten Systemen. Da die Menschheit hierüber schon sehr viel weiß, werden wir nur kleine Teile erforschen können. Aus diesen Gründen habe ich versucht, Themen für die Veranstaltung auszusuchen, die *exemplarisch* sowohl die Möglichkeiten wie auch die Probleme aufzeigen; die *praktisch* sind in dem Sinne, dass reale Menschen diese Probleme wirklich gelöst haben wollen und das auch noch schnell; und die *interessant* sind in dem Sinne, dass ihre Lösungen überraschend oder trickreich oder einfach nur schön sind.

Beginnen werden wir mit *parallelen Systemen*. Stellen Sie sich vor, Sie bekommen ein dickes, fettes Übungsblatt, das Sie in einer Woche bearbeiten müssen. Stellen Sie sich weiter vor, dass Sie aufgrund einer ganzen Verkettung von widrigen Umständen erst eine Stunde vor Abgabeschluss damit beginnen, das Blatt zu bearbeiten. Sie stellen schnell fest, dass Sie das Blatt alleine nicht werden lösen können. Die Lösung liegt aber auf der Hand: Arbeitsteilung! Hat das Blatt drei Teilaufgaben, so kann man die Arbeit logischerweise auf drei Leute gut verteilen. (Dies werden wir einen *Speedup* von 1:1 nennen.) Nehmen wir an, auch dies reicht noch nicht aus, da jede Aufgabe alleine schon über eine Stunde dauert. Sie probieren wieder, auf Arbeitsteilung zu setzen: Pro Aufgabe diskutieren nun vier Studierende, wie man sie lösen sollte. Wie Sie sicherlich aus eigener Erfahrung wissen, können Sie nicht erwarten, dass vier Leute eine Aufgabe vier Mal so schnell gelöst bekommen wie eine Person. Aber etwas schneller wird es schon gehen, sagen wir doppelt so schnell. (Wir werden davon reden, dass die *Effizienz* sinkt.) Wenn Sie nun auch mit den mittlerweile zwölf Studierenden nicht rechtzeitig fertig werden, dann könnten Sie auf die Idee kommen, das ganze Semester an den Aufgaben arbeiten zu lassen. Dabei werden Sie aber wahrscheinlich feststellen, dass Sie hauptsächlich damit beschäftigt sein werden, Kaffee von A nach B zu transportieren und die Leute davon zu überzeugen versuchen, doch bitte weiter zu arbeiten. (Wir werden sagen, dass auch mit beliebig vielen fleißigen Helfern eine Zeit T^∞ nicht unterschritten werden kann und dass irgendwann mehr kommuniziert wird als gerechnet wird.) Moral: Bitte beginnen Sie rechtzeitig mit den Übungsblättern.

Ein paralleles System ist das elektronische Analogon zu einer Horde Studierenden, die gemeinsam ein Übungsblatt zu lösen versuchen. Je nach Art des Übungsblatts gelingt dies mehr oder weniger leicht. Die Theorie Paralleler Systeme beschäftigt sich ausführlich mit der Frage, welche Probleme eben gerade gut parallelisierbar sind und welche weniger gut.

In *verteilten Systemen* geht es chaotischer zu als in parallelen. Hier stelle man sich eher eine Studentenclique vor, von der einige in der Mensa sitzen, einige zu Hause vor ihrem Rechner, andere vermutlich noch im Bett liegen und von einigen ist der Status völlig unbekannt. Eine solche Clique wird es um einiges schwieriger haben, ein Problem zu lösen, da man recht viel wird telefonieren und e-mailen müssen. Aus diesem Grund werden wir unsere Ansprüche auch gehörig herunterschrauben. Die Problemstellungen, die man bei einem verteilten System lösen möchte, sind eher auf dem Niveau: »Wir wollen uns einigen, welchen Kinofilm wir heute Abend schauen.« oder »Alle sollen erfahren, dass die Vorlesung heute ausfällt.« – beides unter den angegebenen Umständen keine trivialen Probleme.

In der Theorie Verteilter Systeme werden wir untersuchen, welche Probleme dieser Art lösbar sind und welche nicht. Wir werden uns auch Gedanken machen, wie man bei Verteilten Systemen spezifiziert, was man eigentlich erreichen möchte – eine erstaunlich schwierige Aufgabe, wie sich herausstellen wird.

Soviel zum Inhalt der Veranstaltung; nun zu den Zielen. Diese lauten:

1. Modellierung von parallelen und verteilten Systemen kennen.
2. Parallele und verteilte Algorithmen kennen, erstellen und analysieren können.
3. Grenzen der Parallelisierbarkeit kennen.
4. Verteilte Systeme modellieren und spezifizieren können.

Die Worte »kennen« und »können« tauchen dort recht häufig auf. Um etwas wirklich zu können, reicht es nicht, davon gehört zu haben oder davon gelesen zu haben. Man muss es auch wirklich *getan* haben: Sie können sich tausend Fußballspiele im Fernsehen anschauen, sie sind deshalb noch kein guter Fußballspieler; sie können tausend Stunden World of Warcraft spielen, sie werden deshalb trotzdem keinen Frostblitz auf Ihren Professor geschleudert bekommen. Deshalb steht bei dieser Veranstaltung der Übungsbetrieb mindestens gleichberechtigt neben der Vorlesung. Der Ablauf ist dabei folgender: In der Vorlesung werde ich Ihnen die Thematik vorstellen und Sie können schon mit dem Üben im Rahmen kleiner Miniübungen *während der Vorlesung* beginnen. Alle ein bis zwei Wochen gibt es ein Übungsblatt, das inhaltlich zu den Vorlesung gehört. Sie müssen sich die Übungsblätter aber nicht »alleine erkämpfen«. Vielmehr gibt es Tutorien, in denen Sie Aufgaben üben werden, die »so ähnlich« wie die Aufgaben auf den Übungsblättern sind. Sie werden feststellen, dass das Lösen der Übungsblätter mit diesen Vorbereitungen immer möglich sein wird und dies auch mit vertretbarem Aufwand.

Ich wünsche Ihnen viel Spaß mit dieser Veranstaltung.

Till Tantau

Teil I
Parallele Modelle

Wenn Sie einen Theoretiker in Verlegenheit bringen wollen, dann fragen Sie ihn doch mal Folgendes: »Wie schnell kann man zwei Matrizen multiplizieren?« Vor vierzig Jahren wäre die Antwort einfach gewesen: »Na, $O(n^3)$ und das scheint mir auch optimal zu sein.« Mit einer gewissen Verblüffung hat die Theoretische Informatik dann das Ergebnis von Volker Strassen zur Kenntnis genommen, dass es auch in Zeit $O(n^{\log_2 7})$ geht. Heute würde der befragte Theoretiker wohl antworten »$O(n^{2+\text{Forschungskonstante}})$«. Richtig ärgern können Sie ihn dann, wenn Sie entgegnen »Also, auf meinem Parallelrechner geht das in Zeit $O(\log n)$.«

In der Welt der Parallelverarbeitung geht es recht flott zu. Probleme, die man normalerweise mit viel Mühe in quadratischer oder kubischer Zeit gelöst bekommt, lassen sich in logarithmischer Zeit oder sogar noch schneller lösen. In diesem ersten Teil der Veranstaltung wollen wir zunächst einmal klären, was man eigentlich genau damit meint, dass ein Problem »parallel in Zeit XY lösbar ist«. Dazu werden wir uns in klassischer Theoretikermanier auf einen Modellbegriff für Parallelrechner einigen und dann das Ressourcemaß *Zeit* definieren. Die Aussage »man kann Matrizen parallel in Zeit $O(\log n)$ multiplizieren« lässt sich dann präziser (und wie so oft bei präzisen Formulierungen: weniger verständlich) fassen als »Es gibt ein PRAM-Programm mit $T^\infty \in O(\log n)$, dass das Produkt von beliebigen Matrizen berechnet.«

Wenn parallele Computer so viel schneller sind als sequentielle, warum haben wir eigentlich nicht Computer mit einer Milliarde Prozessoren auf dem Schreibtisch stehen? Es stellt sich heraus, dass der Geschwindigkeitsvorteil der Parallelverarbeitung teuer erkauft ist: Parallele Rechner mögen schnell sein, in der Regel sind sie in Bezug auf Stromverbrauch und »Miete« ziemlich verschwenderisch. Wir werden deshalb besondere Ressourcemaße einführen, die solche Dinge wie Stromverbrauch messen – solche Maße machen im sequentiellen interessanteweise keinen besonderen Sinn.

Am Ende dieses ersten Teils definieren wir dann den heiligen Gral der Parallelverarbeitung: *Schnelle arbeitsoptimale Algorithmen*. Im Teil II werden wir eine Reihe solcher Algorithmen kennen lernen, wobei einige ziemlich kompliziert sein werden.

1-1

Kapitel 1
Parallele Modelle
Wie rechnet man parallel?

1-2

Lernziele dieses Kapitels

1. Grundidee des parallelen Algorithmus verstehen
2. DAGs als paralleles Modell kennen und anwenden können
3. PRAMS als paralleles Modell kennen und anwenden können
4. Das Netzwerkmodell und Netztopologien kennen

Inhalte dieses Kapitels

Worum es heute geht

Bevor man parallele Programme schreiben kann benötigt man zunächst eine Programmiersprache, die Parallelität unterstützt. Ganz ähnlich benötigt man zum Entwurf paralleler Algorithmen zunächst ein *paralleles Maschinenmodell*. Ohne ein solches Modell machen Aussagen der Art »dieser Algorithmus löst das Problem in Zeit $O(\log n)$« keinen Sinn (was genau bedeutet denn »in Zeit XY«?). Noch trostloser wird es ohne ein genaues Maschinenmodell, wenn man Aussagen der Form »dieses Problem lässt sich parallele nicht schneller als in Zeit $O(\log\log n)$ lösen« zeigen möchte.

Wie sollte nun ein geeignetes Maschinenmodell aussehen? Im Prinzip geht man beim Entwurf eines solchen Modells genauso vor wie im sequenziellen Fall: Man beginnt mit einem realen parallelen Computer und modelliert mathematisch alles, was dort so passiert. Dann beginnt man, von Besonderheiten der Architektur zu abstrahieren. Beispielsweise ist es wohl eher unerheblich, ob nun 2 Gigabyte oder 4 Gigabyte Speicher zur Verfügung stehen. Ebenso ist es reichlich unerheblich, ob es ein A20-Gate gibt. Resultat dieser Vereinfachungen ist idealerweise ein einfaches, elegantes mathematisches Modell mit der Eigenschaft, dass alle anderen sinnvollen Modelle ohne großen Zeit- oder Platzverlust simuliert werden können. Im Sequenziellen hat dieser Ansatz gut funktioniert: Es sind dabei das RAM-Modell und die Turing-Maschine herausgekommen; zwei sehr einfache Modelle, die beide einander simulieren können.

Im Parallelen ist die Sache leider komplexer. Es gibt kein Modell von dem man behaupten könnte, es sein gleichzeitig einfach, elegant und universell einsetzbar. Ist das Modell schön einfach (wie das PRAM-Modell), so lässt es leider zu viele reale Details weg, in denen bekanntlich der Teufel steckt. Ist es geradezu übermächtig (wie das Netzwerkmodell, in dem die Kommunikation der Prozessoren komplett modelliert wird), so kann man kaum etwas Sinnvolles beweisen.

In der heutigen Vorlesung wollen wir die Problematik eines geeigneten Maschinenmodells für parallele Algorithmen erörtern. Das Resultat kann man auch gleich vorwegnehmen: Wir werden später auf das PRAM-Modell setzen. Wie schon gesagt ist dieses Modell eigentlich »zu optimistisch«. Dies hat aber auch einen wichtigen Vorteil: Können wir zeigen, dass sich ein Problem *noch nicht mal auf einer* PRAM *schnell lösen lässt*, so gilt dies sicher auch für alle »weniger optimistischen« Modelle.

1.1 Einführung

1.1.1 Mehr Prozessoren = Schneller?

Das Versprechen der Parallelverarbeitung. 1-4

- Die Kosten, Prozessoren schneller zu machen, steigen exponentiell.
- Die Kosten, die Anzahl der Prozessoren zu erhöhen, steigen linear.

Versprechen
Durch Parallelverarbeitung kann man Probleme billiger schnell lösen als durch schnellere Prozessoren.

Zur Diskussion
Welche Einwände lassen sich gegen dieses Argument erheben?

1.1.2 Arten der Parallelverarbeitung

Grundsätzliche Arten von Parallelverarbeitung. 1-5

1. SISD: single instruction, single data
   »Eine Instruktion auf einen Datensatz anwenden«
   Beispiel: Normale CPU
2. SIMD: single instruction, multiple data
   »Eine Instruktion auf mehrere Datensätze anwenden«
   Beispiel: Graphikprozessor
3. MIMD: multiple instructions, multiple data
   »Unterschiedliche Instruktionen auf unterschiedliche Datensätze anwenden«
   Beispiel: Dual-Core-Prozessoren, große Parallelrechner

1.1.3 Fallbeispiel

1-6

Die Matrix-Vektor-Multiplikation.

Problem

Eingabe Eine $n \times n$ Matrix A und ein Vektor v.

Ausgabe $b = A \cdot v$.

Zur Diskussion

Wie löst man das Problem, wenn man ...

- ... einen Prozessor hat?
- ... zwei Prozessoren hat?
- ... drei Prozessoren hat?
- ... $n/2$ Prozessoren hat?
- ... $n/2+1$ Prozessoren hat?
- ... n Prozessoren hat?
- ... $2n$ Prozessoren hat?
- ... n^2 Prozessoren hat?
- ... $2n^2$ Prozessoren hat?

1-7

Was bringen mehr Prozessoren?

Für das Problem Matrix-Vektor-Multiplikation hat die Verdoppelung der Prozessoranzahl folgenden Effekt:

- Für kleine Prozessorzahlen halbiert sich jedesmal die Rechenzeit.
- Für Prozessorzahlen über n gilt dies nicht mehr exakt, aber fast.
- Für Prozessorzahlen in der Nähe von n^2 verringert sich die Rechenzeit kaum.
- Für Prozessorzahlen über n^2 ändert sich die Rechenzeit nicht mehr.

1.2 Das DAG-Modell

1-8

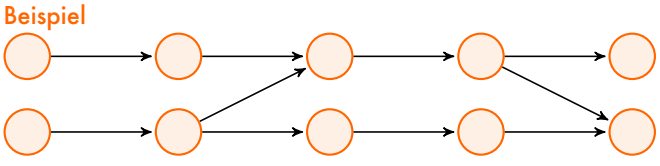
Die Idee hinter dem DAG-Modell.

- Das DAG-Modell macht *Abhängigkeiten* sichtbar.
- Das DAG-Modell blendet die Implementierung aus.

Beschreibung des DAG-Modells

- Im DAG-Modell wird ein Algorithmus durch einen DAG (directed acyclic graph) beschrieben.
- Die Eingabe liegt an den Knoten mit Eingrad 0 an.
- Die anderen Knoten führen Berechnungen durch.
- Sie können ihre Berechnung erst durchführen, wenn die Ergebnisse der Vorgänger vorliegen.
- Berechnungen können recht beliebig sein.

Beispiel



Das Konzept des Schedules.

1-9

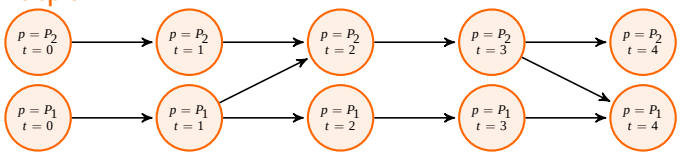
Definition (Schedule)

Sei $G = (V, E)$ ein DAG und P eine Menge von Prozessoren. Ein *Schedule* S ist eine Abbildung, die jedem Knoten von G einen Prozessor $p \in P$ zuordnet und einen Zeitpunkt $t \in \mathbb{N}$. Dabei muss gelten:

1. Keinen zwei Knoten darf derselbe Zeitpunkt und gleichzeitig derselbe Prozessor zugeordnet werden.
2. Der Zeitpunkt jedes Prozessors muss größer sein als der seiner Vorgänger.

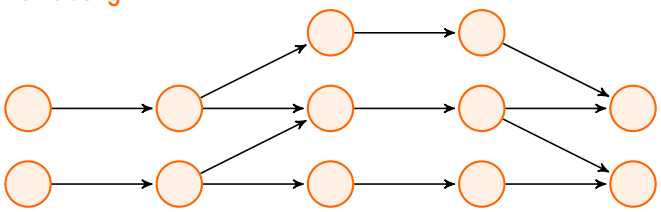
Der maximale in S vorkommende Zeitpunkt heißt *Zeitbedarf*, abgekürzt $T(S)$.

Beispiel



Zur Übung

1-10



Gruppe 1

Geben Sie einen optimalen Schedule für den DAG mit $P = \{P_1, P_2\}$ an.

Gruppe 2

Geben Sie optimale Schedules für den DAG mit $P = \{P_1\}$ an und dann für $P = \{P_1, P_2, P_3\}$.

1.3 Das PRAM-Modell

1.3.1 Die Idee

PRAMs im Überblick

1-11

Aufbau und Initialisierung

- Die Abkürzung PRAM steht für Parallel Random Access Machine.
- Jeder Prozessor ist eine eigene RAM. Das bedeutet:
  - Eine RAM hat Zugriff auf einen Satz Register, in denen Zahlen stehen.
  - Eine RAM arbeitet ein festes Programm ab.

  Neu im Vergleich zur einfachen RAM *ist:*
  - Es gibt noch *globale Register*.
  - Es gibt noch ein *nur lesbares Prozessor-ID-Register*.
  - Es gibt neue Befehle, um die globalen Register zu lesen und zu beschreiben.
- Die Prozessoren arbeiten synchron (exakt gleichgetaktet).
- Alle Prozessoren führen das gleiche Programm aus, aber bei jedem Prozessor ist PID anders initialisiert.
- Ein- und Ausgabe liegen in bestimmten Bereichen des globalen Speichers.

1-12

PRAMs im Überblick

Kommunikation

- Die Kommunikation erfolgt mittels des globalen Speichers (shared memory).
- Man unterscheidet, ob der Lesezugriff ausschließend (exclusive read) oder parallel (concurrent read) sein darf.
- Schreibzugriff kann man ebenfalls unterschiedlich regeln:
  - Nur ein bestimmter Prozessor darf eine bestimmte globale Zelle beschreiben (owner write).
  - Immer nur ein Prozessor darf schreiben (exclusive write).
  - Paralleles Schreiben (concurrent write) ist erlaubt. Hier gibt es die Untervarianten Common, Arbitrary und Priority.
- Insgesamt gibt es folgende sinnvolle Zugriffsarten: EROW, EREW, CROW, CREW, Common-CRCW, Aribtrary-CRCW, Priority-CRCW.

1.3.2 Ein Beispiel

1-13

Beispiel eines PRAM-Programms.

Folgendes Programm berechnet die Summe der Zahlen im Array A im globalen Speicher.

```
1 for h ← 1 to log n seq do
2     if pid ≤ n/2^h then
3         global read a ← A[2 · pid]
4         global read b ← A[2 · pid − 1]
5         c ← a + b
6         global write A[pid] ← c
7 if pid = 1 then
8     global write S ← c
```

Zur Übung

Welche Schreib-/Lese-Zugriffsart wird im Programm verwendet?

Skript

Das PRAM-Modell wird das »zentrale« Modell sein, welches wir im Rest der Veranstaltung benutzen werden. Aus diesem Grund ist im Folgenden (nur im Skript) für an Formalisierungen Interessierte eine genaue Beschreibung des Modells angegeben. *Es ist aber nicht nötig, diese genau zu kennen, Algorithmen und Beweise werden wir eher auf der Pseudo-Code-Ebene führen.* (Man beweist auch nicht die Eigenschaften eines Quicksorts anhand einer Implementation in Maschinensprache...) Es gibt verschiedene Arten, RAMs und PRAMs zu formalisieren, die folgende Formalisierung ist eher für Komplexitätsbetrachtungen geeignet, da es nur wenige Befehle gibt und »aufwendige« Berechnungen wie die Multiplikation explizit *nicht* als ein Befehl zur Verfügung stehen.

Definition (Syntax eines PRAM-Programms)

Ein PRAM-*Programm* besteht aus einer endlichen, durchnummerierten Folge von Befehlen, wobei die Zählung mit 1 beginnt. In Befehlen kommen *Registeradressierungen* vor, wovon es genau folgende vier Arten gibt (im Folgenden sei immer $i, j, k \in \mathbb{N}$):

1. Ri bezeichnet eine *direkte lokale Adressierung*
2. RRi bezeichnet eine *indirekte lokale Adressierung*
3. GRi bezeichnet eine *direkte globale Adressierung*
4. GRRi bezeichnet eine *indirekte globale Adressierung*

Folgende Befehle sind möglich:

- **Die Transportbefehle**
  1. Ri ← Rj

2. $\text{R}i \leftarrow \text{RR}j$
3. $\text{R}i \leftarrow \text{GR}j$
4. $\text{R}i \leftarrow \text{GRR}j$
5. $\text{RR}i \leftarrow \text{R}j$
6. $\text{GR}i \leftarrow \text{R}j$
7. $\text{GRR}i \leftarrow \text{R}j$

– **Die arithmetischen Befehle**

8. $\text{R}i \leftarrow k$
9. $\text{R}i \leftarrow pid$
10. $\text{R}i \leftarrow \text{R}j + \text{R}k$
11. $\text{R}i \leftarrow \text{R}j - \text{R}k$
12. $\text{R}i \leftarrow \lfloor \frac{1}{2}\text{R}j \rfloor$

– **Die Sprungbefehle**

13. *if* $\text{R}i = 0$ *goto* k
14. *if* $\text{R}i > 0$ *goto* k
15. *goto* k

Bei allen Sprungbefehlen muss k die Nummer einer Programmzeile sein.

– **Die Stopp- und Idlebefehle**

16. *idle while* $\text{GRR}i = 0$
17. *stop*

Der letzte Befehl eines Programms muss der Stoppbefehl sein.

Ein Beispiel eines PRAM-Programms wäre folgendes:

```
1  R0 ← pid
2  R3 ← GR0
3  R2 ← 1
4  R1 ← R0 + R2
5  R4 ← R1 − R3
6  if R4 > 0 goto 13
7  R5 ← GRR0
8  R6 ← GRR1
9  R7 ← R6 + R7
10 GRR0 ← R7
11 R2 ← R2 + R2
12 goto 4
13 stop
```

Definition (Semantik eines PRAM-Programms)
Sei P ein PRAM-Programm. Eine PRAM hat einen *globalen Speicher* sowie eine unbeschränkte Anzahl an Prozessoren, die alle einen *lokalen Speicher* besitzen. Eine PRAM arbeitet getaktet, wobei mit Takt $t = 0$ begonnen wird. Der Inhalt des i-ten globalen Registers zum Zeitpunkt t ist eine natürliche Zahl, welche wir mit $\langle \text{GR}i \rangle_t$ bezeichnen. Der Inhalt des i-ten lokalen Registers des p-ten Prozessors zum Zeitpunkt t bezeichnen wir mit $\langle \text{R}i \rangle_t^p$. Weiterhin verfügt jeder Prozessor über einen Programmschrittzähler, dessen Wert zum Zeitpunkt t mit $\langle \text{PC} \rangle_t^p$ bezeichnet wird.
Die Gesamtheit der Inhalte aller lokalen und globalen Register (dies sind abzählbar unendlich viele) bezeichnen wir als *Konfiguration*. Eine *Berechnung* ist eine Folge von Konfigurationen, die den weiter unten angegebenen Bedingungen genügt. Eine *Eingabe* für eine PRAM ist ein Vektor v von natürlichen Zahlen. Zu einer Eingabe gehört die *Anfangskonfiguration* bei der alle lokalen Register 0 sind, alle Programmschrittzähler 1 sind, das erste globale Register (GR0) die Dimension n von v enthält und die globalen Register GR1 bis GRn die Komponenten von v enthalten. Eine *Endkonfiguration* ist eine Konfigurationen, in der der Programmschrittzähler aller Prozessoren eine Nummer enthält, für die der Programmbefehl »Stopp« lautet.
Die *Berechnungsrelation* gibt an, welche neue Konfiguration zum Zeitpunkt $t + 1$ aus der Konfiguration zum Zeitpunkt t hervorgeht. Dazu führen alle Prozessoren gleichzeitig den

jeweiligen Befehl aus, der im PRAM-Programm P in der Zeile $\langle \text{PC} \rangle_t^p$ steht. Die Effekte dieser Befehle sind in der folgenden Tabelle angegeben:

Befehl	*Wirkung, wenn von Prozessor p ausgeführt*
$\text{R}i \leftarrow \text{R}j$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \langle \text{R}j \rangle_t^p$
$\text{R}i \leftarrow \text{RR}j$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \langle \text{R} \langle \text{R}j \rangle_t^p \rangle_t^p$
$\text{R}i \leftarrow \text{GR}j$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \langle \text{GR}j \rangle_t$
$\text{R}i \leftarrow \text{GRR}j$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \langle \text{GR} \langle \text{R}j \rangle_t^p \rangle_t$
$\text{RR}i \leftarrow \text{R}j$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R} \langle \text{R}i \rangle_t^p \rangle_{t+1}^p = \langle \text{R}j \rangle_t^p$
$\text{GR}i \leftarrow \text{R}j$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{GR}i \rangle_{t+1} = \langle \text{R}j \rangle_t^p$, aber siehe unten
$\text{GRR}i \leftarrow \text{R}j$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{GR} \langle \text{R}i \rangle_t^p \rangle_{t+1} = \langle \text{R}j \rangle_t^p$, aber siehe unten
$\text{R}i \leftarrow k$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = k$
$\text{R}i \leftarrow pid$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = p$
$\text{R}i \leftarrow \text{R}j + \text{R}k$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \langle \text{R}j \rangle_t^p + \langle \text{R}k \rangle_t^p$
$\text{R}i \leftarrow \text{R}j - \text{R}k$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \max\{\langle \text{R}j \rangle_t^p - \langle \text{R}k \rangle_t^p, 0\}$
$\text{R}i \leftarrow \lfloor \frac{1}{2} \text{R}j \rfloor$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, $\langle \text{R}i \rangle_{t+1}^p = \lfloor \langle \text{R}j \rangle_t^p / 2 \rfloor$
goto k	$\langle \text{PC} \rangle_{t+1}^p = k$,
if $\text{R}i = 0$ *goto* k	$\langle \text{PC} \rangle_{t+1}^p = k$, falls $\langle \text{R}i \rangle_t^p = 0$, sonst $\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$
if $\text{R}i > 0$ *goto* k	$\langle \text{PC} \rangle_{t+1}^p = k$, falls $\langle \text{R}i \rangle_t^p > 0$, sonst $\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$
idle while $\text{GRR}i = 0$	$\langle \text{PC} \rangle_{t+1}^p = \langle \text{PC} \rangle_t^p + 1$, falls $\langle \text{GR} \langle \text{R}i \rangle_t^p \rangle_t > 0$, sonst keine Änderung
stop	keine Änderungen

Es kann vorkommen, dass mehrere Prozessoren gleichzeitig auf ein globales Register schreibend zugreifen. Dann können Konflikte entstehen und der Wert von $\langle \text{GR}i \rangle_{t+1}$ ist nicht wohldefiniert, da unterschiedliche Angaben vorliegen können. Dieses Problem wird wie folgt aufgelöst:

1. Beim *Owner-Write-Modell* darf auf das Register $\text{GR}i$ nur Prozessor i schreiben. Die Nachfolgekonfiguration ist nicht definiert, falls ein Prozessor sich nicht daran hält.
2. Beim *Exclusive-Write-Modell* darf in jedem Takt für jedes globale Register i nur genau ein Prozessor $\langle \text{GR}i \rangle_{t+1}$ einen Wert zuweisen. Versuchen dies mehrere, so gibt es wiederum keine Nachfolgekonfiguration.
3. Beim *Arbitrary-Write-Modell* gibt es im Falle von unterschiedlichen Werten, die von verschiedenen Prozessor in dasselbe globale Register $\text{GR}i$ geschrieben werde, für jeden Wert eine eigene Nachfolgekonfiguration, in der dieser Wert eingenommen wird (das Verhalten ist also nichtdeterministisch).
4. Beim *Common-Write-Modell* müssen alle Prozessoren denselben Wert schreiben. Geschieht dies nicht, ist die Nachfolgekonfiguration wieder nicht definiert.
5. Beim *Priority-Write-Modell* sei P_i die Menge aller Prozessornummern von Prozessoren, die versuchen, im Zeittakt t den Wert von $\langle \text{GR}i \rangle_{t+1}$ zu verändern. Sei $p_i = \min P_i$ und sei x_i der Wert, den p_i schreiben möchte. Dann ist $\langle \text{GR}i \rangle_{t+1} = x_i$.

Während ein Prozessor den Idle-Befehl ausführt (und somit schläft) zählt er nicht zu den aktiven Prozessoren und liefert keinen Beitrag zur so genannten *Arbeit* des Systems (siehe dazu die nächste Vorlesung).

1.4 Das Netzwerkmodell

1.4.1 Das Modell

1-14

Die Idee hinter dem Netzwerk-Modell.

- Man modelliert ein festes Kommunikationsnetzwerk durch einen Graphen.
- Die *Knoten* des Graphen sind die Prozessoren.
- Die *Kanten* des Graphen sind Kommunikationskanäle.
- Die Prozessoren arbeiten in der Regel *asynchron* und schicken sich gegenseitig Nachrichten.

Das Netzwerk-Modell – parallel oder verteilt? 1-15

Gemeinsamkeiten mit verteilten Systemen

- Modellierung des Kommunikationsnetzwerks.
- Nachrichtenbasiertes Modell.

Unterschiede zu verteilten Systemen

- Feste Topologie (fest verdrahtet)
- Sichere Kommunikation

Wichtige Parameter eines Netzwerks 1-16

Parameter Durchmesser

Maximale Entfernung zwischen Knoten.
Dies ist die Zeit, die eine Nachricht höchstens braucht.
Kleiner ist besser.

Parameter Grad

Maximale Anzahl von Nachbarn eines Knoten.
Je höher der Grad, desto mehr Kabel muss man ziehen.
Kleiner ist besser.

Parameter Zusammenhänge

Minimale Anzahl von Knoten/Kanten, die man löschen muss, um das Netz in zwei Teile zu spalten.
Je höher der Zusammenhang, desto ausfallsicherer.
Größer ist besser.

1.4.2 Wichtige Topologien

Liste wichtiger Netztopologien. 1-17

1. Lineares Array (Pfad)
2. Ring
3. 2D-Gitter
4. 3D-Gitter
5. Hyperwürfel
6. Binärbaum

Zur Übung

Bestimmen Sie für eine der obigen Topologien für p Prozessoren den Durchmesser, Grad und Kantenzusammenhang.

Zusammenfassung dieses Kapitels

1. *Parallele Algorithmen* führen mehrere Berechnungen koordiniert aus. 1-18
2. Probleme haben häufig *parallele Anteile* und *sequentielle Anteile*.
3. Das *DAG-Modell* beschreibt abstrakt, welche Anteile parallel und welche sequentiell sind.
4. Das PRAM-*Modell* beschreibt Algorithmen konkreter.
5. Das *Netzwerkmodell* modelliert die *Kommunikation* bei parallelen Programmen realistischer als das PRAM-Modell.

Übungen zu diesem Kapitel

Übung 1.1 (Einfache Programmanalyse, leicht)

Gegeben sei ein globales Array A der geraden Länge n, sowie das folgende Programm:

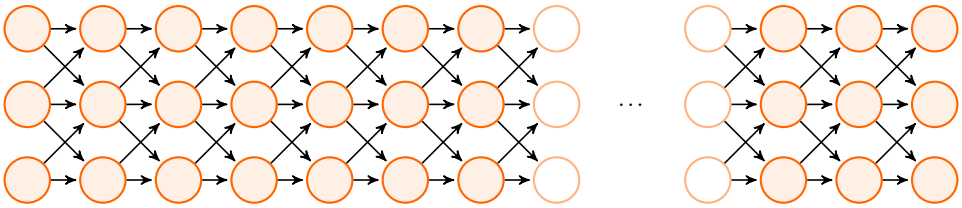
```
1 global read x ← A[pid]
2 global read y ← A[pid + n/2]
3 z ← x + y
4 global write A[pid] ← z
5 global write A[pid + n/2] ← z
```

Beantworten Sie folgende Fragen in Bezug auf die Programm:

1. Was macht das Programm?
2. Welche Zugriffsart (wie zum Beispiel EREW, Priority-CRCW, etc.) benötigt das Programm? Begründen Sie Ihre Antwort.
3. Wie schnell lässt sich das Problem sequentiell lösen ?

Übung 1.2 (Optimale Schedules, mittel)

Folgender DAG gibt die Abhängigkeiten in einem parallelen Programm an:



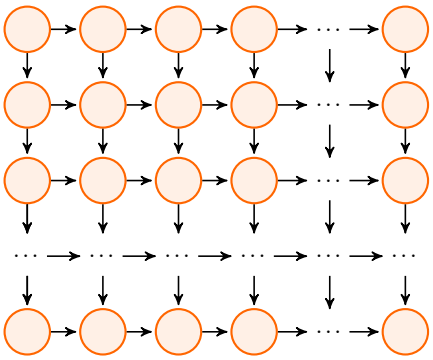
Die Anzahl der Knoten im DAG sei n (also ist $n = 3m$, wenn m die Anzahl der Knoten in einer Reihe ist). Geben Sie für jedes $p = 1, 2, 3, \ldots$ einen optimalen Schedule für diesen DAG an. Beweisen Sie für jedes p, dass Ihr Schedule optimal ist.

Da n und p variabel sind, müssen Sie Ihren Schedule allgemein beschreiben; Ihr Tutor muss daraus schlau werden. Vermeiden Sie bitte sowohl unverständlichen Formelsalat als auch Beschreibungen, die sich nur auf ein einziges konkretes n beziehen. Graphiken »mit Pünktchen« sind in Ordnung.

Tipps: Der interessanteste Fall ist $p = 2$. Alle $p > 3$ kann man auf die gleiche Art behandeln.

Übung 1.3 (Optimale Schedules in einer Gitterstruktur, mittel)

Folgender DAG der Größe $n \times n$ gibt die Abhängigkeiten in einem parallelen Programm an:



Die Anzahl der Knoten im DAG ist also n^2. Geben Sie für jedes $p = 1, 2, 3, \ldots$ einen optimalen Schedule für diesen DAG an. Beweisen Sie für jedes p, dass Ihr Schedule optimal ist. Überlegen Sie, was für $p = 1$, $p = n$ und $p > n$ gilt.

2-1

Kapitel 2
Ressourcemaße bei parallelen Programmen
Lohnt sich der ganze Aufwand?

2-2

Lernziele dieses Kapitels

1. Die Ressourcen Zeit, Arbeit, Kosten, Speedup und Effizienz kennen
2. Optimalitätsbegriffe bei parallelen Algorithmen kennen
3. Ressourceverbrauch von Algorithmen bestimmen können
4. Das Konzept der Arbeit-Zeit-Repräsentation kennen

Inhalte dieses Kapitels

Worum es heute geht

Frage: *Welche drei Ressourcen sind bei sequentiellen Programmen am wichtigsten?* Antwort: *Zeit, Zeit und Zeit.* Wenn man möchte, kann man als vierte noch den Arbeitsplatz hinzufügen; aber eigentlich sind die ersten drei Ressourcen doch wichtiger. Bei parallelen Programmen liegt die Sachlage kompliziert. Wir werden viele Probleme kennen lernen, die sich auf einem parallelen System theoretisch rasend schnell lösen lassen. Eine Laufzeit von $O(\log n)$ ist da schon langsam – und bekanntermaßen wächst der Logarithmus ausgesprochen gemütlich mit der Eingabegröße.

Trotzdem werden uns diese Algorithmen nicht befriedigen: Sie mögen schnell sein, sie gehen aber oft sehr verschwenderisch mit der zur Verfügung stehenden Rechenkraft um. Dann tritt folgender Effekt ein: Sie haben sich einen niegelnagelneuen Rechner mit vier Dual-Core-Prozessoren unter Ihrem Schreibtisch gestellt und erwarten nun zu Recht, dass dieser Probleme achtmal schneller lösen wird als ein Rechner mit nur einem Prozessor. Vielleicht auch nur sieben- oder sogar nur sechsmal schneller – schließlich müssen die Prozessoren etwas mehr kommunizieren. Sie wären zu Recht beleidigt, wenn aber das System mit seinen acht Kernen das Problem *langsamer* löst als ein einzelner Prozessor. Genau das passiert aber schnell bei parallelen Algorithmen: Diese sind oft so verschwenderisch, dass sie 20- oder 30-mal mehr an Rechnungen insgesamt durchführen.

Von *guten* parallelen Algorithmen dürfen wir verlangen, dass dieser Effekt nicht auftritt – wir werden sie (etwas verwirrlicherweise) *optimale* Algorithmen nennen. Um das genau zu definieren benötigen wir aber erstmal etwas (recht einfache) Theorie und einige neue (recht intuitive) Ressourcemaße wie Speedup und Effizienz.

2.1 Ressourcen und Maße

2.1.1 Zeit

2-4

Zeitmaße bei parallelen und sequentiellen Algorithmen.

Sei Q ein Problem.

Definition

Die Zeitkomplexität eines *schnellsten sequentiellen Algorithmus* zur Lösung von Q bezeichnen wir mit $T^*(n)$.

Man beachte, dass $T^*(n)$ in der Regel unbekannt ist (und auch nicht so sauber definiert ist).

Definition

Die Rechenzeit eines *parallelen Algorithmus P für p Prozessoren* bezeichnen wir mit $T_p^P(n)$ oder auch nur mit $T_p(n)$.

Definition

Die Rechenzeit eines *parallelen Algorithmus, der beliebig viele Prozessoren benutzen darf,* bezeichnen wir mit $T_\infty(n)$.

2-5

Analyse des Zeitverbrauchs eines parallelen Algorithmus.

Problemstellung und Algorithmus

Problemstellung

Es soll eine $m \times m$ Matrix mit einem Vektor der Größe m multipliziert werden.
(Beachte: Bei Eingabegröße n ist $m \approx \sqrt{n}$.)

Paralleler Algorithmus

1. Falls $p < m$, behandle je n/p Zeilen parallel.
2. Falls $p \geq m$, behandle alle Zeilen parallel.
   - 2.1 In jeder Zeile bilde Blöcke der Größe m^2/p.
   - 2.2 Summiere parallel alle Werte der Blöcke.
   - 2.3 Summiere die Summe der Blöcke baumartig auf.

2-6

Analyse des Zeitverbrauchs eines parallelen Algorithmus.

Zeitverbrauch

- Offenbar gilt $T^*(n) = m^2 = n$ mittels eines einfachen Matrix-Vektor-Multiplikationsprogramms.
- Es gilt $T_1(n) = m^2 = n$.
- Es gilt $T_2(n) = m^2/2 = n/2$.
- Es gilt $T_3(n) = m^2/3 = n/3$.
- Es gilt $T_{\sqrt{n}}(n) = m^2/m = \sqrt{n}$.
- Es gilt $T_n(n) = \log_2 m = \frac{1}{2}\log_2 n$.
- Es gilt $T_\infty(n) = \frac{1}{2}\log_2 n$.

(Konstanten wurden ignoriert.)

2.1.2 Speedup und Effizienz

2-7

Maße der Güte von parallelen Algorithmen.

Definition (Speedup)

Der *Speedup* ist das Verhältnis

$$S_p(n) = \frac{T^*(n)}{T_p(n)}.$$

Definition (Effizienz)

Die *Effizienz* ist

$$E_p(n) = \frac{T_1(n)}{p \cdot T_p(n)}.$$

2.1.3 Kosten und Arbeit

Kosten versus Arbeit

2-8

Definition (Kosten)
Die *Kosten* eines Algorithmus sind das Produkt

$$C_p(n) = p \cdot T_p(n).$$

Idee
Würde man einen Parallelrechner mieten, so müsste man die Kosten bezahlen.

Definition (Arbeit)
Die *Arbeit* $W(n)$ eines Algorithmus ist die Anzahl der Zeitschritte, in denen Prozessoren tatsächlich etwas tun.

Idee
Würde man einen Parallelrechner mieten, so wäre die Arbeit der Stromverbrauch.

Zur Übung

2-9

Bestimmen Sie Kosten und Arbeit für $p = n$ für folgendes Programm.

```
1 for h ← 1 to log n seq do
2     if pid ≤ n/2^h then
3         global read a ← A[2·pid]
4         global read b ← A[2·pid − 1]
5         c ← a + b
6         global write A[pid] ← c
7 if pid = 1 then
8     global write S ← c
```

Zusammenhang zwischen Kosten und Zeit

2-10

Satz
Sei ein PRAM-Programm P für p Prozessoren geben und eine Prozessoranzahl $p' \leq p$. Dann gibt es ein PRAM-Programm P' für das gilt

$$T_{p'}^{P'}(n) = O\left(\frac{C_p^P(n)}{p'}\right).$$

Beweis. Es gilt $C_p^P(n) = p \cdot T_p^P(n)$. Die Behauptung ist also $T_{p'}^{P'}(n) = O(T_p^P(n) \cdot p/p')$. Das Programm P' darf also um den Faktor p/p' langsamer sein als P. Dies ist aber einfach zu bewerkstelligen: Jeder der p' Prozessoren simuliert sequentiell $\lceil p/p' \rceil$ Prozessoren von P. Dann dauert die Simulation *eines Schrittes von P* gerade $O(p/p')$ *Schritt von P'*. □

2.1.4 Optimalität

2-11

Was sind die »besten« parallelen Algorithmen?

Definition
Ein paralleler Algorithmus heißt *optimal*, wenn für ihn gilt $W(n) = \Theta\big(T^*(n)\big)$.
Dabei ist die Rechenzeit egal (!).

Beispiel
Der sequentielle Algorithmus zur Berechnung des Matrix-Vektor-Produkts ist, als entarteter paralleler Algorithmus aufgefasst, optimal.

Beispiel
Der parallele Algorithmus zur Berechnung des Matrix-Vektor-Produkts ist optimal, da die Arbeit nur $O(n)$ ist.

2.2 Arbeit-Zeit-Repräsentation

2.2.1 Konzept

2-12

Ein Problem und seine Lösung

Aufschreibe-Schwierigkeiten bei parallelen Algorithmen

- Wir müssen für jeden Prozessor angeben, was er genau machen soll.
- Wir müssen darauf achten, dass das Timing stimmt.
- Wir müssen uns mit vielen Indizes herumschlagen.

Lösungsidee

- Wir schreiben Programme in der so genannten *Arbeit-Zeit-Repräsentation* auf.
- Dazu geben wir für jeden Zeitpunkt an, was alles parallel passieren könnte.
- Dies geschieht mittels eines Par-Do-Befehls.
- Die Verteilung auf die Prozessoren wird vom Compiler übernommen.

2-13

Beispiel einer WT-Repräsentation

```
1 for h ← 1 to log n seq do
2     for i ∈ {1,2,3,...,n/2^h} par do
3         A[i] ← A[2i−1] + A[2i]
4 S ← A[1]
```

- Für jeden Par-Do-Befehl werden eigentlich $n/2^h$ Prozessoren benötigt.
- Stehen so viele zur Verfügung, führt einfach jeder den Par-Do-Körper für ein anderes i aus.
- Stehen nur $p < n/2^h$ Prozessoren zur Verfügung, so kümmert sich jeder dieser Prozessoren *nacheinander* um den Par-Do-Körper für einen Block von $(n/2^h)/p$ vielen unterschiedlichen i.

2.2.2 Beschleunigungssatz

Arbeit und Zeit von Programmen in WT-Repräsentation

2-14

Definition

Der *Zeitbedarf* $T(n)$ eines Programms in WT-Repräsentation ist die Rechenzeit, wenn alle Par-Do-Befehle parallel ausgeführt werden.

Satz

Sei ein Programm P in WT-Repräsentation gegeben und p Prozessoren. Dann gibt es ein PRAM-Programm P' mit

$$T_p(n) \leq \left\lfloor \frac{W(n)}{p} \right\rfloor + T(n).$$

Merke: Für kleine p gilt $T_p(n) \approx W(n)/p$, falls $W(n) \gg T(n)$.

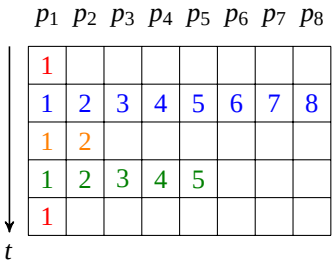
Beweis. Die PRAM arbeitet wie folgt: Sie arbeitet das Programm sequentiell ab bis zum ersten Par-Do-Befehl. Bei diesem möge eine Variable v gerade x unterschiedliche Werte annehmen.

Falls $p \leq x$ (falls also zu wenig Prozessoren zur Verfügung stehen), so beginnen alle Prozessoren parallel zu arbeiten. Jeder von ihnen führt sequentiell den Körper des Par-Do-Befehls für $\lceil x/p \rceil$ unterschiedliche Werte aus.

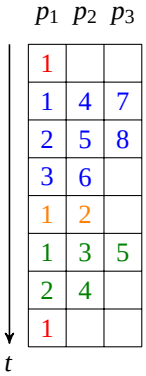
Falls $p > x$ (falls also mehr als genügend Prozessoren zur Verfügung stehen), so führen die ersten x Prozessoren parallel jeder einmal den Körper des Par-Do-Befehls aus (natürlich jeder für anderen Wert von v). Die restlichen Prozessoren machen zunächst nichts, sie werden aber vorrätig gehalten für eventuelle verschachtelte Par-Do-Befehle.

Als Beispiel sei die Simulation eines Programms P durch P' angegeben:

Programm P



Programm P' für $p = 3$



Sei nun W^t die Menge an Arbeit, die P in Zeitschritt t leistet. Dann gilt $W(n) = \sum_{t=1}^{T(n)} W^t$ und

$$T_p(n) \leq \sum_{t=1}^{T(n)} \lceil W^t/p \rceil \leq \sum_{t=1}^{T(n)} \left(\lfloor W^t/p \rfloor + 1 \right) \leq \left\lfloor \frac{W(n)}{p} \right\rfloor + T(n).$$

□

Zusammenfassung dieses Kapitels

2-15

1. Die wichtigsten Ressourcen bei parallelen Programmen sind *Zeit* und *Arbeit*.
2. *Optimale* Algorithmen *arbeiten* nicht mehr als der beste sequentielle Algorithmus.
3. Bei der WT-Repräsentation wird angegeben, was *prinzipiell parallel* ausgeführt werden kann.
4. *Für kleine* p *gilt* $T_p(n) \approx W(n)/p$, *falls* $W(n) \gg T(n)$.

Übungen zu diesem Kapitel

Übung 2.1 (Programmanalyse, leicht)

Gegeben sei das folgende Programm, welches $p = n$ Prozessoren benutzt:

```
1  for i ← 1 to n seq do
2      if pid ist gerade und pid < n then
3          global read a ← A[pid]
4          global read b ← A[pid + 1]
5          if a > b then
6              global write A[pid] ← b
7              global write A[pid + 1] ← a
8      // Sync
9      if pid ist ungerade und pid < n then
10         global read a ← A[pid]
11         global read b ← A[pid + 1]
12         if a > b then
13             global write A[pid] ← b
14             global write A[pid + 1] ← a
15     // Sync
```

Wir nehmen an, dass Indizierungen bei 1 beginnen und dass die Prozessoren an den angegebenen Stellen synchronisiert werden (wenn also ein Prozessor einen If-Block nicht ausführt, dann wartet er genau so lange, wie die anderen Prozessoren zur Ausführung brauchen).

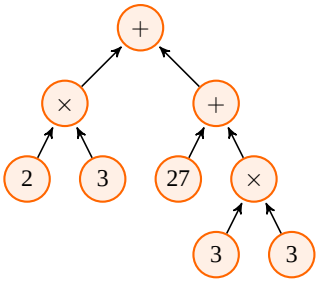
Beantworten Sie folgende Fragen in Bezug auf das Programm:

1. Was macht das Programm?
2. Welche Zugriffsart (wie zum Beispiel EREW, Priority-CRCW, etc.) benötigt das Programm? Begründen Sie Ihre Antwort.
3. Wie lautet $T^*(n)$?
4. Berechnen Sie für $p = n$ die Arbeit, die Kosten, die Effizienz und den Speedup des Programms (grob, O-Klasse reicht).
5. Ist der Algorithmus optimal? Begründen Sie Ihre Antwort.

Teil II
Parallele Algorithmen

Manche Probleme lassen sich ganz einfach parallelisieren. Schon fast trivial ist das Problem, zwei Vektoren zu addieren: Hier kann man einfach alle Addition unabhängig voneinander und damit parallel durchführen. Eine Spur schwieriger ist das Problem, die Summe von n Zahlen zu berechnen: Hier wird man baumartig vorgehen und zunächst parallel die Summe von jeder Zahl an einer ungerade Position und der neben ihr stehenden Zahl an der nächsthöheren Position; dies verringert die Anzahl der zu addierenden Zahlen um die Hälfte, weshalb man nach $O(\log n)$ Schritten das Ergebnis in seinen virtuellen Händen halten kann.

Einige Probleme sind kniffliger. Das wohl schönste Beispiel ist das Auswerten eines arithmetischen Baumes wie dem folgenden, welcher selbstverständlich zu 42 auswertet:



Wie wertet man einen solchen Baum parallel aus? Wenn er zufälligweise schön ausgeglichen ist, dann kann man sich gut von den Blättern her in Richtung Wurzel vorarbeiten. Bei obigem Baum und noch schlimmer bei den entarteten Bäumen, die aus den Horner-Schema entstehen, geht dies nicht. Es ist zunächst nicht einmal klar, ob sich dieses Problem *überhaupt* parallelisieren lässt.

In dem Teil II werden wir viel Zeit und Energie darauf verwenden, dieses Problem optimal zu parallelisieren. Dazu beginnen wir mit einigen ganz einfachen Algorithmen, nämlich Präfixsummen und Pointer-Jumping. Dann werden wir uns eine Reihe von algorithmischen Designmethoden für parallele Algorithmen anschauen, welche wir auch für das Auswertungsproblem werden anwenden können. Dann wird es um List-Ranking gehen, das im sequentiellen triviale Problem, für jedes Element einer verketteten Liste seine Position in der Liste zu bestimmen. Es sei gewarnt, dass es dabei in der Vorlesung über optimales List-Ranking etwas blutrünstig zugehen wird (niedliche Schafe werden von bösen Wölfen verspeist; aber wie im Märchen werden sie später wieder aus den Wölfen herausgeschnitten werden). Kombiniert man dann List-Ranking mit parallelen Euler-Touren und erweitert man die Auswertungsbäume um Linearformen so, voilà, hat man den gewünschten Algorithmus.

3-1

Kapitel 3
Einfache parallele Algorithmen
Präfixsumme hilft immer

3-2

Lernziele dieses Kapitels

1. Algorithmus für parallele Präfixsumme kennen
2. Algorithmus Pointer-Jumping kennen
3. Die Algorithmen in anderen Problemen anwenden können

Inhalte dieses Kapitels

Worum es heute geht

Es gibt Probleme, die so grundlegend sind, dass sie immer und immer wieder auftauchen. Zwei Standardprobleme sind Suchen und Sortieren – ständig sind Computer (genau wie Menschen) damit beschäftigt etwas zu suchen und Ordnung zu schaffen (wobei sie weitaus erfolgreicher den zweiten Hauptsatz der Thermodynamik bekämpfen als Menschen dies im Allgemeinen gelingt).

In der Parallelverarbeitung gibt es auch zwei Standardprobleme, die immer und immer wieder auftauchen: Die Bestimmung von Präfixsummen und die Bestimmung der Wurzeln eines Waldes. Auf den ersten Blick scheinen diese Probleme etwas exotisch, im Sequentiellen spielen sie jedenfalls nur eine eher untergeordnete Rolle. Dies täuscht aber, der Punkt ist lediglich, dass diese Probleme im Sequentiellen in der Regel nicht *explizit* gelöst werden.

Dazu ein kleines Beispiel: Sie führen eine elektronische Einkaufsliste mit beim Einkaufen beim Bio-Discounter Ihrer Wahl, auf der Sie erstandene Produkte von der Liste streichen können. Ein Algorithmus soll Ihre Einkaufsliste »komprimieren«, also nur noch die fehlenden Objekte im Speicher halten. Ihre Einkaufsliste ist ein Array von Strings, ein zweiter Array A gibt für jedes Objekt an, ob sie es schon besorgt haben (kodiert als 0 oder 1).

Ein sequentieller Algorithmus, der die Dinge ja per Definition »nacheinander« angehen muss, sucht nach dem ersten Objekt für das $A[i]$ eine 1 ist und schiebt dieses an die erste Position. Das nächste Objekt mit $A[i] = 1$ kommt an Position 2, das nächste an Position 3 und so weiter. Ein paralleler Algorithmus wird nun versuchen, alle Objekte gleichzeitig an den richtigen Platz zu schieben. Wohin aber mit dem zweiundvierzigsten Objekt? Die korrekte Stelle hat der sequentielle Algorithmus implizit in einem Zähler gehalten. Der parallele Algorithmus braucht die Position »explizit« und – das ist der entscheidende Punkte – diese Stelle ist gerade $A[1] + A[2] + \cdots + A[42]$. Die Präfixsummen sagen also dem parallelen Algorithmus »wo er arbeiten muss«.

3.1 Präfix-Summen

3.1.1 Problemstellung

Ein Problem, das in vielen Anwendungen auftaucht. 3-4

Das Präfix-Summen-Problem

Eingabe Array X von n Zahlen.

Ausgabe Array S von n Zahlen, wobei $S[i]$ die Summe der ersten i Zahlen in X ist.

Beispiel

Eingabe $X = (3,4,1,2,3)$

Ausgabe $S = (3,7,8,10,13)$.

Zur Diskussion

Wie lautet $T^*(n)$?

Zur Übung 3-5

Wir kennen bereits einen Algorithmus für die Summenbildung mit Laufzeit $O(\log n)$ und Arbeit $O(n)$.

Hieraus können wir einen parallelen Algorithmus für das Präfix-Summen-Problem bauen: Wir berechnen einfach parallel alle $S[i]$, indem wir mittels des Summenbildungsalgorithmus die Summe der ersten i Zahlen in X berechnen.

- Schreiben Sie den skizzierten Algorithmus in WT-Repräsentation auf.
- Wie viele Prozessoren braucht der Algorithmus?
- Was ist die Laufzeit?
- Was ist die Arbeit?

3.1.2 Algorithmus

Ein optimaler Algorithmus für das Präfix-Summen-Problem 3-6

```
1  for j ∈ {1,...,n} par do // Input
2      B_0[j] ← X[j]
3
4  // Berechnung von immer längeren Teilsummen
5  for i ← 1 to log n seq do
6      for j ∈ {1,...,n/2^i} par do
7          B_i[j] ← B_{i-1}[2j-1] + B_{i-1}[2j]
8
9  C_{log n}[1] ← B_{log n}[1]
10
11 // Berechnung der finalen Partialsummen; C_i[0] sei immer 0
12 for i ← log n - 1 to 0 seq do
13     for j ∈ {1,...,n/2^i} par do
14         if j ist gerade then
```

```
15            C_i[j] ← C_{i+1}[j/2]
16        else
17            C_i[j] ← C_{i+1}[(j−1)/2] + B_i[j]
18
19  for j ∈ {1,...,n} par do // Output
20      S[j] ← C_0[j]
```

3-7

Der Algorithmus in Aktion.

1

j	0	1	2	3	4	5	6	7	8
$X[j]=B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$									
$B_2[j]$									
$C_3[j]=B_3[j]$									
$C_2[j]$	0								
$C_1[j]$	0								
$S[j]=C_0[j]$	0								

2

j	0	1	2	3	4	5	6	7	8
$X[j]=B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$									
$C_3[j]=B_3[j]$									
$C_2[j]$	0								
$C_1[j]$	0								
$S[j]=C_0[j]$	0								

3

j	0	1	2	3	4	5	6	7	8
$X[j]=B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$		6	3						
$C_3[j]=B_3[j]$									
$C_2[j]$	0								
$C_1[j]$	0								
$S[j]=C_0[j]$	0								

4

j	0	1	2	3	4	5	6	7	8
$X[j]=B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$		6	3						
$C_3[j]=B_3[j]$		9							
$C_2[j]$	0								
$C_1[j]$	0								
$S[j]=C_0[j]$	0								

5

j	0	1	2	3	4	5	6	7	8
$X[j]=B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$		6	3						
$C_3[j]=B_3[j]$		9							
$C_2[j]$	0	6	9						
$C_1[j]$	0								
$S[j]=C_0[j]$	0								

6

j	0	1	2	3	4	5	6	7	8
$X[j]=B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$		6	3						
$C_3[j]=B_3[j]$		9							
$C_2[j]$	0	6	9						
$C_1[j]$	0	3	6	7	9				
$S[j]=C_0[j]$	0								

7

j	0	1	2	3	4	5	6	7	8
$X[j]=B_0[j]$		1	2	0	3	1	0	0	2
$B_1[j]$		3	3	1	2				
$B_2[j]$		6	3						
$C_3[j]=B_3[j]$		9							
$C_2[j]$	0	6	9						
$C_1[j]$	0	3	6	7	9				
$S[j]=C_0[j]$	0	1	3	3	6	7	7	7	9

3.1.3 Analyse

Analyse des Algorithmus 3-8

Satz
Der Algorithmus berechnet die Präfix-Summen in Zeit $O(\log n)$ und Arbeit $O(n)$.

Beweis. Zur Korrektheit siehe Übungsaufgabe 3.1.
Die Laufzeit erhält man sofort dadurch, dass sequentiell lediglich zwei Schleifen jeweils $\log n$ oft durchlaufen werden. Die behauptete Arbeit ergibt sich daraus, dass in der Schleife in Zeilen 5 bis 8 die Arbeit $n\sum_{i=1}^{\log n} 1/2^i < n$ geleistet wird und genauso in der Schleife zwischen 13 und 19. □

3.1.4 Erweiterungen

Es müssen nicht Präfix-Summen sein. 3-9

- Der Algorithmus funktioniert offenbar auch, wenn die Präfix-*Produkte* statt Präfix-*Summen* berechnen wollen.
- Allgemein funktioniert er für alle *assoziativen* Operationen.

Zur Diskussion
Für welche weiteren assoziativen binären Operationen könnte das Präfix-Summen-Problem interessant sein?

3.2 Pointer-Jumping

3.2.1 Problemstellung

Ein zweites Problem, das in vielen Anwendungen auftaucht. 3-10

Das Wurzelproblem

Eingabe Gerichteter Wald.
Ausgabe Für jeden Knoten die Wurzel des Baumes, in dem er liegt.

Zur Diskussion
Sollten die Kanten zur Wurzel hin gerichtet oder weg gerichtet sein?

3.2.2 Algorithmus

Das Problem wird mittels Pointer-Jumping gelöst. 3-11

```
1 for i ∈ {1,...,n} par do
2     while S[i] ≠ S[S[i]] do
3         S[i] ← S[S[i]]
```

3.2.3 Analyse

3-12

Analyse des Pointer-Jumping-Algorithmus.

Satz

Der Pointer-Jumping-Algorithmus löst das Wurzelproblem in Zeit $O(\log h)$ und Arbeit $O(n \log h)$, wobei h die Höhe des Waldes ist (Länge des längsten Pfades).

Beweis. Man zeigt per Induktion, dass sich in jeder Iteration die Höhe aller Bäume halbiert. Dies liegt daran, dass der Weg von einem Knoten zur Wurzel nach einem Jump-Schritt nur noch halb so lang ist, da jeder zweite Knoten weggelassen wurde.
Hieraus ergibt sich eine Laufzeit von $\log h$ und damit eine Arbeit von $O(n \log h)$. □

Zur Übung

Dies ist nicht optimal. Wie lautet $T^*(n)$ und wie lautet der Algorithmus?

3.2.4 Erweiterungen

3-13

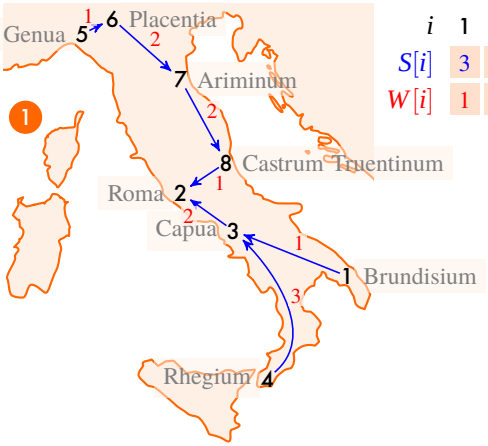
Man kann mehr berechnen als nur die Wurzel.

- Während des Pointer-Jumpings kann man auch gleich noch die Entfernung das Knotens von der Wurzel bestimmen.
- Allgemeiner kann jede Kante eine Zahl oder einen Wert erhalten und wie bei den Präfix-Summen kann man mittels Pointer-Jumpings Pfad-Summen bestimmen.

```
1 for i ∈ {1,...,n} par do
2     while S[i] ≠ S[S[i]] do
3         W[i] ← W[i] ⊗ W[S[i]]
4         S[i] ← S[S[i]]
```

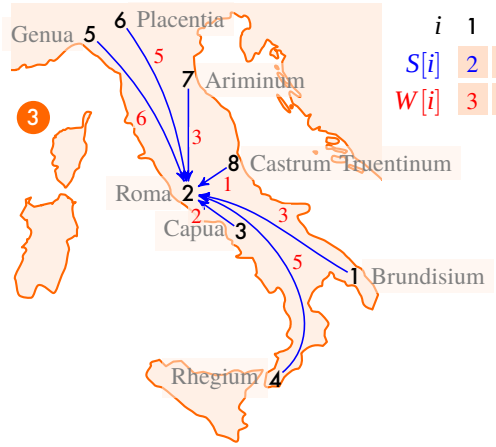
3-14

Beispiel: Alle Wege führen nach Rom.



i	1	2	3	4	5	6	7	8
$S[i]$	3	2	2	3	6	7	8	2
$W[i]$	1	0	2	3	1	2	2	1

i	1	2	3	4	5	6	7	8
$S[i]$	2	2	2	2	7	8	2	2
$W[i]$	3	0	2	5	3	4	3	1



i	1	2	3	4	5	6	7	8
$S[i]$	2	2	2	2	2	2	2	2
$W[i]$	3	0	2	5	6	5	3	1

Zusammenfassung dieses Kapitels

1. *(Verallgemeinerte) Präfix-Summen* kann man in Arbeit $O(n)$ und Zeit $O(\log n)$ berechnen.
2. Mit *Pointer-Jumping* kann man Baumwurzeln in Arbeit $O(n \log h)$ und Zeit $O(\log h)$ berechnen.

Übungen zu diesem Kapitel

Übung 3.1 (Beweis der Korrektheit des Präfixsummenalgorithmus, mittel)

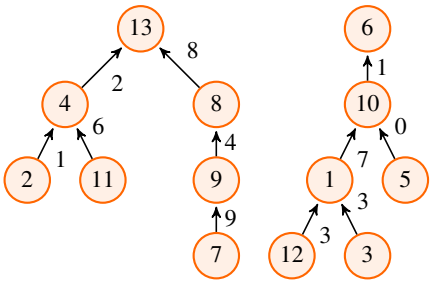
Zeigen Sie, dass der Präfixsummenalgorithmus von Projektion 3-6 korrekt ist.

Tipps: Dies ist eine einfache, aber etwas fummelige Induktion, in der man sich klar machen muss, welche Teilsummen die $B_i[j]$ und $C_i[j]$ jeweils darstellen.

Übung 3.2 (Pointer-Jumping nachvollziehen, leicht)

Es soll Pointer-Jumping benutzt werden, um in einem Wald mit gewichteten Kanten für jeden Knoten das maximale Gewicht auf dem Pfad von der Wurzel des Knotens zur Wurzel zu berechnen.

Geben Sie dazu zu folgendem Wald den Inhalt der Arrays S und W zum Anfang und nach jedem »Jump-Schritt« an:



Übung 3.3 (Sortieren und Präfixsummen, schwer)

Im Speicher einer PRAM ist eine Folge $Z(1), \ldots, Z(n)$ von Zeigern auf n Objekte gespeichert und zusätzlich eine Folge $B(1), \ldots, B(n)$, die jedem Objekt den Wert 0 oder 1 zuordnet. Die Objekte sollen jetzt anhand der B-Werte sortiert werden, das heißt in der Ergebnisliste stehen zuerst alle Zeiger $Z(i)$ mit $B(i) = 0$ und dann die mit $B(i) = 1$. Die Sortierung soll dabei *stabil* sein: Objekte mit gleichem B-Wert sollen nach dem Sortieren in derselben Reihenfolge stehen wie vorher.

1. Geben Sie einen Algorithmus mit Laufzeit $O(\log n)$ und Arbeit $O(n)$ an, der die Objekte wie beschrieben sortiert.
2. Zeigen Sie, dass es einen Sortieralgorithmus für n Zahlen mit k Bits gibt, der eine Laufzeit von $O(k \log n)$ und Arbeit von $O(kn)$ besitzt.
   Tipp: Benutzen Sie Radix-Sort.

Übung 3.4 (Segmentierte Präfix-Summen-Problem, mittel)

Wir betrachten eine Variante des Präfix-Summen-Problems, das sogenannte *segmentierte Präfix-Summen-Problem*. Gegeben ist eine Liste $x = (x_1, \ldots, x_n)$ von Elementen aus einer Menge S und ein assoziativer Operator $\otimes\colon S \times S \to S$. Weiterhin gibt es eine Bitfolge $b = (b_1, \ldots, b_n)$ mit $b_1 = 1$. Ausgegeben wird eine Liste $y = (y_1, \ldots, y_n)$ von Elementen aus S. Die Bitfolge b definiert eine Unterteilung der Listen x und y in Segmente. Ein neues Segment beginnt immer dann, wenn $b_i = 1$ gilt; bei $b_i = 0$ wird das aktuelle Segment fortgesetzt. Das segmentierte Präfix-Summen-Problem besteht nun darin, die Präfix-Summen getrennt für jedes Segment zu berechnen.

Im folgenden Beispiel wird die Liste x durch die Bitfolge b in vier Segmente eingeteil:

b =	1	0	0	1	0	1	1	0	0	0	0
x =	1	2	3	4	5	6	7	8	9	10	11
y =	1	3	6	4	9	6	7	15	24	34	45

1. Wir definieren den Operator $\widehat{\otimes}$ auf Paaren $(a,z), (a',z') \in \{0,1\} \times S$ wie folgt:

$$(a,z) \mathbin{\widehat{\otimes}} (a',z') = \begin{cases} (a, z \otimes z') & \text{if } a' = 0, \\ (1, z') & \text{if } a' = 1. \end{cases}$$

   Zeigen Sie, dass $\widehat{\otimes}$ assoziativ ist.
2. Geben Sie einen Algorithmus an, der das segmentierte Präfix-Summen-Problem einer n-elementigen Liste auf einer EREW-PRAM in Zeit $O(\log n)$ löst.

4-1

Kapitel 4
Paralleles Teilen und Herrschen
Teilbar = parallelisierbar

4-2

Lernziele dieses Kapitels

1. Divide-and-Conquer Algorithmus für die konvexe Hülle kennen
2. Divide-and-Conquer Algorithmus für parallelen Merge-Sort kennen

Inhalte dieses Kapitels

Worum es heute geht

Wie jeder Informatikfeldherr weiß, lassen sich mit dem antiken »Teile und herrsche« viele Probleme in den Griff bekommen. Jedoch: Erstens ist die Redewendung gar nicht antik (siehe unten) und zweitens ist im Parallelen das *Teilen* oft leicht, das *Herrschen* (sprich: Zusammenfügen) weniger. Wie dies doch geht, darum soll es heute gehen.

Aus *de.wikipedia.org/wiki/Divide_et_impera*

> *Divide et impera* (lateinisch für »Teile und herrsche«) ist eine Redewendung und steht für das Prinzip, unter Gegnern Zwietracht und Uneinigkeit zu säen, um so in der Machtausübung ungestört zu bleiben. Es ist angeblich ein Ausspruch des französischen Königs Ludwigs XI. (*diviser pour régner*), die lateinische Version geht vermutlich auf die Renaissance zurück. Heinrich Heine schrieb am 12. Januar 1842 aus Paris: »König Philipp hat die Maxime seines mazedonischen Namensgenossen, das ›Trenne und herrsche‹, bis zum schädlichsten Übermaß ausgeübt.« Der gemeinte Namensgenosse war Philipp II. (359–336 vor Christus), der den größten Teil des Balkans beherrschte. Die Redewendung ist wahrscheinlich nicht antik, wenngleich die damit bezeichnete Strategie sehr alt ist und zum Beispiel in der römischen Außenpolitik ohne Zweifel wiederzuerkennen ist. Sunzi (um 500 vor Christus) beschreibt »Teile und herrsche« als eine Strategie der chinesischen Kriegskunst. [. . .]
>
> Goethe formuliert in »Sprichwörtliches« einen Gegenvorschlag:
>
> Entzwei und gebiete! Tüchtig Wort. –
> Verein und leite! Besserer Hort.

4.1 Konvexe Hüllen

4.1.1 Problemstellung

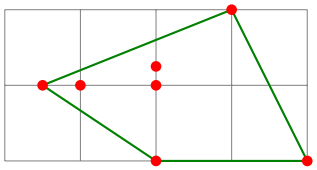
4-4

Das Konvexe-Hülle-Problem in 2D.

Problem

Eingabe Eine *Menge von Punkten* in der Ebene.

Output Die *konvexe Hülle* der Punkte (kleinstes Polygon, das alle Punkte enthält).



Eingabe

Ausgabe

Dieses Problem ist wichtig, wenn man *maximale Entfernungen* berechnen möchte oder testen möchte, ob ein Punkt *innerhalb* einer Punktwolke liegt.

4.1.2 Sequentieller Algorithmus

4-5

Grahams Algorithmus für das Problem.

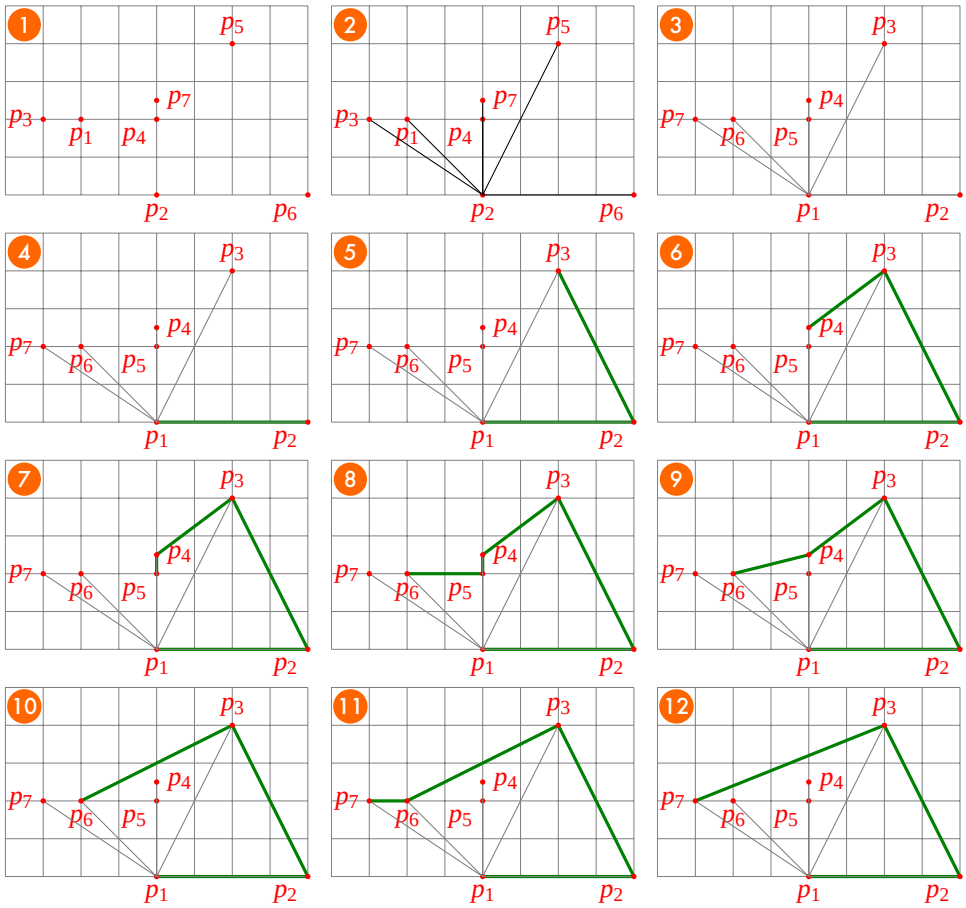
Graham's Scan berechnet *sequentiell* konvexe Hüllen in Zeit $O(n \log n)$.

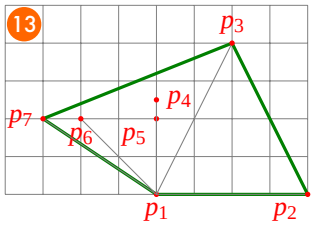
Er funktioniert wie folgt:

1. Bestimme den Punkt mit der kleinsten y-Koordinate.
2. Erklären diesen Punkt zum Ursprung.
3. Sortiere die Punkte nach ihrem Polarwinkel (beispielsweise mittels Merge-Sort).
4. Für jeden Punkt tue nun folgendes:
   - **4.1** Füge den Punkt zur konvexen Hülle hinzu.
   - **4.2** Ziehe an der Hüllenschnur.

4-6

Graham's scan in action.





4.1.3 Paralleler Algorithmus

Ziel: Ein paralleler Algorithmus für die konvexe Hülle. 4-7

- Graham's Scan ist »sehr sequentiell«.
- Für einen (vorzugsweise optimalen) parallelen Algorithmus müssen wir uns etwas Neues einfallen lassen.
- Idee: Divide-and-Conquer.

Divide-and-Conquer besteht aus drei Phasen:

1. Teile die Eingabe in mehrere gleichgroße Teile.
2. Löse das Problem rekursiv auf den Teilen.
3. Füge die Teile zu einem Ganzen zusammen.

Offenbar ist der zweite Schritt leicht zu parallelisieren, die Schwierigkeiten liegen beim ersten und letzten Schritt.

Ein paralleler Algorithmus für die konvexe Hülle. 4-8

Der Algorithmus beginnt mit einer Vorverarbeitung:

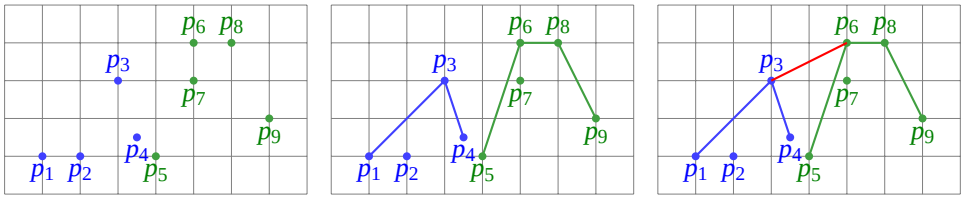
1. Wir sortieren die Punkt nach ihren x-Koordinaten. Dies geht in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$, wie wir später sehen werden.
2. Wir beschränken uns auf das Problem, die obere konvexe Hülle zu finden.

Der eigentliche D&C-Teil geht nun wie folgt:

1. Teile die Punkte die erste und die zweite Hälfte bezüglich den x-Koordinaten.
2. Berechne rekursiv die konvexen Hüllen der Teile.
3. Zum Zusammenfügen berechne die beiden Punkte auf den konvexen Hüllen, »unterhalb« denen alle Punkte liegen.

Die Phasen des parallelen Algorithmus. 4-9

1. Teile die Punkte in die erste und die zweite Hälfte bezüglich den x-Koordinaten.
2. Berechne rekursiv die konvexen Hüllen der Teile.
3. Zum Zusammenfügen berechne die beiden Punkte p^0 und p^1 auf den konvexen Hüllen, »unterhalb« derer Verbindung alle Punkte liegen.



4.1.4 Analyse

4-10

Die Komplexität des Algorithmus.

Lemma
Das Punktepaar (p^0, p^1) kann in Zeit $O(\log n)$ sequentiell berechnet werden.

Beweis. Übungsaufgabe 4.1. □

Satz
Der parallele Algorithmus für die konvexe Hülle benötigt Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$.

Da, wie wir spätern sehen werden, das Sortieren $O(\log^2 n)$ dauert und Arbeit $O(n \log n)$ verursacht, müssen wir nur die Zeit und die Arbeit des Divide-and-Conquer-Teil berechnen.

4-11

Beweis des Satzes – Die Laufzeit.

Analyse der Laufzeit. Für die benötigte Rechenzeit lässt sich folgende Rekursionsformel aufstellen:

$$T(n) = T(n/2) + c \cdot \log n,$$

wobei $T(n/2)$ die in der Rekursion verbrachte Zeit ist und $c \cdot \log n$ die Zeit zur Berechnung der Punkte p^0 und p^1. Das Umkopieren der Listen im Speicher geschieht in Zeit $O(1)$.
Wir behaupten nun, dass für die Lösung $T(n)$ der Rekursion gilt $T(n) \leq d \log^2 n$. Dies zeigt man durch Induktion. In der folgenden Rechnung wird die Induktionsvoraussetzung in der zweiten Zeile angewandt, im letzten Schritt sei d hinreichend groß.

$$\begin{aligned} T(n) &= T(n/2) + c \log n \\ &\leq d \log^2 (n/2) + c \log n \\ &= d \log^2 n + (c - 2d) \log n + d \leq d \log^2 n. \end{aligned}$$

□

4-12

Beweis des Satzes – Die Arbeit.

Analyse der Arbeit. Die Hauptarbeit fällt beim Zusammenfügen der zwei Listen an, nachdem die Rekursion fertig ist. Diese Arbeit ist einmal $O(\log n)$ für die Berechnung der Punkte p^0 und p^1 und dann nochmal $O(n)$, um die Teillisten im Speicher nebeneinander zu kopieren.
Dies ergibt die Formel:

$$W(n) = 2W(n/2) + O(n).$$

Dies ist die gleiche Rekursionsformel wie bei Merge-Sort und hat bekanntermaßen die Lösung $W(n) = O(n \log n)$. □

4.2 Verschmelzen

4.2.1 Problemstellung

Sortieren

4-13

Das Sortierproblem

Eingabe Liste von Zahlen (oder allgemein, Objekten)

Ausgabe Sortierte Liste

Zur Übung

4-14

Geben Sie einen parallelen Sortier-Algorithmus an, der in Zeit $O(\log n)$ und mit Arbeit $O(n^2)$ auskommt.

Tipp: Bestimmen Sie für jedes Element parallel, wo es in der sortierten Liste auftaucht. Diese Position ist gleich der Anzahl der Element, die kleiner als es selbst sind.

Verschmelzen

4-15

- Um das Sortierproblem mit weniger Arbeit zu lösen, können wir Merge-Sort verwenden.
- Offenbar ist nun das Problem, das *Verschmelzen* zu parallelisieren.

Das Verschmelzungsproblem

Eingabe Zwei sortierte Listen von Zahlen

Ausgabe Verschmolzene sortierte Liste

4.2.2 Einfacher paralleler Algorithmus

Eine Definition, die beim einfachen parallelen Verschmelzungs-Algorithmus hilfreich ist.

4-16

Definition

Für eine sortierte Liste A von Zahlen und eine Zahl x ist der *Rang* von x in A die Stelle, an die x in A einsortiert werden müsste.

Beispiel

Der Rang von 15 in der Liste $(1,3,9,14,20,30)$ ist 5.

Offenbar kann der Rang eines Elementes sequentiell in Zeit $O(\log n)$ mittels binärer Suche ermittelt werden.

4-17

Zur Übung

Gegeben seien eine sortierte Liste A und zwei Elemente x_1 und x_2 mit $x_1 < x_2$.

- Wie groß sind parallele Zeit und Arbeit, die Ränge von x_1 und x_2 zu bestimmen?
- Seien r_1 und r_2 die Ränge. An welchen Stellen landen dann x_1 und x_2, wenn man sie *beide* einfügt?

Zur Übung

Wie die vorherige Aufgabe, nur mit drei Zahlen $x_1 < x_2 < x_3$.

4-18

Der einfache parallele Verschmelzungsalgorithmus.

Einfacher Verschmelzungsalgorithmus

Gegeben seien zwei sortierte Listen A und B.

- Bestimme parallel für jedes a_i seinen Rang in B.
- Bestimme gleichzeitig parallel für jedes b_i seinen Rang in A.
- Platziere parallel alle a_i und b_i im verschmolzenen Array nach folgender Vorschrift:
  - Ist r_i der Rang von a_i in B, so platziere a_i an die Stelle $r_i + i - 1$.
  - Ist s_i der Rang von b_i in A, so platziere b_i an die Stelle $s_i + i - 1$.

4-19

Beispielablauf des Verschmelzungsalgorithmus.

Die Schritte des Algorithmus für die Eingaben

$$A = (10, 20, 30, 40, 50, 60, 70, 80, 90)$$
$$B = (1, 2, 3, 41, 42, 43, 81, 82, 100)$$

1. Die Ränge werden bestimmt. Die Ränge der a_i in B und der b_i in A lauten jeweils:

$$r = (4, 4, 4, 4, 7, 7, 7, 7, 9)$$
$$s = (1, 1, 1, 5, 5, 5, 9, 9, 10)$$

2. Die a_i und b_i werden deshalb wie folgt platziert:

Position $r_i + i - 1$	4	5	6	7	11	12	13	14	17
Wert a_i	10	20	30	40	50	60	70	80	90

Position $s_i + i - 1$	1	2	3	8	9	10	15	16	18
Wert b_i	1	2	3	41	42	43	81	82	100

3. Das Resultat ist

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Wert	1	2	3	10	20	30	40	41	42	43	50	60	70	80	81	82	90	100

Die Komplexität des einfachen Verschmelzungsalgorithmus. 4-20

Satz
Der einfache Verschmelzungsalgorithmus benötigt Zeit $O(\log n)$ und Arbeit $O(n \log n)$.

Beweis. Es wird parallel n mal etwas getan, was $O(\log n)$ lange dauert. □

Eine einfache, aber noch nicht ausreichende Folgerung. 4-21

Folgerung
Das Sortierproblem kann in Zeit $O(\log^2 n)$ und Arbeit $O(n \log^2 n)$ gelöst werden.

Beweis. Wieder lassen sich zwei Rekursionsgleichungen aufstellen. Auf jeder Rekursionsstufe wird $O(\log n)$ lange gearbeitet und das macht $O(n \log n)$ Arbeit. Dies liefert:

$$T(n) = T(n/2) + O(\log n),$$
$$W(n) = 2W(n/2) + O(n \log n).$$

Mit Induktion zeigt man nun, dass $T(n) = O(\log^2 n)$ und $W(n) = O(n \log^2 n)$ hiervon Lösungen sind. □

4.2.3 Ausblick: Optimaler Algorithmus

Wie bekommen wir einen optimalen parallelen Algorithmus? 4-22

- Wir sind nicht optimal, da die Arbeit $O(n \log n)$ statt $O(n)$ ist.
- Wir müssen also Arbeit einsparen.
- Der Trick ist, *weniger zu parallelisieren*.
- In der Vorlesung zum *Accelerated Cascading* werden wir sehen, wie dies geht.
- Mit diesem verbesserten Algorithmus werden wir dann in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$ sortieren können.

Zusammenfassung dieses Kapitels

4-23

1. Divide-and-Conquer kann in parallelen Algorithmen ebenfalls genutzt werden.
2. Man kann die konvexe Hülle von n Punkten in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$ berechnet werden.
3. Man kann zwei sortierte Listen der Länge n in Zeit $O(\log n)$ und Arbeit $O(n \log n)$ verschmelzen.

Übungen zu diesem Kapitel

Übung 4.1 (Schnelles Finden einer Tangente, schwer)
Beweisen Sie das Lemma von Projektion 4-10.

Tipps: Führen Sie gleichzeitig auf beiden Teilhüllen eine binäre Suche durch. In jedem Schritt sollten Sie in beiden Teilhüllen jeweils die Hälfte der Punkte eliminieren. Um zu ermitteln, welche Punkte eleminierte werden können, berechnen Sie Tangenten an die aktuellen Punkte und die Teilhüllen. Berechnen Sie, wo diese Tangenten sich treffen und wo dieser Schnittpunkt relativ zu den aktuellen Punkten auf der Tangente liegt (links oder rechts).

Übung 4.2 (Anordnung von Punkten bestimmen, leicht)
Geben Sie ein möglichst einfaches und numerisch stabiles Verfahren an um festzustellen, ob ein Punkt p links oder rechts von der durch die Punkte q_1 und q_2 gehenden Gerade liegt.

Übung 4.3 (Programmodifikation, mittel)

Bei den Algorithmen zum Verschmelzen zweier Listen und beim Partitionierungsalgorithmus hatten wir vorausgesetzt, dass alle Elemente paarweise verschieden sind. Wir betrachten nun Listen, die auch gleiche Elemente enthalten dürfen.

Zeigen Sie, wie man mit Hilfe eines Vorverarbeitungsschrittes und eines Nachbearbeitungsschrittes die Algorithmen auch für solche Listen verwenden kann. Geben Sie für die Vorverarbeitung und die Nachbearbeitung je ein Verfahren an, das konstante Zeit und lineare Arbeit benötigt.

Tipps: Addieren Sie kleine Störwerte auf die Elemente.

Übung 4.4 (Pointer-Jumping, schwer)

Im Speicher einer PRAM sei ein Wald mit n Knoten gespeichert. Dieser ist durch eine Liste $P(1)$, $\ldots$, $P(n)$ repräsentiert, wobei $P(i)$ den Elternknoten des Knotens i angibt.

1. Geben Sie einen Algorithmus an, der die Anzahl der Bäume im Wald bestimmt.
2. Geben Sie einen Algorithmus an, der die Anzahl der Blätter im Wald bestimmt.
3. Geben Sie einen Algorithmus an, der die Größe des größten Baumes im Wald bestimmt. Die Größe eines Baumes ist dabei die Anzahl seiner Knoten. Der Algorithmus soll eine Laufzeit von $O(\log^2 n)$ und eine Arbeit von $O(n \log n)$ haben. Sie dürfen davon ausgehen, dass man n Zahlen in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$ sortieren kann.

5-1

Kapitel 5
Pipelining Algorithmen
Arbeiter am Fließband

5-2

Lernziele dieses Kapitels

1. Konzept des Pipelining verstehen und anwenden können
2. Konzept des 2-3-Baumes verstehen
3. Beispiel eines Pipelining Algorithmus kennen (Einfügen in 2-3-Bäume)

Inhalte dieses Kapitels

Worum es heute geht

Spätestens seit Charly Chaplins Film *Modern Times* weiß man, dass Menschen für Fließbandarbeit nicht geschaffen sind. Ganz anders Computer: Ohne Murren können sie auch die stupideste Aufgabe hingebungsvoll viele Billiarden Male wiederholen, bevor es ihnen zu langweilig wird und sie zur Abwechselung – nur so zur allgemeinen Erheiterung – mal einen Non-Maskable-Interrupt auslösen.

An einem Fließband arbeiten mehrere Arbeiter parallel. Genauso werden beim Pipelining im Rechner unterschiedliche Prozessoren gleichzeitig an unterschiedlichen »Arbeitspaketen« arbeiten. Ist ein Arbeiter mit einem Arbeitsschritt fertig (zum Beispiel dem Anschrauben einer Mutter), so wird das Ergebnis zum nächsten Arbeiter befördert. Analog wird beim einem Pipelining-Algorithmus jeder Rechner ein kleines Arbeitspaket abschließen und dann das Ergebnis an den nächsten Prozessor weiterreichen.

Wir werden heute das Pipelining an einem eher theoretischen Beispiel kennenlernen und auch sonst werden die Pipeliningbeispiel so komplex sein, dass man sie eher nicht in realem Code umsetzen würde. Jedoch ist Pipelining auch vom praktischen Standpunkt aus enorm wichtig: Ohne die *Prozessorpipeline* oder auch *Instruktionspipeline*, welche in allen modernen Prozessoren vorhanden ist und welche letztendlich genau eine parallele Pipeline ist, wie wir sie in der heutigen Vorlesung betrachten werden, wären moderne Prozessoren locker um den Faktor 10 langsamer.

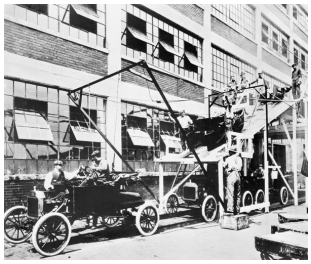
5.1 Pipelining

5-4

Die Idee beim Pipelining.

- *Pipelining* ist *Fließbandarbeit*.
- Verschiedene Personen (Prozessoren) arbeiten *parallel* and mehreren Dingen.
- Nach jedem Arbeitsschritt wird das Ergebnis an den nächsten Prozessor weitergereicht.

5-5



Unknown author, public domain

(Zu) einfache Formalisierung des Pipelining.

Problemstellung

Eingabe Ein Folge von Werte $w_1, w_2, \ldots, w_k$.

Ausgabe Die Folge $f(w_1), f(w_2), \ldots, f(w_k)$.
Dabei ist $f = f_m \circ f_{m-1} \circ \cdots \circ f_1$.

Lösung mittels Pipelining

1. Im ersten Zeitschritt berechnet
   - P_1 den Wert $f_1(w_1)$.
2. Im zweiten Zeitschritt berechnet
   - P_1 den Wert $f_1(w_2)$,
   - P_2 den Wert $f_2(f_1(w_1))$.
3. Im dritten Zeitschritt berechnet
   - P_1 den Wert $f_1(w_3)$,
   - P_2 den Wert $f_2(f_1(w_2))$,
   - P_3 den Wert $f_3(f_2(f_1(w_1)))$.
4. Und so weiter.

5.2 2-3-Bäume

5.2.1 Was sind 2-3-Bäume?

5-6

Was sind 2-3-Bäume?

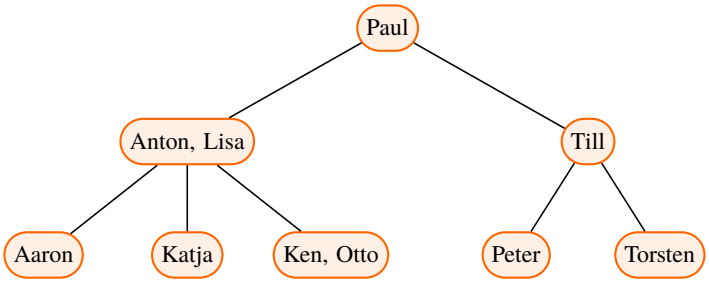
- 2-3-Bäume sind eine *Suchbaumstruktur*.
- Sie unterstützen die Operationen *Einfügen*, *Suchen* und *Löschen*.
- Sie unterscheiden sich von gewöhnlichen (binären) Suchbäumen dadurch, dass
  1. alle Blätter dieselbe Höhe h haben.
  2. jeder innere Knoten genau 2 oder 3 Nachfolger hat.
- Wir nehmen auch noch an, dass nur die Blätter Daten halten (nicht unbedingt nötig).

Zur Diskussion

Welche Vor- und Nachteile haben 2-3-Bäume gegebenüber normalen Suchbäumen?

Beispiel eines 2-3-Baums.

5-7



Die Datenstruktur im Detail.

5-8

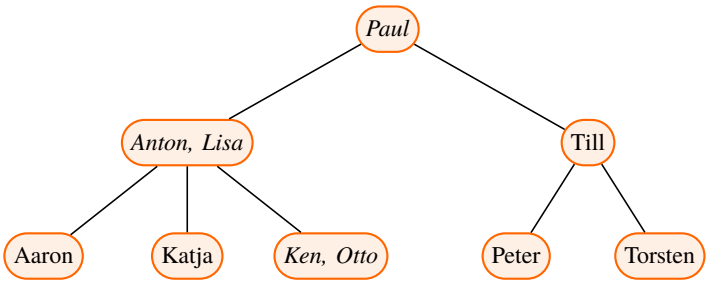
- Jedes *Blatt* hält einen Schlüssel s_i.
- Für innere Knoten mit zwei Kindern v' und v'' gilt:
  - Alle Schlüssel im Teilbaum von v' sind kleiner als die Schlüssel im Teilbaum von v''.
  - $L[v]$ speichert einen Wert, der größer als alle Schlüssel im Teilbaum von v' ist und kleiner als alle Schlüssel im Teilbaum von v''.
    Wir sagen, $L[v]$ *trennt die Kindbäume*.
- Für innere Knoten mit drei Kindern v', v'' und v''' gilt:
  - Alle Schlüssel im Teilbaum von v' sind kleiner als die Schlüssel im Teilbaum von v'', die wiederum kleiner als die Wert von v''' sind.
  - $L[v]$ trennt die Werte in den Teilbäumen von v' und v''.
    $M[v]$ trennt die Werte in den Teilbäumen von v'' und v'''.

5.2.2 Suchen in 2-3-Bäumen

Suchen in 2-3-Bäumen

5-9

In 2-3-Bäumen sucht man wie in normalen Binärbäumen (hier nach *Otto*).



Zur Übung

Es soll parallel nach k Werten $w_1, \ldots, w_k$ in einem 2-3-Baum gesucht werden. Überlegen Sie sich einen Algorithmus und geben Sie $T(n,k)$ und $W(n,k)$ an.
(Wem das zu leicht ist: Das Problem für EREW-PRAMs.)

5.2.3 Einfügen in 2-3-Bäumen

5-10

Einfügen in 2-3-Bäume

Das *Einfügen* eines neuen Knoten geschieht wie folgt:

- Zunächst wird die Einfügestelle *gesucht*.
- Dann wird das Element als neues *Blatt* eingefügt.
  Hierdurch kann ein Knoten bis zu 4 Kinder bekommen.
- Ein Algorithmus *Bereinigen* sorgt dafür, dass Knoten mit mehr als 3 Kindern verschwinden.

5-11

Der Bereinigungsalgorithmus.

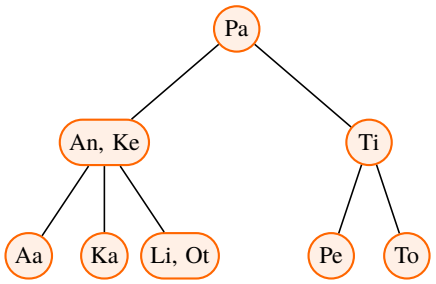
Bereinigung eines Knotens

- *Eingabe* ist ein Knoten *mama* mit *bis zu 6 Kindern*. Sei *großmutter* der Elternknoten von *mama*.
- Falls *mama höchstens 3* Kinder hat, ist nichts zu tun.
- Sonst teile die Kinderliste in zwei Mengen K_1 und K_2. Jeder Teil besteht dann aus höchstens 3 Knoten.
- Mache *mama* zum Elternknoten von K_1.
- Bilde einen neuen Knoten *papa* und mache ihn zum Elternknoten von k_2.
- Füge *papa* als Kind (Schwiegersohn) von *großmutter* hinzu.
- Bereinige *großmutter*.

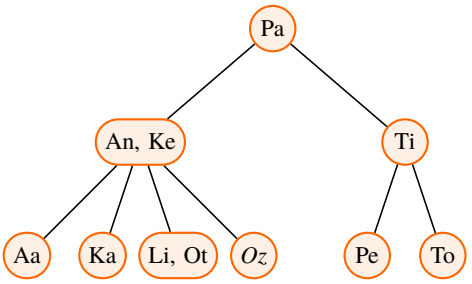
5-12

Einfügen von Ozzie.

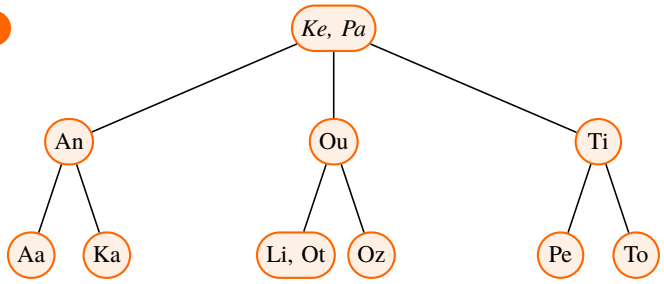
1



2



3



5.2.4 Paralleles Einfügen

Paralleles Einfügen in 2-3-Bäume. 5-13

Problemstellung

Eingabe Ein 2-3-Baum und eine *sortierte* Liste $(b_1, \ldots, b_k)$ von Werten.

Ausgabe 2-3-Baum, in den die Werte eingefügt wurden.

Gesucht ist ein paralleler Algorithmus, der in Zeit $O(\log k + \log n)$ und Arbeit $O(k \log n)$ läuft.

Zur Übung
Wie lautet $T^*(n)$?

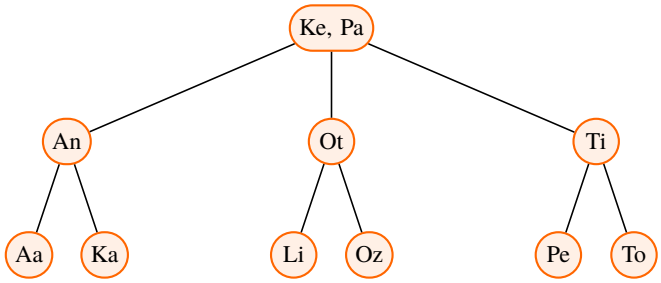
Der Algorithmus für das parallele Einfügen in 2-3-Bäume. 5-14

Der einfache Fall.

Algorithmus (einfacher Fall)

1. Wir ermitteln für jeden Wert, wo er eingefügt werden muss.
2. Wir fassen alle Elemente zu einem *Block* zusammen, welche beim Einfügen Kind desselben inneren Knotens würden.
3. Im *einfachen Fall* haben alle Blöcke Größe höchstens 3.
4. Dann fügen wir alle Werte parallel ein und rufen parallel *Bereinigen* für allen inneren Knoten auf, wo wir etwas eingefügt haben.

Beispiel für den einfachen Fall. 5-15



- Nehmen wir an, wir wollen folgende Liste einfügen: $(A, B, C, Lo, Lp, Lq, Y, Z)$
- Dann bilden sich die *Blöcke* (A, B, C) und (Lo, Lp, Lq) und (Y, Z).
- Da jeder Block Größe höchstens 3 hat, liegt der einfache Fall vor.

Analyse des einfachen Falls. 5-16

Satz
Der Algorithmus für den einfachen Fall fügt die Werte in Zeit $O(\log n)$ und Arbeit $O(k \log n)$ ein.

Beweis.
- Die Baumtiefe ist $O(\log n)$ und der Algorithmus läuft einmal »zu dem Blättern« und dann einmal »zur Wurzel zurück«.
- Da wir dies für alle Elemente der Liste gleichzeitig tun, ist die Arbeit $O(k \log n)$. □

5-17

Der Algorithmus für das parallele Einfügen in 2-3-Bäume.

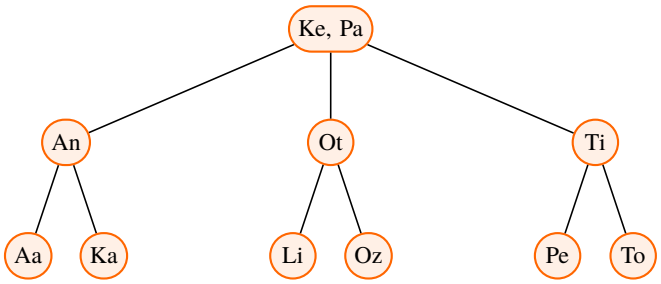
Der schwierige Fall.

Algorithmus (schwieriger Fall)

1. Wir ermitteln für jeden Wert, wo er eingefügt werden muss.
2. Wir fassen alle Elemente zu einem *Block* zusammen, welche beim Einfügen Kind desselben inneren Knoten würden.
3. Im *schwierigen Fall* sind machen Blöcke größer als 3.
4. Solange noch Blöcke größer 3 vorhanden sind, verfahre wie folgt:
   - Von jedem großen Block nehmen wir die *mittleren drei Elemente* heraus. Dies teilt den Rest in zwei neue Blocke auf.
   - Alle herausgenommenen Elemente fügen wir mittels des einfachen Algorithmus ein.

5-18

Beispiel für den schwierigen Fall.



- Nehmen wir an, wir wollen folgende Liste einfügen: (A,B,C,D,E,F,G,H,I).
- Dann bildet sich nur *ein Block*.
- Im ersten Schritt werden nun (D,E,F) mit dem einfachen Algorithmus eingefügt.
- Im zweiten Schritt dann parallel (A,B,C) und (G,H,I).

5-19

Analyse des schwierigen Falls.

Satz

Der Algorithmus für den schwierigen Fall fügt die Werte in Zeit $O(\log k \cdot \log n)$ *und Arbeit* $O(k \log n)$ *ein.*

Beweis.
- Jeder Aufruf des einfachen Verfahrens dauert $O(\log n)$ lange.
- Nach jedem Aufruf ist die Blockgröße halbiert – also gibt es nur $O(\log k)$ Aufrufe. □

5.3 Pipelining und 2-3-Bäume

5.3.1 Verbesserung durch Pipelining

5-20

Was noch zu tun ist.

- Wir wollen die Laufzeit von $O(\log k \log n)$ auf $O(\log k + \log n)$ drücken.
- Dazu wollen wir Pipelining verwenden.
- Es helfen uns zwei Beobachtungen:
  1. Wenn wir die neuen Blöcke bilden, wissen wir schon, wo sie gleich eingefügt werden müssen (nämlich als Kinder der neugebildeten Mama und Papa).
  2. Während *Bereinigen* auf die Großmutter angewandt wird, kann man schonmal die neuen Kinder zur Mama hinzufügen und dann die Mama bereinigen.

Der Algorithmus mit Pipelining.

5-21

Algorithmus (pipelined)

1. Wir ermitteln für jeden Wert, wo er eingefügt werden muss.
2. Wir fassen alle Elemente zu einem Block zusammen, welche beim Einfügen Kind desselben inneren Knoten würden.
3. Im schwierigen Fall sind manche Blöcke größer als 3.
4. Solange noch Blöcke größer 3 vorhanden sind, verfahre wie folgt:
   - Von jedem großen Block nehmen wir die mittleren drei Elemente heraus. Dies teilt den Rest in zwei neue Blocke auf.
   - Starte den einfachen Algorithmus für alle herausgenommen Elemente.
   - Warte nicht, bis der einfache Algorithmus fertig ist, sondern starte sofort mit den neuen Blöcken.

Zusammenfassung dieses Kapitels

5-22

1. *2-3-Bäume* sind immer balanciert.
2. Man kann eine sortierte Liste *optimal parallel* in einen 2-3-Baum einfügen.
3. Zur Beschleunigung haben wir *Pipelining* benutzt.
4. *Pipelining* bedeutet, dass Prozessoren Ergebnisse an andere Prozessoren weiterreichen, während sie selber an neuen Eingaben arbeiten.

6-1

Kapitel 6
Accelerated-Cascading
Schnell und gut

6-2

Lernziele dieses Kapitels

1. Konzept des Accelerated-Cascading verstehen und anwenden können
2. Beispiele von Accelerated-Cascading-Algorithmen kennen

Inhalte dieses Kapitels

Worum es heute geht

Wenn man naiv an den Entwurf paralleler Algorithmen herangeht, so parallelisiert man schnell »alles und jedes«. Dies führt leider schnell dazu, dass die Prozessoren ganz eifrig rechnen, die insgesamt geleistete Arbeit aber schnell uferlos steigt. Dabei ist schon eine parallele Arbeit, die auch nur um einen logarithmischen Faktor größer ist als die optimale sequentielle Laufzeit, nicht akzeptabel: Für eine realitische Eingabegröße wie $n = 1000$ werden wir zu recht die Nase über einen Algorithmus rümpfen, der auf einem System mit zwei Quad-Core-Prozessoren langsamer ist als ein ganz einfacher sequentieller Algorithmus.

Oberstes Gebot bei parallelen Algorithmen muss sein, die Arbeit niedrig zu halten. Was aber tun, wenn durch die Parallelisierung die Arbeit steigt? Weniger parallelsieren! »Accelerated-Cascading« ist der vornehme Fachausdruck hierfür, letztendlich geht es aber einfach darum, weniger zu arbeiten indem man weniger parallelisiert. Natürlich wollen wir das Kind nicht mit dem Bade ausschütten und zu radikal »sequentialisieren«. Der Trick ist, gerade so viel weniger zu parallelisieren, dass die Geschwindigkeit nicht wächst, die Arbeit aber merklich fällt. Dieser Balanceakt gelingt nicht immer, aber meistens.

Das erste Beispiel an dem Accelerated-Cascading heute demonstriert wird ist auf den ersten Blick schon fast beleidigend einfach: Die Bestimmung des Maximums von n Zahlen. Trivialerweise ist dieses Problem sequentiell in Zeit $O(n)$ lösbar und parallel natürlich in Zeit $O(\log n)$ und Arbeit $O(n)$. Die hohe Kunst ist nun, es *deutlich schneller* hinzubekommen. Verblüffenderweise kann man das Maximum von n Zahlen sogar in *konstanter (!) Zeit* berechnen, dies geht aber massiv auf Kosten der Arbeit, welche auf $O(n^2)$ hochschnellt. Mittels Accelerated-Cascasing und weiteren Raffinessen werden wir dies ausbalancieren und eine Laufzeit von $O(\log\log n)$ bei einer Arbeit von $O(n)$ hinbekommen.

6.1 Vorbereitung: Das Maximumproblem

6.1.1 Problemstellung

Die Hauptproblemstellung für heute. 6-4

Das Maximumproblem

Eingabe Eine Liste A von n Zahlen.

Ausgabe Das Maximum von A.

Wie schnell lässt sich das Problem lösen?

1. Offenbar ist $T^*(n) = n$.
2. Es gibt einen EREW-Algorithmus mit $T(n) = O(\log n)$ und $W(n) = O(n)$.
3. Es gibt einen CRCW-Algorithmus mit $T(n) = O(1)$ und $W(n) = O(n^2)$.
4. *Unser Ziel:* Ein CRCW-Algorithmus mit $T(n) = O(\log\log n)$ und $W(n) = O(n)$.

6.1.2 Langsamer, optimaler Algorithmus

Ein langsamer, aber optimaler EREW-Algorithmus für das Maximumproblem. 6-5

Sie sollten in der Lage sein, folgende Aufgabe in 10 Minuten zu lösen.

Lernkontrolle

Schreiben Sie den Programmtext eines EREW-Algorithmus auf Papier auf, der das Maximumproblem in Zeit $O(\log n)$ und Arbeit $O(n)$ löst.

6.1.3 Schneller, verschwenderischer Algorithmus

Ein schneller CRCW-Algorithmus für das Maximumproblem. 6-6

Ziel und Vorbereitung.

Ziel

Ein CRCW-Algorithmus, der das Maximumproblem in Zeit $O(1)$ löst.

Hierzu ist eine kleine Vorüberlegung nützlich.

Satz

Das Maximumproblem kann in Zeit $O(1)$ und Arbeit $O(n)$ gelöst werden, wenn alle Zahlen nur 0 oder 1 sind.

Beweis.

```
1 input A[1], ..., A[n]
2 m ← 0
3 for i ∈ {1,...,n} par do
4     if A[i] = 1 then
5         m ← 1
6 output m
```

□

6-7

Ein schneller CRCW-Algorithmus für das Maximumproblem.
Der Algorithmus.

```
1  input A[1], ..., A[n]
2  // Vorbereitung
3  for i ∈ {1,...,n} par do
4      for j ∈ {1,...,n} par do
5          if A[j] > A[i] then B[i,j] ← 1 else B[i,j] ← 0
6
7  // Zeilenmaxima
8  for i ∈ {1,...,n} par do
9      M[i] ← max{B[i,1],...,B[i,n]}
10
11 // Gesamtmaximum
12 for i ∈ {1,...,n} par do
13     if C[i] = 0 then
14         m ← i
15 output m
```

6-8

Ein schneller CRCW-Algorithmus für das Maximumproblem.
Die Analyse.

Satz
Der CRCW-Algorithmus berechnet das Maximum von n Zahlen in Zeit $O(1)$ und Arbeit $O(n^2)$.

Beweis.
- Die Zeit ist konstant, da nur konstant viele sequentielle Schritte auszuführen sind (nämlich 3 oder 4, je nach Rechnung).
- Es werden nur $O(n^2)$ Prozessoren benutzt. Da diese nur eine konstante Anzahl Schritte arbeiten, kann die Arbeit auch nur $O(n^2)$ sein.

□

6.1.4 Schneller, fast optimaler Algorithmus

6-9

Ein schneller, fast optimaler CRCW-Algorithmus.
Ziel und Vorbereitung.

Ziel
Ein optimaler CRCW-Algorithmus, der das Maximumproblem möglichst schnell löst.

Zur Erinnerung:
- »optimal« bedeutet hier »in Arbeit $O(n)$«.
- Wir können das Problem optimal in Zeit $O(\log n)$ auf einer EREW-PRAM lösen.

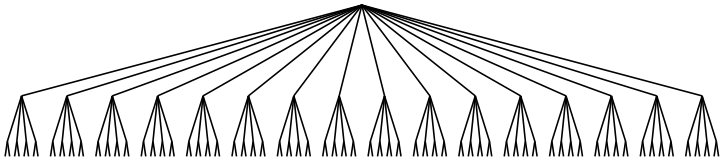
Wir suchen deshalb:
- Einen Algorithmus, der schneller als $O(\log n)$ ist.
- Wir werden in einem späteren Kapitel sehen, dass man nicht schneller als $O(\log\log n)$ werden kann.
- Also setzen wir uns $O(\log\log n)$ *als Ziel*.

6-10

Hilfsmittel: Doppelt-logarithmische Bäume.

Definition (Doppelt-logarithmischer Baum)
Ein *doppelt-logarithmischer Baum* ist wie folgt aufgebaut:
- Alle Blätter liegen auf einer Höhe, genannt *Ebene 0*.
- Ihre Eltern bilden die Ebene 1, deren Eltern die Ebene 2, und so weiter.
- Jeder Knoten auf Ebene 1 hat $2 = 2^1 = 2^{2^0}$ Kinder.
- Jeder Knoten auf Ebene 2 hat $4 = 2^2 = 2^{2^1}$ Kinder.
- Jeder Knoten auf Ebene 3 hat $16 = 2^4 = 2^{2^2}$ Kinder.
- Jeder Knoten auf Ebene 4 hat $256 = 2^8 = 2^{2^3}$ Kinder.
- Jeder Knoten auf Ebene 5 hat $65536 = 2^{16} = 2^{2^4}$ Kinder.
- Jeder Knoten auf Ebene i hat 2^{2^i} Kinder.

Visualisierung des doppelt-logarithmischen Baums mit drei Ebenen. 6-11



Tiefe und Blätteranzahl von doppelt-logarithmischen Bäumen. 6-12

Zur Übung

Wie viele Blätter hat ein doppelt-logarithmischer Baum mit

1. einer Ebene?
2. zwei Ebenen?
3. drei Ebenen?
4. vier Ebenen?
5. fünf Ebenen?
6. i Ebenen?

Folgern Sie: Hat ein doppelt-logarithmischer Baum n Knoten, so ist seine Tiefe $\log(1+\log n) \leq 1+\log\log n$.
(Daher der Name.)

Maximale Anzahl von Kindern in doppelt-logarithmischen Bäumen. 6-13

Lemma

In einem doppelt-logarithmischen Baum mit n Knoten hat die Wurzel $\sqrt{2n}$ Kinder.

Beweis.
- Die Wurzel liegt auf der Ebene der Höhe des Baumes, also auf Ebene $\log(1+\log n)$.
- Knoten auf Höhe i haben $2^{2^{i-1}}$ Nachfolger.
- Also hat die Wurzel

$$2^{2^{\log(1+\log n)-1}} = 2^{2^{\log(1+\log n)}/2} = 2^{(1+\log n)/2} = (2n)^{1/2} = \sqrt{2n}.$$

Kinder.

□

Ein schneller, fast optimaler CRCW-Algorithmus. 6-14

Der Algorithmus.

- Nehmen wir an, dass n die Größe eines doppelt-logarithmischen Baumes ist (also $n = 2^{2^k-1}$ für ein k).
- Wir »stellen uns vor«, dass ein Baum über die Eingaben gelegt wird.

```
1 input A[1], ..., A[n]
2 for i ← 1 to log(1+log n) seq do
3     for alle Knoten k auf Ebene i par do
4         Berechne das Maximum der Kinder von k mit dem schnellen Algorithmus
5         Speichere das Maximum im Knoten k
6 output Wert an der Wurzel
```

6-15

Ein schneller, fast optimaler CRCW-Algorithmus.

Die Analyse.

Satz

Der neue CRCW-Algorithmus berechnet das Maximum von n Zahlen in Zeit $O(\log\log n)$ und Arbeit $O(n\log\log n)$.

Beweis.
- Für die Zeit:
  - Es gibt $O(\log\log n)$ serielle Schleifendurchläufe.
  - In jedem Schleifendurchlauf wird nur konstant lange gerechnet, also ist die Laufzeit $O(\log\log n)$.
- Für die Arbeit:
  - Wir zeigen, dass in jedem Durchlauf höchstens $O(n)$ gearbeitet wird.
  - In jedem Durchlauf werden die Wurzeln von x Teilbäumen der Größe n/x behandelt.
  - Diese Wurzeln haben $\sqrt{2n/x}$ Kinder.
  - Also ist die Arbeit $O\left(x\sqrt{2n/x}^2\right) = O(2n)$. □

6.2 Methodik: Accelerated Cascading

6.2.1 Idee

6-16

Ziel des Accelerated Cascading: Gut und schnell.

- Wir haben einen schnellen, aber nicht optimalen Algorithmus gegeben.
- Weiterhin haben wir einen langsamen, aber optimalen Algorithmus gegeben.
- Ziel von *Accelerated Cascading* ist, sie zu vereinigen und einen schnellen optimalen Algorithmus zu erhalten.

Allgemeine Vorgehensweise

- Teile die Eingabe in viele kleine Teile auf.
- Wende den langsamen Algorithmus auf die kleinen Teile parallel an.
  Dies ist langsam, aber bei kleinen Eingaben nicht schlimm.
- Wende den schnellen Algorithmus auf die Ergebnisse an.
  Dies kostet viel Arbeit, aber es gibt nur noch wenige Eingaben.

6.2.2 Beispiel: Maximumproblem

6-17

Beispiel eines Accelerated Cascading

Ein schneller optimaler CRCW-Algorithmus.

Ziel

Ein CRCW-Algorithmus, der das Maximumproblem in Zeit $O(\log\log n)$ und Arbeit $O(n)$ löst.

Zur Erinnerung:

- Wir haben einen schnellen, aber nicht optimalen Algorithmus, der Zeit $O(\log\log n)$ und Arbeit $O(n\log\log n)$ benötigt.
- Wir haben einen langsamen, aber optimalen Algorithmus mit Zeit $O(\log n)$ und Arbeit $O(n)$.

6-18

Beispiel eines Accelerated Cascading

Ein schneller optimaler CRCW-Algorithmus.

Algorithmus zur Maximumsbildung

1. Teile die Eingabe in Blöcke der Größe $\log\log n$ auf.
2. Bestimme für alle Blöcke parallel das Maximum.
3. Wende den schnellen Algorithmus auf die $n/\log\log n$ Ergebnisse an.

Analyse

- Die ersten beiden Schritte dauern $O(\log\log\log n)$ und machen $O(n)$ Arbeit.
- Der letzte Schritt dauert höchstens $O(\log\log n)$ und die Arbeit ist $O(n/\log\log n \cdot \log\log n) = O(n)$.

6.2.3 Beispiel: Verschmelzung

Beispiel eines Accelerated Cascading

6-19

Ein schneller optimaler Verschmelzungsalgorithmus.

Ziel

Ein Verschmelzungsalgorithmus, der zwei sortierte Listen in Zeit $O(\log n)$ und Arbeit $O(n)$ verschmilzt.

Zur Erinnerung:

- Es war einfach, einen Algorithmus anzugeben, der in Zeit $O(\log n)$ und Arbeit $O(n \log n)$ zwei Listen A und B verschmilzt: Suche mit binärer Suche für jedes Element a_i seine Position r_i in B und platziere a_i und Stelle $r_i + i - 1$, umgekehrt mit den b_i.
- Wir können sequentiell in Zeit $O(n)$ und Arbeit $O(n)$ zwei Listen verschmelzen.

Die Ideen beim optimalen parallelen Algorithmus

6-20

- Anstatt alle Positionen parallel zu bestimmen, tun wir dies nur für jede $(\log n)$-te Zahl.
  - Wir unterteilen also A in Intervalle der Länge $\log n$. Nenne wir die Intervalle $A_1, A_2, \ldots, A_{n/\log n}$.
  - Dann bestimmen wir für *das erste Elemente* jedes Intervalls den Rang dieses Elements in B.
  - Dies unterteilt B in $n/\log n$ Intervalle $B_1, \ldots, B_{n/\log n}$.
- Wären alle B_i von der Größe $O(\log n)$, so könnten wir parallel jedes A_i mit seinem B_i sequentiell verschmelzen und hätten den gewünschten Algorithmus.
- Da die B_i aber im Allgemeinen zu groß sind, wiederholen wir den Algorithmus nun nocheinmal für jedes Paar (A_i, B_i), aber mit vertauschten Rollen.
- Dies liefert Subblöcke $A_{i,j}$ und $B_{i,j}$ die Größe höchstens $O(\log n)$ haben und deshalb alle gemeinsam in Zeit $O(\log n)$ und Arbeit $O(n)$ verschmolzen werden können.

Beispiel einer Verschmelzung

6-21

Aufteilung des ersten Arrays.

Die Eingaben seien

i	1	2	3	4	5	6	7	8	9	10
$A[i]$	1	3	4	7	9	15	17	19	22	24
r_i										
$B[i]$	2	5	8	12	13	14	16	20	21	25

Wir bestimmen nun parallel lediglich die Ränge der *Blockanfänge*.

i	1	2	3	4	5	6	7	8	9	10
$A[i]$	1	3	4	7	9	15	17	19	22	24
r_i	1			3			8			10
$B[i]$	2	5	8	12	13	14	16	20	21	25

Wir müssen nun noch verschmelzen:

1. $A_1 = (1,3,4)$ mit $B_1 = (2,5)$.
2. $A_2 = (7,9,15)$ mit $B_2 = (8,12,13,14,16)$.
3. $A_3 = (17,19,22)$ mit $B_3 = (20,21)$.
4. $A_4 = (24)$ mit $B_4 = (25)$.

Dies ist einfach, außer bei B_2.

6-22

Beispiel einer Verschmelzung

Aufteilung der Blöcke im zweiten Array.

Für die »zu großen« B_i-Blöcke wiederholen wir das Spiel (hier nur für den einzigen zu großen Block gezeigt):

i	1	2	3	4	5
$A_2[i]$	7	9	15		
$B_2[i]$	8	12	13	14	16
s_i					

Wir bestimmen nun parallel wieder lediglich die Ränge der *Subblockanfänge*.

i	1	2	3	4	5
$A_2[i]$	7	9	15		
$B_2[i]$	8	12	13	14	16
s_i	2			3	

Wir müssen also für A_2 und B_2 nun noch verschmelzen:

1. $A_{2,1} = (9)$ mit $B_{2,1} = (8, 12, 13)$.
2. $A_{2,2} = (15)$ mit $B_{2,2} = (14, 16)$.

Nun enthalten sicherlich alle $A_{i,j}$ und $B_{i,j}$ höchstens $\log n$ Elemente.

6-23

Die Komplexität des optimalen Verschmelzungsalgorithmus.

Satz

Der optimale Verschmelzungsalgorithmus benötigt Zeit $O(\log n)$ und Arbeit $O(n)$.

Folgerung

Das Sortierproblem kann in Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$ gelöst werden.

Zusammenfassung dieses Kapitels

6-24

1. Man kann das *Maximum von n Zahlen* bestimmen in
   - *Zeit $O(\log n)$ und Arbeit $O(n)$* auf einer EREW-PRAM.
   - *Zeit $O(1)$ und Arbeit $O(n^2)$* auf einer CRCW-PRAM.
   - *Zeit $O(\log\log n)$ und Arbeit $O(n)$* auf einer CRCW-PRAM.
2. Accelerated Cascading bedeutet, einen schnellen und einen optimalen Algorithmus zu einem *schnellen, optimalen* Algorithmus zu verschmelzen.
3. Man kann zwei sortierte Listen der Länge n in Zeit $O(\log n)$ und Arbeit $O(n)$ verschmelzen.

Übungen zu diesem Kapitel

Herr Lenz ist Manager bei der *Bunte-Bilder AG*, einer großen Beratungsfirma. Derzeit führt Herr Lenz die Planung mehrerer Projekte durch, die alle am 1. Juni beginnen werden. Jedes Projekt besteht aus mehreren Arbeitspaketen. Für jedes Arbeitspaket hat er sich in einer Tabelle aufgeschrieben, wie viele Tage das Paket vermutlich dauern wird und zu welchem Projekt es gehört:

Paketnummer	1	2	3	4	5	6	7	8	...
Arbeitsaufwand	1d	10d	5d	1d	9d	2d	4d	1d	...
Projekt	IBM	SAP	IBM	IBM	SAP	Bund	SAP	Bund	...

Herr Lenz möchte nun wissen, wann welches Arbeitspaket anfangen kann. Beispielsweise können die ersten beiden Pakete am 1. Juni starten, das dritte kann am 2. Juni starten, das vierte am 7. Juni, das fünfte am 11. Juni, das sechste am 1. Juni und so weiter.

Bei den folgenden Aufgaben geht es nun darum, dieses Problem mit einer PRAM in Zeit $O(\log n)$ und Arbeit $O(n)$ zu lösen, wenn n die Anzahl der Pakete ist und es höchstens $\log n$ unterschiedliche Projekte gibt. Dazu soll im Speicher der PRAM ein Array A mit den Arbeitsaufwänden (also $A[1] = 1$ und $A[2] = 10$) gespeichert sein und ein Array P mit Projektnummern (die Projekte seien also durchnummeriert von 1 bis $\log n$, dabei steht dann 1 für die IBM, 2 für SAP und so weiter).

Übung 6.1 (Umwandlung in Vektordarstellung, leicht)

Jedes Arbeitspaket kann als Vektor mit $\log n$ Elementen aufgefasst werden, der an Stelle $P[i]$ den Eintrag $A[i]$ hat und ansonsten nur Nullen. Geben Sie einen Algorithmus an, der zu jedem Paket einen solchen Vektor anlegt. Wie groß sind Zeitbedarf und Arbeit Ihres Algorithmus? (Sie dürfen voraussetzen, dass der Speicher mit Nullen initialisiert ist.)

Übung 6.2 (Präfixsummen bei Vektoren, mittel)

Geben Sie den Code eines Algorithmus an, der auf der Liste der Vektoren die Präfixsummen berechnet. Die Summe zweier Vektoren ist dabei komponentenweise zu verstehen. Der Algorithmus soll eine Laufzeit von $O(\log n)$ besitzen. Beantworten Sie folgende Fragen:

1. Wie groß ist seine Arbeit?
2. Welcher Zusammenhang besteht zwischen den Präfixsummen der Vektoren und den eigentlich gesuchten Anfangszeiten der Pakete?

Übung 6.3 (Accelerated-Cascading, schwer)

Um die Arbeit des Algorithmus zu verringern, soll nun Accelerated-Cascading verwendet werden. Dazu unterteilen wir die Liste der Pakete in $n/\log n$ Blöcke einer Länge von jeweils $\log n$. Für jeden Block lässt sich nun *ein* Vektor mit $\log n$ Einträgen bestimmen, der die Summe der aller Vektoren des Blockes beinhaltet.

Geben Sie einen Algorithmus an (eine Beschreibung genügt, Code ist aber auch in Ordnung), der zu jedem Block einen solchen Vektor bestimmt und anschließend auf alle diese $n/\log n$ Vektoren den Präfixsummen-Algorithmus aus der vorherigen Aufgabe anwendet. Die Laufzeit soll $O(\log n)$ betragen und die Arbeit $O(n)$.

Welcher Zusammenhang besteht zwischen den hier errechneten Präfixsummen der $n/\log n$ Vektoren und den eigentlich gesuchten Präfixsummen?

Übung 6.4 (Bestimmung der fehlenden Werte, schwer)

Erweitern Sie den Algorithmus aus der vorherigen Aufgabe, so dass nun die von Herrn Lenz gesuchten Werte in einer Laufzeit von $O(\log n)$ und Arbeit von $O(n)$ berechnet werden.

7-1

Kapitel 7
Aufbrechen von Symmetrien
Alle Tiere sind gleich, aber manche sind gleicher als andere

7-2

Lernziele dieses Kapitels

1. Problematik der Symmetrie kennen
2. Einfachen 3-Färbealgorithmus kennen
3. Schnellen 3-Färbealgorithmus kennen
4. Optimalen 3-Färbealgorithmus kennen

Inhalte dieses Kapitels

Worum es heute geht

Heute wird es bunt; wir werden munter Knoten in Kreisen und Listen anpinseln, wobei wir aus haushaltstechnischen Gründen mit den Farben sparsam umgehen müssen. Ziel dabei ist weniger eine besonders ästhetische Bemalung einer Liste als vielmehr das Finden von *unabhängigen Mengen*. Zur Erinnerung: Bei einer (gültigen) *Färbung* eines Graphen haben durch Kanten verbundene Knoten unterschiedliche Farben. Folglich bilden alle Knoten der selben Farbe eine *unabhängige Menge*, was sie dazu wiederum prädestiniert, unabhängig (sprich: parallel) verarbeitet zu werden.

Das Färben von Graphen ist eine Wissenschaft für sich. Der berühmte Vier-Farben-Satz besagt, dass sich jeder planare Graph mit vier Farben färben lässt. Anderseits wissen Sie auch, dass die Frage, ob sich ein Graph mit drei Farben färben lässt, NP-vollständig ist. Schließlich ist es wiederum leicht, einen Graphen mit zwei Farben zu färben, falls dies möglich ist (durch eine Breitensuche). Wir wollen heute eine besonders einfache Art von Graphen färben: Listen (eine andere Bezeichnung für gerichtete Pfade in diesem Kontext). Offenbar lässt sich jede Liste mit zwei Farben (zum Beispiel »schwarz« und »weiß«, auch wenn dies nicht gerade Paradebeispiele von »Farben« sind) färben, nämlich immer schön abwechselnd.

Die Kunst ist nun aber, Listen parallel zu färben. Da Listen prinzipbedingt »kreuz und quer« im Speicher liegen, gibt es da ein kleines Problem: Sollte das Listenelement an Speicherstelle 234255 nun schwarz oder weiß gefärbt werden? Das Vorgängerelement liegt übrigens an Speicherstelle 1024 und der Nachfolger an Stelle 65500. Offenbar »sieht man einer Speicherstelle nicht an«, ob sie nun schwarz oder weiß werden soll.

Mit etwas List und Tücke werden wir dieses Problem aber lösen: Zunächst schrauben wir unsere Ansprüche herunter und versuchen lediglich, eine Färbung mit einer »kleinen Anzahl« an Farben zu berechnen. Das ist dann ja schonmal etwas. In weiteren Schritten werden wir die Farbanzahl dann immer weiter reduzieren, bis wir nur noch drei Farben brauchen. Damit werden wir es für heute gut sein lassen, um auf zwei Farben zu kommen werden wir noch eine Menge weiterer List und Tücke brauchen.

7.1 Motivation

Das Problem der Symmetriebrechung. 7-4

Problemstellung

Eingabe Eine durch Zeiger verkettete Liste oder ein Ring von Objekten.

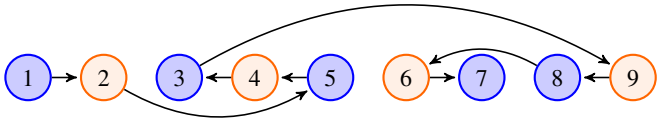
Ausgabe Ein gültige Färbung der Objekte.

Eine *gültige Färbung* bedeutet, dass durch einen Zeiger verbundene Objekte unterschiedlich gefärbt sind.

Offenbar kann man eine Liste immer mit zwei Farben gültig färben, einen Ring immer mit drei Farben.

Eine Liste im Speicher und als Bild. 7-5

Visualisierung einer gültig gefärbten Liste



Speicherung der Liste

Position i	1	2	3	4	5	6	7	8	9
Nachfolger $S(i)$	2	5	9	3	4	7	–	6	8
Farbe $c(i)$	0	1	0	1	0	1	0	0	1

Warum Färbealgorithmen wichtig sind. 7-6

- Gleich gefärbte Objekte stehen *nie nebeneinander*.
- Dies ist unter Umständen beim *Parallelisieren* wichtig: Man erzeugt zuerst eine Färbung, dann bearbeitet man alle gleich gefärbten Objekte gleichzeitig – dabei kommt man sich nicht »ins Gehege«.
- Man spricht statt von Färben auch von *Symmetriebrechung*: In der ungefärbten Liste sieht lokal »alles gleich aus«, in der gefärbten nicht mehr.
- *Anwendungen* werden wir in den nächsten Wochen kennen lernen.

Das Produkt vieler Matrizen. 7-7

Problemstellung

Eingabe Verkettete Liste der Länge n von (4×4)-Matrizen.

Ausgabe Produkt der Matrizen in der durch die Verkettung gegebenen Reihenfolge.

Sie sollten in der Lage sein, folgende Frage beantworten zu können:

Lernkontrolle

Mit welcher Methode kann man dieses Problem in Zeit $O(\log n)$ und Arbeit $O(n \log n)$ lösen?

Eine Anwendung von Färbungen. 7-8

Ziel

Wir wollen die Arbeit auf $O(n)$ zu drücken.

Idee

Wiederhole folgendes:

1. Jeder zweite Prozessor in der Liste berechnet das Produkt seiner Matrix und der Matrix des Nachfolgers.
2. Lösche den Nachfolger aus der Liste.

Problem

Einem Prozessor sieht man nicht an, ob er an einer »ungeraden Stelle« in der Liste auftaucht.

Lösung (grobe Idee)

Wir erstellen eine legale Färbung und bearbeiten dann alle mit einer der Farben gefärbten Matrizen.

7.2 Färbealgorithmen

7.2.1 Einfacher Algorithmus

7-9

Die genaue Problemstellung.

Allgemeine Problemstellung für heute

Eingabe Ein gerichteter Zyklus $G = (V, E)$ mit $V = \{1, \ldots, n\}$. Dieser ist gegeben durch einen Array S, wobei $S[i]$ der Nachfolger von i ist.

Ausgabe Ein gültige Färbung der Objekte, also eine Funktion $C\colon V \to \mathbb{N}$ mit $C[i] \neq c[S[i]]$.

- Unser Ziel ist natürlich, mit möglichst wenigen Farben auszukommen (idealerweise nur 3).
- Offenbar gilt $T^*(n) = O(n)$.

7-10

Ein ganz einfacher Algorithmus.

```
1 for i ∈ {1,...,n} par do
2     C[i] ← i
```

- Dieser Algorithmus liefert eine gültige Färbung.
- Er ist sehr schnell und optimal.
- Er braucht viel zu viele Farben.

7-11

Ein Algorithmus zur Farbreduktion: *ReduceColors*.

```
1 for i ∈ {1,...,n} par do
2     k ← Position des niederwertigsten Bits, wo sich C[i] und C[S[i]] unterscheiden
3     b ← der Wert des k-ten Bits von C[i]
4     C[i] ← 2k + b
```

Beispiel

	Farbe (dezimal)	Farbe (binär)
$C[i]$	36	100100
$C[S[i]]$	8	1000

Wir erhalten $k = 2$ und $b = 0$ und $2k + b = 4$. Die Farbe von $C[i]$ würde also von 36 zu 4 geändert.

7-12

Der Algorithmus erhält gültige Färbungen.

Lemma

Der Algorithmus ReduceColors werde mit einer Liste gestartet, die mit den Farben 0, 1, $\ldots$, $c-1$ *gültig gefärbt ist. Dann ist die Liste hinterher immernoch gültig gefärbt und es werden höchstens* $2\lceil \log_2 c \rceil + 1$ *Farben verwendet.*

Beweis.

- Die behauptete Anzahl der Farben ergibt sich daraus, dass immer $2k + b \leq 2\lceil \log_2 c \rceil + 1$ gilt.
- Für die Korrektheit siehe Übungsaufgabe 7.1.

□

7-13

Eine subtile Fragestellung.

Zur Diskussion

Wie lauten $T(n)$ und $W(n)$ bei einer Anwendung von *ReduceColors*?

Zusammenfassung zum einfachen Färbungsalgorithmus. 7-14

Satz
Wenden wir ReduceColors einmal an, so erhalten wir eine gültige $O(\log n)$-Färbung.
Die Rechenzeit ist $O(1)$ und die Arbeit ist $O(n)$.

Satz
Wenden wir ReduceColors zweimal an, so erhalten wir eine gültige $O(\log\log n)$-Färbung.
Die Rechenzeit ist $O(1)$ und die Arbeit ist $O(n)$.

7.2.2 Schneller Algorithmus

Ein schneller, einfacher Färbungsalgorithmus. 7-15

- Mit Hilfe das einfachen *ReduceColors*-Algorithmus können wir in Zeit $O(1)$ die Farbenzahl von c auf $2\lceil\log_2 c\rceil+1$ verringern.
- Starten wir mit n Farben und wenden wir den Algorithmus
  1. Einmal an, so werden noch $O(\log n)$ Farben verwendet.
  2. Zweimal an, so werden noch $O(\log\log n)$ Farben verwendet.
  3. Dreimal an, so werden noch $O(\log\log\log n)$ Farben verwendet.
- Nach $O(\log^*(n))$ Anwendungen hat man nur noch 6 Farben.

Lemma
Wenden wir ReduceColors $O(\log^* n)$ mal an, so erhalten wir eine gültige 6-Färbung.
Die Rechenzeit ist $O(\log^* n)$ und die Arbeit ist $O(n\log^* n)$.

Von sechs Farben zu drei Farben. 7-16

- Wir wollen nun noch die Farbenanzahl von sechs auf drei reduzieren.
- Dazu benutzen wird folgenden *KnockOutColor* Algorithmus:

```
1 input Färbung C und spezielle Farbe X > 2
2 for i ∈ {1,...,n} par do
3     if C[i] = X then
4         h ← eine Farbe aus {0,1,2}, die weder der Vorgänger noch
5         der Nachfolger von i hat
6         C[i] ← h
```

- Rufen wir nacheinander *KnockOutColor*(3), dann *KnockOutColor*(4) und schließlich *KnockOutColor*(5) auf, so haben wir eine gültige 3-Färbung.

Zur Übung 7-17

1. Machen Sie sich die Funktionsweise von *KnockOutColor* an einem Beispiel klar.
2. Wie groß sind $T(n)$ und $W(n)$?

Zusammenfassung zum schnellen, einfachen Färbungsalgorithmus. 7-18

Satz
Wenden wir ReduceColors $O(\log^* n)$ mal an und dann dreimal KnockOutColor, so erhalten wir eine gültige 3-Färbung.
Die Rechenzeit ist $O(\log^* n)$ und die Arbeit ist $O(n\log^* n)$.

7.2.3 Optimaler Algorithmus

7-19

Ein optimaler Algorithmus.

- Die Arbeit $O(n \log^* n)$ ist zwar nicht optimal, *aber für alle (!) praktischen Zwecke ist* $\log^* n \leq 6$.
- Ein optimaler Algorithmus fängt mit den Aufrufen von *KnockOutColor* früher an:

```
1 call ReduceColors()
2 Sortiere alle Knoten nach ihrer Farbe
3 for i ← 3 to 2⌈log n⌉ seq do
4     call KnockOutColor(i)
```

7-20

Zusammenfassung zum optimalen Färbungsalgorithmus.

Satz

Wenden wir ReduceColors einmal an, sortieren wir die Knoten nach ihrer Farbe und wenden dann $O(\log n)$ mal KnockOutColor an, so erhalten wir eine gültige 3-Färbung. Die Rechenzeit ist $O(\log n)$ und die Arbeit ist $O(n)$.

Beweisskizze.
- Die Korrektheit sollte klar sein.
- Die Sortierung kann in Zeit $O(\log n)$ und Arbeit $O(n)$ erfolgen, siehe Übungsblätter.
- Durch die Sortierung ist sichergestellt, dass in jedem Schleifendurchlauf nur so viele Prozessoren arbeiten müssen, wie Knoten der aktuellen Farbe vorhanden sind. □

Zusammenfassung dieses Kapitels

7-21

1. Man kann einen Ring mit $\log\log n$ Farben in *Zeit* $O(1)$ und *Arbeit* $O(n)$ färben.
2. Man kann einen Ring mit *3* Farben in *Zeit* $O(\log^* n)$ und *Arbeit* $O(n \log^* n)$ färben.
3. Man kann einen Ring mit *3* Farben in *Zeit* $O(\log n)$ und *Arbeit* $O(n)$ färben.

Übungen zu diesem Kapitel

Übung 7.1 (Beweis der Korrektheit von ReduceColors, mittel)

Geben Sie einen detaillierten Beweis des Lemmas von Projektion 7-12 an.

Tipps: Man führe einen Widerspruchsbeweis: Wären am Ende des Algorithmus $C[i]$ und $C[S[i]]$ gleich, so müssten sie auch schon vorher gleich gewesen sein.

8-1

Kapitel 8
Einfache List-Ranking-Algorithmen
1, 2, 3, ...

Lernziele dieses Kapitels

1. Problematik des List-Rankings kennen und anwenden können
2. Einfachen optimalen List-Ranking Algorithmus kennen

8-2

Inhalte dieses Kapitels

Worum es heute geht

Die verkettete Liste gehört unzweifelhaft zu den grundlegendsten Datenstrukturen der Informatik. Man begegnet ihr in solidem, äußerst bodenständigem Code einer industriellen Oberflächenprogrammierung, in den tiefsten Eingeweiden des Betriebssystem innerhalb des guru-gehacktem Assemblercodes des Scheduler, aber auch in der hohen Theorie der reinen Lehre der funktionalen Typtheorie.

Eines der Wesensmerkmale einer Liste ist, dass ihre Elemente in einer *Reihenfolge* stehen. Es gibt eben ein erstes Element, ein zweites, ein drittes und so weiter; und diese Reihenfolge ist im Allgmeinen äußerst wichtig. Nun die »einfache« Problemstellung: Gegeben eine Liste im Speicher, bestimme für jedes Element der Liste, das wievielte es ist (dies nennt man auch den *Rang* des Listenelements). Dieses Problem ist sequentiell sehr einfach mit einem Einzeiler zu lösen (etwa `while(c){c.rank=i++; c=c.next;}`). Jedoch ist diese Schleife »sehr« sequentiell und man wird sich zum parallelisieren etwas Neues ausdenken müssen.

Wenn man im Parallelen nicht mehr weiter weiß, dann helfen bekanntlich immer Präfixsumme oder Pointer-Jumping. So auch hier: Eine Liste ist ja ein entarteter Baum und folglich kann man auf sie Pointer-Jumping anwenden. Damit lässt sich List-Ranking in Zeit $O(\log n)$ bewerkstelligen, jedoch mit einer Arbeit von $O(n \log n)$. Dies ist alles andere als optimal. In der heutigen Vorlesung soll es darum gehen, diese Arbeit zu »drücken«. Dazu kombinieren wir Pointer-Jumping und die Färbealgorithmen aus dem letzten Kapitel mittels Accelerated-Cascading.

Diese Kombination von Algorithmen ist schon eher komplex, obwohl es laut Titel dieses Kapitels um »einfache« List-Ranking-Algorithmen geht. Dies ist dem Umstand geschuldet, dass es im nächsten Kapitel noch deutlich komplizierter wird. Aber für einen schnellen optimalen List-Ranking-Algorithmus ist uns natürlich kein Opfer zu groß.

8.1 Motivation

8.1.1 Ein Anwendungsbeispiel

8-4

Wiederholung: Das Produkt vieler Matrizen.

Problemstellung

Eingabe Verkette Liste der Länge n von (4×4)-Matrizen.

Ausgabe Produkt der Matrizen in der durch die Verkettung gegebenen Reihenfolge.

Lösungsidee 1

Berechne das Produkt durch Pointer-Jumping. Die Rechenzeit ist dabei $O(\log n)$, die Arbeit ist $O(n \log n)$.

8-5

Wiederholung: Das Produkt vieler Matrizen.

Problemstellung

Eingabe Verkette Liste der Länge n von (4×4)-Matrizen.

Ausgabe Produkt der Matrizen in der durch die Verkettung gegebenen Reihenfolge.

Lösungsidee 2

- Bestimmen für jede Matrix ihren Rang in der Liste.
- Schreibe jede Matrix in einen Array an die Stelle ihres Rangs.
- Bilde das Produkt der Elemente im Array (oder sogar Präfixsumme).

Zur Übung

Nehmen wir an, wir können in Zeit $O(\log n)$ und Arbeit $O(n)$ die Position einer Matrix in der Liste bestimmen. Welchen Zeit- und Arbeitsaufwand hat dann obiger Algorithmus?

8.1.2 Das Ranking-Problem

8-6

Das Ranking-Problem

Definition

Eingabe Verkettete Liste der Länge n, gegeben durch die Nachfolger $S[i]$ und Vorgänger $P[i]$ für jeden Knoten in der Liste.

Ausgabe Abstand $R[i]$ des Knotens vom Ende der Liste (der Rang ist dann $n - R[i]$).

Ziele

1. Ein Algorithmus mit Zeit $O(\log n)$ und Arbeit $O(n \log n)$. (Einfach.)
2. Ein Algorithmus mit Zeit $O(\log n \log\log n)$ und Arbeit $O(n)$. (Trickreich.)
3. Ein Algorithmus mit Zeit $O(\log n)$ und Arbeit $O(n)$. (Wahnsinnig kompliziert.)

8.2 Einfaches Ranking

Ein einfacher Ranking-Algorithmus: Pointer-Jumping. 8-7

```
1 for i ∈ {1,...,n} par do
2     if S[i] ≠ i then
3         R[i] ← 1
4     else
5         R[i] ← 0
6 for i ∈ {1,...,n} par do
7     while S[i] ≠ i
8         R[i] ← R[i] + R[S[i]]
9         S[i] ← S[S[i]]
```

Merke
Der Wert $R[i]$ speichert, wie weit $S[i]$ entfernt ist.

Satz
Durch Pointer-Jumping kann man den Rang aller Elemente in Zeit $O(\log n)$ und Arbeit $O(n \log n)$ bestimmen.

8.3 Optimales Ranking

8.3.1 Idee

Die Ideen hinter einem optimalen Ranking-Algorithmus. 8-8

1. Hauptziel ist, die Liste auf die *Größe* $n/\log n$ zu verkleinern. Dann macht *Pointer-Jumping* Arbeit $O(\log n \cdot n/\log n) = O(n)$.
2. Dazu wird wiederholt *ein Viertel der Elemente der Liste* gestrichen und später wieder eingefügt.
3. Zur Identifizierung der streichbaren Elemente benutzen wird *Färbungen*.

8.3.2 Ausklinken eines Elementes

Wir wollen ein Element loswerden. 8-9

Ausgangssituation:

- Für jedes Listenelement i gibt $S[i]$ ein Element an, das in der ursprünglichen Liste später kommt und $R[i]$ gibt die Entfernung zu diesem Element in der ursprünglichen Liste an.
- Ein spezielles Element x ist gegeben.

Ziel:

- Das Element x soll aus der Liste temporär entfernt werden.
- Dazu soll der Vorgänger von x auf den Nachfolger von x zeigen und sein R-Wert soll entsprechend angepasst werden.

Die Algorithmen zum Ausklinken. 8-10

Algorithmus *Suspend*(x).

```
1 Speichere die Werte S[x], P[x] und R[x] lokal am Knoten x
2 S[P[x]] ← S[x]
3 P[S[x]] ← P[x]
4 R[P[x]] ← R[P[x]] + R[x]
```

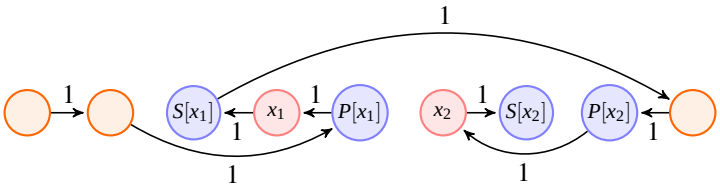
Algorithmus *Reinstall*(x).

```
1 Seien S_orig, P_orig und R_orig die gespeicherten Werte am Knoten x
2 R[x] ← R[S_orig] + R_orig
3 S[x] ← S[S_orig]
```

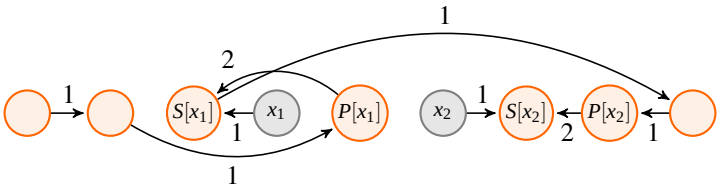
8-11

Beispiel eines Suspends.

Vor dem Suspend von x_1 und x_2



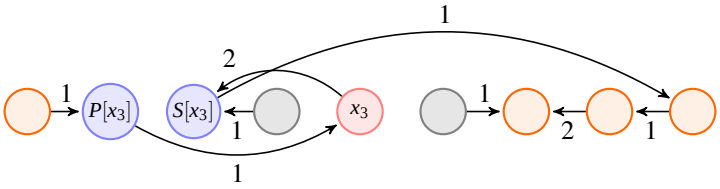
Nach dem Suspend von x_1 und x_2



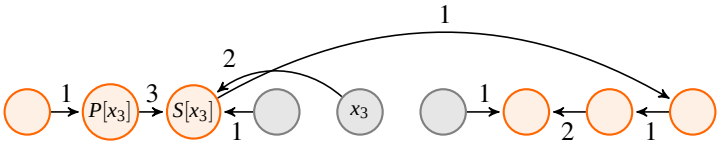
8-12

Beispiel eines weiteren Suspends.

Vor dem Suspend von x_3



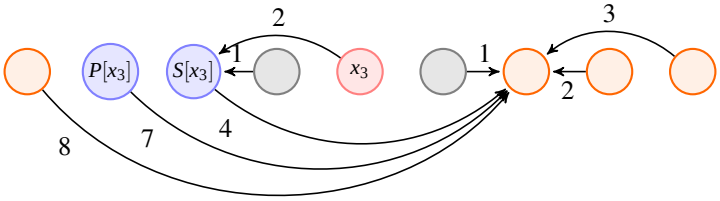
Nach dem Suspend von x_3



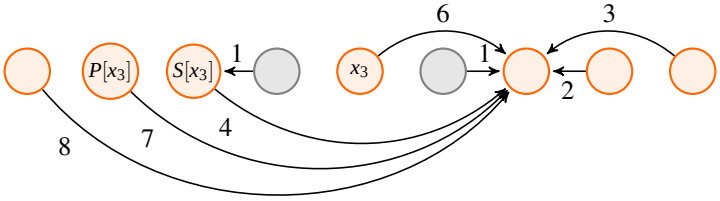
8-13

Beispiel eines Reinstall von x_3.

Die Liste, nachdem Pointer-Jumping stattgefunden hat



Nach dem Reinstall von x_3.



8.3.3 Unabhängige Mengen

Wir wollen viele Elemente loswerden. 8-14

Ausgangssituation:

- Für jedes Listenelement i gibt $S[i]$ ein Element an, das in der ursprünglichen Liste später kommt und $R[i]$ gibt die Entfernung zu diesem Element in der ursprünglichen Liste an.
- Eine Menge X von Elementen.

Ziel:

- Alle Elemente in X sollen parallel aus der Liste temporär entfernt werden.

Wann können wir Elemente parallel ausklinken? 8-15

- Wir können nicht zwei benachbarte Elemente gleichzeitig ausklinken.
- Deshalb muss X eine *unabhängige Menge* sein. Das heißt, kein Element in X ist Nachfolger eines Elementes in X.
- Damit stellt sich das Problem, eine möglichst große unabhängige Menge zu bestimmen.

Wie bestimmt man unabhängige Mengen? 8-16

Idee 1

Man bestimmt eine Färbung. Dann bilden alle Elemente mit der gleichen Farbe eine unabhängige Menge.

Problem

Selbst bei einer 3-Färbung können wir nicht sicher sein, dass eine konkrete Farbe oft vorkommt.

Beispiel

Wir bestimmen eine 3-Färbung und wollen dann alle, sagen wir, *roten* Elemente ausklinken. Dann könnte es passieren, dass es *gar keine roten Elemente gibt*.

Der Trick mit den lokalen Minima. 8-17

Definition

Ein Knoten heißt *lokales Minimum*, wenn die Nummer seiner Farbe kleiner ist als die Farben beider seiner Nachbarn.

Satz

Die lokalen Minima einer Färbung von n Knoten mit k Farben bilden eine unabhängige Menge der Größe mindestens $n/(2k-1)$.

(Beweis ist eine Übungsaufgabe.)

8.3.4 Algorithmus

»Langsames«, optimales Ranking. 8-18

Reduktionsteil

```
1  n_0 ← n
2  k ← 0
3  while n_k > n/log_2 n do
4      Färbe die Liste mit 3 Farben.
5      for i ∈ {1,...,n_k} par do
6          if i ist lokales Farbminimum then
7              call Suspend(i)
8      Kompaktifiziere die Liste
9      k ← k+1
10     n_k ← Länge der kompaktifizierten Liste
11 call PointerJump() auf die Restliste
```

8-19

»Langsames«, optimales Ranking.

Expansionsteil

```
1 for j ← k downto 1 seq do
2     Mache die j-te Kompaktifizierungen rückgängig
3     for i ∈ {1,...,n_k} par do
4         if i wurde bei diesem Schritt ausgeklinkt then
5             call Reinstall(i)
```

8-20

Performance des Algorithmus

Satz

Der Algorithmus berechnet ein Ranking in Zeit $O(\log n \log\log n)$ und Arbeit $O(n)$.

Beweisteil 1: Die Arbeit ist $O(n)$.

- Die Arbeit in jedem While-Schleifendurchlauf ist $O(n_k)$.
- In jedem While-Schleifendurchlauf sinkt die Größe n_k um mindestens ein Fünftel.
- Insbesondere sinkt die Größe in vier Durchläufen um mehr als die Hälfte.

□

Beweisteil 2: Die Zeit ist $O(\log n \log\log n)$.

- Jeder While-Schleifendurchlauf dauert $O(\log n_k) \subseteq O(\log n)$.
- Die Schleife wird nur $O(\log\log n)$ oft durchlaufen:
  - Nach vier Runden gibt es höchstens $n/2$ Knoten.
  - Nach acht Runden gibt es höchstens $n/4$ Knoten.
  - Nach zwölf Runden gibt es höchstens $n/8$ Knoten.
  - Nach sechzehn Runden gibt es höchstens $n/16$ Knoten.
  - Nach $4r$ Runden gibt es höchstens $n/2^r$ Knoten.

  Also gibt es nach $4\log\log n$ Runden höchstens $n/2^{\log\log n} = n/\log n$ Knoten.

□

Zusammenfassung dieses Kapitels

8-21

1. List-Ranking kann mittels Pointer-Jumping in *Zeit* $O(\log n)$ und *Arbeit* $O(n \log n)$ gelöst werden.
2. List-Ranking kann mittels wiederholtem Ausklinken von unabhängigen Mengen in *Zeit* $O(\log n \log\log n)$ und *Arbeit* $O(n)$ gelöst werden.

Übungen zu diesem Kapitel

Übung 8.1 (Eingabeformate ändern, leicht)

Im Speicher einer PRAM liegt eine einfachverkettete Liste mit n Elementen vor, das heißt es gibt für jedes Element i einen Zeiger $S(i)$ auf den Nachfolger von i.

Geben Sie jeweils einen Algorithmus für folgende Probleme an, die jeweils in Zeit $O(1)$ und Arbeit $O(n)$ arbeiten (oder besser):

1. Umwandlung der Eingabe in eine doppeltverkette Liste. Dazu muss zu jedem Element i der Vorgängen $P(i)$ bestimmte werden.
2. Umwandlung der Liste in einen Ring, indem das erste Element zum Nachfolger des letzten Elements gemacht wird.

Übung 8.2 (Modifikation der Suspend-Methode, leicht)

Der Algorithmus *suspend*(x) setzt voraus, dass x weder das erste noch das letzte Element der Liste ist. Modifizieren Sie den Algorithmus so, dass er auch in diesen beiden Fällen korrekt arbeitet.

Wir vereinbaren, dass der Vorgängerzeiger des ersten Elements sowie der Nachfolgerzeiger des letzten Elements auf das jeweilige Element selbst zeigt. Weiterhin gibt es eine Variable *anfang*, die einen Zeiger auf das erste Element der Liste enthält. Stellen Sie sicher, dass das auch nach Ausführung Ihres Algorithmus noch der Fall ist.

Übung 8.3 (Algorithmenanalyse, leicht)

Professor Schlauhmayer hat sich eine Beschleunigung des schnellen optimalen 3-Färbealgorithmus von Seite 7.19 ausgedacht. Sein neuer Algorithmus lautet wie folgt:

```
1 ReduceColors()
2 ReduceColors()
3 Sortiere alle Knoten nach ihrer Farbe
4 for i ← 3 to 2⌈log n log n⌉ + 4 seq do
5     KnockOutColor(i)
```

Der Professor behauptet, dass dieser Algorithmus eine 3-Färbung in Zeit $O(\log\log n)$ und Arbeit $O(n)$ berechnet. Was stimmt hieran nicht? Begründen Sie Ihre Antwort.

Übung 8.4 (Einen Kompaktifizierungsalgorithmus entwerfen, schwer)

Ein Programm benutzt einen Memory-Heap zur Speicherung einer Datenstruktur. Der Memory-Heap besteht aus einem Array M von Zellen, die von 1 bis n nummeriert sind. Jede Zelle i enthält entweder den Wert *leer* oder sie enthält ein Tripel bestehend aus einer reellen Zahl R_i sowie zwei Zeigern p_i und q_i. Die Zeiger p_i und q_i enthalten die Nummer einer nichtleeren Memory-Zelle oder den Wert 0.

Sie sollen einen in Zeit $O(\log n)$ und Arbeit $O(n)$ arbeitenden Algorithmus entwerfen, der den Memory-Heap kompaktifiziert. Dies bedeutet, dass nach Ablauf Ihres Algorithmus die nichtleeren Zellen am Anfang des Memory-Heaps stehen und die Werte und Pointer korrekt angepasst wurden. Dazu soll für jeden Index i einer nichtleeren Zelle im alten Memory-Heap der Inhalt dieser Zelle an eine neue Stelle i' kopiert werden und die Zeiger p_i und q_i sollen zu p'_i und q'_i verändert werden.

Hier ein Beispiel: Der Memory-Heap habe anfangs den Inhalt

Zellenummer i:	1	2	3	4	5	6
R_i:	5	π	`leer`	$-3{,}2$	`leer`	100
p_i:	0	1	–	0	–	4
p_i:	6	6	–	0	–	1

Dann könnte er nach der Kompaktifizierung folgenden Inhalt haben:

Zellenummer i:	1	2	3	4	5	6
R_i:	5	π	$-3{,}2$	100	`leer`	`leer`
p_i:	0	1	0	3		
p_i:	4	4	0	1		

Tipps: Sie dürfen Hilfsarrays verwenden. Führen Sie eine Präfixsumme über einen Array durch, der eine 1 für jede nichtleere Zelle hat.

Übung 8.5 (Beweis des Lokale-Minima-Lemmas, mittel)

Beweisen Sie: Eine Liste der Länge n sei mit k Farben gültig gefärbt. Dann bilden die lokalen Minima der Farben eine unabhängige Menge der Größe mindestens $n/(2k-1)$.

Übung 8.6 (Schneller Algorithmus für Erreichbarkeit in Mengen von Kreisen, mittel)

Als Eingabe sei ein Graph gegeben, der die Vereinigung von disjunkten gerichteten Kreisen ist, sowie zwei Knoten s und t. Wie üblich ist der Graph durch ein Array S gegeben, wobei $S(i)$ der Nachfolgeknoten von i ist. Geben Sie einen Algorithmus an, der entscheidet, ob s und t im selben Kreis liegen. Der Algorithmus soll in Zeit $O(\log n)$ arbeiten, die Arbeit ist egal.

Tipp: Machen Sie t zum Nachfolger von sich selbst und führen Sie dann Pointer-Jumping durch.

Übung 8.7 (Arbeitsoptimalen Algorithmus finden, schwer)

Geben sie einen Algorithmus für dasselbe Problem wie in der vorigen Aufgabe an mit einer Laufzeit von $O(\log n \log\log n)$ und Arbeit $O(n)$. Es genügt eine grobe Beschreibung des Algorithmus und jeweils ein kurzes Argument, weshalb die Arbeit und die Laufzeit die behaupteten sind.

9-1

Kapitel 9
Schneller optimaler List-Ranking-Algorithmus
Wenn es unbedingt schnell gehen muss

9-2

Lernziele dieses Kapitels

1. Schnellen optimalen List-Ranking Algorithmus grob kennen
2. Analyse des Algorithmus grob verstehen

Inhalte dieses Kapitels

Worum es heute geht

Dieses Kapitel ist nichts für Leute mit schwachen Nerven und dies gleich aus mehreren Gründen:

1. Es geht recht blutig zu: süße kleine Schafe werden von bösen roten Wölfen gefressen. (In alten Fassungen dieser Vorlesung gab es statt der »Wölfe« etwas tierlieber »Hirten«. Aber auch bei den Hirten hatte nach jeder Runde jeder Hirte ein Schaf weniger, was nie zufriedenstellend begründet werden konnte.)
2. Es geht algorithmisch komplex zu: der Algorithmus besteht aus mehreren verwobenen Ideen, wobei wir alles, was wir bisher gelernt haben, in verschiedener Weise einsetzen werden.
3. Es wird analysetechnisch schwierig: Es ist schon nicht offensichtlich, dass der Algorithmus korrekt arbeitet – aber man kann sich mit etwas gutem Willen davon überzeugen. Richtig schwierig ist es zu zeigen, dass der Algorithmus tatsächlich auch arbeitsoptimal ist. Wir werden dazu eine amortisierte Analyse durchführen müssen, bei der zu allem Überfluss auch noch die so genannte *Kostenverteilung* nicht schön gleichmäßig ist, sondern eher die Komplexität des Bundeshaushalts erreicht.

Nehmen wir an, Sie haben starke Nerven. Was bringt dann alle die Mühe? Sie werden sich an einem *arbeitsoptimalen* List-Ranking-Algorithmus mit einer Laufzeit von $O(\log n)$ erfreuen können. Ein solcher Algorithmus ist in der Tat eine erfreuliche Sache: Ab sofort können Sie in Ihren Algorithmen nicht nur schreiben ». . . und dann einmal Präfixsumme anwenden, was weder die Arbeit noch die Zeit erhöht, und dann. . . «, sondern nun auch ». . . und dann einmal List-Ranking anwenden, was weder die Arbeit noch die Zeit erhöht, und dann. . . «.

9.1 Das Ziel

Die Problemstellung und was wir schon wissen. 9-4

Eingabe Verkettete Liste der Länge n, gegeben durch die Nachfolger $S[i]$ und Vorgänger $P[i]$ für jeden Knoten in der Liste.

Ausgabe Abstand $R[i]$ des Knotens vom Ende der Liste (der Rang ist dann $n - R[i]$).

- Lösbar in *Zeit* $O(\log n)$ und *Arbeit* $O(n \log n)$ (Pointer-Jumping).
- Lösbar in *Zeit* $O(\log n \log\log n)$ und *Arbeit* $O(n)$ (iteriertes Färben und Ausklinken).

Ziel
Algorithmus mit *Zeit* $O(\log n)$ und *Arbeit* $O(n)$.

9.2 Die Lösung

9.2.1 Idee

Die grobe Idee. 9-5

Wir beginnen ähnlich wie beim letzten Mal:

- Wir führen eine *Vorverarbeitung* durch.
- Dabei wird die Liste von n Knoten auf $n/\log n$ *Knoten geschrumpft*.
- Dazu werden wiederholt *unabhängige Mengen* von Knoten »ausgeklinkt«.
- Sobald die Länge der Liste auf $n/\log n$ geschrumpft ist, können wir Pointer-Jumping durchführen und die Knoten wieder einfügen.

Neu ist nun:

- Wir *garantieren*, dass die Liste schnell schrumpft.
- Dazu klinken wir Knoten nicht »irgendwo« aus, sondern etwas kontrollierter.

Zur Erinnerung: Ausklinken. 9-6

Ausgangssituation:

- Für jedes Listenelement i gibt $S[i]$ ein Element an, das in der ursprünglichen Liste später kommt und $R[i]$ gibt die Entfernung zu diesem Element in der ursprünglichen Liste an.
- Ein spezielles Element x ist gegeben.

Ziel:

- Das Element x soll aus der Liste temporär entfernt werden.
- Dazu soll der Vorgänger von x auf den Nachfolger von x zeigen und sein R-Wert soll entsprechend angepasst werden.
- Da wir nicht zwei benachbarte Elemente gleichzeitig ausklinken können, muss die Menge X der ausgeklinkten Elemente eine *unabhängige Menge* sein.

9.2.2 Die Blockbildung

9-7

Wir teilen die Liste in Blöcke ein.

Idee der Blockbildung

- Um mehr Kontrolle über das Ausklinken zu erhalten, teilen wir die Liste in Blöcke B_j der Länge $\log n$ auf.
- Diese Aufteilung erfolgt in Bezug auf die Reihenfolge, wie die Daten im Speicher stehen, nicht in Bezug auf die Listenreihenfolge.
- Die Position eines Knotens innerhalb eines Blocks nennen wir seine *Höhe*.

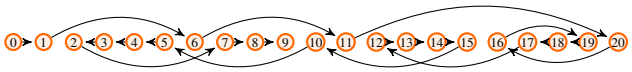
Grobes Ziel

- Wir wollen (idealerweise) in jedem Schritt aus jedem Block ein Element ausklinken.
- Dies soll in Zeit $O(1)$ und Arbeit $O(n/\log n)$ geschehen.
- Dann sind nach $O(\log n)$ Schritten nur noch $O(n/\log n)$ Knoten vorhanden.

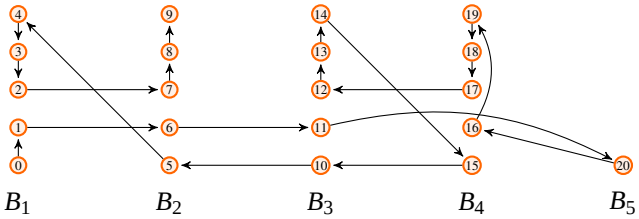
9-8

Visualisierung der Ausgangssituation.

Die Liste im Speicher



Die Liste in Blöcke der Größe $\lceil \log_2 20 \rceil = 5$ aufgeteilt.



9-9

Der Fokus in den Blöcke.

Definition

- Für jeden Block B_j bezeichnet f_j den aktuellen *Fokus*.
- Der Fokus wir in der Regel in jedem Schritt hochgesetzt, er kann aber auch gleich bleiben.
- Wenn der Block leer ist, so ist der Fokus nicht definiert.

9.2.3 Knotenzustände

9-10

Woran gerade gearbeitet wird.

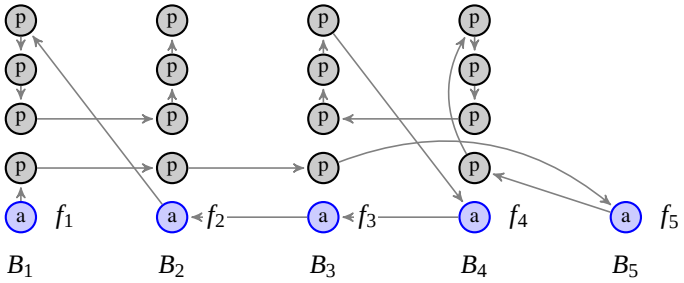
Jeder Knoten hat immer genau einen der folgenden Zustände:

active	Gerade fokussierter Knoten, den man am liebsten loswerden würde (blau).
passive	Kommt erst später dran (schwarz).
suspended	Schon ausgeklinkt. Wird nicht mehr beachtet. (grau).
isolated	Isolierter Knoten (orange).
wolf	Ein Wolfsknoten (rot).
sheep	Ein Schafsknoten (grün).

Da Knoten heute recht komplexe Gebilde sind, werden wir ab jetzt die *Objektnotation* verwenden und *i.successor* statt $S[i]$ schreiben und *i.state* für den Zustand und so weiter.

Visualisierung der Ausgangssituation

9-11



Ideen, wie wir Knoten loswerden.

9-12

Idee

1. Wir wollen in jeder Runde möglichst alle aktive Knoten loswerden (durch Ausklinken).
2. Danach wollen wir die Fokusse um 1 erhöhen und wiederholen dann Schritt 1.

Zur Diskussion

1. Warum können wir nicht einfach in jeder Runde alle aktiven Knoten ausklinken?
2. Was könnte man stattdessen tun?

9.2.4 Wölfe und Schafe

Wie finden wir Knoten zum Ausklinken?

9-13

Ziel und Problem

Ziel Wir wollen in jeder Runde möglichst viele aktiven Knoten loswerden (durch Ausklinken).

Problem Diese Knoten bilden im Allgemeinen keine unabhängige Menge.

Idee 1

- Wir ermitteln eine 3-Färbung der Knoten.
- Dies induziert (zum Beispiel durch die Farbminima) eine unabhängige Menge.
- Diese können wir ausklinken.

Problem: Die Rechenzeit von $O(\log n)$ ist viel zu hoch, denn wir haben nur $O(1)$ Zeit.

Eine neue Idee, um schneller Knoten zum Ausklinken zu finden.

9-14

Idee 2

- Wir ermitteln in Zeit $O(1)$ eine $\log\log n$-Färbung der Knoten (zweimal *ReduceColors*).
- Dies induziert (zum Beispiel durch die Farbminima) eine unabhängige Menge.
- Diese können wir ausklinken.

Vorteil: Die Rechenzeit ist nur $O(1)$.
Problem: Wir entfernen nur einen Anteil von $O(1/\log\log n)$ Knoten.

Wie man mehr Knoten los wird.

9-15

Idee 3

- Nehmen wir an, wir haben eine unabhängige Menge ermittelt.
- Es ist aber nur jeder $(\log\log n)$-te Knoten in der Menge.
- Dann bilden aber die *Nachfolger* dieser Knoten wieder eine unabhängige Menge. Diese können wir also danach entfernen.
- Und ebenso deren Nachfolger. Diese können wir dann danach entfernen.

Vorteil: Wenn alles gut läuft, dann kann in jedem Schritt jeder $(\log\log n)$-te Knoten ausgeklinkt werden.
Problem: Spätestens nach $\log\log n$ Runden bilden die Nachfolger keine unabhängige Menge mehr.

9-16

Mit Pipelining zum Sieg.

Idee 4

- Die vorherige Idee war gut, auch wenn sie nicht ganz funktionierte.
- Das Problem ist, dass gegen Ende viele Knoten nichts mehr tun, da sie schon ausgeklinkt wurden.
- Deshalb benutzen wir *Pipelining*, um diesen Knoten Arbeit zu verschaffen.

Problem: Jetzt wird es unübersichtlich.

9-17

Einteilung in Wölfe und Schafe.

Wie wir die aktiven Knoten aufteilen

In jedem Schritt wechseln die *aktiven Knoten* ihren Zustand in einen der folgenden:

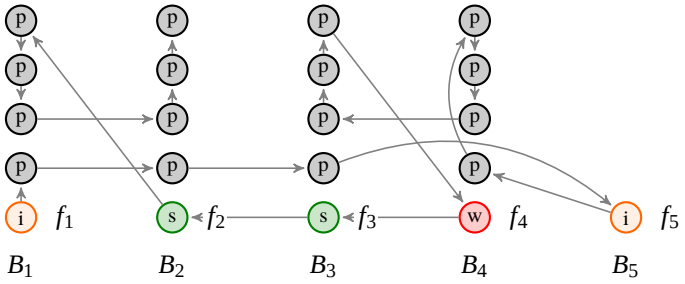
isolated Knoten, bei denen weder der Vorgänger noch der Nachfolger aktiv ist.

wolf Lokale Farbminima einer $(\log\log n)$-Färbung der aktiven Knoten.

sheep Alle anderen aktiven Knoten.

9-18

Einteilung der Knoten in isolierte Knoten, Schafe und Wölfe



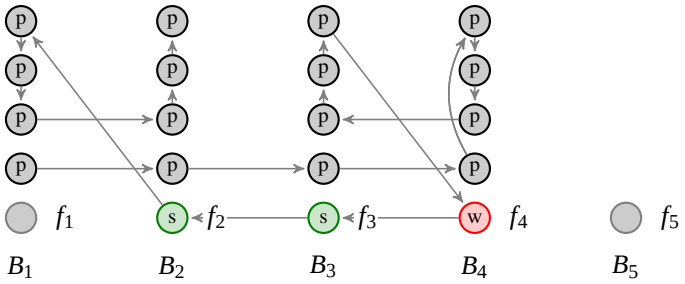
9-19

Beobachtungen zu Wölfen und Schafen.

- Wir können die Einteilung in isolierte Knoten, Wölfe und Schafe in Zeit $O(1)$ durchführen.
- Wir mögen isolierte Knoten, da man sie sofort ausklinken kann.
- Wölfe bilden eine unabhängige Menge. Deshalb bilden auch die den Wölfen direkt folgenden Schafe eine unabhängige Menge.

9-20

Die isolierten Knoten werden suspendiert.



9-21

Was Wölfe und Schafe tun.

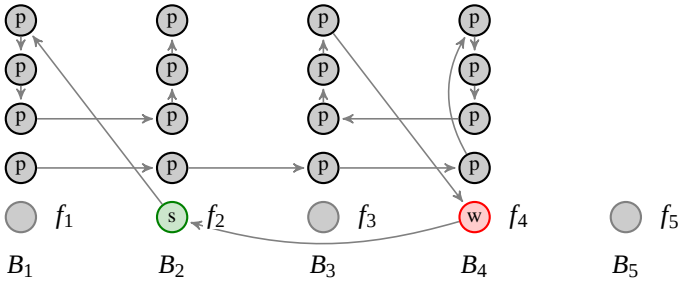
Was Wölfe tun:

- In jeder Runde *fressen* die Wölfe die Schafe, die ihnen direkt folgen.
- Hat ein Wolf alle seine Schafe gefressen, ist er kein Wolf mehr und wechselt in den Zustand aktiv zurück.
- Er kann dann als isolierter Knoten oder als Schaf oder als Wolf wiedergeboren werden.
- Wölfe behalten den Fokus.

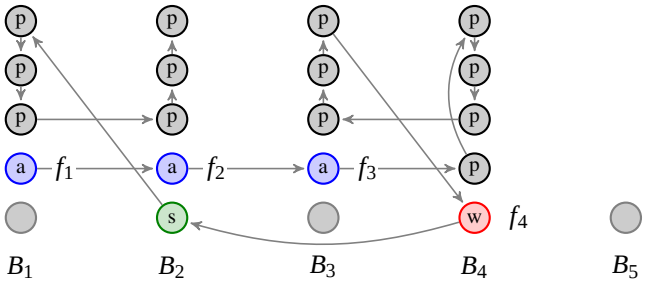
Was Schafe tun:

- Schafe warten geduldig darauf, gefressen zu werden.
- Schafe behalten *nicht* den Fokus, dieser wird einfach hochgeschoben.

Wölfe fressen ein Schaf durch Suspendierung. 9-22



Die Fokusse rücken vor und Knoten werden aktiv. 9-23



9.2.5 Algorithmus

Der ganze Algorithmus. 9-24

- Der Algorithmus verläuft in Runden.
- In jeder Runde geschehen folgende Dinge nacheinander:
  1. Die aktiven Knoten werden in isolierte Knoten, Wölfe und Schafe eingeteilt.
  2. Isolierte Knoten werden entfernt und Wölfe fressen ihre Schafe. Dadurch verlieren sie vielleicht ihren Status als Wolf und werden wieder aktive Knoten.
  3. Der Fokus wird aktualisiert: Bei Schafen und suspendierten Knoten wird der Fokus erhöht und neu fokussierte Knoten wechseln von passiv auf aktiv.
- Nach $O(\log n)$ Runden stoppen wir und wenden Pointer-Jumping an.

Die Einteilung im Detail. 9-25

```
1 call ReduceColors () für die aktiven Knoten
2 call ReduceColors () für die aktiven Knoten
3 for alle Blocknummern j par do // das heißt j ∈ {1,...,n/log n}
4     if f_j.state = active und f_j ist isoliert then
5         f_j.state ← isolated
6     else if f_j.state = active und f_j.color ist Farbminimum then
7         f_j.state ← wolf
8     else
9         f_j.state ← sheep
```

(Tatsächlich machen wir auch den ersten und letzten Knoten einer Kette zum Wolf und den zweiten Knoten einer Kette zum Schaf.)

9-26

Das große Fressen im Detail.

```
1  // Fressen
2  for alle Blocknummern j par do
3      if f_j.state = wolf
4          s ← f_j.successor
5          call Suspend(s)
6          if s ist letztes Schaf then
7              f_j.state ← active
8
9  // Aufräumen isolierter Knoten
10 for alle Blocknummern j par do
11     if f_j.state = isolated then
12         call Suspend(f_j)
```

9-27

Vorrücken des Fokus.

```
1  for alle Blocknummern j par do
2      if f_j.state ∈ {suspended, sheep} then
3          // Kann weitergehen
4          f_j ← f_j + 1
5          if f_j ist über die Blockgrenze hinausgelaufen then
6              ignoriere f_j und Block j im Folgenden
7          else
8              f_j.state ← active
```

9-28

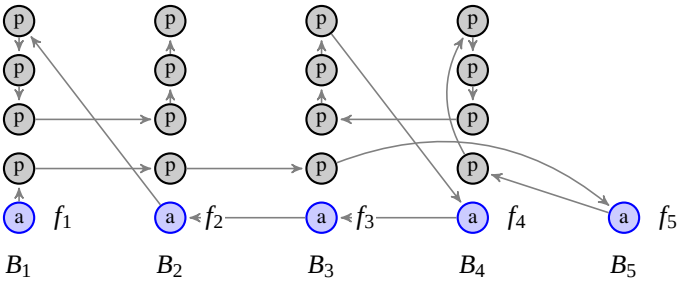
Zusammenfassende Beobachtungen zu den Zuständen.

- Jeder Knoten ist immer in einem der Zustände suspended, active, passive, isolated, wolf und sheep.
- Der Fokus liegt am Anfang einer Runde immer auf Wölfen und auf aktiven Knoten.
- *Unterhalb* des Fokus finden sich in einem Block nur Schafe und ausgeklinkte Knoten.
- *Oberhalb* des Fokus finden sich in einem Block nur passive Knoten.

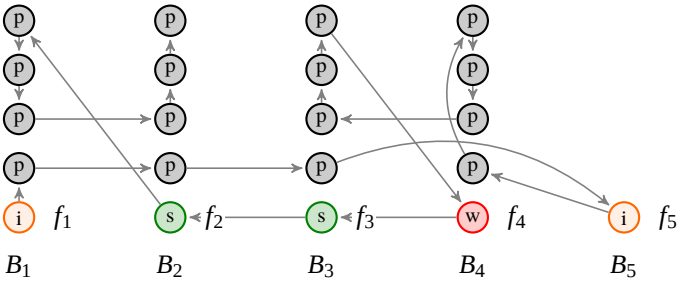
9-29

Der Algorithmus in Aktion.

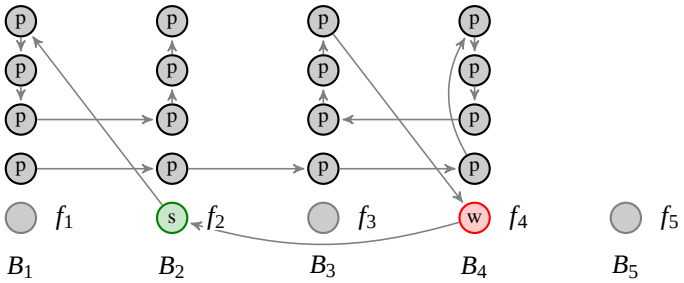
Ausgangssituation:



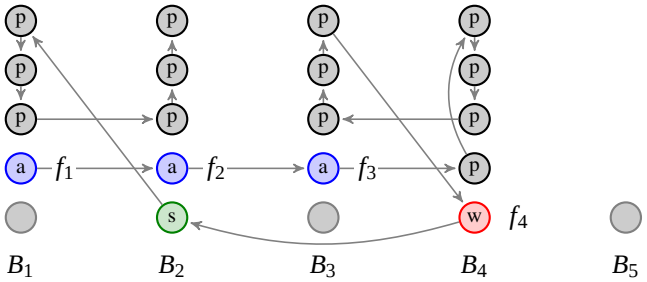
. . . Einteilung der aktiven Knoten in isolierte Knoten, Schafe und Wölfe. . .



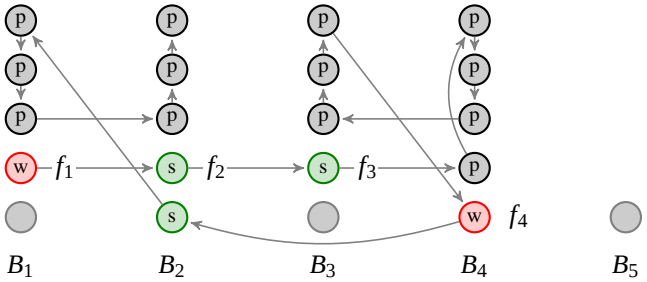
. . . Wölfe fressen ein Schaf durch Suspendierung und isolierte Knoten werden ebenfalls suspendiert. . .



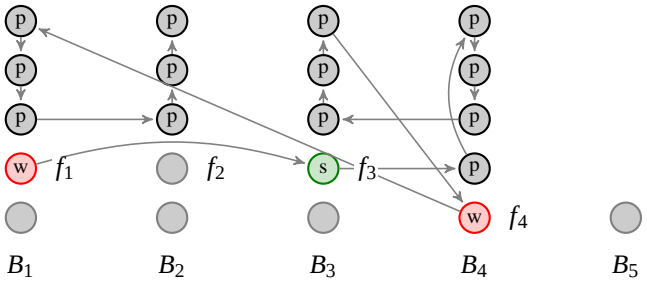
...Die Fokusse werden vorgerückt und Knoten aktiv gemacht ...



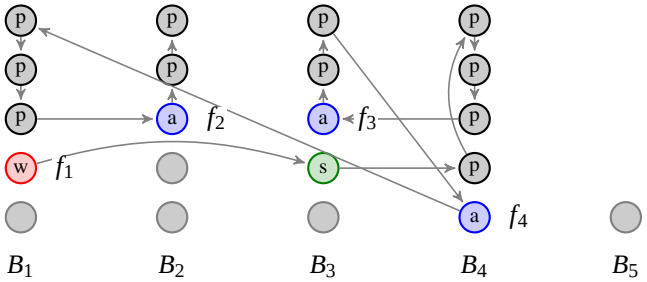
...Einteilung der aktiven Knoten in isolierte Knoten, Schafe und Wölfe...



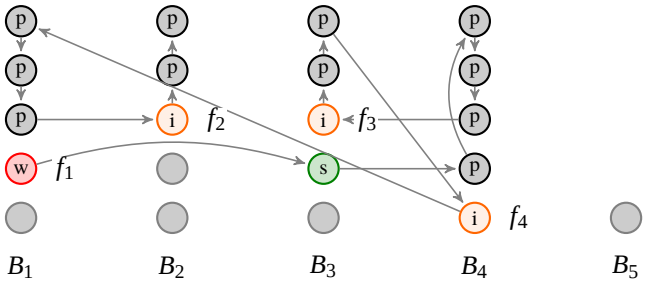
...Wölfe fressen ein Schaf durch Suspendierung und isolierte Knoten werden ebenfalls suspendiert...



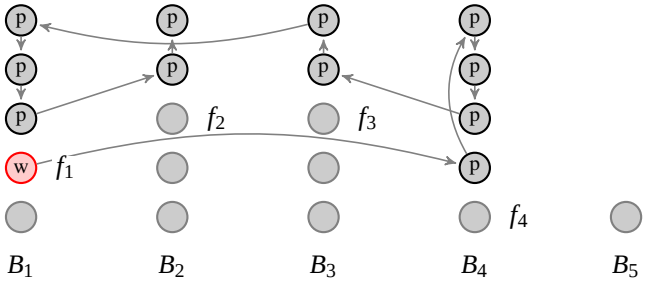
...Die Fokusse werden vorgerückt und Knoten aktiv gemacht, ein Wolf ohne Schafe wird als aktiver Knoten wiedergeboren...



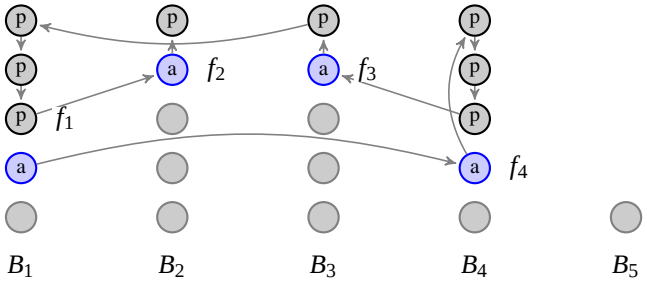
...Alle aktiven Knoten sind isoliert...



. . . Wölfe fressen ein Schaf durch Suspendierung und isolierte Knoten werden ebenfalls suspendiert. . .



. . . Die Fokusse werden vorgerückt und Knoten aktiv gemacht, ein Wolf ohne Schafe wird als aktiver Knoten wiedergeboren.



9.3 Die Analyse

9.3.1 Plan

9-30

Die Laufzeit und die Arbeit.

Unser Ziel ist es nun, folgenden Satz zu beweisen:

Satz

Der Algorithmus berechnet List-Rankings in Zeit $O(\log n)$ und Arbeit $O(n)$.

Beweisplan

- Wir analysieren zuerst die Zeit und Arbeit einer Runde.
- Dann zeigen wir, dass nach $O(\log n)$ Runden die Anzahl der Knoten, die nicht im Zustand suspended sind, nur noch $O(n/\log n)$ ist.

9-31

Die Laufzeit für eine Runde.

Hier nochmal, was alles in einer Runde passiert:

1. Die aktiven Knoten werden in isolierte Knoten, Wölfe und Schafe eingeteilt.
2. Isolierte Knoten werden entfernt und Wölfe fressen ihre Schafe.
3. Der Fokus wird aktualisiert: Bei ehemals aktive Knoten, die jetzt keine Wölfe sind, wird der Fokus erhöht und neu fokussierte Knoten wechseln von passiv auf aktiv.

Lemma

Jede Runde kann in Zeit $O(1)$ und mit Arbeit $O(n/\log n)$ durchgeführt werden.

Beweis. In jeder Runde passieren nur Dinge, die $O(1)$ lange dauern und es arbeiten nur $O(n/\log n)$ Prozessoren. □

Was zu tun bleibt. 9-32

- Wir tun x mal etwas, was $O(1)$ lange dauert und Arbeit $O(n/\log n)$ verursacht.
- Falls also $x = O(\log n)$ ist, so brauchen wir $O(\log n)$ Schritte und die Arbeit ist $O(n)$.

Was zu zeigen bleibt.
Nach $O(\log n)$ Runden gibt es nur noch $O(n/\log n)$ Knoten, die nicht suspendiert wurden.

Probleme hierbei

- Es ist *nicht* der Fall, dass in jeder Runde alle Fokus-Zeiger hochgesetzt werden.
- Es ist *nicht* der Fall, dass alle Fokus-Zeiger in $O(\log n)$ Runden überhaupt das Blockende erreichen.

9.3.2 Nochmal Wölfe und Schafe

Eine neue Definition von Wölfen und Schafen. 9-33

Für die folgende Analyse ist es wichtig, dass Wölfe folgende Eigenschaft haben:

> Der Wolf soll mindestens so hoch wie alle seine Schafe sein.

Diese Bedingung ist normalerweise nicht garantiert, wir können sie aber durch folgenden Tricks erzwingen:

1. Wenn die aktiven Knoten in Wölfe und Schafe eingeteilt werden, werden auch solche Knoten Wölfe (statt Schafen), die ein lokales Höhenmaximum bilden.
2. Gleiche Höhen werden durch leichtes »Schütteln« ausgeschlossen (wie beim ersten Übungsblatt).
3. Die Herde eines Wolfs ist nun die Menge aller Vorgänger und Nachfolger bis zum nächsten lokalen Höhenminimum oder bis zum nächsten Wolf.

(Details an der Tafel)

9.3.3 Kostenanalyse

Die groben Ideen hinter der Kostenanalyse. 9-34

Sei $q = 1/\log\log n$.

- Wir machen Schulden bei einer Bank, und zwar gerade $\frac{n}{q \log n}$.
- Wir verteilen die Schulden (ungleichmäßig) auf die Knoten, die nicht suspended sind, jeder Knoten bekommt aber mindestens die Schulden $\frac{1}{2}(1-q)^{\log n}$.
- In jeder Runde zahlen wir die Schulden der in der Runde ausgeklinkten Knoten zurück.
- Dies wird jede Runde die Gesamtschulden um den Faktor mindestens $1-q/4$ senken.
- Nach $6\log n$ Runden sind unsere Schulden dann gesunken auf

$$\frac{n}{q\log n}(1-q/4)^{6\log n} \leq \frac{n}{\log n}(1-q)^{\log n}.$$

- Dann kann es nur noch $2n/\log n$ Knoten geben, denn sonst müssten die Schulden ja höher sein.

Die Verteilung der Kosten und die Blockschulden. 9-35

- Jeder nicht ausgeklinkte Knoten bekommt die Schulden $(1-q)^h$ aufgebrummt, wobei h die Höhe des Knotens ist.
- Schafe bekommen allerdings nur die halben Schulden.
- Die *Schulden eines Blocks* sind die Schulden aller passiven Knoten plus, falls es in dem Block einen Wolf gibt, die Schulden des Wolfs und seiner Herde.

Beobachtungen

- Jeder nicht ausgeklinkte Knoten bekommt mindestens den Wert $\frac{1}{2}(1-q)^{\log n}$ aufgebrummt, da $\log n$ die Blockhöhe ist.
- Blockschulden sind anfangs höchstens $\sum_{i=0}^{\log n}(1-q)^i \leq 1/q$.
- Also sind die Gesamtschulden anfangs höchstens $n/(q\log n)$.

9-36

Die Rückzahlungen bei isolierten Knoten.

Betrachten wir nun, was passiert, wenn ein *isolierter Knoten* gelöscht wird:

- Die Schulden seines Blocks waren *vorher*

$$\sum_{i=h}^{\log n-1} (1-q)^i.$$

- Die Schulden seines Blocks sind *hinterher*

$$\sum_{i=h+1}^{\log n-1} (1-q)^i \leq (1-q) \sum_{i=h}^{\log n-1} (1-q)^i.$$

- Die Kosten des Blocks sind also um den Faktor $1-q < 1-q/4$ gesunken.

9-37

Die Rückzahlungen bei Herdenbildung.

Betrachten wir nun, was passiert, wenn eine *Schafherde gebildet* wird:

- Betrachten wir die Gesamtschulden aller an der Herde beteiligten Blöcke.
- Die Gesamtschulden aller Blöcke *vorher* sind

$$Q := \sum_{j=1}^{k} \sum_{i=h_j}^{\log n-1} (1-q)^i.$$

  Hierbei ist h_j die Höhe des j-ten Schafs und h_1 die Höhe des Wolfs.
- Die Gesamtschulden aller Blöcke *nachher* sind

$$Q - \sum_{j=2}^{k} \frac{1}{2}(1-q)^{h_j}$$

9-38

Die Rückzahlungen bei Herdenbildung. Fortsetzung.

- Es gilt

$$\begin{aligned} Q &= \sum_{j=1}^{k} \sum_{i=h_j}^{\log n-1} (1-q)^i \\ &\leq \sum_{j=1}^{k} \frac{1}{q}(1-q)^{h_j} \\ &\leq \sum_{j=2}^{k} \frac{2}{q}(1-q)^{h_j}. \end{aligned}$$

- Andererseits reduzierten sich die Kosten von Q auf

$$Q - \sum_{j=2}^{k} \frac{1}{2}(1-q)^{h_j} \leq Q - Q\frac{q}{4}.$$

9-39

Die Rückzahlungen beim Schaffressen.

Betrachten wir nun, was passiert, wenn ein *Schaf gefressen* wird:

- Die Schulden des Blocks waren *vorher*

$$\underbrace{\sum_{i=h}^{\log n-1} (1-q)^i}_{\text{passive Knoten}} + \underbrace{\frac{1}{2} \sum_{j=2}^{k} (1-q)^{h_j}}_{\text{Schafherde}}.$$

- Seien h_j die Höhen der Schafe und sei h_2 die Höhe des gefressenen Schafs. Wir dürfen annehmen, dass h_2 minimal ist, sonst verteilen wir die Gewichte um.
- Die Schulden des Blocks sind *hinterher*

$$\sum_{i=h}^{\log n-1}(1-q)^i+\frac{1}{2}\sum_{j=3}^{k}(1-q)^{h_j}.$$

- Es gilt $\sum_{i=h}^{\log n-1}(1-q)^i \leq \frac{1}{q}(1-q)^h$ und $\sum_{j=3}^{k}(1-q)^{h_j} \leq k(1-q)^{h_2} \leq \frac{1}{q}(1-q)^{h_2}$. Hier haben wir benutzt, dass jede Schafherde Größe höchstens $\log\log n = 1/q$ hat.
- Die Größe vorher ist also höchstens $\frac{3}{2q}(1-q)^{h_2}$ und reduziert sich um $\frac{1}{2}(1-q)^{h_2}$, also um den Faktor $1-q/3$.

Zusammenfassung dieses Kapitels

1. Es gibt einen List-Ranking Algorithmus mit *Zeit* $O(\log n)$ und *Arbeit* $O(n)$.
2. Die Analyse ist komplex und beruht auf einer Amortisierungsanalyse.

10-1

Kapitel 10
Euler-Touren
Die Brücken von Königsberg

10-2

Lernziele dieses Kapitels

1. Das Eulertourproblem kennen.
2. Einen Baum in eine Eulertour umwandeln können.
3. Blätter eines Baumes in eine Reihenfolge bringen können.

Inhalte dieses Kapitels

Worum es heute geht

Dieses Kapitel präsentiert das letzte fehlende Puzzelstück für den Algorithmus zur Auswertung arithmetischer Ausdrücke: Ein paralleler Algorithmus zur Bestimmung der Inorder-Traversierung der Blätter eines Baumes. Dazu betrachten wir heute *Euler-Touren* genauer, welche natürlich auf Leonhard Euler zurückgehen. Über diesen weiß Wikipedia folgende interessante Dinge zu berichten (beachten Sie beispielsweise sein Alter zum Zeitpunkt seiner Berufung zum Professor und überlegen Sie, was Sie in diesem Alter getan haben):

Autor Petar Marjanovix, GNU Free Documentation License

Aus *de.wikipedia.org/wiki/Leonhard_Euler*

1707 wurde Leonhard Euler in der Deutschschweiz als der älteste Sohn des Pfarrers Paul Euler und Margarethe Bruckner geboren. Er besuchte das Gymnasium am Münsterplatz in Basel und nahm gleichzeitig Privatunterricht beim Mathematiker Johannes Burckhardt. Ab 1720 studierte er an der Universität Basel und hörte hier Vorlesungen von Johann Bernoulli. 1723 erlangte er durch einen Vergleich der Newtonschen und Kartesischen Philosophie in lateinischer Sprache die Magisterwürde. Seinen Plan, auch Theologie zu studieren, gab er 1725 auf. Am 17. Mai 1727 berief ihn Daniel Bernoulli an die Universität Sankt Petersburg. Er erbte die Professur des 1726 verstorbenen Nikolaus II Bernoulli. Hier traf er auf Christian Goldbach. 1730 erhielt Euler die Professur für Physik und trat schließlich 1733 die Nachfolge von Daniel Bernoulli als Professor für Mathematik an. Er bekam in den folgenden Jahren immer stärkere Probleme mit seinem Augenlicht und war ab 1740 halbseitig blind.

1741 wurde er von Friedrich dem Großen an die Berliner Akademie berufen. Euler korrespondierte und verglich seine Theorien weiterhin mit Christian Goldbach. Nach 25 Jahren in Berlin kehrte er 1766 zurück nach St. Petersburg. An seine Tätigkeit und sein damaliges Wohnhaus in Berlin erinnert eine Gedenktafel an der Behrenstraße 22/23, das heutige Haus der Bayerischen Landesvertretung in Berlin. Im St. Petersburg der Zarin Katharina der Großen wurde ihm an der Akademie der Wissenschaften ein ehrenvoller Empfang bereitet. Er arbeitete wie in der ersten Sankt Petersburger Periode in der Kunstkamera und lebte in einem von Katharina der Großen geschenkten Palais mit seinem Sohn Johann Albrecht direkt an der Newa.

1771 erblindete er vollständig. Trotzdem entstand fast die Hälfte seines Lebenswerks in der zweiten Petersburger Zeit. Hilfe erhielt er dabei von seinem Sekretär Niklaus Fuß, der nach seinem Tod als erster eine Würdigung verfasste, und seinen Söhnen Johann Albrecht, Karl und Christoph. 1783 starb er an einer Hirnblutung. Trotz seiner Forderung wurde er nie Präsident der Universität, dieses Amt besetzte meist einer der Liebhaber Katharinas, aber sein Einfluss in der Universität war fast dem des Präsidenten ebenbürtig.

Euler war extrem produktiv: Insgesamt gibt es 866 Publikationen von ihm. Ein großer Teil der heutigen mathematischen Symbolik geht auf Euler zurück (zum Beispiel e, π, i, Summenzeichen $\sum$, $f(x)$ als Darstellung für eine Funktion). 1744 gibt er ein Lehrbuch der Variationsrechnung heraus. Euler kann auch als der eigentliche Begründer der Analysis angesehen werden. 1748 publiziert er das Grundlagenwerk *Introductio in analysin infinitorum*, in dem zum ersten Mal der Begriff der Funktion die zentrale Rolle spielt.

In den Werken *Institutiones calculi differentialis* (1765) und *Institutiones calculi integralis* (1768–1770) beschäftigt er sich außer mit der Differential- und Integralrechnung unter anderem mit Differenzengleichungen, elliptischen Integralen sowie auch mit der Theorie der Gamma- und Betafunktion. Andere Arbeiten setzen sich mit Zahlentheorie, Algebra (zum Beispiel *Vollständige Anleitung zur Algebra*, 1770), angewandter Mathematik (zum Beispiel *Mechanica, sive motus scientia analytica exposita*, 1736 und *Theoria motus corporum solidorum seu rigidorum*, 1765) und sogar mit der Anwendung mathematischer Methoden in den Sozial- und Wirtschaftswissenschaften auseinander (zum Beispiel Rentenrechnung, Lotterien, Lebenserwartung).

In der Mechanik arbeitete er auf den Gebieten der Hydrodynamik (Eulersche Bewegungsgleichungen, Turbinengleichung) und der Kreiseltheorie (Eulersche Kreiselgleichungen). Die erste analytische Beschreibung der Knickung eines mit einer Druckkraft belasteten Stabes geht auf Euler zurück; er begründete damit die Stabilitätstheorie. In der Optik veröffentlichte er Werke zur Wellentheorie des Lichts und zur Berechnung von optischen Linsen zur Vermeidung von Farbfehlern.

Seine 1736 veröffentlichte Arbeit *Solutio problematis ad geometriam situs pertinentis* beschäftigt sich mit dem Königsberger Brückenproblem und gilt als eine der ersten Arbeiten auf dem Gebiet der Graphentheorie.

Über seinen wenig rezipierten Beitrag zur mathematischen Musiktheorie (*Tentamen novae theoriae musicae*, 1739), bemerkte ein Biograph: »für die Musiker zu anspruchsvolle Mathematik, für die Mathematiker zu musikalisch.«

1745 übersetzte Leonhard Euler das Werk des Engländers Benjamin Robins *New principles of gunnery* ins Deutsche, das im selben Jahre in Berlin unter dem Titel *Neue Grundsätze der Artillerie - enthaltend die Bestimmungen der Gewalt des Pulvers nebst einer Untersuchung über den Unterschied des Widerstands der Luft in schnellen und langsamen Bewegungen aus dem Englischen des Herrn Benjamin Robins übersetzt und mit den nötigen Erläuterungen und vielen Anmerkungen versehen*. Das Buch beschäftigt sich mit der so genannten inneren Ballistik und – als Hauptthema – mit der äußeren Ballistik. Seit Galilei hatten die Artilleristen die Flugbahn der Geschosse als Parabeln angesehen, indem sie den Luftwiderstand wegen der »Dünnheit« der Luft glaubten vernachlässigen zu dürfen. Robins hat als einer der ersten wertvolle Experimente ausgeführt und gezeigt, dass dem nicht so ist; dass im Gegenteil die Flugbahn durch den Einfluss des Luftwiderstandes wesentlich abgeändert werde. Somit wurde dank Robins und Eulers Mithilfe »das erste Lehrbuch der Ballistik« geschaffen. Da solch ein Lehrbuch einer Armee einen Vorteil verschaffte, wurde es 1777 wieder ins Englische und 1783 ins Französische übersetzt. In Frankreich wurde es sogar als offizielles Lehrbuch in den Militärschulen eingeführt, sodass sogar Napoléon Bonaparte es (als Leutnant) studieren musste.

Besondere Bedeutung in der breiten Öffentlichkeit erlangte seine populärwissenschaftliche Schrift *Lettres à une princesse d'Allemagne* von 1768, in der er in Form von Briefen an die Prinzessin von Anhalt-Dessau, einer Nichte Friedrichs des Großen, die Grundzüge der Physik, der Astronomie, der Mathematik, der Philosophie und der Theologie vermittelt.

Zeitgenossen Eulers waren unter anderen Christian Goldbach, Jean le Rond d'Alembert, Alexis-Claude Clairaut, Johann Heinrich Lambert und einige Mitglieder der Familie Bernoulli.

10.1 Problemstellung

10.1.1 Eulertouren

10-4 **Die Brücken von Königsberg**

Unknown author, Public Domain

10-5 **Definition von Eulertouren.**

Definition
Eine *Eulertour* durch einen Graphen ist eine Folge von Knoten, so dass

1. je zwei aufeinanderfolgende Knoten durch eine Kante verbunden sind,
2. der letzte und der erste Knoten in der Folge durch eine Kante verbunden sind,
3. jede Kante des Graphen genau einmal besucht wird.

Bekanntermaßen gilt folgender Satz:

Satz
Ein zusammenhängender Graph hat genau dann eine Eulertour, wenn jeder Knoten einen geraden Grad (Anzahl Nachbarn) hat.

10.1.2 Sortieren von Blättern

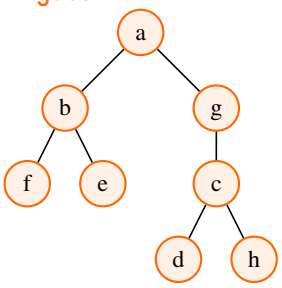
10-6 **Ziel: Reihenfolge der Blätter bei einer Inorder-Traversierung.**

Der Problem INORDER-RANKING

Eingabe Ein gerichteter Baum.

Ausgabe Die Blätter in der Reihenfolge einer Inorder-Traversierung.

Eingabe



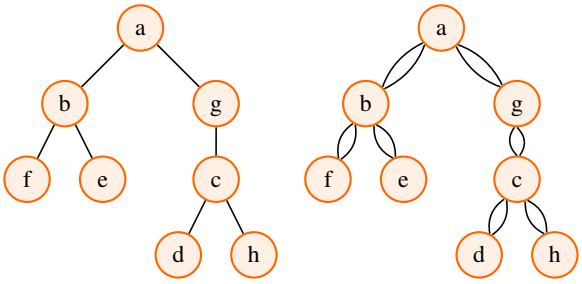
Ausgabe
f, e, d, h

10.2 Bäume und Eulertouren

10.2.1 Vom Baum zur Eulertour

Wir können Bäume in Eulertouren verwandeln. 10-7

- Gegeben sei ein Baum, dargestellt als Adjazenzliste.
- Wir ersetzen nun jede Kante durch zwei Kanten.
- Im resultierenden Graph hat jeder Knoten einen geraden Grad.
- Also gibt es einen Eulertour, die wir nun finden wollen.



Wie findet man die Eulertour? 10-8

Algorithmus

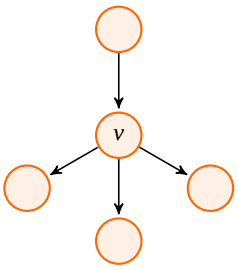
1. Eingabe sei die Adjazenzliste des Baumes.
2. Lege folgende Reihenfolge auf den Nachbarn jedes Knotens fest:
   Erst die Kinder von links nach rechts, dann der Elternknoten.
3. Erzeuge daraus eine Liste der Kanten für die Eulertour.
4. Verkette *die Kanten* wie folgt:
   - Sei (u,v) eine Kante.
   - Dann ist u ein Nachbar von v. Sei u' der »nächste« Nachbar von v.
   - Dann ist der Nachfolger der Kante (u,v) die Kante (v,u').

Satz

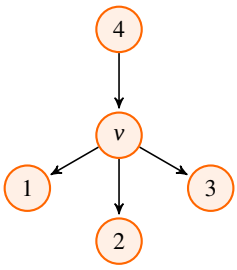
Zu einem durch eine Adjazenzliste gegebenen Baum kann man in Zeit $O(1)$ und Arbeit $O(n)$ eine Eulertour berechnen.

Der Algorithmus lokal für einen Knoten v. 10-9

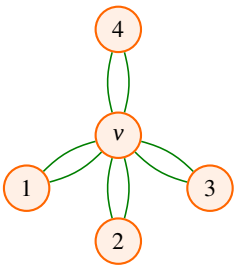
1 Der Knoten v in der Ausgangssituation



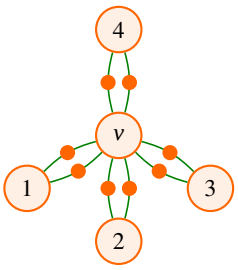
2 Eine Reihenfolge für die Nachbarn von v



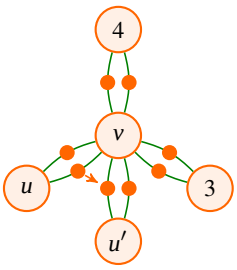
3 Erzeuge Kanten eines Graphen



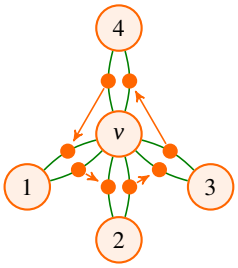
4.1 Diese Kanten sind die Knoten (!) der Liste



4.2 Die Kante zwischen (u,v) und (v,u')



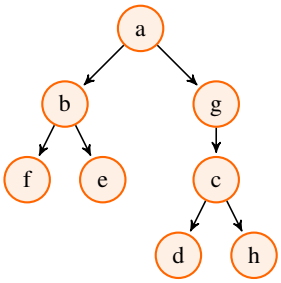
4.3 Alle lokalen Kanten



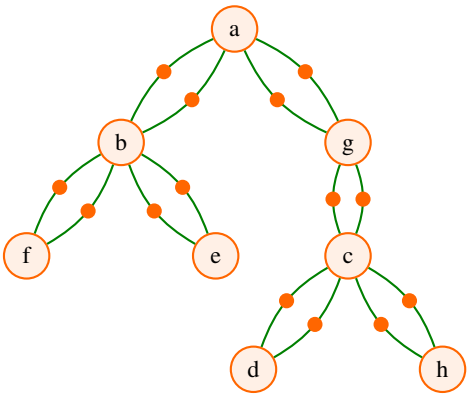
10-10

Der Algorithmus für einen ganzen Baum.

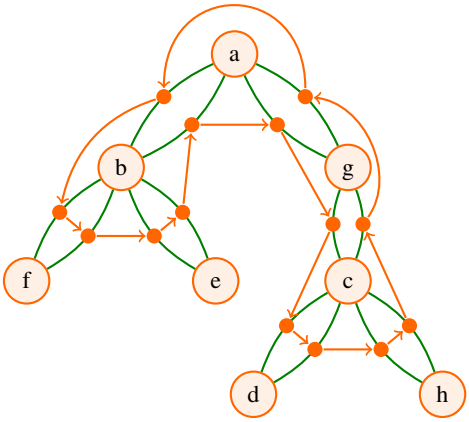
Beispiel (Der Ausgangsbaum)



Beispiel (Die Knoten der Eulertour als Liste)



Beispiel (Die Verkettung der Eulertour als Liste)



10.2.2 Von der Eulertour zum Ranking

Von der Eulertour zur Sortierung der Blätter.

10-11

Unser Hauptziel ist immernoch, die Blätter eines Baumes in ihre Inorder-Reihenfolge zu bringen.

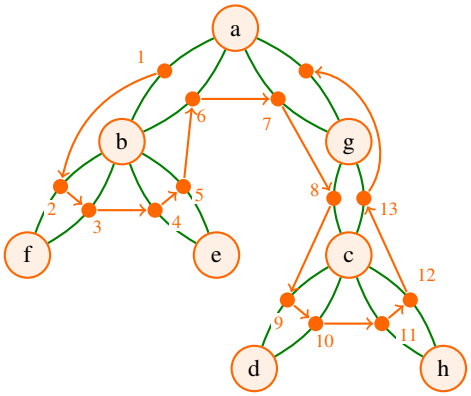
Dazu gehen wir wie folgt vor:

1. Wir berechnen eine Eulertour.
2. Wir *brechen diese am Ende auf* durch Löschen der letzten Kante.
3. Wir wenden einen *Ranking-Algorithmus* auf die Kanten an.
4. Wir *ziehen die Liste gerade*.
5. Wir löschen alle Kanten, die *nicht zu Blättern führen*.
6. Wir wenden *Präfixsummen* an, um jedem Blatt eine Position zuzuweisen.

Die Schritte des Algorithmus.

10-12

Beispiel (Das Ranking der aufgebrochenen Liste)



Beispiel (Die geradegezogene Liste im Speicher)

Rang	1	2	3	4	5	6	7	8	9	10	11	12	13
Kante von	a	b	f	b	e	b	a	g	c	d	c	h	c
nach	b	*f*	b	*e*	b	a	g	c	*d*	c	*h*	c	g

Wie schnell geht das Sortieren der Blätter?

10-13

Satz

Die Inorder-Traversierungsreihenfolge eines Baumes, der als Adjazenzliste gegeben ist, kann in Zeit $O(\log n)$ und Arbeit $O(n)$ berechnet werden.

Zur Übung

Schätzen Sie zum Beweis die Zeit und den Aufwand der einzelnen Schritte des Algorithmus ab:

1. Berechnen der Eulertour.
2. Aufbrechen der Eulertour.
3. Anwenden des Ranking-Algorithmus.
4. Geradeziehen der Liste.
5. Löschen aller Kanten, die nicht zu Blättern führen.
6. Präfixsummen.

Zusammenfassung dieses Kapitels

10-14

1. Beim *Eulertourproblem* geht es darum, in einem Baum jede Kante genau einmal zu besuchen.
2. Man kann einen Baum in eine Eulertour umwandeln, indem man jede Kante verdoppelt.
3. Man kann mittels der Eulertour die *Inorder-Reihenfolge* der Blätter bestimmen.

11-1

Kapitel 11
Auswerten von arithmetischen Ausdrücken
Gemeinschaftliches Rechnen

11-2

Lernziele dieses Kapitels

1. Konzept der Baumkontraktion kennen
2. Schnelle und optimale Algorithmen zur Auswertung arithmetischer Ausdrücke kennen und eigene erstellen können

Inhalte dieses Kapitels

Worum es heute geht

Über verschlungene Seitenpfade (Sie erinnern sich hoffentlich noch an bunte Listen, Italienkarten, schaffressende Wölfe und die Brücken von Königsberg) haben wir uns dem zentralen Problem dieses Teils der Veranstaltung genähert: Dem Auswerten arithmetischer Bäume.

Die Problematik liegt, wie schon früher angedeutet, darin, dass arithmetische Ausdrücke nicht schön symmetrisch sein müssen, sondern beliebig degenerieren können. Sind sie vollständig degeneriert (also im Wesentlichen Listen), so ist die Sache auch wieder übersichtlicher, wirklich problematisch sind die Bäume, die nicht Fisch und nicht Fleisch sind – weder einigermaßen ausgeglichene Bäume noch Listen. Um solche Ausdrücke zu verarbeiten werden wir zwei »Tricks« benutzen: Erstens besorgen wir uns eine Inorder-Traversierung der Blätter, welche es uns erlauben wird, jedes »zweite« Blatt parallel zu verarbeiten.

Der zweite Trick ist, »schon mal anzufangen« mit dem Auswerten von Teilbäumen, auch wenn nicht alle nötigen Informationen vorliegen. Dies geht grob wie folgt: Ein Additionsknoten habe als linkes Kind ein Blatt mit dem Wert x. Das rechte Kind sei ebenfalls ein Additionsknoten, der als linkes Kind ebenfalls ein Blatt (mit dem Wert y) habe und als rechtes einen großen Teilbaum. Auf den Wert des großen Teilbaums wird man eventuell länger warten müssen; den Effekt des Additionsknoten und seines rechten Kindes kann aber leicht »schon mal« zusammengefasst werden zu einer einzigen Addition mit $x+y$. Ebenso kann man natürlich zwei Multiplikationen zu einer zusammenfassen. Kommen Additionen und Multiplikationen gemischt vor, so muss man etwas trickreicher vorgehen, aber auch dies ist möglich.

Wir werden heute nur Additionen und Multiplikationen betrachten. In den Übungsaufgaben wird angedeutet, wie man auch weitere Operationen integrieren kann; allerdings wird das nötige »trickreiche Vorgehen« dabei immer komplizierter.

11.1 Problemstellung

11-4

Worum es heute geht.

- Wir wollen *arithmetische Ausdrücke* ausrechnen.
  - Beispiele sind $3+4$ oder $5\cdot(6+10)$.
  - Es kommen nur Additionen und Multiplikationen vor.
  - Dies Ausdrücke können beliebig komplex sein.
- Wir wollen sie *parallel* auswerten.
- Die Addition und die Multiplikation von zwei Zahlen gilt als Elementaroperation, dauert also $O(1)$.

11-5

Die genaue Problemstellung.

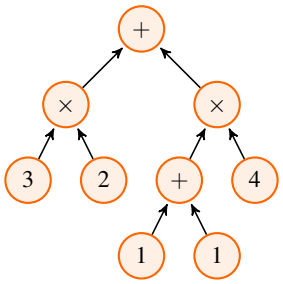
Problemstellung

Eingabe Die Adjazenzliste eines binären Baumes, dessen innere Knoten mit den Symbolen $+$ und $\times$ beschriftet sind und dessen Blätter mit ganzen Zahlen beschriftet sind.

Ausgabe Wert des Baumes.

Beispiel

Eingabe:



Ausgabe: 14

11.2 Ein einfacher Fall

11-6

Zur Übung

Nehmen wir an, der Baum ist *ausgeglichen*. (Alle Blätter sind auf einer Höhe.)

1. Geben Sie einen Algorithmus an, der den Baum in Zeit $O(\log n)$ und Arbeit $O(n)$ auswertet, wobei n die Anzahl der Zahlen in dem Baum ist.
2. Falls Ihr Algorithmus bei nicht ausgeglichenen Bäumen funktioniert, wie lange benötigt er bei einem beliebigen Baum der Tiefe h?

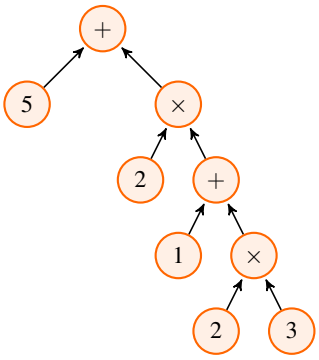
11.3 Der allgemeine Fall

11.3.1 Problematik

Die Problematik im allgemeinen Fall.

11-7

- Besonders problematisch scheint der Fall eines degenerierten Baumes.
- Hier kann man in jedem Schritt immer nur für einen Knoten den Wert sofort ausrechnen.
- Die entsprechenden arithmetischen Ausdrücke kommen sehr häufig vor (zum Beispiel im Horner-Schema).

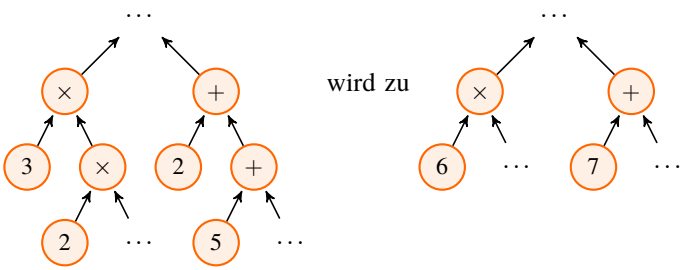


11.3.2 Lösungsidee

Lösungsidee: Schon Mal anfangen...

11-8

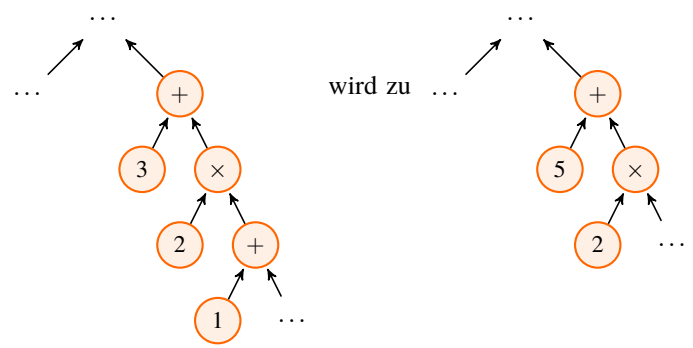
- Wir können den Wert von Knoten »in der Mitte« nicht komplett auswerten.
- Wir können aber »schon mal anfangen«.
- Dazu fassen wir die »Wirkung« mehrerer Knoten zusammen.



Neues Problem: Mischung von Addition und Multiplikation

11-9

- Kommen Multiplikationen und Additionen durcheinander, so ist unklar, wie man »schon mal anfangen« kann.
- Der Trick ist, dass man *mehrere* gemischte Multiplikationen und Additionen zu *einer* Multiplikation gefolgt von *einer* Addition zusammenfassen kann.



11-10

Grobe Lösungsidee.

- In jeder Runde werden wir parallel die Hälfte aller Blätter entfernen.
- Dazu wird an jedem zweiten Blatt »schon mal angefangen zu rechnen«.
- Dies verkleinert den Baum in jedem Schritt um die Hälfte.
- Nach $\log n$ Schritten haben wir dann das Ergebnis.

11.3.3 Linearformen als Kantengewichte

11-11

Was ist eine Linearform?

Definition

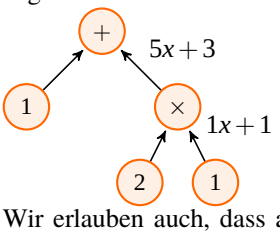
Eine *Linearform* ist eine Funktion der Form $f(x) = a \cdot x + b$ für zwei Konstanten a und b.

- Linearformen beschreiben genau den gemeinsamen Effekt mehrerer Additionen und Multiplikationen.
- Wir werden Linearformen einfach schreiben als $ax + b$. Man kann sie sogar noch kürzer schreiben als (a, b).
- Die Linearform $1x + 0$ ist offenbar die Identität.

11-12

Was machen wir mit den Linearform?

- Wir schreiben Linearformen an *Kanten* des Baumes.
- Die *Bedeutung* einer Linearformen an einer Kante ist: Wenn das Ergebnis des Teilbaumes unterhalb der Kante »hochgereicht« wird, dann muss dabei die Linearformen angewandt werden.



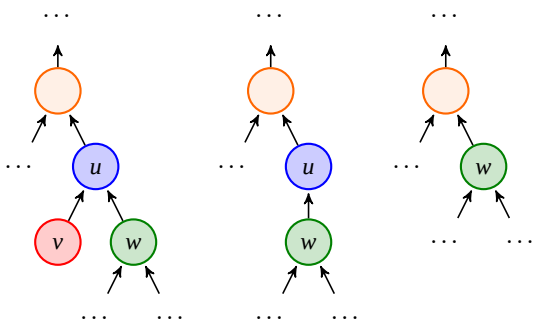
- Wir erlauben auch, dass an einer Kante keine Linearform steht. Wen dies stört, der schreibt an solche Kanten $1x + 0$.

11.3.4 Die Rake-Operation

11-13

Die *Rake*-Operation.

- Ziel ist es nun, ein Blatt »loszuwerden«.
- Das Blatt darf auch mitten im Baum stehen und die Kanten sind schon mit Linearformen belegt.
- Das Entfernen des Blattes geschieht in zwei Schritten.
  1. Das Blatt wird entfernt.
  2. Der Eltern-Knoten wird entfernt.



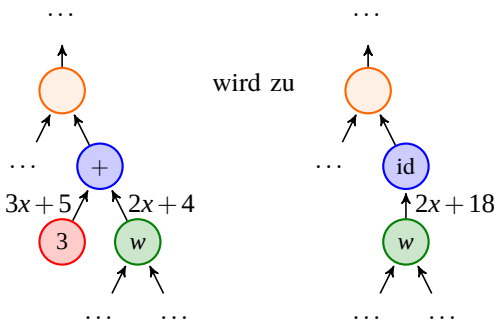
Details der *Rake*-Operation.

11-14

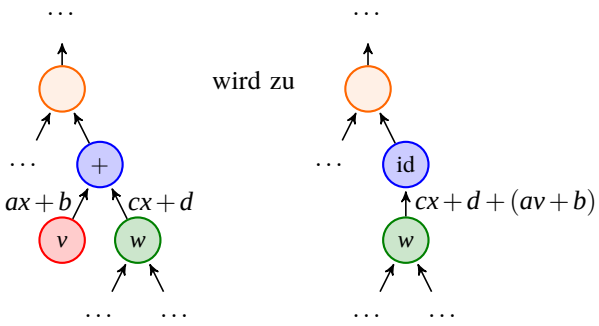
Teil 1.1: Blatt löschen bei einer Addition.

Wir wollen das Blatt v löschen und der Elternknoten u ist ein Additionsknoten.

Beispiel



Allgemeine Umwandlung



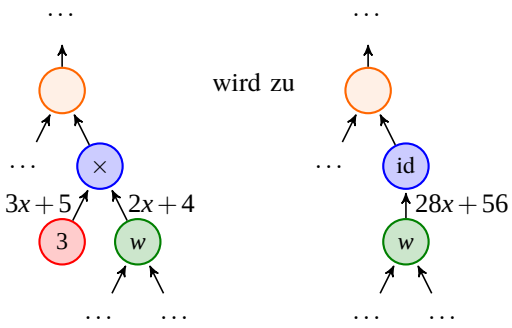
Details der *Rake*-Operation.

11-15

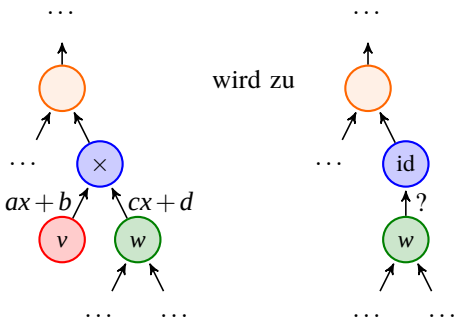
Teil 1.2: Blatt löschen bei einer Multiplikation.

Wir wollen das Blatt v löschen und der Elternknoten u ist ein Multiplikationsknoten.

Beispiel



Zur Übung



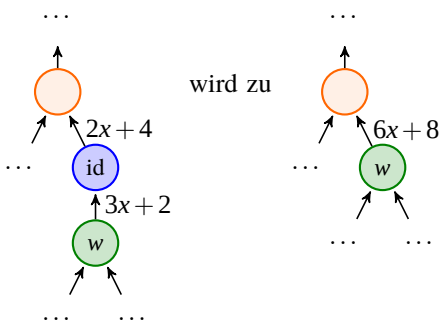
11-16

Details der *Rake*-Operation.

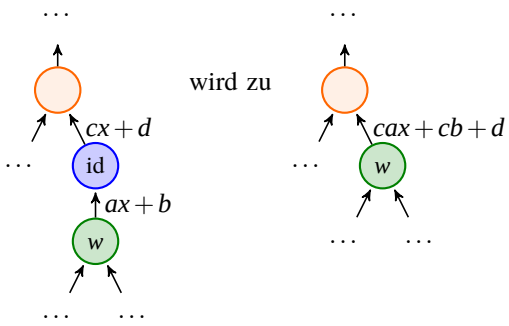
Teil 2: Löschen des Knotens mit nur einem Kind.

Wir wollen den alten Elternknoten löschen, der jetzt nur noch ein Kind hat.

Beispiel



Allgemeine Umwandlung



11-17

Zur Laufzeit und Korrektheit des Algorithmus

Satz

Der Algorithmus Rake wandelt einen arithmetischen Baum in Zeit $O(1)$ und Arbeit $O(1)$ um, ohne seinen Wert zu ändern.

11.3.5 Algorithmus

11-18

Der Algorithmus – wir setzen jetzt alles zusammen.

Algorithmus *Evaluate*

1. Lies einen arithmetischen Baum als Eingabe, gegeben als Adjazenzliste.
2. Erstelle mittels der Euler-Tour-Technik einen Blätter-Array, in dem die Blätter nacheinander in ihrer Traversierungsreihenfolge im Baum stehen.
3. Wenn der Baum nur noch aus der Wurzel besteht, so steht dort das Ergebnis. Sonst wiederhole:
   - 3.1 Betrachte nur die Blätter mit *gerader Nummer*.
   - 3.2 Teile diese (konzeptionell) in die Mengen L und R der Blätter auf, die *linke Kinder* und die *rechte Kinder* sind.
   - 3.3 Führe *Rake* parallel für alle Elemente von L aus.
   - 3.4 Führe *Rake* parallel für alle Elemente von R aus.
   - 3.5 Streiche (konzeptionell) alle Blätter mit gerader Nummer aus dem Array.

Zur Laufzeit und Korrektheit des Algorithmus

11-19

Satz

Der Algorithmus Evaluate wertet arithmetische Ausdrücke in Zeit $O(\log n)$ und Arbeit $O(n)$ aus.

Beweis.

- Die *Laufzeit ist* $O(\log n)$, da die Vorverarbeitung Zeit $O(\log n)$ braucht und die Hauptschleife $\log n$ mal durchlaufen wird und in der Schleife immer nur Zeit $O(1)$ benötigt wird.
- Die *Arbeit ist* $O(n)$, da die Vorverarbeitung $O(n)$ Arbeit macht und ein der Hauptschleife jedes Blatt nur einmal betrachtet wird.
- *Der Algorithmus ist korrekt,* da sichergestellt ist, dass sich je zwei *Rake*-Operationen nicht »in die Quere kommen«.

□

Zusammenfassung dieses Kapitels

11-20

1. Arithmetische Ausdrücke lassen sich in Zeit $O(\log n)$ und Arbeit $O(n)$ auswerten.
2. Bei nicht ausgeglichenen Bäumen muss man mittels Linearformen »Berechnungen schon mal in der Mitte beginnen«, obwohl nicht alle Daten vorliegen.

Übungen zu diesem Kapitel

Übung 11.1 (Auswertung von monotonen booleschen Formeln, leicht)

Eine *monotone boolesche Formel ohne Variablen* kann man als Baum auffassen, an dessen Blättern die Konstanten `true` und `false` stehen und dessen innere Knoten mit den Junktoren $\wedge$ und $\vee$ beschriftet sind.

Beschreiben Sie, wie ein Algorithmus für die Auswertung solcher Formeln funktioniert, der eine Laufzeit von $O(\log n)$ und Arbeit $O(n)$ hat. Der Baum sei wie üblich als Adjazenzliste gegeben. Der Detailgrad Ihrer Beschreibung des Algorithmus sollte sich am Skript zur Vorlesung über arithmetische Ausdrücke orientieren.

Übung 11.2 (Auswertung von booleschen Formeln, mittel)

Eine allgemeine boolesche Formel ohne Variablen enthält neben den Und- und Oder-Junktoren auch noch Negationsknoten mit nur einem Kind.

Beschreiben Sie, wie Sie Ihren Algorithmus aus der vorherigen Aufgabe erweitern können, so dass er auch mit solchen Formeln umgehen kann (und noch dieselbe Laufzeit und Arbeit aufweist). Beachten Sie, dass Negationsgatter in langen Ketten auftreten können (es gibt verschiedene Arten, dieses Problem zu lösen).

Übung 11.3 (Auswertung für Addition und Maximum, mittel)

Geben Sie einen Algorithmus an, der einen arithmetischen Ausdruck auf den rationalen Zahlen über den binären Operationen Addition ($+$) und Maximumsbildung (max) auswertet. Der Ausdruck ist dabei als Binärbaum mit n Blättern gegeben und der Algorithmus soll eine Laufzeit von $O(\log n)$ besitzen.

Tipp: Benutzen Sie statt der Linearformen $a \cdot x + b$ als Kantenbeschriftung Ausdrücke der Form $\max(a + x, b)$.

Übung 11.4 (Auswertung von verallgemeinerten Formeln, schwer)

Lösen Sie die vorherige Übungsaufgabe für arithmetische Ausdrücke auf den positiven rationalen Zahlen über den Operationen $+$, $\times$ und max.

Tipp: Überlegen Sie sich zunächst einen geeigneten Ersatz für die Linearformen.

12-1

Kapitel 12
Paralleles Sortieren
Wie man in der Gruppe einen Kartenstapel sortiert

12-2

Lernziele dieses Kapitels

1. Einfachen Merge-Sort kennen
2. Bitonischen Merge-Sort kennen

Inhalte dieses Kapitels

Worum es heute geht

Zum Abschluss der Thematik »Parallele Algorithmen« soll es um ein Thema gehen, um das kein Informatiker herumkommt: Sortieren. Computer sind ständig mit Sortieren beschäftigt, sie verbringen einen nicht unerheblichen Prozentsatz ihrer gesamten Rechenzeit (und damit ihres Stromverbrauchs) damit. Deshalb erscheint es geboten, dem Sortierproblem mindestens so viel Aufmerksamkeit zu schenken wie dem – im direkten Vergleich schon fast elfenbeinturmhaften – Problem der Auswertung arithmetischer Ausdrücke.

Wir haben bereits einen parallelen Merge-Sort-Algorithmus kennen gelernt. Der Hauptteil dieses Kapitels geht um Sortiernetze, mit deren Hilfe wir Daten zwar nicht schneller sortieren können als mit dem parallelen Merge-Sort-Algorithmus, aber dafür wesentlich einfacher und sehr leicht praktisch umsetzbar. Dies ist ein ganz wesentlicher Aspekt, wenn man Algorithmen tatsächlich implementieren möchte. Sortiernetzwerke lassen sich nicht nur sehr einfach in Algorithmen verwandeln, sie lassen sich sogar elegant in Hardware ätzen. Damit sind sie prädestiniert für den Einsatz in eingebettenen Systemen, bei denen es um Geschwindigkeit geht. Zum Vergleich: Versuchen Sie doch mal, den Schafe-und-Wölfe-Algorithmus in ein FPGA zu verwandeln (es erscheint unwahrscheinlich, dass dies in diesem Universum jemals geschehen wird).

Natürlich gibt es einfachere und komplexere Sortiernetzwerke; wir werden nur einfache in diesem Kapitel behandeln. Das Ajtai-Komlós-Szemerédi-Sortiernetzwerke ist (theoretisch) wesentlich flacher als die einfachen in diesem Kapitel vorgestellten: Es hat Tiefe $O(\log n)$ statt $O(\log^2 n)$. Erkauft wird dies mit einer komplexen Konstruktion und einer in der O-Notation versteckten Konstante, die man in verbesserten Versionen auf unter 6100 drücken konnte. Solche Netzwerke sind also nur dann »flach«, wenn man mehr als 10^{2000} Zahlen sortieren möchte, was in der Praxis nur sporadisch vorkommt.

12.1 Einfacher Merge-Sort

Unsere Ziele

12-4

Sei $<$ eine feste Lineare Ordnung auf Objekten.

Problemstellung

Eingabe Liste von Objekten

Ausgabe Sortierte Liste der Objekte

Bekanntermaßen ist $T^*(n) = O(n \log n)$.

Ziele

1. Ein Algorithmus mit Zeit $O(\log^2 n)$ und Arbeit $O(n \log n)$ (heute).
2. Ein einfacher Algorithmus mit Zeit $O(\log^2 n)$ und Arbeit $O(n \log^2 n)$ (heute).
3. Ein Algorithmus mit Zeit $O(\log n)$ und Arbeit $O(n \log n)$ (möglich, aber sehr kompliziert).

Ein einfacher Merge-Sort-Algorithmus.

12-5

Algorithmus

Sei A der Eingabearray. Falls seine Länge 1 ist, so sind wir fertig. Sonst:

1. Teile A in die vordere und in die hintere Hälfte auf.
2. Sortiere (parallel) die vordere und die hintere Hälfte rekursiv.
3. Verschmelze die beiden sortierten Listen zu einer Liste.

Zur Erinnerung:

- Wir können zwei Listen der Länge n in Zeit $O(\log n)$ und Arbeit $O(n)$ verschmelzen (mittels Accelerated Cascading).
- Merge-Sort hat eine Rekursionstiefe von $O(\log n)$.
- Also ist die Gesamtzeit $O(\log^2 n)$ und die Arbeit $O(n \log n)$.

12.2 Sortiernetze

12.2.1 Warum noch ein Algorithmus?

Warum noch ein Algorithmus zum Sortieren?

12-6

- Sortieren ist eines der *wichtigsten Probleme* überhaupt.
- Es kommt in *sehr vielen Anwendungen* als Teilproblem vor.
- Der einfache Merge-Sort Algorithmus enthält einen *komplexen* Verschmelzungalgorithmus.
- Der einfache Merge-Sort Algorithmus lässt sich *schlecht in Hardware implementieren*.
- Es lohnt sich also, nach einem *praktischeren* Sortieralgorithmus Ausschau zu halten.

12.2.2 Idee des Sortiertnetzes

12-7

Die Elementaroperation, mit der wir die Daten sortieren.

Operation *CompareSwap*

Eingabe Zwei Objekte x und y.

Ausgabe Die Objekte in sortierter Reihenfolge, also (x,y), falls $x \leq y$, sonst (y,x).

Bemerkungen:

- Die Operation vergleicht zwei Objekte und vertauscht sie, falls nötig.
- Wir wollen die Objekte *nur mittels dieser Operation sortieren*.
- Wir nehmen an, dass diese Operation *eine Zeiteinheit* dauert.

12-8

Sortiernetzwerk bestehen aus vielen Anwendungen von *CompareSwap*.

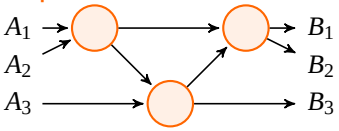
Definition (Vergleichsnetzwerk, Sortiernetzwerk)

Ein *Vergleichsnetzwerk für n Objekte* ist ein DAG mit:

- n Eingabeknoten.
- n Ausgabeknoten.
- Alle inneren Knoten haben zwei Eingänge und zwei Ausgänge und führen *CompareSwap* aus.

Ein Vergleichsnetzwerk ist ein *Sortiernetzwerk*, wenn es jeden Array von n Knoten sortiert.

Beispiel

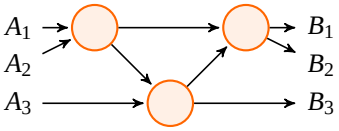


12-9

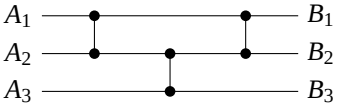
Eine andere Art, ein Sortiernetzwerk aufzuschreiben.

Es ist günstig, die Knoten eines Sortiernetzwerks als »Kanten zwischen Linien« aufzuschreiben.

Runde Knoten



Kantenknoten



12-10

Die wichtigsten Parameter von Sortiertnetzwerken.

- Die Anzahl der inneren Knoten nennen wir die *Größe* des Netzwerks.
- Die Länge des längsten Pfades nennen wir die *Tiefe* des Netzwerks.
- Offenbar entspricht die Größe von Netzwerken der Arbeit und die Tiefe der Rechenzeit bei einer PRAM.

Zur Übung

Geben Sie ein Sortiernetzwerk möglichst kleiner Tiefe für $n = 4$ an.

12.2.3 Das 0-1-Prinzip

Das 0-1-Prinzip. 12-11

Satz
Ein Vergleichsnetzwerk ist genau dann ein Sortiernetzwerk, falls es jede 0-1-Folge sortiert.

Für den Beweis brauchen wir zunächst das *Monotonielemma*.

Lemma
Sei N ein Vergleichsnetzwerk und $f: A \to \{0,1\}$ sei monoton (das bedeutet, aus $a \leq b$ folgt $f(a) \leq f(b)$). Permutiert N die Eingabe $(a_1,\ldots,a_n)$ zu $(b_1,\ldots,b_n)$, so gibt es bei Eingabe $\big(f(a_1),\ldots,f(a_n)\big)$ auch $\big(f(b_1),\ldots,f(b_n)\big)$ aus.

Beweis. Eine Induktion über die Anzahl der Knoten. □

Beweis des 0-1-Prinzips 12-12

Beweis. Beweis durch Widerspruch. Sei $(a_1,\ldots,a_n)$ ein Tupel, das nicht korrekt sortiert wird. Dann gibt es in der Ausgabe $(b_1,\ldots,b_n)$ des Vergleichsnetzwerk eine Stelle i, für die $b_i > b_{i+1}$ gilt.
Die Funktion $f: A \to \{0,1\}$ sei wie folgt definiert:

$$f(a) = \begin{cases} 1 & \text{falls } a \geq b_i \\ 0 & \text{falls } a < b_i. \end{cases}$$

Offenbar ist f monoton. Nach dem Monotonielemma wird $\big(f(a_1),\ldots,f(a_n)\big)$ auf das Tupel $\big(f(b_1),\ldots,f(b_n)\big)$ abgebildet. Dann ist aber $f(b_i) = 1 > 0 = f(b_{i+1})$. Also ist die Ausgabe nicht sortiert. □

12.3 Bitonischer Merge-Sort

12.3.1 Bitonische Folgen

Was sind bitonische Folgen? 12-13

Definition (Bitonische Folgen)
Eine *bitonische Folge* ist

- entweder erst aufsteigend und dann absteigend
- oder erst absteigend und dann aufsteigend.

Beispiel
Zwei bitonische Folgen:

- 1, 4, 9, 8, 7, 6, 2.
- 1, 1, 1, 0, 1, 1, 1.

Eine nicht bitonische Folge:

- 1, 1, 2, 1, 0, 1.

Plan zum Bitonischen Merge-Sort. 12-14

Zum Sortieren einer Listen gehen wir in zwei Schritten vor:

1. Wir verwandeln die unsortierte Liste (rekursiv) in eine bitonische Liste.
2. Wir verwandeln die bitonische Liste (rekursiv) in eine sortierte Liste.

12.3.2 Der Halbreiniger

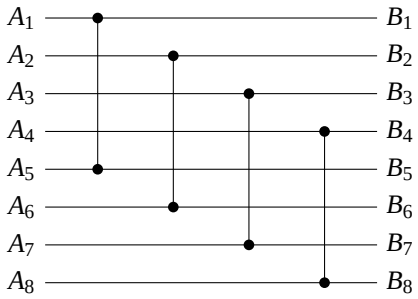
12-15

Die Definition von Halbreinigern.

Definition (Halbreiniger)

Ein *Halbreiniger* ist das Vergleichsnetzwerk auf $2n$ Eingaben, das parallel jeweils auf allen Eingaben $i \in \{1,\dots,n\}$ und Eingaben $i+n$ ein *CompareSwap* ausführt.

Beispiel



12-16

Zentrale Eigenschaft von Halbreinigern.

Satz

Sei A eine bitonische 0-1-Folge der Länge $2n$.
Wendet man auf dieses Array einen Halbreiniger an, so ist die Ausgabe

1. *bitonisch und*
2. *die erste Hälfte ist konstant 0 oder die zweite Hälfte ist konstant 1.*

Beweis. Übungsaufgabe 12.1. □

12-17

Wie man mit Halbreinigern sauber macht.

- Unser *Ziel* ist es, eine bitonische Folge in eine sortierte (saubere) Folge umzuwandeln.
- Was wir schon können ist, ein bitonische Folge *zur Hälfte* zu säubern.
- Säubern wir nun die Hälften rekursiv immer wieder, so erhalten wir insgesamt eine *saubere* Liste.

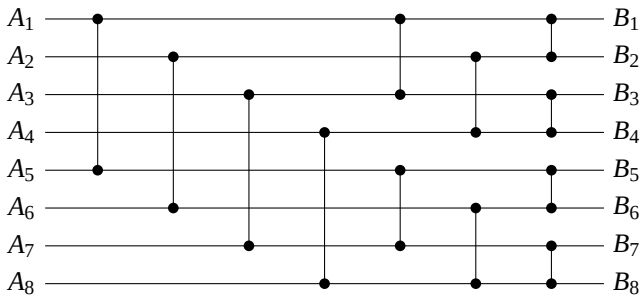
Algorithmus *SortBitonicList*.

1. Die *Eingabe* ist eine *bitonische Liste* der Länge $2n$.
2. Wende den Halbreiniger auf die Liste an.
3. Rufe rekursiv und parallel *SortBitonicList* auf die ersten n und die letzten n Elemente an.

12-18

Das Vergleichsnetz von *SortBitonicList*.

Beispiel



Zur Übung

Bestimmen Sie $T(n)$ und $W(n)$ für *SortBitonicList*.

12.3.3 Das Sortiernetz

Wo sind wir? Wie geht es weiter? 12-19

- Wir haben eine *unsortierte Liste als Eingabe*.
- Wir können eine *bitonische Liste* in eine *sortierte Liste* umwandeln.
- Was uns fehlt ist, eine *unsortierte Liste* in eine *bitonische* umzuwandeln.

Ideen

1. Wir wenden Rekursion an, um die *bitonische Liste* zu erzeugen.
2. Die Rekursion liefert uns zwei *sortierte Listen*.
3. Um daraus *eine bitonische Liste* zu machen, hängen wir einfach die zweite umgedreht hinter die erste.

Der Algorithmus *BitonicSort*(A). 12-20

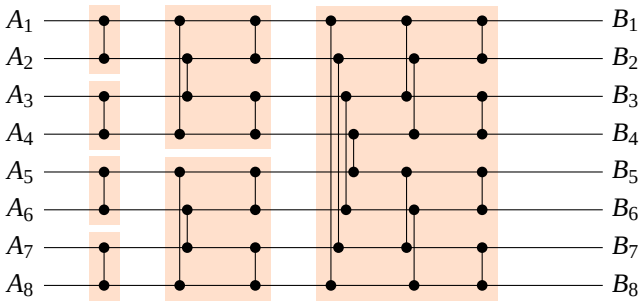
```
1 call parallel BitonicSort für die erste und für die zweite Hälfte von A
2 Drehe die zweite (nun sortierte) Hälfte von A um.
3 call SortBitonicList(A)
```

Satz
Der Algorithmus BitonicSort sortiert eine Liste in Zeit $O(\log^2 n)$ *und Arbeit* $O(n \log^2 n)$.

Beweis. Nach der Übungsaufgabe weiter vorne benötigt *SortBitonicList* Zeit $O(\log n)$ und Arbeit $O(n \log n)$. Da die Rekursionstiefe des obigen Algorithmus $\log n$ ist, ergibt sich sofort eine Laufzeit von $O(\log^2 n)$ und eine Arbeit von $O(n \log^2 n)$. □

Das Sortiernetz von *BitonicSort*. 12-21

Beispiel



- Jeder Block ist ein Aufruf von *SortBitonicList*.
- Die Struktur ist anders als bei *SortBitonicList*, da die zweite Liste ja jedesmal umgedreht wird.

Zusammenfassung dieses Kapitels

1. Man kann eine Liste von Zahlen in *Zeit* $O(\log^2 n)$ und *Arbeit* $O(n \log n)$ sortieren mittels *Merge-Sort*. 12-22
2. Der bitonische Merge-Sort ist *sehr einfach zu »programmieren«*.
3. Er arbeitet in *Zeit* $O(\log^2 n)$ und *Arbeit* $O(n \log^2 n)$.

Übungen zu diesem Kapitel

Übung 12.1 (Haupteigenschaft von Halbreinigern, mittel)

Beweisen Sie den Satz von 12-16, zeigen Sie also: Ist A eine bitonische 0-1-Folge der Länge $2n$ und wendet man auf A einen Halbreiniger an, so ist die Ausgabe bitonisch und die erste Hälfte ist konstant 0 oder die zweite Hälfte ist konstant 1.

Übung 12.2 (Odd-Even-Merge-Algorithmus, mittel)

Seien $A = (a_1, a_2, \ldots, a_n)$ und $B = (b_1, b_2, \ldots, b_n)$ zwei sortierte Folgen, wobei n eine Zweierpotenz sei. Der *Odd-Even-Merge-Algorithmus* verschmilzt rekursiv die »ungeraden« Folgen $(a_1, a_3, \ldots, a_{n-1})$ und $(b_1, b_3, \ldots, b_{n-1})$ zu einer Folge $(c_1, c_2, \ldots, c_n)$ sowie die »geraden« Folgen $(a_2, a_4, \ldots, a_n)$ und $(b_2, b_4, \ldots, b_n)$ zu einer Folge $(c'_1, c'_2, \ldots, c'_n)$. Die sortierte Folge ist dann durch

$$\left(c_1, \min\{c'_1, c_2\}, \max\{c'_1, c_2\}, \min\{c'_2, c_3\}, \max\{c'_2, c_3\}, \ldots, \min\{c'_{n-1}, c_n\}, \max\{c'_{n-1}, c_n\}, c'_n\right)$$

gegeben.

1. Geben Sie den Odd-Even-Merge-Algorithmus für $n = 4$ als Vergleichsnetzwerk an.
2. Zeigen Sie mit Hilfe des 0-1-Prinzips, dass der Odd-Even-Merge-Algorithmus A und B korrekt verschmilzt.
3. Es sind $2n$ Elemente $(a_1, a_2, \ldots, a_{2n})$ gegeben, die in die n kleinsten und n größten aufgeteilt werden sollen. Dazu werden zuerst $(a_1, \ldots, a_n)$ und $(a_{n+1}, \ldots, a_{2n})$ getrennt sortiert. Zeigen Sie, dass es ein Vergleichsnetzwerk konstanter Tiefe gibt, das aus diesen beiden sortierten Arrays einen Array mit den n kleinsten und einen mit den n größten Elementen erzeugt.

Teil III
Schaltkreise und untere Schranken

Das Motto der zwei vorherigen Teilen könnte sein – frei nach Huxley – *Oh brave new parallel world, that has such algorithms in it!* In diesem dritten Teil wird die allgemeine Euphorie etwas gedämpft werden, denn es soll hauptsächlich um die Frage gehen, was sich *nicht* parallelisieren lässt, ob es also untere Schranken für die Parallelisierbarkeit von Problemen gibt.

Bevor wir uns in die Details stürzen werden wir uns zunächst mit Schaltkreisen beschäftigen. Der Grund ist, dass PRAMs etwas unhandlich für untere Schranken sind (obwohl wir auch für bestimmte Arten von PRAMs untere Schranken werden angeben können). Schaltkreise sind hingegen wirklich »schön einfach zu formalisieren« und gleichzeitig massiv parallel. Wenn wir es also schaffen, für Schaltkreise untere Schranken zu beweisen, werden wir implizit auch untere Schranken für komplexere Modelle wie PRAMs erhalten. Damit nicht der Eindruck entsteht, Schaltkreise seien nur dazu da, böse untere Schranken zu beweisen, werden auch schöne, sehr effiziente Schaltkreise für die vier Grundrechenarten vorgestellt.

Versucht man, untere Schranken für die Tiefe (= Rechenzeit) von Schaltkreisen zu beweisen, hat man schnell mit den gleichen Problemen zu kämpfen wie wenn man versucht, die P-NP-Frage zu beantworten: Man schafft es nicht. Anstatt das aber einfach zuzugeben und sich eine ehrliche Arbeit zu suchen, forschen Theoretiker trotzdem munter weiter daran herum. In Bezug auf die P-NP-Frage sind dabei Resultate der folgenden Art herausgekommen: »Ehrlich gesagt wissen nicht, ob das Erfüllbarkeitsproblem schwierig ist, aber wir wissen ganz sicher, dass es in NP nicht *noch schwierigere* Problem gibt.« Ganz ähnliche Sachen lassen sich in Bezug auf Parallelisierbarkeit aussagen: »Niemand weiß, ob Schaltkreisauswertung parallelisierbar ist, aber wir wissen ganz sicher, dass es in P nicht *noch schlechter parallelisierbare* Probleme gibt.«

Ein Hoffnungsschimmer bleibt jedoch: Wir werden untere Schranken für einige Probleme bei PRAMs zeigen können. Jedoch erweist sich auch dieser beim genaueren Hinschauen als eher dürftig: Wir müssen »etwas schummeln« und das Maschinenmodell verändern und einschränken.

13-1

Kapitel 13
Boole'sche Schaltkreise
Die einfachsten Parallelrechner

13-2

Lernziele dieses Kapitels

1. Formalisierung von Schaltkreisen kennen
2. AC- und NC-Schaltkreisklassen kennen
3. Verhältnis der Klassen kennen
4. NC^1-Additionsalgorithmus kennen

Inhalte dieses Kapitels

Worum es heute geht

Schaltkreise sind die einfachste Form paralleler »Programme«. In der Tat lässt sich kaum leugnen, dass Schaltkreise massiv parallel arbeiten, denn alle Gatter arbeiten gleichzeitig. Den Unterschied zwischen Theorie und Praxis von Schaltkreisen erkennt man aber schon am Namen: Schalt*kreise* heißen elektrotechnisch so, weil da Elektronen »im Kreis« fließen; Schaltkreise heißen mathematisch so, obwohl sie definiert sind als besondere Graphen, die kreis*frei* sind. Trotzdem ist die mathematische Modellierung von Schaltkreisen als Graphen nicht sonderlich weit weg von der Wirklichkeit. Anders als dies bei der Übertragung von PRAMS in reale Programme gelingt, kann man die Graphen, die Schaltkreise modellieren, recht gut in reale Hardware umwandeln ohne größere Zeit- oder Platzverluste: Ein Und-Gatter ist auch in Hardware ein Und-Gatter, schlimmstenfalls eine kleine Ersatzschaltung aus mehreren NAND- oder NOR-Gattern.

In dem Film *Die Matrix* haben Sie gelernt »time is always against us«. Bei Schaltkreisen entspricht der Zeit die *Tiefe* des Schaltkreises, denn so lange müssen wir warten, bis ein Ergebnis vorliegt. Deshalb wird unser Hauptziel sein, *flache* Schaltkreise zu finden. Dazu definieren wir – wir sollte es in der Komplexitätstheorie anders sein – ein paar Klassen, die die Tiefe von Schaltkreisen messen, und sortieren dann Probleme in diese Klassen ein. Die größte mathematische Hürde wird dabei das Problem der so genannten *Uniformität* sein. Diese etwas unschöne Thematik entsteht dadurch, dass Schaltkreise prinzipbedingt nur eine feste Anzahl von Eingaben haben (anders als Programme, wo die Eingabelänge ja beliebig sein kann). Damit Schaltkreise aber Sprachen entscheiden und wir ihre Mächtigkeit damit vergleichbar machen mit uns bereits bekannten Sprachklassen

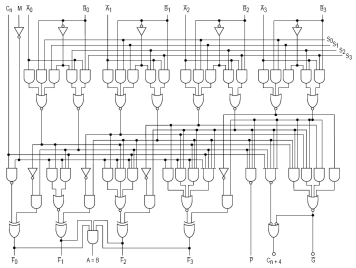
(wie P oder NP) ist definitorisches Getrickse nötig, vornehm eben »Uniformität« genannt.

13.1 Formalisierung von Schaltkreisen

13.1.1 Physikalische Schaltkreise

Zwei Schaltkreise. 13-4

Copyright by Qurren, GNU Free Documentation License


Copyright by Stephane Tsacas, GNU Free Documentation License

13.1.2 Modellierung

Was sollten wir in unser Schaltkreismodell aufnehmen? 13-5

- Wir wollen ein *Modell* von Schaltkreisen erstellen.
- Offenbar wollen wir *nicht alle* physikalischen Eigenschaften von Transistoren modellieren.
- Die Frage ist nun, welche Eigenschaften wir modellieren sollten oder wollen.

Zur Diskussion

Erstellen Sie zwei Listen von Eigenschaften oder Aspekten von Schaltkreisen, die

1. in einem Modell vorkommen sollten
2. in einem Modell ignoriert werden sollten.

13.1.3 Gatter und Verbindungen

Wie unsere Schaltkreise prinzipiell aufgebaut sind. 13-6

- Wir werden nur *Boole'sche Schaltkreise* betrachten.
  Das bedeutet, dass nur die Werte 0 und 1 vorkommen, entsprechend Strom fließt und kein Strom fließt oder hohe Spannung und niedrige Spannung.
- Schaltkreise zeichnen sich dadurch aus, dass sie *keine Kreise* bilden.
  (Gibt es Rückkopplungen, also Kreise, so spricht man von Schaltwerken.)
- Der Schaltkreis besteht also aus zwei Dingen:
  1. *Gattern*, an denen (mehrere) 0 und 1 Werte verarbeitet werden, und
  2. *Verbindungen*, also Drähten, die die Gatter miteinander verbinden.

13-7

Welche Arten von Gattern gibt es?

Ein *Gatter* leistet folgendes:

- Es hat (eventuell mehrere) Eingänge, an denen von vorherigen Gattern in jedem Takt 0en und 1en anliegen.
- Das Gatter errechnet aus diesen Werten einen neuen Wert (wieder eine 0 oder eine 1) und liefert diesen an seinen Ausgang.
- Wir werden *Und-*, *Oder-* und *Nicht*-Gatter zulassen. (Bekanntermaßen würden auch beispielsweise Nand-Gatter genügen.)
- Vom Ausgang aus wandern die Werte dann zu den nachgeschalteten Gattern.

13.1.4 Definition von Schaltkreisen

13-8

Die mathematische Definition eines Schaltkreis.

Definition

Ein *Schaltkreis* ist ein gerichteter azyklischer Graph mit folgenden Eigenschaften:

- Die *Knoten* des Graphen sind die Gatter.
- Die *Kanten* des Graphen sind die Verbindungen.
- Jedes Gatter hat einen der folgenden *Gattertypen*:
  - *Eingabegatter* – solche Knoten haben Eingrad 0.
  - *Ausgabegatter* – solche Knoten haben Ausgrad 0.
  - *Negationsgatter* – solche Knoten haben Eingrad 1.
  - *Und-Gatter*
  - *Oder-Gatter*
- Die Eingabegatter benennen wir x_1 bis x_n.
- Die Ausgabegatter benennen wir y_1 bis y_m.

Es gibt keine feste Notation für Schaltkreise, sie ist Geschmackssache.

13-9

Wie eine Berechnung mittels eines Schaltkreises funktioniert.

- Wir wollen Schaltkreise natürlich benutzen, um Eingaben in Ausgaben umzuformen.
- Dazu *legt man die Eingabe an den Eingängen an* und schaut, wie die Berechnung vorangeht.
- Jedes Gatter kann erst dann beginnen zu arbeiten, wenn alle seine (lokalen) Eingänge einen Wert vorweisen.

13-10

Formale Definition der Berechnung.

Definition (Berechnete Funktion)

Sei C ein Schaltkreis mit n Eingängen und m Ausgängen. Dann berechnet C eine Funktion $f_C\colon \{0,1\}^n \to \{0,1\}^m$ wie folgt:

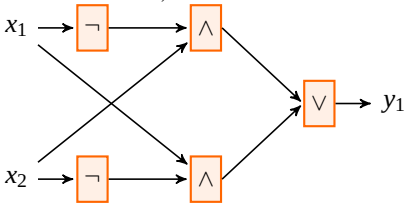
- Seien b_1 bis b_n Bits.
- Wir definieren rekursiv den *Wert* eines Gatter als
  - b_i für Eingabegatter x_i.
  - $1-v$ für Negationsgatter, wenn v der Wert des Vorgängers ist.
  - das logische Oder aller Vorgängergatter für Oder-Gatter und
  - das logische Und aller Vorgängergatter für Und-Gatter.
- Sind v_1 bis v_m dann die Werte der Ausgabegatter, so ist $f_c(b_1 \cdots b_n) = v_1 \cdots v_m$.

13-11

Beispiel eines Schaltkreises

Beispiel

Ein Schaltkreis, der die XOR-Funktion berechnet.



13.2 Klassen von Schaltkreisen

13.2.1 Ressourcen bei Schaltkreisen

Welche Ressourcen sind bei Schaltkreisen wichtig?

13-12

Sicherlich sind folgende Ressourcen bei Schaltkreisen wichtig:

- *Die Größe* des Schaltkreise (Anzahl Gatter), da dies der Chip-Fläche entspricht.
- *Die Tiefe* des Schaltkreises (maximale Weglänge von Eingabe- zu Ausgabegattern), da dies der Rechenzeit entspricht.
- *Der Fan-In* des Schaltkreises (maximaler Eingrad von Gattern), da Gatter mit hohem Fan-In schwer oder gar nicht zu bauen sind.

Zur Diskussion

Welche weiteren Ressourcen fallen Ihnen ein, die bei Schaltkreisen wichtig sind?

13.2.2 Schaltkreisfamilien

Vom Schaltkreis zur Schaltkreisfamilie.

13-13

Problem

- Ein einzelner Schaltkreis kann nur genutzt werden, Eingaben *einer festen Länge zu bearbeiten*.
- Im Allgemeinen wollen wir aber Eingaben *beliebiger Länge* bearbeiten.

Lösung

Der Trick ist, für jede mögliche Eingabelänge n einen eigenen Schaltkreis C_n zu benutzen.
Eine solche Liste $C = (C_0, C_1, C_2, \dots)$ nennt man eine *Schaltkreisfamilie*.
Wir schreiben einfach $C(x)$ für $f_{C_{|x|}}(x)$.

Für eine Schaltkreisfamilie C bezeichnet

- $\text{depth}_C(n)$ die *Tiefe von* C_n und
- $\text{size}_C(n)$ die *Größe von* C_n.

Uniforme Schaltkreisfamilien.

13-14

Problem

- Schaltkreisfamilien können sehr »wild« sein.
- So könnte C_{17} eine Addition durchführen und C_{18} eine Multiplikation und C_{19} Schach spielen.
- Schlimmer noch: Schaltkreis C_i könnte ein logisches Und oder ein logisches Oder sein, je nachdem, ob die i-te Turingmaschine anhält.
- Es gibt also Schaltkreisfamilien, die *nicht sinnvoll konstruierbar sind*.

Lösung

- Wir interessieren uns besonders für *logspace-uniforme* Schaltkreisfamilien.
- Eine Schaltkreisfamilie $(C_0, C_1, \dots)$ ist *logspace-uniform*, wenn es eine Logspace-Maschine gibt, die bei Eingabe von i Einsen den Code von C_i ausgibt.

13.2.3 Die Klasse POLYSIZE

13-15

Die Klasse POLYSIZE.

Definition (POLYSIZE)
Die Klasse POLYSIZE ist eine *Sprachklasse*. Sie enthält alle Sprachen L, deren charakteristische Funktion von einer Schaltkreisfamilie C

- polynomieller Größe berechnet werden kann.

Im Einzelnen bedeutet dies:

- Für alle Binärstrings $b \in \{0,1\}^*$ gilt $b \in L$ genau dann, wenn $C(b) = 1$.
- Es gibt ein k, so dass $\text{size}_C(n) \leq n^k$ für alle n gilt.

Verlangt man zusätzlich, dass die Schaltkreisfamilie logspace-uniform ist, so spricht man vom *logspace-uniformen POLYSIZE*, sonst vom *nichtuniformen POLYSIZE*.

13.2.4 Die AC- und NC-Klassen

13-16

Die Klasse POLYSIZE »ist nicht parallel«.

- Die Klasse POLYSIZE hat den Nachteil, dass Berechnungen polynomiell dauern können.
- Wir hätten aber lieber Laufzeiten (=Tiefe) wie $O(\log n)$.
- Deshalb führen wir Klassen ein, in denen nur Sprachen sind, für die es »flache« Schaltkreise gibt.

13-17

Flache Schaltkreisklassen: AC-Klassen und NC-Klassen.

Definition (AC-Klassen)
Sei i eine natürliche Zahl. Die *Klasse* AC^i enthält alle Sprachen L, deren charakteristische Funktion von einer Schaltkreisfamilie C

- polynomieller Größe und
- der Tiefe $O(\log^i n)$ berechnet werden kann.

Definition (NC-Klassen)
Sei i eine natürliche Zahl. Die *Klasse* NC^i enthält alle Sprachen L, deren charakteristische Funktion von einer Schaltkreisfamilie C

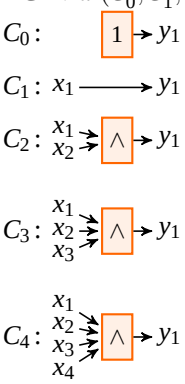
- mit polynomieller Größe,
- mit Tiefe $O(\log^i n)$ und
- mit Fan-In maximal 2 berechnet werden kann.

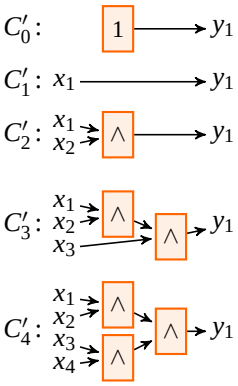
Wieder gibt es *uniforme* und *nichtforme* AC- und NC-Klassen.

13-18

Ein ganz einfaches Beispiel.

Beispiel
Die Sprache $\{1\}^*$ liegt sowohl in AC^0 via der Schaltkreisfamilie $(C_0, C_1, \dots)$ als auch in NC^1 via $(C'_0, C'_1, \dots)$:





13.3 Additionsschaltkreis

13.3.1 Ziel

Die Problemstellung zum Additionsproblem. 13-19

Problemstellung

Für ein gegebenes n wollen wir die Funktion $p_n\colon \{0,1\}^{2n} \to \{0,1\}^{n+1}$ berechnen.
Sie addiert die ersten n Bits der Eingabe auf die zweite n Bits und liefert das Ergebnis zurück.

Beispiel

- $p_4(01001101) = 10001$.
- $p_4(00010001) = 00010$.

Ziel

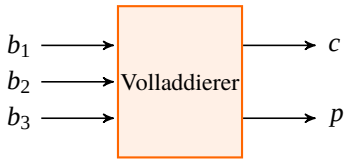
Einen »NC^1-Schaltkreis« für die Funktion.
Genauer ist eine Schaltkreisfamilie polynomieller Größe und logarithmischer Tiefe gesucht, die alle p_n berechnet.

13.3.2 Der naive Addierer

Der naive Addierer. 13-20

Definition (Volladdierer)

Ein *Volladdierer* ist ein Schaltkreis mit drei Eingängen und zwei Ausgängen. Er liefert die Binärdarstellung der Summe der drei Eingänge.



Hierbei sind c der Übertrag (Carry) und p die Parität.

Der naive Addierer. 13-21

Die Konstruktion eines naiven Addierers:

- Mit einem Volladdierer kann man zwei Bits und noch ein Carry addieren.
- Man beginnt deshalb mit einem Volladdierer, der die ersten beiden Bits addiert. Die Parity ist das erste Ausgabebit.
- Das Carrybit des ersten Volladdierers wird mit den zweiten Eingabebits in einen zweiten Volladdierer gefüttert. Die Parity hiervon ist das zweite Ausgabebit.
- Das Carrybit des zweiten Volladdierer wird mit den dritten Eingabebits in einen dritten Volladdierer gefüttert. Die Parity hiervon ist das dritte Ausgabebit.
- Und so weiter.

Zentraler Nachteil: Die Tiefe des Schaltkreises ist $\Omega(n)$.

13.3.3 Der Carry-Look-Ahead-Addierer

13-22

Warum der naive Addierer so lange braucht.

- Der naive Addierer braucht lange, da die Carry-Bits hinten einen Einfluss auf das Ergebnis vorne haben können.
- Bei einem *Carry-Look-Ahead-Addierer* versucht man deshalb, den Einfluss der Carry-Bits schneller zu berechnen.

13-23

Die drei Effekte, die zwei Bits haben können.

Sei $a_n \dots a_1 b_n \dots b_1$ die Eingabe und betrachten wir zwei Bits a_i und b_i. Ihr Einfluss auf das Carry-Bit lässt sich in drei Klassen unterteilen:

generate Falls beide Bits 1 sind, erzeugen sie *auf jeden Fall ein Carry-Bit*, unabhängig davon, was an den niederwertigeren Stellen passiert.

kill Falls beide Bits 0 sind, erzeugen sie *auf keinen Fall ein Carry-Bit*.

propagate Falls genau ein Bit 0 ist, so *reichen sie das Carry-Bit der niederwertigen Bits einfach durch*.

13-24

Wie die Effekte zusammenwirken.

Nehmen wir nun an, der Effekt zweier Bits a_i und b_i ist g, k oder p und der Effekt von a_{i-1} und b_{i-1} ist ebenfalls g, k oder p. Dann ist der Gesamteffekt:

Effekt von a_i, b_i	Effekt von a_{i-1}, b_{i-1}	Gesamteffekt
g	g	g
g	k	g
g	p	g
p	g	g
p	p	p
p	k	k
k	g	k
k	k	k
k	p	k

13-25

Der Carry-Look-Ahead-Addierer.

Ein Carry-Look-Ahead-Addierer ist wie folgt aufgebaut:

- Zunächst wird parallel in Tiefe $O(1)$ für jedes Bitpaar (a_i, b_i) der Effekt ausgerechnet.
- Dann wird für jede Stelle der Effekt ausgerechnet, den alle Bits haben, die niederwertiger sind.
- Dazu kann man einen *Postfixsumme* berechnen. Dies geht in Tiefe $O(\log n)$ und Platz $O(n)$.
- Ist für jede Stelle der Übertrag der vorherigen Stellen bekannt, kann die Ausgabe in Tiefe $O(1)$ und Platz $O(n)$ berechnet werden.

Beispiel

Bitposition i	4	3	2	1	0
a_i		0	1	1	1
b_i		0	0	0	1
Effekt von a_i und b_i		k	p	p	g
Gesamteffekt bis i ...		k	g	g	g
... und folglicher Übertrag c_i		0	1	1	1
Summe $a_i + b_i + c_{i-1}$	0	1	0	0	0

Zusammenfassung dieses Kapitels

13-26

1. Das mathematische Schaltkreismodell *abstrahiert* von vielen Details.
2. Die wichtigsten Schaltkreisklassen sind POLYSIZE, NC^i und AC^i.
3. Man kann Zahlen in Tiefe $O(\log n)$ und Platz $O(n)$ addieren.

Übungen zu diesem Kapitel

Übung 13.1 (Simmulation der Sprache der Palindrome durch Schaltkreise, leicht)

Geben Sie eine AC^0- oder eine NC^1-Schaltkreisfamilie für die Sprache der Palindrome an. Begründen Sie auch, warum Ihre Schaltkreisfamilie logspace-uniform ist.

Übung 13.2 (AC versus NC, mittel)

Zeigen Sie, dass für alle $i \in \mathbb{N}$ gilt: $NC^i \subseteq AC^i \subseteq NC^{i+1}$. Folgern Sie daraus, dass für $NC := \bigcup_{i\in\mathbb{N}} NC^i$ und $AC := \bigcup_{i\in\mathbb{N}} AC^i$ gilt: $AC = NC$.

Übung 13.3 (Schaltkreiskomplexität von regulären Sprachen, schwer)

Zeigen Sie, dass jede reguläre Sprache in NC^1 ist.

Aus Ihrem Beweis sollte einem »geübten Leser« aufgrund Ihrer Beschreibung klar sein, wie der Schaltkreis zu konstruieren ist; es müssen nicht alle Gatter einzeln beschrieben werden.

Tipp: Benutzen Sie die Charakterisierung von regulären Sprachen durch deterministische endliche Automaten und verknüpfen Sie die Zustände eines solchen Automaten ähnlich wie beim Carry-Look-Ahead-Addierer.

14-1

Kapitel 14
Multiplizieren und Dividieren
Computer rechnen nicht so, wie wir es in der Schule gelernt haben

14-2

Lernziele dieses Kapitels

1. NC^1-Multiplikationsalgorithmus kennen
2. NC^2-Divisionsalgorithmus kennen

Inhalte dieses Kapitels

Worum es heute geht

Zwei Zahlen zu multiplizieren oder gar zu dividieren ist eine Kunst für sich. Beide Operationen sind so grundlegend, dass die Menschheit sich wirklich *viele* Gedanken darüber gemacht hat, wie man diese beiden Operationen möglichst effizient hinbekommt. Um so verblüffender erscheint es, dass die Komplexität der Division aus komplexitätstheoretischer Sicht erst im Jahr 2001 endgültig geklärt werden konnte (das Dividieren ist vollständig für TC^0, eine Klasse zwischen AC^0 und NC^1).

Dabei erscheinen beide Operationen zunächst ganz einfach, schließlich lernt jedes Kind in der Schule schon recht früh, wie dies geht. Die »Schulmethoden« haben aber den Nachteil, dass sie (a) »sehr sequentiell« sind, insbesondere die Methode zur Division, und (b) quadratischen Aufwand (in der Länge der Zahlen) haben, was bei sehr großen Zahlen – wie sie zum Beispiel in der Kryptologie oft auftreten – zu viel erscheint.

Ist man nicht an einer Parallelisierung interessiert, dann bietet der Algorithmus von Anatolij Alexejewitsch Karazuba eine schnellere Methode. Soll es aber parallel gehen, so muss man sich etwas neues ausdenken. Mit etwas Informatiksachverstand sieht man schnell, dass die Multiplikation mit einem Divide-and-Conquer-Algorithmus recht gut parallelisiert werden kann, was recht einfach einen NC^2-Algorithmus liefert. Für einen NC^1-Algorithmus, unser erstes Ziel für dieses Kapitel, ist ein netter kleiner Zusatztrick nötig.

Bei der Division muss man schon mit größeren Kalibern hantieren. In diesem Kapitel wird eine Newton-Iteration für die Division vorgestellt, welche einen NC^2-Algorithmus liefert. Dies ist noch ein sehr kleines Geschütz verglichen mit den mathematischen Schwergewichten, die im Beweis für die TC^0-Vollständigkeit der Division notwendig sind. Da sie für diese Veranstaltung eine Nummer zu groß sind, lassen wir deshalb lieber die Finger davon.

14.1 Multiplikationsschaltkreis

14.1.1 Ziel

Die Problemstellung zum Multiplikationsproblem. 14-4

Problemstellung

Für ein gegebenes n wollen wir die Funktion $m_n\colon \{0,1\}^{2n} \to \{0,1\}^{2n}$ berechnen.
Sie multipliziert die ersten n Bits der Eingabe mit den zweiten n Bits und liefert das Ergebnis zurück.

Beispiel

- $m_4(01001101) = 00110100$.
- $m_4(00010001) = 00000001$.

Ziel

Eine NC^1-Schaltkreisfamilie für die Funktion.

Wie ein NC^2-Schaltkreis für die Multiplikation funktioniert. 14-5

1. Mit einem Schaltkreis *konstanter Tiefe* werden n Zahlen ermittelt, deren Summe das gesuchte Produkt ist. (Schulmethode.)
2. Wir wissen schon, dass sich zwei Zahlen in Tiefe $O(\log n)$ addieren lassen.
3. Addieren wir n Zahlen baumartig, so müssen erhalten wir einen Additionsbaum der Tiefe $O(\log n)$.
4. Dies liefert eine Gesamttiefe von $O(\log^2 n)$.

Beispiel (Ein Produkt)

$1001 \cdot 1101 =$

$00001001 +$
$00000000 +$
$00100100 +$
$01001000 =$

$00001001 +$
$01101100 =$

01110101

14.1.2 Ein Trick

Ein Trick, mit dem man den Schaltkreis flach bekommt. 14-6

- Wir können die Addition von zwei Zahlen *nicht beschleunigen*.
- Wir können aber *schnell aus drei Zahlen zwei Zahlen machen*.

Neue Aufgabenstellung

Eingabe Drei Zahlen a, b, c als Bitfolge.
Ausgabe Bitfolge zweier Zahlen d und e mit $a+b+c = d+e$.

Zur Übung

Überlegen Sie, wie ein »aus 3 mach 2 Schaltkreis« *konstanter Tiefe* aussieht.

14.1.3 Der Schaltkreis

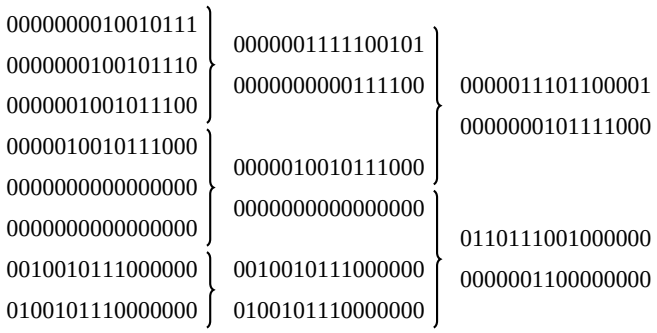
14-7 **Wie ein NC^1-Schaltkreis für die Multiplikation funktioniert.**

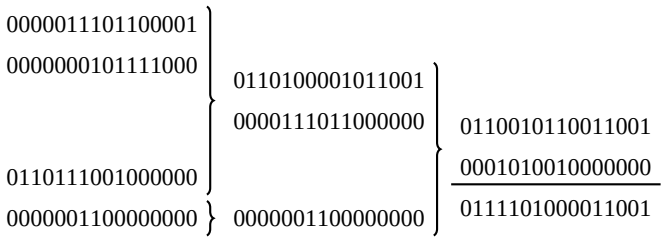
Aufbau einer NC^1-Multiplikationsschaltkreisfamilie

1. In einer ersten Schicht *konstanter Tiefe* werden aus der Eingabe n Zahlen erzeugt, deren Summe das Ergebnis liefert.
2. In den n Zahlen wird parallel in *konstanter Tiefe* jeder Block von drei Zahlen durch zwei Zahlen ersetzt.
3. In den verbleibenden $\frac{2}{3}n$ Zahlen wird parallel in *konstanter Tiefe* jeder Block von drei Zahlen durch zwei Zahlen ersetzt.
4. In den verbleibenden $\frac{4}{9}n$ Zahlen wird parallel in *konstanter Tiefe* jeder Block von drei Zahlen durch zwei Zahlen ersetzt.
5. Und so weiter, bis nur noch zwei Zahlen übrig sind.
6. Diese werden in Tiefe $O(\log n)$ addiert.

14-8 **Verdeutlichung der Schritte des Schaltkreises an einem Beispiel.**

1. Eingabe $10010111 \cdot 11001111 = ?$
2. Bildung von zu summierenden Zahlen
3. »3 auf 2«-Reduktion (wiederholt)
4. Endsumme = Endprodukt





14.2 Divisionsschaltkreis

14.2.1 Problemstellung

14-9 **Die Problemstellung zum Divisionsproblem.**

Problemstellung

Für ein gegebenes n wollen wir die Funktion $d_n \colon \{0,1\}^{2n} \to \{0,1\}^n$ berechnen.
Sie dividiert die ersten n Bits der Eingabe durch die zweiten n Bits und liefert den ganzzahligen Anteil zurück.

Beispiel

- $m_4(11010010) = 0110$.
- $m_4(00010001) = 0001$.

Ziel

Eine möglichst flache Schaltkreisfamilie für die Division.

14.2.2 Division in modernen Prozessoren

Division beim Itanium Prozessor. 14-10

Literatur

[1] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual* Volume 1, Revision 2.2, 2002.

```
     frcpa.s0 f8,p6 = f6,f7;;
(p6) fnma.s1 f9 = f7,f8,f1 ;;
(p6) fma.s1 f8 = f9,f8,f8
(p6) fma.s1 f9 = f9,f9,f0 ;;
(p6) fma.s1 f8 = f9 ,f8,f8
(p6) fma.s1 f9 = f9,f9,f0 ;;
(p6) fma.s1 f8 = f9,f8,f8 ;;
(p6) fma.d.s1 f9 = f6,f8,f0 ;;
(p6) fnma.d.s1 f6 = f7,f9,f6 ;;
(p6) fma.d.s0 f8 = f6,f8,f9
```

- `frcpa` liefert eine Approximation das Kehrwerts.
- `fma` steht für »floating point multiply and add«.
- `fnma` steht für »floating point negate multiply and add«.

14.2.3 Vorbereitungen

Vorbereitungen: Ein paar Vereinfachungen. 14-11

- Es ist im Prinzip *egal*, ob wir mit *Nachkommastellen* rechnen oder nicht – ein einfaches Verschieben des Kommas genügt.
- Wenn wir mit Kommazahlen rechnen, *dann reicht es* offenbar, einen *Kehrwert zu berechnen*. Das Ergebnis a/b errechnet sich dann als $(1/b) \cdot a$ – und wir wissen schon, wie man schnell multipliziert.
- Da wir wieder das Komma beliebig verschieben können, wollen wir nun *Kehrwerte von Zahlen zwischen* $1/2$ *und* 1 berechnen. Solche Zahlen haben in Binärschreibweise die Form 0,1....

Vorbereitungen: Approximative Lösungen und unser neues Ziel. 14-12

Definition (n-Approximation)
Sei $c \in \mathbb{R}$ eine Zahl. Eine *n-Approximation* von c ist eine Zahl $\tilde{c}$ mit

$$|c - \tilde{c}| \leq 2^{-n}.$$

Problemstellung

Eingabe n-Bit Zahl d mit $1/2 \leq d \leq 1$.

Ausgabe $2n$-Approximation von $1/d$.

Die *Motivation* für die neue Problemstellung ist, dass für zwei n-Bit Zahlen c und d und für eine $2n$-Approximation d' von $1/d$ gilt: $c \cdot d'$ ist eine n-Approximation von c/d.

14.2.4 Das Newton-Verfahren

14-13

Eigenschaften des Newton-Verfahrens.

- Das *Newton-Verfahren* geht auf Isaac Newton zurück.
- Es ist ein *numerisches Verfahren*, um Nullstellen von Funktionen zu bestimmen.
- Es arbeitet *iterativ*: In jeder Iteration wird die bestehende Approximation (im Allgemeinen sehr stark) verbessert.
- Wie alle numerischen Verfahren funktioniert es nur unter bestimmten Bedingungen.

14-14

Ein Satz, der die Konvergenz des Newton-Verfahrens allgemein beschreibt.

Satz

Sei f eine im Intervall $[a,b]$ zweifach stetig differenzierbare Funktion mit $f'(x) \neq 0$ für alle $x \in [a,b]$. Sei

$$\max_{x\in[a,b]} \left| \frac{f(x) \cdot f''(x)}{f'(x)^2} \right| \leq 1.$$

Dann existiert exakt eine Nullstelle x von f im Intervall $[a,b]$ und die Folge

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

konvergiert gegen die Nullstelle für alle $x_0 \in [a,b]$.

14-15

Zur Übung

Wir wollen den Kehrwert einer Zahl d berechnen.

- Wie lautet eine (sinnvolle) Funktion f, deren Nullstelle gerade dieser Kehrwert ist?
- Wie lautet die Iterationsvorschrift

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

  für Ihre Funktion f?

14-16

Die Fixpunktiteration für die Division.

- Die Funktion $f(x) = d - 1/x$ hat offenbar die Nullstelle $1/d$.
- Die Iterationsvorschrift für diese Funktion lautet wegen $f'(x) = 1/x^2$ folglich:

$$x_{n+1} = x_n - \frac{d - 1/x_n}{x_n^2} = x_n - dx_n^2 + x_n = x_n(2 - dx_n).$$

- Leider ist die Bedingungen für die Konvergenz der Newtoniteration nur in einer Umgebung des Kehrwertes erfüllt (also für gute anfängliche Näherungen).

Satz über die Konvergenz der Iterationsvorschrift für die Division. 14-17

Satz
Sei d mit $1/2 \leq d \leq 1$ gegeben. Sei $x_0 = 1$ der feste Startwert und sei

$$x_{k+1} = x_k(2 - dx_k).$$

Dann ist x_k eine 2^{-2^k+2}-Approximation von $1/d$.

Beweis. Wir zeigen zwei Behauptungen:

1. Es gilt $x_k = \sum_{i=0}^{2^k-1}(1-d)^i$.
2. Es gilt, dass $\sum_{i=0}^{2k+1}(1-d)^i$ eine $2k$-Approximation von $1/d$ ist.

□

Zur Behauptung $x_k = \sum_{i=0}^{2^k-1}(1-d)^i$. 14-18

- Wir zeigen die Behauptung durch Induktion.
- Der Induktionsanfang für $k = 0$ ist korrekt.
- Für den Induktionschritt rechnen wir wie folgt:

$$\begin{aligned}
x_{k+1} &= x_k(2 - dx_k) = 2x_k - dx_k^2 = 2x_k + \big(-1 + (1-d)\big)x_k^2 \\
&= 2x_k - x_k x_k + x_k(1-d)x_k \\
&= 2x_k - x_k - \textstyle\sum_{i=1}^{2^k-1}(1-d)^i x_k + x_k(1-d)x_k \\
&= x_k - \textstyle\sum_{i=1}^{2^k-1}(1-d)^i x_k + \sum_{i=0}^{2^k-1}(1-d)^{i+1}x_k \\
&= x_k - \textstyle\sum_{i=1}^{2^k-1}(1-d)^i x_k + \sum_{i=1}^{2^k}(1-d)^i x_k \\
&= x_k + (1-d)^{2^k}x_k \\
&= \textstyle\sum_{i=0}^{2^k-1}(1-d)^i + (1-d)^{2^k}\sum_{i=0}^{2^k-1}(1-d)^i \\
&= \sum_{i=0}^{2^{k+1}-1}(1-d)^i
\end{aligned}$$

Zur Behauptung, dass $\sum_{i=0}^{2k+1}(1-d)^i$ eine $2k$-Approximation von $1/d$ ist. 14-19

Es gilt:

$$\begin{aligned}
\left|1/d - \sum_{i=0}^{2k+1}(1-d)^i\right| &= \left|\sum_{i=0}^{\infty}(1-d)^i - \sum_{i=0}^{2k+1}(1-d)^i\right| \\
&= \sum_{i=2k+2}^{\infty}(1-d)^i \leq \sum_{i=2k+2}^{\infty}\frac{1}{2^i} \\
&= 2^{-2k-1} < 2^{-2k}.
\end{aligned}$$

14.2.5 Der Schaltkreis

Von der Iteration zum Schaltkreis. 14-20

Satz
Es gibt eine NC^2-Schaltkreisfamilie für die Division.

Beweis.
1. Für gegebene Eingaben a und b wird erstmal das »Komma bei b zurechtgerückt«.
2. Dann wenden wir $\log n$ mal die Iterationsvorschrift $x_{k+1} = x_k(2 - dx_k)$ an. Jede dieser $\log n$ Iteration benötigt Tiefe $O(\log n)$.
3. Dann multiplizieren wir das Ergebnis mit a, was in Tiefe $O(\log n)$ geht.
4. Die Gesamttiefe ist also $O(\log^2 n)$. □

Bemerkung: Es gibt einen (wahnsinnig komplizierten) NC^1-Schaltkreis für die Division.

Zusammenfassung dieses Kapitels

14-21

1. Es gibt eine Schaltkreisfamilie für die Multiplikation der *Tiefe* $O(\log n)$.
2. Der Haupttrick ist, dass man zu je drei Zahlen leicht zwei Zahlen finden kann, deren Summe dieselbe ist.
3. Es gibt eine Schaltkreisfamilie für die Division der *Tiefe* $O(\log^2 n)$.
4. Der Trick ist, eine *Newton-Iteration* mittels der Vorschrift $x_{k+1} = x_k(2 - dx_k)$ durchzuführen.

Übungen zu diesem Kapitel

Übung 14.1 (Die Newton-Iteration, leicht)

Führen Sie für die Werte $a = \frac{2}{3}$, $b = \frac{3}{4}$ und $c = 0{,}9$ die Newton-Iteration zur Bestimmung des Kehrwerts durch. Geben Sie jeweils die ersten vier Werte der Iteration exakt an und bestimmen Sie, auf wie viele Dezimalstellen dies genau ist.

Übung 14.2 (Konzept der approximativen Lösung, leicht)

Gegeben sind m Zahlen $\tilde{a}_1, \ldots, \tilde{a}_m$, die n-Approximationen der Zahlen $a_1, \ldots, a_m$ sind.

1. Bestimmen Sie ein möglichst großes k, so dass $\sum_{i=1}^{m} \tilde{a}_i$ eine k-Approximation für $\sum_{i=1}^{m} a_i$ ist.
2. Beweisen Sie Ihre Behauptung aus dem ersten Teil.

Übung 14.3 (Schnelle Addition, schwer)

Zeigen Sie, dass es eine AC^0-Schaltkreisfamilie für die Addition gibt.

Tipps: Gehen Sie zunächst wie bei der NC^1-Schaltkreisfamilie vor, vermeiden Sie dann aber den Baum zur Berechnung des Effektes.

15-1

Kapitel 15
Kleine Platzklassen
(Hoffentlich) eine Wiederholung

15-2

Lernziele dieses Kapitels

1. Kleine Platzklassen wiederholen
2. Logspace-Many-One-Reduktion wiederholen

Inhalte dieses Kapitels

Worum es heute geht

Die Untersuchung von kleinen Platzklassen erscheint auf den ersten Blick reichlich akademisch – oder wie ich vor kurzem in einem Review zu einem Papier von mir lesen konnte »mainly of scholary interest« –, kurz gesagt: überflüssig. Speicherplatz ist (im Gegensatz zu Zeit) in aller Regel wahrlich nicht unser Problem. Wenn eine Ingenieurin ein Gigabyte an Eingabedaten hat, dann wird sie sicherlich auch irgendwo noch Platz für ein Gigabyte weiteren Speicher finden. Man könnte einwenden, dass kleine Systeme wie Smartcards oder eingebettete Systeme wie Waschmaschinensteuerungen nicht mit übermäßigem Speicherplatz gesegnet sind und es sich sehr wohl lohne, sparsam damit umzugehen. Wohl wahr – jedoch erscheint es schon von dagobertduckhaftem Geiz zu zeugen, nur *logarithmisch* viel Platz vorzusehen.

Hat man beispielsweise ein Gigabyte an Eingabedaten, so dürfte man bei einem erlaubten Platz von $\log_2 n$ Bits gerademal 33 Bits an Speicher verbrauchen, also großzügig gerundet zwei 32-Bit-Register. Wächst die Eingabemenge auf 1000 Terabyte, so hätte man immerhin 53 Bits zur Verfügung, also immernoch nicht mehr als zwei Register. Bei der größten in diesem Universum überhaupt denkbaren Eingabe (10^{80}, denn so viele Atome hat das Universum, plus/minus ein paar Zehnerpotenzen), hätte man immerhin zehn 32-Bit-Register zur Verfügung.

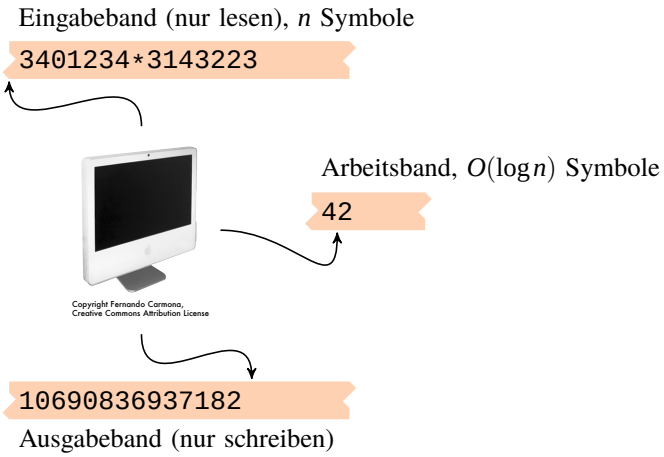
Wozu also diese Klassen untersuchen? Es gibt zwei Gründe. Erstens schätzt man logarithmischen Platz häufig gerade deshalb, weil hier so unrealistisch restriktive Annahmen gemacht werden. Dies bedeutet nämlich, dass, wenn eine Berechnung in logarithmischem Platz möglich ist, wirklich nicht verschwenderisch mit den Ressourcen umgegangen wird. Beispielsweise benötigt man für die Untersuchung der Vollständigkeit von Problemen möglichst schwache oder restriktive Reduktionen, denn die Reduktion selber soll ja gerade möglichst »trivial« sein – es sollen die Probleme sein, welche die »Schwierigkeiten« enthalten. Der zweite Grund für die Untersuchung von kleinen Platzklassen wird im nächsten Kapitel klar werden: Erstaunlicherweise sind logarithmischer Platz und Parallelisierbarkeit von Problemen aufs Engste verwoben.

15.1 Kleine Platzklassen

15.1.1 Die Klasse L

15-4

Wiederholung: Die Klassen L und FL



15-5

Definition der Klassen L und FL

Definition (Die Klasse L und FL)

Eine Funktion $f\colon \Sigma^* \to \Sigma^*$ ist in der *Klasse* FL, falls für sie eine Turingmaschine M mit folgenden Eigenschaften existiert:

- Ihre Bänder sind:
  1. Das *Eingabeband*, auf das nur lesend zugegriffen wird.
  2. Das *Arbeitsband*, das die Größe $O(\log n)$ hat bei Eingaben der Länge n.
  3. Das *Ausgabeband*, auf das nur schreibend zugegriffen wird.
- M schreibt bei Eingabe x das Wort $f(x)$ auf ihr Ausgabeband.

Eine Sprache ist in der *Klasse* L, falls ihre charakteristische Funktion in FL ist.

Zur Diskussion

Ist es egal, ob die Maschine die Ausgabe nur in eine Richtung beschreiben darf? Ist es egal, ob sie die Ausgabe überschreiben darf?

15.1.2 Die Klasse NL

15-6

Definition der Klasse NL

Definition (Die Klasse NL)

Eine Sprache $L \subseteq \Sigma^*$ ist in der *Klasse* NL, falls für sie eine nichtdeterministische Turingmaschine M mit folgenden Eigenschaften existiert:

- Ihre Bänder sind:
  1. Das *Eingabeband*, auf das nur lesend zugegriffen wird.
  2. Das *Arbeitsband*, das die Größe $O(\log n)$ hat bei Eingaben der Länge n.
- Für jede Eingabe $x \in \Sigma^*$ gilt:
  1. Falls $x \in L$, so gibt es (wenigstens) eine akzeptierende Berechnung.
  2. Falls $x \notin L$, so gibt es keine akzeptierende Berechnung.

Zur Diskussion

Welche Probleme treten auf, wenn man die Klasse FNL definieren möchte?

Zur Übung

15-7

Geben Sie für jedes der folgenden Probleme an, was man bezüglich seiner Zugehörigkeit zu L, FL und/oder NL weiß:

1. Regulären Sprachen.
2. Die Additionsfunktion.
3. Die Multiplikationsfunktion.
4. Kreise in Graphen entdecken.
5. Wege finden in Bäumen.
6. Wege finden in ungerichteten Graphen.
7. Wege finden in gerichteten Graphen.
8. 2-SAT (Erfüllbarkeit von Formeln in KNF mit zwei Literalen pro Klausel).
9. 1-Färbbarkeit
10. 2-Färbbarkeit
11. 3-Färbbarkeit
12. 4-Färbbarkeit

15.2 Logspace-Reduktion

15.2.1 Reduktionen allgemein

Wiederholung: Reduktionen.

15-8

Idee

- *Reduktionen* dienen dazu, die Komplexität (Schwierigkeit) von zwei Problemen A und B zu vergleichen.
- Man stellt sich vor, eine Firma/Aliens/Gott könne das Problem B beliebig schnell lösen.
- Die Frage ist nun, ob einem das hilft, das Problem A schnell zu lösen.
- Wenn dies der Fall ist, dann heißt A *reduzierbar auf* B.
- Der Vorgang der Übersetzung der Frage »Ist x in A?« in die Frage »Ist $f(x)$ in B?« heißt *Reduktion*.

Anforderungen an Reduktionen

- Reduktionen sollten *möglichst einfach* sein.
- Reduktionen sollten *transitiv* sein.

15-9

Zur Übung

Für Anfänger
Geben Sie eine Reduktion von CLIQUE auf INDEPENDENT-SET an.

Für Fortgeschrittene
Geben Sie eine Reduktion von 3-COLORABLE auf 4-COLORABLE an.

Für Profis
Geben Sie eine Reduktion von 4-COLORABLE auf 3-COLORABLE an.

Für Gurus
Geben Sie eine Reduktion von 2-SAT auf REACHABILITY an.

15-10

Was man alles bei einer Reduktion festlegen muss.

Folgende Fragen muss man bei Reduktionen klären:

- Wie viele Fragen (an B) darf man stellen?
- Wie aufwändig darf es sein, die Frage(n) zu berechnen?
- Was darf man mit den Antworten anstellen?

Die Logspace-Many-One-Reduktion
Wir werden in dieser Veranstaltung folgende Reduktion betrachten:

- Man darf *eine Frage* stellen.
- Man darf *logarithmischen Platz* benutzen, um die Frage zu berechnen.
- Man *muss die Antwort übernehmen*.

15.2.2 Logspace-Reduktionen

15-11

Definition der Logspace-Many-One-Reduktion.

Definition (Logspace-Many-One-Reduktion)
Seien A und B Sprachen. Wir schreiben $A \leq_{\mathrm{m}}^{\log} B$, falls es eine Funktion $f\colon \Sigma^* \to \Sigma^*$ gibt mit folgenden Eigenschaften:

1. $f \in \mathrm{FL}$.
2. Für alle $x \in \Sigma^*$ gilt $x \in A$ genau dann, wenn $f(x) \in B$.

15.2.3 Kompositionalität

Zentrale Eigenschaften der Logspace-Many-One-Reduktion. 15-12

1. Logspace-Many-One-Reduktionen sind effizient zu berechnen.
2. Die Logspace-Many-One-Reduktion ist transitiv.

Zur zweiten Behauptung:

Satz
Die Klasse FL *ist abgeschlossen unter Komposition. Sind also* $f, g \in \mathrm{FL}$*, so auch* $f \circ g \in \mathrm{FL}$.

Beweis der Abgeschlossenheit von FL unter Komposition. 15-13

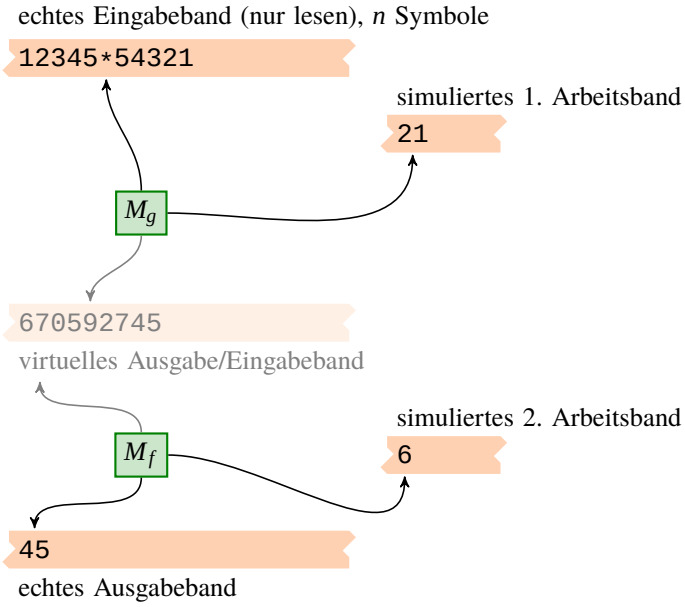
Beweis.

- Sei eine Eingabe x gegeben. Wir wollen $f(g(x))$ in logarithmischem Platz berechnen.
- Man kann *nicht* einfach erst $g(x)$ ausrechnen und dann f auf das Ergebnis anwenden, da das Zwischenergebnis viel zu lang ist.
- Der Trick ist, ein »virtuelles« Arbeitsband zu benutzen:
  - Man beginnt mit der Berechnung von $f(g(x))$ und tut so, als wüsste man, wie $g(x)$ lautet.
  - Wenn nun tatsächlich ein Bit von $g(x)$ benötigt wird, wird in diesem Moment erst eine Simulation von g auf der (echten) Eingabe x gestartet.
  - Die Ausgaben der Simulation werden unterdrückt, bis auf das interessierende Bit. □

Beispiel zweier logspace-berechenbaren Funktionen, die wir komponieren wollen. 15-14

- Die erste Funktion g sei die Multiplikationsfunktion.
  Genauer: die Funktion, die durch einen Stern getrennte Ziffernfolgen auf das Produkt der Ziffernfolgen abbilden.
- Sei M_g eine Logspace-Maschine, die g berechnet.
  (Dass eine solche existiert, sehen wir nächstes Mal.)
- Die zweite Funktion f sei die Quersummenfunktion.
  Genauer: Die Funktion, die eine Ziffernfolge auf die Summe dieser Ziffern abbildet.
- Sei M_f eine Logspace-Maschine, die f berechnet.
- Die Komposition $f \circ g$ bildet beispielsweise `12345*54321` auf das Wort `45` ab, da $12345 \cdot 54321 = 670592745$ und die Quersumme von 670592745 gerade 45 ist.

Visualisierung des »virtuellen Bandes«. 15-15



Zusammenfassung dieses Kapitels

15-16

1. Logspace-Maschinen dürfen ihre *Eingabe nur lesen*, ihre *Ausgabe nur schreiben* und *nur logarithmisch viel Platz* auf ihrem Arbeitsband benutzen.
2. Die *Standardreduktion* ist die *Logspace-Many-One-Reduktion*.
3. Diese Reduktion ist *abgeschlossen und Komposition*, was *nicht trivial* ist.

16-1

Kapitel 16
Verhältnis von Platz und Parallelisierbarkeit

Wenig Platz = gut parallelisierbar

16-2

Lernziele dieses Kapitels

1. Inklusionsstruktur der NC-Klassen und der kleinen Platzklassen kennen
2. Die grundlegenden Beweisideen verstehen

Inhalte dieses Kapitels

Worum es heute geht

Platz und Parallelisierbarkeit haben ein intimes Verhältnis, sie sind ineinander verschlungen wie die Leiber zweier Liebenden. Wer mit solche kitschigen Antropomorphisierungen nichts anfangen kann, der merkt sich statt dessen: $NC^1 \subseteq L \subseteq NL \subseteq AC^1$. Diese Inklusionskette liefert die fehlende Motivation nach, sich überhaupt mit Platzklassen zu beschäftigen. Sie besagt nämlich, dass jedes Problem, das sich mit logarithmischem Platz (falls gewünscht auch nur nichtdeterministisch) entscheiden lässt, gut parallelisierbar ist. Umgekehrt lassen sich alle sehr gut parallelisierbaren Probleme auch platzeffizient lösen, was ja zunächst nicht klar ist.

Als einfaches Beispiel für die Vorteile einer solchen Charakterisierung sei an Übungsaufgabe 13.3 erinnert. Dort soll gezeigt werden, dass jede reguläre Sprache in NC^1 liegt. Falls Sie sich an dieser Aufgabe versucht haben, werden Sie sich vielleicht erinnern, dass dies alles andere als trivial zu zeigen ist. Jedoch *ist* es trivial, dass alle regulären Sprachen in logarithmischem Platz entschieden werden können (logarithmischer Platz ist nämlich mehr als gar kein Platz). Hieraus folgt dann sofort, dass alle regulären Sprachen in AC^1 liegen, was schon »fast« das gewünschte Resultat ist.

Die Inklusionen $NC^1 \subseteq L$ und $NL \subseteq AC^1$ sind beide nicht ganz einfach zu zeigen. Für das erste Resultat ist die wesentliche Beweisidee, dass man zum Auswerten eines Schaltkreises logarithmischer Tiefe mit beschränktem Eingangsgrad lediglich einen »Call-Stack« logarithmischer Tiefe benötigt. Für die zweite Inklusion macht man sich zu nutze, dass man das Erreichbarkeitsproblem auf die Berechnung von Potenzen von Graph-Adjazenzmatrizen reduzieren kann. Potenzen wiederum lassen sich recht gut mit einem flachen Schaltkreis berechnen durch iteriertes Quadrieren.

16-4

Zentraler Zusammenhang von Platz und Parallelisierbarkeit.

Motto

Logarithmischer Platz = gute Parallelisierbarkeit

Genauer zeigen wir:

$$NC^1 \subseteq L \subseteq NL \subseteq AC^1.$$

- Der Zusammenhang zwischen Platz und Parallelisierbarkeit ist überraschend.
- Er ist eine wichtige Motivation, sich mit kleinen Platzklassen zu beschäftigen.
- Es ist oft einfacher, einen platzeffizienten Algorithmus anzugeben, als einen AC^1-Schaltkreis.

16.1 NC^1 liegt in L

16.1.1 Die Behauptung

16-5

NC^1 liegt in L.

Satz

$NC^1 \subseteq L$.

Bemerkungen:

- Wir gehen von *logspace-uniformem* NC^1 aus.
- Die Behauptung ist also, dass sich jede Berechnung, die eine flache Schaltkreisfamilie durchführen kann, auch in wenig Platz möglich ist.
- Dies ist nicht gerade offensichtlich, da man zur »naiven« Auswertung eines Schaltkreises viel Platz braucht.

16.1.2 Beweisplan

16-6

Plan des Beweis des Satzes.

Voraussetzungen und Ziel des Beweises.

- Sei C eine logspace-uniforme Schaltkreisfamilie der Tiefe $O(\log n)$ für eine Sprache A.
- Wir wollen zeigen, dass $A \in L$ gilt.
- Dazu konstruieren wir eine Logspace-Maschine M, die A entscheidet.
- Sei $x \in \Sigma^*$ eine Eingabe.

Grobes Vorgehen.

1. Zunächst berechnen wir den Code des Schaltkreises $C_{|x|}$, der für x »zuständig« ist.
2. Dann berechnen wir, beginnend beim Ausgabegatter, rekursiv den Wert des Schaltkreises.
3. Während der Rekursion müssen wir uns pro Gatter auf dem Pfad immer nur wenig merken.

16.1.3 Beweisdetails

Die Berechnung des Schaltkreises. 16-7

- Der erste Schritt war, den Schaltkreis $C_{|x|}$ zu berechnen.
- Da C logspace-uniform ist, gibt es eine Logspace-Turingmaschine M_C, die bei Eingabe $1^{|x|}$ gerade $C_{|x|}$ ausgibt.
- Diese Turingmaschine benutzen wir, wobei wir »ihr vorspiegeln«, dass die Eingabe $1^{|x|}$ ist.
- Das Ergebnis, nämlich der Code von $C_{|x|}$, wird allerdings nicht auf das Arbeitsband geschrieben.
  Vielmehr werden die Bits des Ergebnis wie beim Beweis, dass FL unter Komposition abgeschlossen ist, immer wieder »on the fly« berechnet.

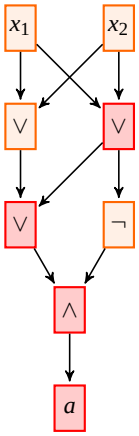
Informationspfade speichern unser bisheriges Wissen. 16-8

Definition (Informationspfad)
Ein *Informationspfad* ist ein Folge von Tripeln. Jedes Tripel besteht aus:

1. Einem *Richtungswert* »links« oder »recht«.
2. Einem *linken Elternwert* aus der Menge $\{0,1,?\}$.
3. Einem *rechten Elternwert* aus der Menge $\{0,1,?\}$.

Definition (Ausgewähltes Gatter)
Gegeben seien ein Informationspfad und ein Schaltkreis C mit einem Ausgabegatter. Das *von dem Pfad ausgewählte Gatter* ist dasjenige Gatter, das man vom Ausgabegatter erreicht, wenn man den Richtungsanweisungen des Pfades folgt. Bei Gattern mit nur einem Elternknoten ist die Richtung nicht wichtig.

Beispiel eines Informationspfades und des ausgewählten Gatters. 16-9



Informationspfad:
$(\text{links},?,1),(\text{links},?,?),(\text{rechts},0,?)$

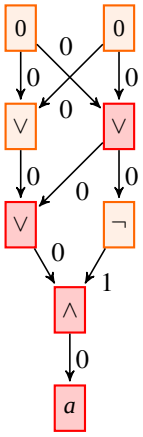
Gültige Pfade. 16-10

Definition (Gültiges Ende)
Sei p ein Informationspfad und C ein Schaltkreis. Wir sagen, p *hat ein gültiges Ende*, wenn für das von dem Pfad ausgewählte Gatter g folgendes gilt: Ist der linke oder rechte Elternwerte im letzten Tripel des Pfades kein Fragezeichen, so ist dieser Wert gerade die Ausgabe des linken oder rechten Elterngatters von g.

Definition (Gültige Pfade)
Ein Informationspfad heißt *gültig in Bezug auf einen Schaltkreis C*, wenn jedes seiner Anfangsstücke ein gültiges Ende hat.

16-11

Die Anfangsstücke des Pfades haben gültige Enden.



Informationspfad:

- $(\text{links}, ?, 1), (\text{links}, ?, ?), (\text{rechts}, 0, ?)$

Anfangsstücke:

- $(\text{links}, ?, 1)$
- $(\text{links}, ?, 1), (\text{links}, ?, ?)$
- $(\text{links}, ?, 1), (\text{links}, ?, ?), (\text{rechts}, 0, ?)$

16-12

Gewünschte Eigenschaften des Algorithmus *EvaluateGate*.

Ein-/Ausgabeverhalten

Eingabe Ein gültiger Informationspfad p.

Ausgabe Wert des von p ausgewählten Gatters, wenn an die Eingänge des Schaltkreises $C_{|x|}$ der Wert x angelegt wird.

Zentrale Eigenschaften

1. Der Informationspfad ist immer gültig.
2. Der Pfad wird nie länger als $O(\log n)$, da der Schaltkreis Tiefe $O(\log n)$ hat.
3. Folglich brauchen wir nur $O(\log n)$ Bits zur Speicherung des Informationspfades.

16-13

Der Algorithmus *EvaluateGate*.

```
1  i ← das von p ausgewählte Gatter
2  // Bestimmung durch Ablaufen des Schaltkreises entsprechend den Pfadangabe
3
4  if i ist k-tes Eingabegatter then
5      output k-tes Bit von x
6  else
7      last ← das letzte Paar/Tripel von p
8      for Richtungen r ∈ {links, rechts} seq do
9          if der r-Elternwert innerhalb last ist ? then
10             Füge (r, ?, ?) temporär an p an
11             call EvaluateGate
12             r-Wert in last ← Ergebnis
13             Entferne das temporäre Tripel wieder
14     Berechne den Wert von Gatter i aufgrund der linken und rechten Elternwerte und
           gibt diesen zurück
```

16.2 NL liegt in AC^1

16.2.1 Die Behauptung

NL liegt in AC^1. 16-14

Satz
$NL \subseteq AC^1$.

Bemerkungen:

- Die Behauptung ist, dass sich jede nichtdeterministische Logspace-Berechnung auch mittels eines Schaltkreises logarithmischer Tiefe durchführen lässt.
- Es ist oft einfacher, eine Logspace-Maschine für ein Problem zu finden, als einen AC^1-Schaltkreis.

16.2.2 Beweisplan

Plan des Beweises des Satzes. 16-15

Voraussetzungen und Ziel des Beweises.

- Sei M eine nichtdeterministische Logspace-Turingmaschine, die eine Sprache A akzeptiert.
- Wir konstruieren eine logspace-uniforme AC^1-Schaltkreisfamilie der Tiefe $O(\log n)$ für A.
- Sei $x \in \Sigma^*$ eine Eingabe.

Grobes Vorgehen.

1. Wir betrachten den *Konfigurationsgraph* von M.
2. Wir *halbieren Entfernungen* in diesem Graphen $O(\log n)$ mal.
3. Wir geben 1 aus, falls nun eine direkte Kante von der Anfangskonfiguration zur akzeptierenden Konfiguration führt.

16.2.3 Beweisdetails

Der Konfigurationsgraph von M. 16-16

Definition
Der *Konfigurationsgraph G_x von M bei Eingabe x* hat als Knoten alle Konfigurationen von M bei Eingabe x. Er enthält eine Kante von einer Konfigurationen c zu einer Konfiguration c', wenn M in einem Schritt von c zu c' kommen kann.

Wie dürfen annehmen, dass G_x genau eine Anfangskonfiguration enthält und auch genau eine akzeptierende Endkonfiguration.

Zur Diskussion
Wieso ist der Konfigurationsgraph nur polynomiell groß?

16-17

Schritt 1: Berechnen der Adjazenzmatrix des Konfigurationsgraphen.

- Unser erstes Ziel ist eine Schicht Gatter mit folgenden Eigenschaften:
  1. Für jedes Paar (c, c') von Konfigurationen gibt es ein Gatter $g_{c,c'}$ in der Schicht.
  2. Ist c' eine Nachfolgerkonfiguration von c, so soll $g_{c,c'}$ gerade 1 liefern, sonst 0.
- Häufig kann man sofort ausschließen, dass c' Nachfolger von c ist (beispielsweise, wenn c' eine ganz andere Kopfposition als c hat). In diesem Fall ist $g_{c,c'}$ einfach das konstante 0-Gatter.
- Manchmal weiß man auch sofort, dass c' auf jeden Fall Nachfolgerkonfiguration von c ist und $g_{c,c'}$ ist das 1-Gatter.
- In den restlichen Fällen ist c' nur dann Nachfolgerkonfiguration von c, wenn ein bestimmtes Eingabesymbol (nämlich das an der Stelle des Schreib-Lese-Kopfes von c) einen bestimmten Wert hat. In diesem Fall liefert $g_{c,c'}$ diesen Wert (oder seine Negation).

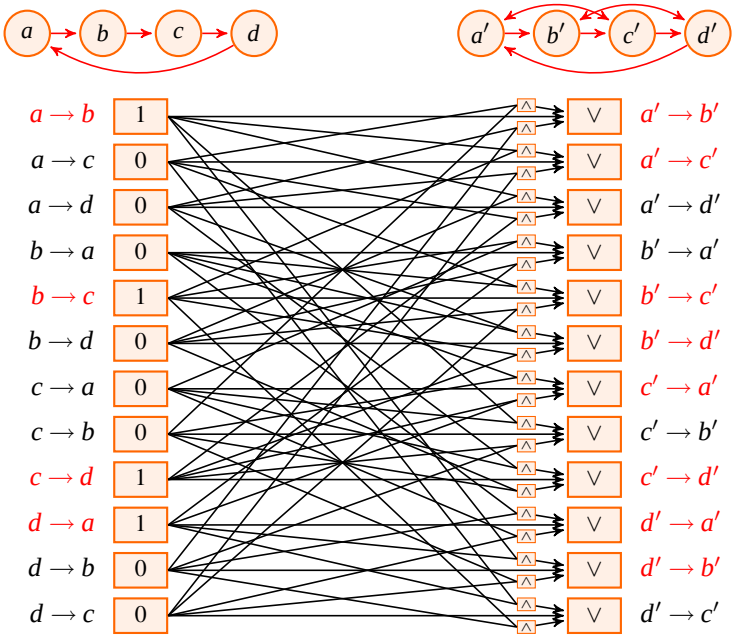
16-18

Schritt 2: Das Verkürzen von Entfernungen.

- Wir fügen für jedes Paar u und v von Knoten in dem Graphen eine Kante ein, falls ein Knoten x *existiert*, für den es Kanten $u \to x$ und $x \to v$ gibt.
- Dies lässt sich durch eine *Schicht konstanter Tiefe* in unserem Schaltkreis bewerkstelligen.
  Eingaben jeder Schicht liegen an Gattern an, die die Adjazenzmatrix des Graphen kodieren, Ausgaben werden an Gattern geliefert, die die Adjazenzmatrix des Graphen mit halbierten Entfernungen darstellen.
- Jede Schicht hat konstante Tiefe, allerdings brauchen wir ein *Oder-Gatter mit Fan-in* n, weshalb wir einen AC^1-Schaltkreis erhalten und keinen NC^1-Schaltkreis.
- Nach $O(\log n)$ *Schichten* kann man das Ergebnis am Gatter für die Kante von der Anfangs- zur akzeptierenden Konfiguration ablesen.

16-19

Eine Schicht von Gattern, die Entfernungen halbiert.



Zusammenfassung dieses Kapitels

16-20

1. Logarithmischer Platz = gute Parallelisierbarkeit.
2. Genauer: $\mathrm{NC}^1 \subseteq \mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{AC}^1$.

Übungen zu diesem Kapitel

Übung 16.1 (Klassen- und Sprachdiagramm, mittel)

Zeichnen Sie ein einziges großes Diagramm, dass die Inklusions- und Elementbeziehungen zwischen den nachfolgenden Klassen und Sprachen darstellt. Die Klassen sollen als Flächen dargestellt werden, die Sprachen als Punkte, die in den passenden Flächen liegen. Inklusionsbeziehungen zwischen Flächen sollen Inklusionsbeziehungen zwischen Klassen andeuten; ist die Beziehung zweier Klassen nicht bekannt, so sollte keine der Flächen in der anderen enthalten sein. Alle Schaltkreisklassen sollen logspace-uniform sein.

Die Sprachklassen:

- Die AC-Klassen AC^0, AC^1, AC^2, ..., sowie AC selber.
- Die NC-Klassen NC^0, NC^1, NC^2, ..., sowie NC selber.
- POLYSIZE
- Die Platzklassen L, NL, PSPACE, NPSPACE.
- Die Zeitklassen P, NP, EXP, NEXP.
- Die Sprachklassen REG und CFL.
- Die Rekursionsklassen REC und RE.

Die Sprachen:

- Palindrome
- Parität
- SAT
- gerichtete Grapherreichbarkeit (REACH oder GAP)
- Halteproblem
- Circuit-Value-Problem

Tipps: Man weiß, dass Parity nicht von polynomiell großen Schaltkreisen der Tiefe $o(\log n/\log\log n)$ berechnet werden kann. Man weiß nicht, ob CFL eine Teilmenge von NL ist.

Übung 16.2 (Abgeschlossenheit von NC unter Logspace-Many-One-Reduktion, leicht)

Zeigen Sie: Die Klasse NC ist abgeschlossen unter Logspace-Many-One-Reduktion, wobei nur Reduktionsfunktionen f zulässig sind mit $|f(x)| = |x|$ für alle x. Solche Reduktionen bilden also jedes Wort auf ein Wort derselben Länge ab.

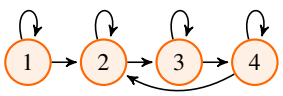
Übung 16.3 (Die Klassen NC und PRAMs, schwer)

Sei L eine Sprache aus logspace-uniformem NC, die also von einer logspace-uniformen Schaltkreisfamilie $(C_n)_{n\in\mathbb{N}}$ entschieden wird. Zeigen Sie, dass es eine PRAM gibt, die L entscheidet und dabei polynomielle Arbeit und polylogarithmische Zeit benötigt. Lösen Sie dazu folgende Teilaufgaben:

1. Nehmen Sie an, dass für ein gegebenes Wort x der Schaltkreis $C_{|x|}$ schon geeignet kodiert im Speicher der PRAM vorliegt. Beschreiben Sie eine PRAM, die mit polynomieller Arbeit und in polylogarithmischer Zeit entscheidet, ob $x \in L$ gilt.
2. Beschreiben Sie eine PRAM, die auf Eingabe eines Wortes x den Schaltkreis $C_{|x|}$ generiert, d. h. in der in (a) benutzten Kodierung in den Speicher schreibt. Die PRAM soll polynomielle Arbeit und polylogarithmische Zeit benötigen.
   Tipps: Die wesentliche Idee ist dieselbe wie im Beweis, dass $\mathrm{NL} \subseteq \mathrm{AC}^1$.

Übung 16.4 (Beispiel Matrixpotenzen und Erreichbarkeit, leicht)

Gegeben sei der folgende Graph:



Geben Sie die Adjazenzmatrix A dieses Graphen an (diese hat eine 1 in Zeile i und Spalte j, wenn es eine Kante von Knoten i zu Knoten j gibt, sonst eine 0). Geben Sie dann die Potenzen A^2, A^3 und A^4 an.

Übung 16.5 (Matrixpotenzen und Erreichbarkeit allgemein, schwer)

Sei G ein gerichteter Graph mit Schlaufen an allen Knoten. Die Knotenmenge sei $V = \{1,\ldots,n\}$ und A sei seine Adjazenzmatrix. Zeigen Sie, dass es genau dann einen Pfad von Knoten 1 zu Knoten n in G gibt, wenn der Eintrag in Zeile 1 und Spalte n von A^{n-1} nicht 0 ist.

Tipp: Induktion für eine stärke Behauptung, die einen Zusammenhang zwischen Pfaden der Länge k und der Matrix A^k herstellt.

17-1

Kapitel 17
Untere Schranken I
P-Vollständigkeit

17-2

Lernziele dieses Kapitels

1. Konzept der Vollständigkeit wiederholen
2. P-Vollständigkeit als untere Schranke begreifen und eigene Beweise führen können

Inhalte dieses Kapitels

Worum es heute geht

Untere Schranken sind die Hohe Kunst in der Theoretischen Informatik. Mit einer unteren Schranke macht man eine Aussage der Form »dieses Problem lässt sich nicht schneller/besser/weiter als so und so lösen«. Zum Vergleich: eine obere Schranke ist eine Aussage der Form »dieses Problem lässt sich so und so schnell/gut/weit lösen«. Untere Schranken sind in der Regel schwieriger zu beweisen als obere Schranken, denn es wird ja behauptet dass *kein* Algorithmus etwas bestimmtes leisten kann – und es gibt immerhin unendlich viele. Wohingegen man bei einer oberen Schranke nur einen konkreten Algorithmus analysieren muss, muss man bei einer unteren Schranke eine allgemeine, ja »grundsätzliche« Aussage über Algorithmen in Bezug auf das gegebene Problem machen.

Theoretiker vermuten viele untere Schranken, können aber nur die wenigstens beweisen. Das bekannteste Beispiel einer vermuteten unteren Schranke ist sicherlich die P-NP-Frage, denn die Behauptung $\mathrm{P} \neq \mathrm{NP}$ ist ja äquivalent zur Aussage »jeder Algorithmus, der das SAT-Problem entscheidet, hat mehr als polynomielle Laufzeit«. Wenn man nun eine solche untere Schranke nicht bewiesen bekommt, so kann man wenigstens Indizien versuchen zu sammeln, was wir in Form von *Vollständigkeitsresultaten* machen.

In diesem Kapitel soll es um eine (weitere) vermutete untere Schranken gehen, diesmal in Bezug auf Parallelverarbeitung: Es wird vermutet, dass das Schaltkreisauswertungsproblem (ein etwas sperriges Wort, das wir handlicher als CVP abkürzen werden) nicht parallelisiert werden kann. Formal: $\mathrm{CVP} \notin \mathrm{NC}$. Auch hier ist die Theoretische Informatik von einem Beweis *sehr* weit entfernt. Alle derzeit bekannten unteren Schranken für Schaltkreise zeigen, dass bestimmte Probleme (teilweise noch viel schwierigere als CVP) nicht mit Schaltkreisen *konstanter Tiefe* lösbar sind. Zwischen »konstanter Tiefe« und »polylogarithmischer Tiefe« (also $O(\log^i n)$ für ein geeignetes i) klaffen leider beweistechnisch Welten.

Wenn man schon nicht direkt zeigen kann, dass $\mathrm{CVP} \notin \mathrm{NC}$ gilt, so wird man versuchen, wenigstens ein Vollständigkeitsresultat zu beweisen. Genau dies ist auch möglich, wir werden zeigen, dass CVP vollständig ist für P.

17.1 Untere Schranken

17.1.1 Motivation

Was sind untere Schranken?

17-4

- Wir möchten die *Komplexität von Problemen verstehen*.
- Wir wollen wissen, *ob* sich Probleme *parallelisieren* lassen, und, wenn ja, *wie gut*.
- Idealerweise wollen wir Aussagen der folgenden Art treffen:
  - Problem XY ist *mindestens so und so schwer*.
  - Problem XY ist *höchstens so und so schwer*.
- Aussagen der ersten Art nennt man *untere Schranken*, Aussagen der zweiten Art heißen *obere Schranken*.

Wie zeigt man untere und obere Schranken?

17-5

- *Obere Schranken* zeigt man, indem man *einen konkreten Algorithmus angibt*, der die behauptete Laufzeit / Güte / etc. hat.
  Dies ist *oft leicht*.
- *Untere Schranken* zeigt man (idealerweise) wie folgt: *Alle Algorithmen*, die das Problem lösen, haben eine Laufzeit / Güte / etc. von mindestens dem behaupteten Wert.
  Dies ist *oft schwierig*.

Wir werden zwei Arten von Schranken betrachten, die mit *Parallelisierbarkeit* zu tun haben:

- Untere Schranken für die *Tiefe* von Schaltkreisen für Probleme in P.
- Untere Schranken für die *Laufzeit* von PRAMs.

17.1.2 Tiefenschranken

Das P-NP-Problem und das P-NC-Problem.

17-6

Bekannt: Das P-NP-Problem
Gilt $\mathrm{P} = \mathrm{NP}$?
(Lassen sich alle Probleme in NP effizient lösen?)

Neu: Das P-NC-Problem
Gilt $\mathrm{P} = \mathrm{NC}$?
(Lassen sich alle Probleme in P effizient parallelisieren?)

Wo begegnen uns untere Schranken?

17-7

- Das P-NP-Problem ist ein *Untere-Schranken-Problem*:
  - Man möchte wissen, ob man ein Problem in NP finden kann, dessen Laufzeit *mindestens superpolynomiell* ist.
  - Dieses Problem ist bekanntermaßen *ungelöst*.
- Analog zur P-NP-Frage liegt die P-NC-Frage:
  - Man möchte wissen, ob man ein Problem in P finden kann, das nicht in NC liegt, dessen Schaltkreistiefe also echt größer ist als $O(\log^i n)$ für alle i.
  - Auch dieses Problem ist *ungelöst*.

»Schwache« untere Schranken.

17-8

Schwache untere Schranken für NP

- Man begnügt sich deshalb mit einer *schwachen* Aussage:
  Man *kann wenigstens* für einige Problem in NP zeigen, dass sie *die schwierigsten Probleme sind.*
- Für solche Probleme gilt: Gibt es für *irgendein* Problem in NP eine superpolynomielle untere Laufzeitschranke, dann auch *für alle schwierigsten* Probleme.

Schwache untere Schranken für P

- Genauso wollen wir mit NC verfahren:
  Wir zeigen, dass es schwierigste Probleme in P gibt.
- Kann man für *irgendein* Problem in P zeigen, dass es nicht in NC liegt (sich nicht parallelisieren lässt), so auch *für alle P-vollständigen Probleme*.

17.1.3 Laufzeitschranken

17-9

Laufzeitschranken für PRAMs.

- Für bestimmte Arten von PRAMs und bestimmte Problem lassen sich *untere Schranken für die Laufzeit* beweisen.
- Wir werden zeigen, dass *Comparison-PRAMs* mit p Prozessoren zum Finden eines Elementes in einer *sortierten Liste der Länge* n mindestens $\Omega(\log n/\log p)$ Schritte benötigt.
- Wir werden zeigen, dass *Comparison-PRAMs* Maxima bestenfalls in Zeit $\Omega(\log\log n)$ berechnen können.

17-10

»Starke« untere Schranken für die Laufzeit von PRAMs.

- Die Resultate für Comparison-PRAMs sind *starke* Resultate – sie gelten absolut und ohne Annahmen.
- Wir beweisen sie durch *Adversary-Argumente*.
- Wir »erkaufen« die starken Resultate dadurch, dass unser Modell (die Comparison-PRAM) nicht sonderlich mächtig ist.

17.2 P-Vollständigkeit

17.2.1 Wiederholung: Vollständigkeit

17-11

Wiederholung: Der Begriff der Abgeschlossenheit.

Definition (Abgeschlossen)

Eine Klasse C von Sprachen heißt *abgeschlossen unter* $\leq_{\mathrm{m}}^{\log}$*-Reduktionen*, falls aus $A \leq_{\mathrm{m}}^{\log} B$ und $B \in C$ folgt $A \in C$.

Motto

Abgeschlossene Klassen kann man mit Reduktionen nicht »verlassen«.

Satz

Die Klassen L, NL, NC, P *und* NP *sind abgeschlossen.*

Andeutungen zu den Beweisen.

- Für L argumentiert man ähnlich wie bei der Abgeschlossenheit von FL unter Komposition.
- Für NC benutzt man $\mathrm{L} \subseteq \mathrm{AC}^1$, für P nutzt man $\mathrm{FL} \subseteq \mathrm{FP}$.
- Für NL und NP argumentiert man wie bei L und P, der Nichtdeterminismus macht keine Probleme.

17-12

Wiederholung: »Schwere« oder »harte« Probleme.

Definition (Schwere Probleme)

Seien

- C eine Klasse von Sprachen und
- X ein Problem (nicht unbedingt in C).

Dann heißt X *schwer für* C *(unter* $\leq_{\mathrm{m}}^{\log}$*-Reduktionen)*, falls

- für *alle* Probleme $A \in C$ gilt:
- $A \leq_{\mathrm{m}}^{\log} X$.

Motto

Kann man ein für C schweres Problem effizient lösen, dann kann man alle Probleme in C effizient lösen.

Wiederholung: Vollständigkeit.

17-13

Definition (Vollständige Probleme)

Seien

- C eine Klasse von Sprachen und
- X ein Problem.

Dann heißt X *vollständig für C (unter $\leq_m^{\log}$-Reduktionen)*, falls

- $X \in C$ und
- X ist schwer für C.

Motto

Vollständige Probleme sind die schwierigsten Probleme einer Klasse.

Zur Übung

17-14

Für Anfänger

Nennen Sie möglichst viele NP-vollständige Probleme.

Für Profis

Nennen Sie ein Problem, das nicht NP-vollständig ist.

Für Gurus

Nennen Sie eine Sprachklasse, die kein vollständiges Problem hat.

17.2.2 Definition: CVP

Welche Probleme lassen sich gut parallelisieren?

17-15

Wir kennen schon viele Probleme in P, die sich gut parallelisieren lassen, die also in NC liegen:

- Grundrechenarten
- Sortieren
- Verschmelzen
- Ranking
- Auswerten von arithmetischen Ausdrücken
- Wurzeln finden in Bäumen
- Alle Probleme in NL
- Insbesondere viele Wege-Finde-Probleme in Graphen

Kandidat für ein Problem, das sich nicht parallelisieren lässt.

17-16

Definition (Circuit-Value-Problem, CVP)

Eingabe Der Code eines Schaltkreises ohne Eingabegatter, in dem dann natürlich 0-Gatter und 1-Gatter vorkommen dürfen.

Ausgabe Wert des Schaltkreises.

Bemerkungen

Schaltkreise kommen hier in zwei unterschiedlichen Rollen vor:

1. Schaltkreise sind hier *Eingaben*.
2. Wollte man zeigen CVP $\in$ NC, so müsst man *eine Schaltkreisfamilie angeben*, die *Schaltkreiscodes als Eingabe bekommt*.

17.2.3 Satz: CVP ist P-vollständig

17-17

Das Hauptresultat.

Satz
CVP *ist* P-*vollständig.*

Dies bedeutet *zwar nicht*, dass sich CVP nicht gut parallel auswerten lässt, macht es aber *»unwahrscheinlich«.*

17-18

Wie beweist man Vollständigkeit?
Die Bootstrapping-Methode.

Ausgangslage
Sei C eine Sprachklasse und X ein Problem. Wir wollen zeigen, dass X vollständig ist für C.

Bootstrapping-Methode

1. Man zeigt zunächst $X \in C$ mittels eines konkreten Algorithmus.
2. Man zeigt dann für jedes Problem in $A \in C$, dass $A \leq_{\mathrm{m}}^{\log} X$ gilt.

- Diese Methode ist »anstrengend«.
- Beim »ersten Mal« ist sie aber nicht zu vermeiden.
- Den Satz von Cook beweist man durch Bootstrapping.
- Wir werden die P-Vollständigkeit von CVP auch so beweisen müssen.

17-19

Wie beweist man Vollständigkeit?
Die Reduktions-Methode.

Ausgangslage
Sei C eine Sprachklasse und X ein Problem. Wir wollen zeigen, dass X vollständig ist für C. Wir kennen schon ein Problem Y, dass vollständig ist für C.

Reduktions-Methode

1. Man zeigt zunächst $X \in C$ mittels eines konkreten Algorithmus.
2. Man zeigt dann $Y \leq_{\mathrm{m}}^{\log} X$.

- Diese Methode ist »oft einfach«.
- Fast alle Vollständigkeitsbeweise benutzen diese Methode.

17-20

CVP ist in P.

Lemma (Erster Beweisteil)
CVP $\in$ P.

Zur Übung
Skizzieren Sie den Beweis.

Reduktion beliebiger Probleme aus P.

17-21

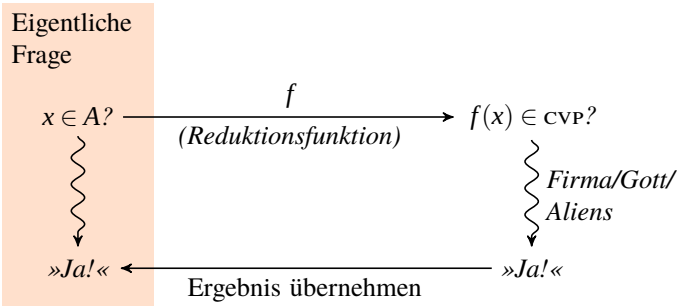
Lemma (Zweiter Beweisteil)
Sei $A \in \mathrm{P}$ ein beliebiges Problem. Dann gilt $A \leq_{\mathrm{m}}^{\log} \mathrm{CVP}$.

Grobes Ziel

- Sei x eine Eingabe für die Reduktion.
- Wir wollen herausfinden, ob $x \in A$ gilt.
- Dazu müssen wir (in logarithmischem Platz) einen Schaltkreis C konstruieren mit folgenden Eigenschaften:
  1. Gilt $x \in A$, so wertet C zu 1 aus.
  2. Gilt $x \notin A$, so wertet C zu 0 aus.

Visualisierung der groben Idee der Reduktion.

17-22



Übersicht zum Beweis.

17-23

Im Folgenden sollen folgende Punkte angesprochen werden:

1. Einige Vereinfachungen
2. Aufbau des Schaltkreises
3. Aufbau der Boxen im Schaltkreis
4. Eigenschaften des Schaltkreises
5. Wie schwierig ist es, den Schaltkreis zu konstruieren?

Von der Maschine zum Schaltkreises.

17-24

- Sei M eine Polynomialzeit-Turingmaschine, die A entscheidet. Dann muss der zu konstruierende Schaltkreis C folgende Eigenschaft haben:
  1. Akzeptiert M die Eingabe x, so wertet C zu 1 aus.
  2. Verwirft M die Eingabe x, so wertet C zu 0 aus.

Wir beginnen mit ein paar Vereinfachungen:

- Die Maschine M habe nur *ein Band*.
- Das Band der Maschine M sei *einseitig beschränkt*.
- Es gibt *genau eine* akzeptierende Endkonfiguration.
- Bei dieser Endkonfiguration ist der *Kopf am Anfang*.

Wir führen folgende Bezeichnungen ein:

- Die Laufzeit der Maschine bei Eingaben der Länge n sei *genau* $p(n)$ für ein festes Polynom p.
- Die Länge der Eingabe x heiße n.

17-25

Grober Aufbau des Schaltkreises.

Wir bauen den Schaltkreis grob wie folgt:

- Er besteht aus $p(n)$ *Schichten*.
- An den Ausgabegattern der k-ten Schicht liegt *kodiert die k-te Bandkonfiguration an*.
- An den Eingabegattern der k-ten Schicht liegt *kodiert die $(k-1)$-te Bandkonfiguration an*.

Die Kodierung einer Bandkonfiguration funktioniert wie folgt:

- Um eine Bandzelle zu kodieren, benutzen wir mehrere Leitungen.
- Ist das Bandalphabet Γ, so können in einer Zelle $|\Gamma|$ verschiedene Symbole stehen; wir benutzen deshalb $\log|\Gamma|$ Leitungen zur Kodierung des Zelleninhalts.
- Zusätzlich benutzt man noch pro Zelle eine Leitung, um zu signalisieren, ob der Kopf dort ist.
- Schließlich benutzen wir noch $\log|Q|$ Leitungen, um den aktuellen Zustand der Turingmaschine zu kodieren.

17-26

Was leistet der Schaltkreis?

- Betrachtet man die Leitungen, die in die *erste* Schicht hineinführen, so kodieren sie genau die Anfangskonfiguration.
- Betrachtet man die Leitungen, die in die *zweite* Schicht hineinführen, so kodieren sie genau den Bandinhalt nach einem Schritt.
- Betrachtet man die Leitungen, die in die *dritte* Schritt hineinführen, so kodieren sie genau den Bandinhalt nach zwei Schritten.
- Betrachtet man die Leitungen, die in die $p(n)$-*te* Schritt hineinführen, so kodieren sie genau den Bandinhalt der Endkonfiguration.
- Schaltet man also noch einen winzigen Schaltkreis nach, der überprüft, ob diese Endkonfiguration akzeptierend ist, so leistet der Schaltkreis das Gewünschte:
  Er wertet zu 1 aus genau dann, wenn die Maschine die Eingabe akzeptiert.

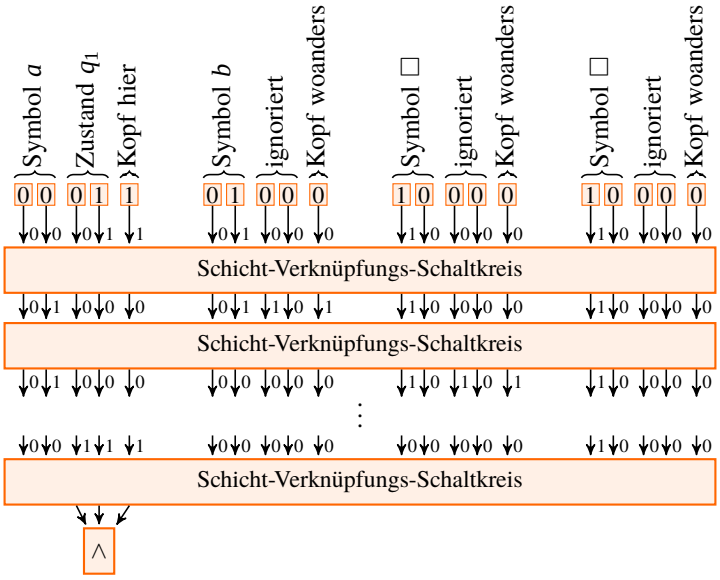
17-27

Eine Beispielsituation.

- Die Symbole des Bandalphabets $\Gamma = \{a, b, \square\}$ werden kodiert durch 00, 01, 10.
- Es gibt vier Zustände $\{q_0, q_1, q_2, q_3\}$, kodiert durch 00, 01, 10 und 11.
- Der initiale Bandinhalt ist $ab\square\square$, der initiale Zustand ist q_1 und der Kopf ist initial auf erstem Zeichen (dem a).
- Nach einem Schritt ist der Bandinhalt $bb\square\square$, der Zustand ist q_2 und der Kopf auf dem zweiten Zeichen.
- Nach zwei Schritten ist der Bandinhalt $ba\square\square$, der Zustand ist q_2 und der Kopf auf dem dritten Zeichen.
- Nach $p(n)-1$ Schritten ist der Bandinhalt $aaa\square$, der Zustand ist q_3 und der Kopf auf dem ersten Zeichen.
- Akzeptierender Zustand ist genau q_3.

17-28

Visualisierung der Schichtenstruktur des Schaltkreises.



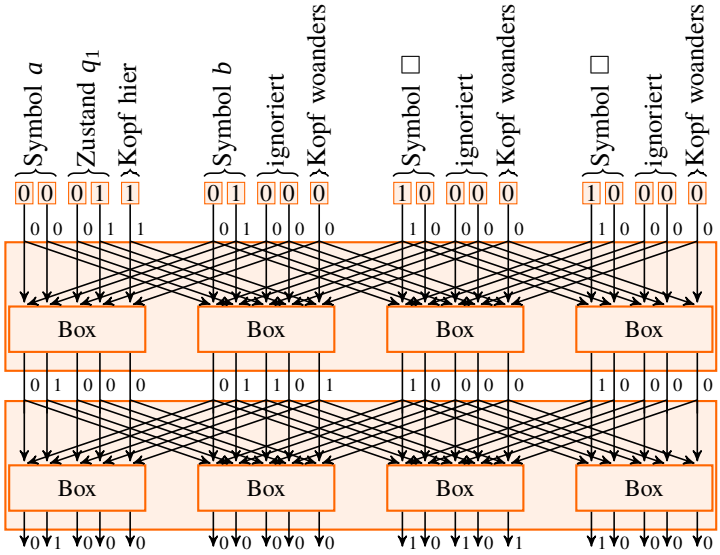
Was in den Schicht-Verknüpfungs-Schaltkreisen passiert.

17-29

- Wir wollen einen (flachen) Schaltkreis angeben, der *Schicht i mit Schicht $i+1$ verknüpft*.
- Dieser Schaltkreis besteht aus $p(n)$ vielen *Boxen*.
- Jede Box kümmert sich darum, den Bandinhalt einer bestimmten Zelle z zu berechnen.
- Dazu muss sie lediglich *die Inhalte der Zelle z in der vorherigen Schicht kennen, sowie die Inhalte der Zelle links und rechts davon*.

Aufbau eines Schicht-Verknüpfungs-Schaltkreises.

17-30



Wie schwierig ist der Schaltkreis zu bauen?

17-31

Was wir erreicht haben:

- Zu jeder beliebigen Eingabe x können wir einen Schaltkreis konstruieren, der genau dann zu 1 auswertet, wenn M das Wort x akzeptiert.

Was fehlt:

- Wie schwierig ist es, den Schaltkreis zu konstruieren?

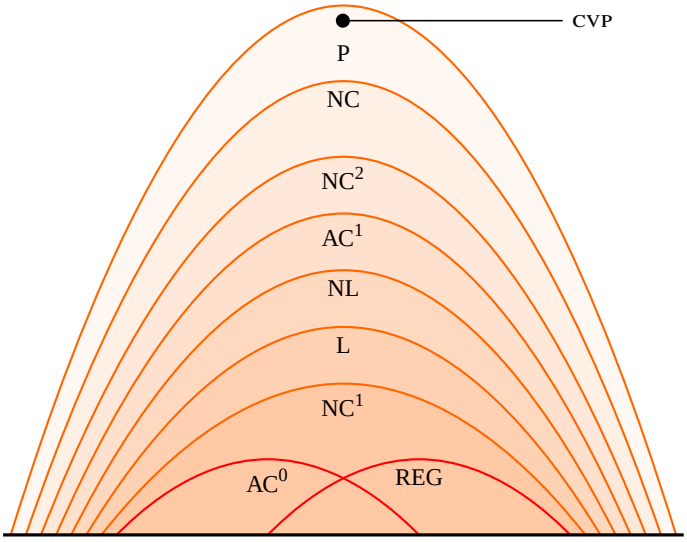
Antwort hierauf:

- Dies ist einfach, da *alle Boxen identisch sind*.
- Der Schaltkreis besteht im Wesentlichen aus $p(n)^2$ Boxen, die in einfacher Weise verdrahtet sind.
- Es ist leicht, in logarithmischem Platz in einer Schleife $p(n)^2$ Boxen (fester Größe!) auszugeben.
- Die Verdrahtung ist etwas fummeliger, aber auch nicht schwierig.

17.2.4 Folgerungen für Schaltkreisklassen

17-32

Visualisierung der Inklusionsstruktur der Schaltkreisklassen.



Zusammenfassung dieses Kapitels

17-33

1. Das P-NC-*Problem* ist die Frage, ob sich alle Probleme in P gut parallelisieren lassen.
2. Dieses Problem ist, wie das P-NP-Problem, *ungelöst*.
3. *P-vollständige Probleme* sind die schwierigsten Probleme in P und Kandidaten für Probleme, die sich nicht gut parallelisieren lassen.
4. Das Circuit-Value-Problem ist P-vollständig.

Übungen zu diesem Kapitel

Übung 17.1 (MON-CVP und P-Vollständigkeit, schwer)

Das Problem MON-CVP ist wie CVP definiert, nur dass im Schaltkreis keine Negationsgatter vorkommen. Zeigen Sie, dass MON-CVP ein P-vollständiges Problem unter Logspace-Many-One-Reduktion ist.

Tipp: Führen Sie eine Reduktion von CVP durch. Verdoppeln Sie dazu den Schaltkreis und verdrahten Sie ihn »geschickt«.

Übung 17.2 (P-Vollständigkeitsbeweis führen, leicht)

Zeigen Sie, dass das Problem NAND-CVP ein P-vollständiges Problem ist. Das NAND-CVP-Problem ist genauso definiert wie das CVP-Problem, nur gibt es statt Und-, Oder- und Nicht-Gattern lediglich NAND-Gatter.

Übung 17.3 (Komplexität des Zusammenhangsproblems, schwer)

Das *Zusammenhangsproblem* ist die Frage, ob es in einem gerichteten Graphen von jedem Knoten zu jedem Knoten einen Pfad gibt. Dieses Problem ist vollständig für eine der Klassen L, NL, NC, P oder NP. Finden Sie heraus für welche und beweisen Sie dies.

18-1

Kapitel 18
Untere Schranken II
Comparison-PRAMS und Adversary-Argumente

18-2

Lernziele dieses Kapitels

1. Konzepte des Vergleichsbaums und des Adversary-Arguments kennen
2. Untere-Schranken-Beweise für Comparison-PRAMS kennen

Inhalte dieses Kapitels

Worum es heute geht

Wenn Sie einen kommerziell erfolgreichen Film drehen wollen, brauchen sie (a) eine Love-Story (heteronormativ und -stereotyp mit heroischem maskulinen Part und erotischem femininen Part), (b) einen möglichst fiesen Bösewicht und natürlich (c) eine Special-Effects-Studio samt Renderfarm. Uns soll in diesem Kapitel lediglich Teil b interessieren: Der böse Widersacher, der dem Helden gerne die Schau stiehlt (der Bösewicht ist fast immer männlich, es sei denn er ist weiblich oder – wie »das Böse« im Fünften Element – einfach so böse, dass keine Geschlecht mehr zugeordnet werden kann). Das Techtelmechtel von Neo und Trinity wäre ohne Mr. Smith nicht der Rede wert. Luke würde ohne seinen Vater wahrscheinlich Saufkumpane von Herrn Hutt sein. Mr. Data wüsste ohne die Borg-Königin heute noch nicht wie sich Wind anfühlt, der zart über die Haut streicht. James würde ohne Goldfinger die Hotelbar nicht mehr verlassen, Hänsel und Grete würden ohne Frau Diplom-Arkanologin B. Hexe als schwererziehbare Problemkinder durch die Talkshows geistern. Und Shrek würde ohne Prinz Charming friedlich in seinem Sumpf leben anstatt sich durch eine dritte Fortsetzung quälen zu müssen.

In der Theoretischen Informatik heißt der böse Widersacher schlicht »der Adversary«. Wie es sich für einen Bösewicht gehört sollte man ihn eigentlich nicht mögen, ohne ihn geht es aber nicht. Zur Erinnerung: In einem Unteren-Schranken-Beweis geht es darum zu zeigen, dass kein Algorithmus ein bestimmtes Problem in einer bestimmten Zeit lösen kann. Der Adversary sieht seine Aufgabe gerade darin, es »jedem Algorithmus« möglichst schwer zu machen. Er wird immer gerade die Eingabe auswählen, die besonders viele Probleme bereitet. Er lässt den Algorithmus sogar anfänglich im Unklaren darüber, wie die Eingabe überhaupt genau lautet; er liefert immer »in letzter Minute« Informationen, die dann auch noch für den Algorithmus möglichst ungünstig ausfallen. Deshalb ist es der Adversary, der uns hilft, untere Schranke zu zeigen: Aufgrund seiner unermüdlichen Suche nach »fiesen Eingaben« werden wir beispielsweise zeigen können, dass sich das Maximum von n Zahlen (unter einigen Zusatzvoraussetzungen) nicht schneller als in Zeit $\Theta(\log\log n)$ berechnen lässt – was übrigens auch unsere obere Schranke für dieses Problem war.

18.1 Wiederholung: Laufzeitschranken

18-4

Laufzeitschranken für PRAMs.

- Für bestimmte Arten von PRAMs und bestimmte Problem lassen sich *untere Schranken für die Laufzeit* beweisen.
- Wir werden zeigen, dass *Comparison*-PRAMs mit p Prozessoren zum Finden eines Elementes in einer *sortierten Liste der Länge* n mindestens $\Omega(\log n / \log p)$ Schritte benötigt.
- Wir werden zeigen, dass *Comparison*-PRAMs Maxima bestenfalls in Zeit $\Omega(\log\log n)$ berechnen können.

18-5

»Starke« untere Schranken für die Laufzeit von PRAMs.

- Die Resultate für Comparison-PRAMs sind *starke* Resultate – sie gelten absolut und ohne Annahmen.
- Wir beweisen sie durch *Adversary-Argumente*.
- Wir »erkaufen« die starken Resultate dadurch, dass unser Modell (die Comparison-PRAM) nicht sonderlich mächtig ist.

18.2 Untere Laufzeitschranken

18-6

Laufzeitschranken für PRAMs.

- Für bestimmte Arten von PRAMs und bestimmte Probleme lassen sich *untere Schranken für die Laufzeit* beweisen.
- Die Resultate für Comparison-PRAMs sind *starke* Resultate – sie gelten absolut und ohne Annahmen.
- Wir beweisen sie durch *Adversary-Argumente*.
- Wir »erkaufen« die starken Resultate dadurch, dass wir sie nur für *Comparison*-PRAMs beweisen.

18.2.1 Das Modell: Comparison-PRAMs

18-7

Das Modell: Die Comparison-PRAM.

- Es hat *noch niemand geschafft*, (nicht triviale) untere Schranke für PRAMs zu beweisen.
- Wir betrachten deshalb ein *schwächeres* Modell von PRAMs: Die *Comparison*-PRAM.
- Die Comparison-PRAM funktioniert *ähnlich einem Vergleichsnetzwerk*.

Definition (Comparison-PRAM)
Eine *Comparison*-PRAM greift auf die Eingabespeicherzellen nur auf folgende Weise zu:

- Sie kann ein zwei Eingabezellen vergleichen.
- Sie kann eine Eingabezelle mit einem gegebenen Wert vergleichen.
- Sie erfährt, wie der Vergleich ausgegangen ist.
- Anderweitig darf sie *nicht* in die Zellen »hineinschauen«.

Skript

Für an Formalisierungen Interessierte sei hier im Skript noch eine genauere formale Definition der Comparison-PRAM angegeben:

Definition (Syntax der Comparison-PRAM)
Syntaktisch ist eine Comparison-PRAM eine normale PRAM mit einem zusätzlichen Befehl:

18. *if* GRRi $<$ GRRj *goto* k

Definition (Semantik der Comparison-PRAM)
Die Semantik einer Comparison-PRAM ist genauso definiert wie die Semantik einer PRAM mit folgenden Erweiterungen:

1. Die Semantik des Befehls *if* GRRi < GRRj *goto* k lautet wie erwartet: $\langle \text{PC} \rangle_{t+1}^{p} = k$, falls $\langle \text{GR} \langle \text{R}i \rangle_t^p \rangle_t < \langle \text{GR} \langle \text{R}j \rangle_t^p \rangle_t$, sonst $\langle \text{PC} \rangle_{t+1}^{p} = \langle \text{PC} \rangle_t^p + 1$.
2. Die Register GR1 bis GRn, wobei n die in GR0 gespeicherte Eingabelänge ist, können nur durch obigen Befehl adressiert werden. Eine Berechnung, bei der durch einen anderen Befehl auf eines dieser Register zugegriffen wird (lesend oder schreibend), endet sofort. Dies wäre zum Beispiel der Fall bei $n = 2$ und dem Befehl GR1 ← R1 oder bei $n = 6$ und dem Befehl R2 ← GRR8 mit $\langle \text{R8} \rangle_t^p = 5$.

18.2.2 Die Methoden: Decision-Trees

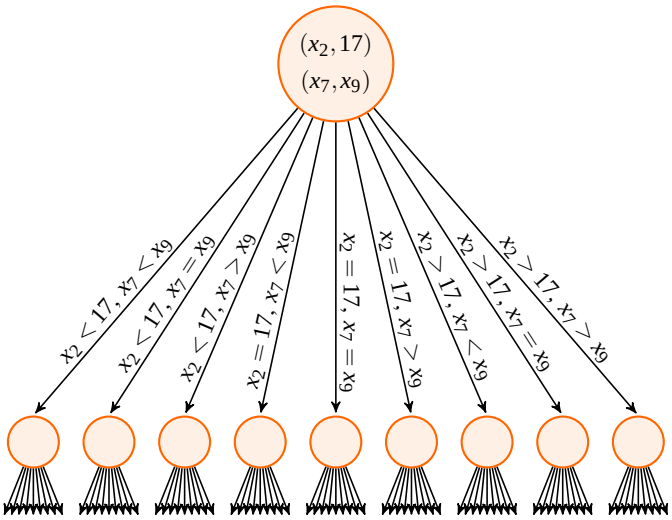
Unser zentrales Mittel zum Beweis unterer Schranken für Comparison-PRAMs. 18-8

- Bei *vergleichsbasierten* Modellen sind *Vergleichsbäume* oft nützlich.
- Vergleichsbäume werden beispielsweise bei dem Beweis verwendet, dass Sortieren Zeit $\Omega(n \log n)$ benötigt.

Definition (Paralleler Vergleichsbaum vom Grad d)

- Ein *paralleler Vergleichsbaum vom Grad d* ist ein Baum, in dem jeder innere Knoten (Nicht-Blatt-Knoten) genau 3^d Kinder hat.
  Dabei gibt es je ein Kind für jedes mögliches Ergebnis von d Vergleichsoperationen, bei denen als Ergebnis $<$, $=$ oder $>$ herauskommen kann.
- Die Blätter sind *mit Ausgaben gelabelt*.
- Das *Ergebnis* eines Baumes für eine Eingabe ist definiert als das Label des Blattes, das man beim Durchschreiten des Baumes »entsprechend der Eingabe« erreicht.

Beispiel eines Decision-Trees. 18-9



Das Verhältnis von Comparison-PRAMs und parallelen Vergleichsbäumen. 18-10

Beobachtung
Für jede Comparison-PRAM *mit p Prozessoren*, die *s Schritte rechnet*, gibt es einen parallelen Vergleichsbaum *vom Grad p* und der *Tiefe s*, der auf allen Eingaben das gleiche Ergebnis liefert.

Folgerung

1. *Kann ein Problem von einer Comparison-PRAM mit p Prozessoren in Zeit s gelöst werden, so gibt es auch einen parallelen Vergleichsbaum vom Grad p und Tiefe s für das Problem.*
2. *Hat jeder parallele Vergleichsbaum vom Grad p für ein Problem Tiefe mindestens s, so braucht eine Comparison-PRAM bei p Prozessoren mindestens Zeit s.*

18.2.3 Die Methoden: Adversary-Argumente

18-11

Allheilmittel bei unteren Schranken: Adversary-Argumente.

- Es gibt ein *generelles Verfahren*, das bei unteren Schranken oft zum Einsatz kommt: Adversary-Argumente.
- Ein *Adversary* ist ein Gegenspieler, der versucht, einer Maschine das Leben möglichst schwer zu machen.
- Für Untere-Schranken-Beweise bedeutet das, dass der Adversary immer *gerade die Lösungen auswählt, die besonders lange brauchen*.

18.2.4 Die Resultate: Suchen

18-12

Untere Schranke für das Suchen.

Satz

*Suchen in sortierten Listen benötigt auf einer Comparison-*PRAM *mit p Prozessoren Zeit* $\Theta(\log n/\log p)$.

Beweis.

- Für die *obere Schranke* müssen wir einfach einen Algorithmus angeben. Grobe Idee: Teile die Liste in Blöcke der Größe n/p auf; jeder Prozessor überprüft für einen Block, ob der gesuchte Wert in seinem Block ist; genau ein Block gewinnt, dort wiederholt man die Suche rekursiv.
- Für die *untere Schranke* zeigen wir, dass jeder Vergleichsbaum vom Grad p mindestens Tiefe $\Omega(\log n/\log p)$ hat. □

18-13

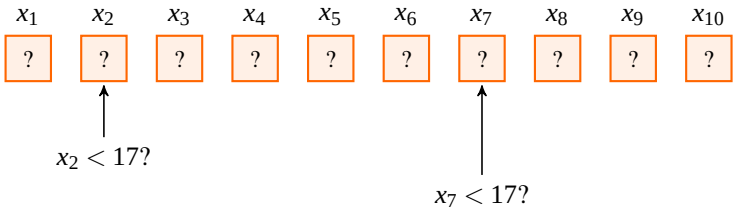
Vorgehen des Adversary.

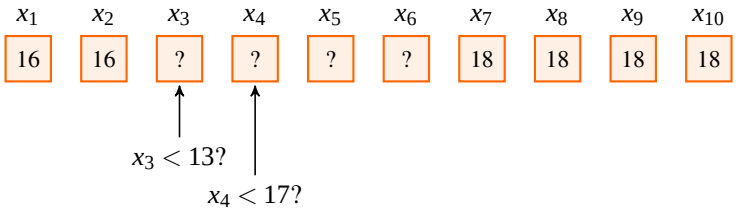
Der Adversary baut eine Eingabe der Länge n, so dass ein korrekter Vergleichsbaum wenigstens Tiefe $\log n/\log p$ haben muss:

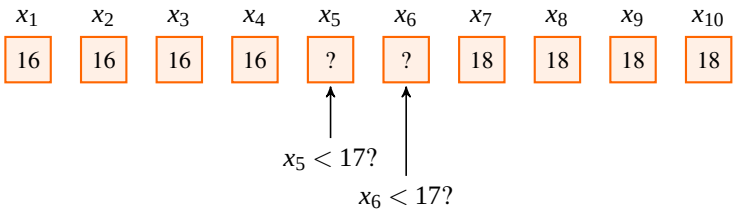
- Er betrachtet die Wurzel und die p Eingaben x_{i_1} bis x_{i_p}, die mit einer Zahl verglichen werden.
- Diese teilen die n Eingaben in Intervalle. Das größte von ihnen muss dann eine Größe von grob n/p haben (genauer $(n-p)/2$).
- Er legt die Eingabe so fest, dass der gesuchte Wert gerade in diesem Intervall landet. *Genauer legt er sich noch nicht fest.*
- Er betrachtet nun das Kind der Wurzel, das den Vergleichsergebnissen für das Intervall entspricht.
- Wieder wird dieses in p Intervalle geteilt, von denen eines mindestens Größe n/p^2 haben muss.
- Nun legt sich der Adversary weitere Werte fest, so dass der gesuchte Wert in diesem Intervall landet.
- Und so weiter.

18-14

Wo ist die 17?



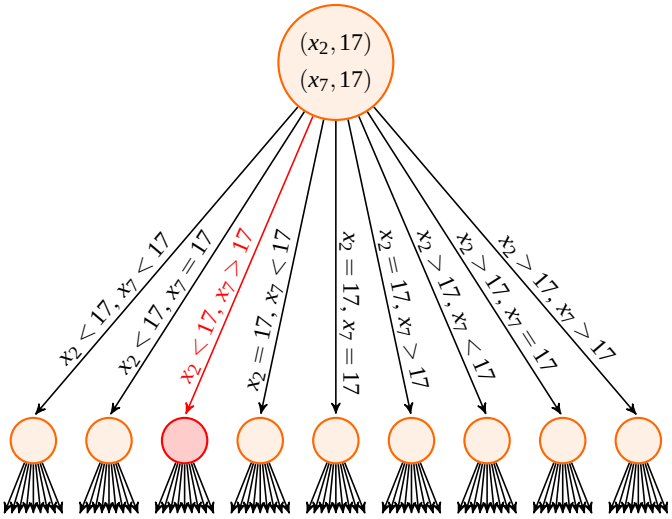




Beispielvorgehen des Adversarys.

18-15

Für $n = 10$ ist das größte Intervall zwischen den Indizes 2 und 7.



Zur Tiefe eines Vergleichsbaumes für die Suche.

18-16

Beobachtungen

1. So lange das größte Intervall, das der Adversary findet, noch Größe mindestens 2 hat, kann in dem Baum noch kein Blatt sein.
2. Für die Tiefe s des Baumes gilt also (grob) $n/p^d \leq 1$, beziehungsweise $\log n/\log p \leq d$.

18.2.5 Die Resultate: Maxima finden

Untere Schranke für Maxima.

18-17

Satz

*Das Finden von Maxima in unsortierten Listen benötigt auf einer Comparison-*PRAM *mit n Prozessoren Zeit $\Theta(\log\log n)$.*

Beweis.

- Für die *obere Schranke* sei an den Algorithmus mit den doppelt-logarithmischen Bäumen erinnert.
- Für die *untere Schranke* benutzen wir wieder ein Adversary-Argument. □

18-18

Vorgehen des Adversarys.

Hauptziel des Adversarys

- Der Adversary versucht zu *verheimlichen, welches Element das Maximum ist*.
- Dazu versucht er, in jeder Runde r eine möglichst große Menge M_r von Knoten zu finden, die alle folgende Eigenschaft haben: *Keiner der bisherigen Vergleiche kann ausschließen, dass dieser Knoten das Maximum ist.*
- Solange M_r noch mindestens zwei Elemente enthält, kann der Algorithmus noch nicht fertig sein.

Aktionen des Adversarys in Runde r

1. Identifikation der Menge M_r.
2. Für alle Listenelemente *außerhalb* M_r, für er sich *noch nicht festgelegt* hat, legt sich der Adversary auf den Wert r fest.

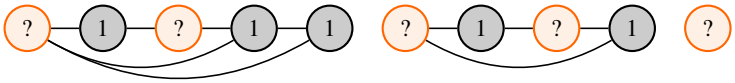
18-19

Identifikation der Menge M_r.

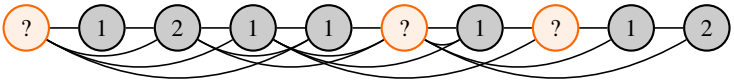
- Der Adversary baut einen Graphen, dessen Knotenmenge gerade die Listenelement in M_{r-1} sind (wobei M_0 alle Listenelemente sind).
- Er zieht eine Kante in dem Graphen, wenn zwei Elemente verglichen wurden.
- Nach dem Satz von Turan (Graphen mit n Knoten und m Kanten enthalten $n^2/(2m+n)$ unabhängige Knoten) gibt es eine unabhängige Menge der Größe $|M_{r-1}|^2/(2n+|M_{r-1}|)$.
- Als M_r wählt er diese Menge.

Die Menge M_0

Die Menge M_1



Die Menge M_2



18-20

Analyse der Größe der unabhängigen Menge.

Die Rekursionsformel lautet $|M_r| \geq |M_{r-1}|^2/(2n+|M_{r-1}|) \geq |M_{r-1}|^2/(3n)$. Also:

- $|M_0| = n$
- $|M_1| \geq n^2 \frac{1}{3n} = n/3$
- $|M_2| \geq \frac{n^2}{3^2} \frac{1}{3n} = n/3^3$
- $|M_3| \geq \frac{n^2}{3^6} \frac{1}{3n} = n/3^7$
- $|M_4| \geq \frac{n^2}{3^{14}} \frac{1}{3n} = n/3^{15}$
- $|M_r| \geq n/3^{2^r-1}$

Wir erreichen also $|M_r| = 1$ erst für $r \in \Omega(\log\log n)$.

Zusammenfassung dieses Kapitels

18-21

1. *Suchen in sortierten Listen* mittels Comparison-PRAMs dauert $\Theta(\log n/\log p)$.
2. *Maximum-Finden in unsortierten Listen* mittels n-Prozessor Comparison-PRAMs dauert $\Theta(\log\log n)$.
3. Dies beweist man mittels *Adversary-Argumenten*.

Teil IV
Verteilte Algorithmen

Die Wörter »Rechner« und »Computer« legen nahe, dass die grauen (oder gerne auch andersfarbigen) Kisten leicht introvertiert in ihrem Rack sitzen und eben rechnen. In Wirklichkeit sind Computer aber geschwätzige soziale Wesen, die gerne ausführlich miteinander plauschen. Wenn dabei auch noch etwas Sinnvolles berechnet wird, wie schön (ein unbeteiligte Beobachter wird sich des Eindrucks nicht erwehren können, dass dies nicht immer das Hauptziel eines Rechnenclusters ist, sondern dass dieser eher tamagotchihaft mit seinem Administrator über seine Probleme redet). Verteilte Systeme modellieren den »sozialen Charakter« der Kommunikation von Computern. Anders als bei den parallelen Systemen geht es hierbei weniger darum, etwas konkretes auszurechnen, sondern eher darum, das soziale Miteinander der Rechner zu regeln.

Um dies zu verdeutlichen, lassen Sie uns zum Beispiel aus der Einleitung zu parallelen Systemen zurückkehren: Dem Studenten, der noch eine Stunde Zeit hat, ein Übungsblatt zu lösen, und dafür Hilfe benötigt. Betrachtet man dieses Szenario als paralleles System, so ist wichtig, wie die Aufgaben gestellt sind und ob man sie gut in mehrere Teilaufgaben aufteilen kann. Betrachtet man das Szenario als verteiltes System, so geht es darum, wie man überhaupt Leute zum Arbeiten motivieren kann, wie man dafür sorgt, dass Dinge nicht doppelt gemacht werden, wie man dafür sorgt, dass die Lösungen auch auf einem Papier in einer Handschrift zusammengeführt werden, und wie man dafür sorgt, dass am Ende tatsächlich genau eine Lösung abgegeben wird. All diese Fragen lassen sich recht unabhängig vom konkreten Übungsblatt beantworten.

Bei verteilten Algorithmen geht es in aller Regel nicht darum, tatsächlich eine Berechnungsaufgabe zu lösen. Vielmehr dienen verteilte Algorithmen dazu, überhaupt erst die »soziale Infrastruktur« für die eigentlichen Berechnungen zu schaffen.

Eines der Probleme, für das wir mehrere verteilte Algorithmen genauer betrachten werden, ist gleichzeitig ebenso schlicht wie vertrackt: die Koordinatorwahl (englisch *leader election*). Eine Gruppe von Computern, die sich mehr oder weniger spontan zusammengefunden haben, sollen einen der ihren auswählen. Hört sich einfach an, ist es aber nicht; zumindest dann nicht, wenn beispielsweise die Kommunikation nicht einhundertprozentig verlässlich ist. Hierauf aufbauend werden wir weitere verteilte Probleme und deren Lösungen untersuchen. Dabei wird sich zeigen, dass auf den ersten Blick recht anders gelagerte Probleme wie Broadcasting und Konsensbildung genau so schwer sind wie die eben erwähnte Koordinatorwahl.

19-1

Kapitel 19
Verteilte versus parallele Systeme
Von Agenten und Schwärmen

19-2

Lernziele dieses Kapitels

1. Unterschiede zwischen parallelen und verteilten Systemen benennen können
2. Aufgaben für verteilte Systeme kennen
3. Ressourcen für und Eigenschaften von verteilten Systeme kennen
4. Einfachen Leader-Election-Algorithmus kennen

Inhalte dieses Kapitels

Worum es heute geht

Wie schon bei parallelen Systemen wird es auch bei verteilten Systemen unser Ziel sein, Algorithmen zur Lösung verschiedener *Probleme* zu finden. Und genau wie bei den parallelen Systemen müssen wir zunächst klären, welches *Modell* eines verteilten Systems wir nutzen wollen.

In diesem Kapitel werden wir das Modell kennen lernen, welches wir die meiste Zeit verwenden werden: nachrichtenbasierte Kommunikationsgraphen mit asynchroner, verlässlicher Kommunikation. Dieses Modell blendet zwei Eigenschaften realer verteilter Systeme aus: Erstens ist in realen Systemen die Topologie nicht fest, zweitens ist die Kommunikation nicht immer verlässlich. Jedoch können mit etwas gutem Willen beide Probleme durch »höhere Protokolle« gelöst werden: Nachrichten können mit Prüfsummen versehen werden, An- und Abmeldungen können registriert und »behandelt« werden. Der eigentliche Grund für die in unserem Modell gemachten Einschränkungen ist aber, dass ohne sie viele Probleme theoretisch nicht lösbar sind. Wir werden dies auch in der Vorlesung über die Unmöglichkeit von Konsens bei ausfallenden Prozessoren beweisen.

Das Grundmodell verteilter Systeme wird in diesem Kapitel eher auf einer »intuitiven« Ebene dargestellt, die saubere Formalisierung wird auf das besagte Kapitel über die Unmöglichkeit von Konsens vertagt.

19.1 Verteilte Systeme

19.1.1 Beispiele

Was sind verteilte Systeme?

19-4

Definition

- Ein *verteiltes System* besteht aus mehreren *Einheiten*.
  - Die Einheiten können *Rechner* sein, aber auch *Messstationen* oder *Mobiltelefone* oder *Roboter*.
  - Die Einheiten haben *in der Regel unterschiedliche Eigenschaften*.
  - Die Einheiten arbeiten *unabhängig voneinander* und mit *unterschiedlichen Geschwindigkeiten*.
- Die Einheiten *kommunizieren* miteinander.
  - Die Kommunikation erfolgt über *Funk* oder *ein Netzwerk* oder *Stromkabel*.
  - Die Kommunikation ist *störanfällig* und *kann ausfallen*.
  - Die Kommunikationstopologie kann *sich ändern*.
- Ein verteiltes System erfüllt *Aufgaben*. Beispiele sind:
  - Buchung einer Reise.
  - Suche nach Auffälligkeiten in astronomischen Daten (SETI@home).
  - Messen der Durchnässung eines Damms.

Was ist alles ein verteiltes System?

19-5

Zur Diskussion

Zu welchem Grad sind die folgenden System verteilte Systeme?

1. Taschenrechner
2. Modernes Auto
3. Waschmaschine
4. Mayflower Parallelrechner
5. Dual-Core Laptop

19.1.2 Unterschiede zu parallelen Systemen

Wichtige Unterschiede zwischen verteilten und parallelen Systemen.

19-6

Parallele Systeme

1. *Gleiche* Prozessoren.
2. *Zuverlässige* Prozessoren.
3. *Schnelle* Kommunikation.
4. *Zuverlässige* Kommunikation.
5. *Lösen Berechnungsprobleme*.

Verteilte Systeme

1. *Unterschiedliche* Einheiten.
2. *Unzuverlässige* Einheiten.
3. *Langsame* Kommunikation.
4. *Unzuverlässige* Kommunikation.
5. *Bearbeiten verschiedenartige Aufgaben*.

19.1.3 Unsicherheiten

19-7

Was kann alles schiefgehen?

- Bei verteilten Systemen gibt es sehr viele *Unsicherheiten*.
- Diese betreffen sowohl die *Rechner* wie auch die *Kommunikation*.

Zur Diskussion
Geben Sie Arten von »ungewöhnlichem Verhalten« an, die ein Rechner aufweisen kann. (Beispiel: Crash)

Zur Diskussion
Geben Sie Arten von »ungewöhnlichem Verhalten« an, die eine Nachrichtenübertragung aufweisen kann. (Beispiel: Verlust)

Zur Diskussion
Geben Sie Arten von »ungewöhnlichem Verhalten« an, die eine Gruppe von Nachrichtenübertragungen aufweisen kann. (Beispiel: Unterschiedliche Übertragungszeiten)

19.1.4 Modellierung

19-8

Modellierung von verteilten Systemen.
Festlegen der Eckdaten.

Festlegung von Eckdaten
Zur Modellierung von verteilten Systemen legt man zunächst *Eckdaten* fest. Dazu gehören:

- Eckdaten des Netzes
  - Ist die Topologie des Netzes fest?
  - Ist die Topologie des Netzes bekannt?
  - Wo liegen die Eingaben und die Daten?
- Eckdaten der Rechner
  - Ist die Menge der Rechner fest?
  - Welche Unsicherheiten bestehen bei den Rechnern?
  - Welche Rechenkraft haben die Rechner?
- Eckdaten der Kommunikation
  - Erfolgt die Nachrichtenübermittlung *synchron* oder *asynchron*?
  - Welche Unsicherheiten bestehen bei der Kommunikation?

19-9

Zur Übung
Wir betrachten ein Flugbuchungssystem als verteiltes System: Beteiligte Rechner sind Ihr Rechner, der Webserver und die Buchungsdatenbankserver.
Geben Sie die Eckdaten des Flugbuchungssystems als verteiltes System an.

Zur Übung
Wir betrachten Wikipedia als verteiltes System: Beteiligte Rechner sind die Server von Wikipedia sowie die Rechner der Nutzer.
Geben Sie die Eckdaten von Wikipedia als verteiltes System an.

19.1.5 Ressourcemaße

Ressourcen, die bei verteilten Systemen eine Rolle spielen. 19-10

Folgende Ressourcen sind bei verteilten Systemen besonders wichtig:

- *Zeit*
- *Kommunikationsmenge*, aufgeteilt in
  - Anzahl der versendeten Nachrichten
  - Gesamtlänge der versendeten Nachrichten
- *Fehlertoleranz* das Systems (wie viele Ausfälle/Angriffe überlebt es?)

Im *Gegensatz* zu gewöhnlichen Algorithmen sind auch noch folgende *Eigenschaften* wichtig:

- Kommt das System *immer zu einem Ergebnis*?
- Ist das System *fair*? (Kommt jeder mal dran?)
- Kommt das System *zu einem Ergebnis, wenn alle fair sind*?

19.2 Aufgaben für verteilte Systeme

19.2.1 Broadcasting

Allen etwas mitteilen. 19-11

Definition (Broadcasting-Problem)
Ein Rechner besitzt eine *Nachricht*. Er möchte sie *allen mitteilen*.

- Dieses Problem kommt oft in kleineren Netzen vor.
- Es kann bei sich ändernder Netztopologie schwierig sein.

19.2.2 Routing

Einem etwas mitteilen. 19-12

Definition (Routing-Problem)
Ein Rechner besitzt eine *Nachricht*. Er möchte sie *einem anderen Rechner mitteilen*.

- Dieses Problem kommt im Internet ständig vor, aber auch in allen anderen verteilten Systemen.
- TCP/IP ist ein gutes Protokoll hierfür, wofür aber *Routingtabellen* vorhanden sein müssen.
- In Netzen mit *unbekannter Netztopologie* ist Routing schwierig.

19.2.3 Synchronisation

19-13

Nur gemeinsam geht es weiter.

Definition (Synchronisationsroblem)
Zwei Rechner arbeiten an einem Problem. Bevor sie weiterarbeiten können, muss garantiert werden, dass *beide* mit einem bestimmten Teilproblem *fertig sind.*

- Dieses Problem kommt beim Multitasking ständig vor.
- Man löst es mit Mutex und Semaphoren.

19.2.4 Konsens

19-14

Einen Konsens finden.

Definition (Konsens-Problem)
Jeder Rechner hat einen *Vorschlag* zu einem Thema. Die Rechner sollen sich auf *einen der Vorschläge einigen*.

- Die Rechner dürfen sich nur auf einen Vorschlag einigen, den anfangs einer der Rechner hatte.
- Dieses Problem tritt beispielsweise bei einer Reisebuchung auf: Jeder Server hat einen Vorschlag, welche Buchung man gemeinsam bearbeiten könnte. Man muss sich auf einen einigen.

19.2.5 Leader-Election

19-15

Einen Anführer finden.

Definition (Das Leader-Election-Problem)
Unter allen Rechner soll *einer ausgewählt werden*.

- Das Problem ist trivial, wenn die Rechnermenge fest ist.
- Finden sich aber die Rechner »spontan zusammen«, so ist nicht klar, welchen Rechner man wählen sollte.
- Der gewählte Rechner, *Leader* genannt, kann im Folgenden Berechnung und anderes *koordinieren*.

19-16

Einfache Leader-Election in Ringen.
Die Problemstellung.

Problemstellung

- Wir wollen Leader-Election durchführen.
  - Die Rechner dürfen eine Zeitlang Nachrichten austauschen.
  - Am Ende muss sich *genau ein Rechner* zum Leader erklären.
  - *Alle anderen Rechner* müssen sich zu Gefolgsleuten erklären.
- Die Netztopologie ist ein *unidirektionaler Ring*.
- Die genau Zahl der Rechner ist *unbekannt*.
- Rechner und Kommunikation sind *absolut verlässlich*.
- Rechner und Kommunikation sind *deterministisch*.

19-17

Einfache Leader-Election in Ringen.
Ein Unmöglichkeitsresultat.

Satz
Leader-Election ist unmöglich, wenn alle Rechner im gleichen Zustand starten.

Beweis. Egal, welchen Algorithmus das System verwendet, gilt: Wenn die Kommunikation synchron abläuft, dann sind immer alle Rechner im gleichen Zustand. Würde sich ein Rechner zum Leader erklären, so auch gleichzeitig alle anderen. □

Wir benötigen offenbar *Symmetriebrechung*.

Einfache Leader-Election in Ringen.

19-18

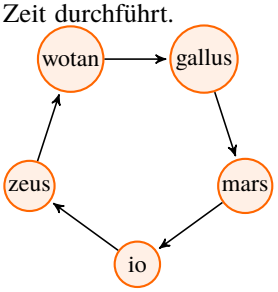
Ein einfacher Algorithmus.

Satz

In einem Ring mit n Prozessor verfüge jeder Prozessor über eine eindeutige Kennung. Dann ist Leader-Election möglich in Zeit $O(n)$ und mit $O(n^2)$ Nachrichten.

Zur Übung

Geben Sie einen verteilten Algorithmus an, der die Leader-Election in der angegebenen Zeit durchführt.



Zusammenfassung dieses Kapitels

19-19

1. *Verteilte Systeme* bestehen aus *unterschiedlichen Einheiten*, die *kommunizieren*.
2. Bei verteilten Systemen gibt es *neue Probleme* wie Ausfälle, Störungen, Topologieveränderungen, Deadlocks oder Unfairness.
3. *Verteilte Algorithmen* lösen Aufgaben wie Broadcast, Routing, Konsens oder Leader-Election.
4. *Leader-Election* ist möglich in Zeit $O(n)$ und mit Kommunikation $O(n^2)$.

Übungen zu diesem Kapitel

Bei allen Aufgaben zum Thema Verteilte Systeme sei angenommen – falls nichts anderes angegeben wurde –, dass alle Prozessoren und alle Kommunikation vollkommen ausfall- und verfälschungssicher ist. Die Kommunikation sei asynchron.

Übung 19.1 (Gittersumme, mittel)

Die Topologie eines Netzes sei ein Gitter. Jeder Prozessor auf dem Gitter hat eine Zahl gespeichert. Genau ein Prozessor initiiert eine Berechnung, und zwar will er die Summe aller Zahlen auf dem Gitter wissen. Geben Sie einen verteilten Algorithmus hierfür an. Die Prozessoren haben *keine* eindeutigen IDs, sie kennen aber ihren nördlichen, südlichen, östlichen und westlichen Nachbarn (falls vorhanden).

20-1

Kapitel 20
Das Konsensproblem
Verabredungen per E-Mail

20-2

Lernziele dieses Kapitels

1. Das Konsensproblem kennen
2. Einfache Konsensalgorithmen kennen
3. Unmöglichkeitsresultat für Absturz von bis zur Hälfte aller Prozesse beweisen können
4. Unmöglichkeitsresultat für Absturz von einem Prozess kennen

Inhalte dieses Kapitels

Worum es heute geht

Sich auf einen Kompromiss zu einigen ist nicht gerade eine Stärke von Menschen, insbesondere wenn einige Dickköpfe dabei sind. In diesem Kapitel – und noch stärker im nächsten – werden wir sehen, dass es Computern ähnlich geht, auch sie tun sich schwer damit, einen Konsens zu erreichen.

Dabei machen wir es Computern beim *Konsensproblem* schon recht einfach. Wir verlangen gar nicht, dass ein besonders guter Kompromiss herauskommt oder dass höfliche Umgangsformen eingehalten werden. Wir sind schon glücklich, wenn die drei Grundforderungen des Konsens erfüllt sind:

1. Man einigt sich (später Agreement-Bedingung genannt).
2. Man einigt sich auf etwas, was auch tatsächlich vorgeschlagen wurde (Validity-Bedingung).
3. Man wird irgendwann fertig (Termination-Bedingung).

Wenn man sich in einem Unigremium nicht einigen kann, dann ist es immer eine gute Idee, einen Ausschuss einzurichten. Dies hilft auch bei Computern: der Trick ist, dass der Ausschuss nur einen Computer enthält. Dieser wird sich schnell mit sich selbst einig und teilt dann das Ergebnis allen mit. Damit reduziert sich das Konsensproblem auf das Problem, einen Koordinator zu wählen. Für den einfachen Fall eines Ringes haben, wie schon gesehen wir dies geht, in späteren Kapiteln werden wir noch allgemeinere Algorithmen hierzu kennen lernen.

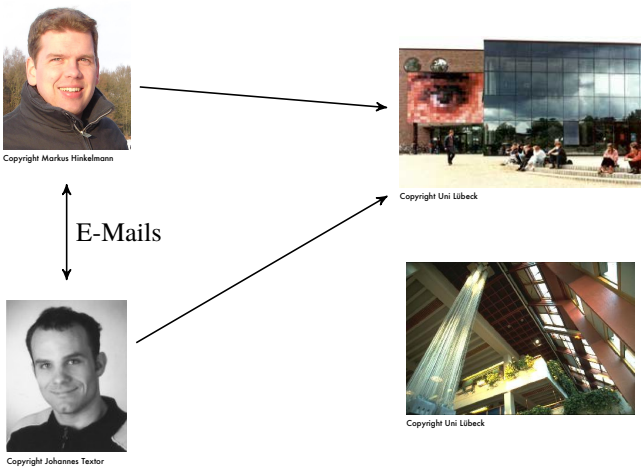
Eine andere Methode einen Konsens herbeizuführen ist, das gewünschte Konsensergebnis möglichst schnell »hinauszubrüllen«. Als Konsens wird dann das genommen, was als erstes gerufen wird. Diese (etwas unhöfliche) Methode liegt dem Konsens durch atomaren Broadcast zugrunde.

20.1 Einführung

20.1.1 Verabredung per E-Mail

Markus und Johannes wollen Mittagessen gehen. 20-4

- Markus und Johannes wollen Mittagessen gehen.
- Wie jeden Tag gehen sie entweder in das Personalcasino oder in die Mensa.



Wie können sich die beiden einigen? 20-5

Erstes Beispiel eines Kommunikationsprotokolls

- Markus schickt die Nachricht »Lass uns in die Mensa gehen.«
- Johannes bekommt diese Nachricht und schickt zurück »Ok.«

Zweites Beispiel eines Kommunikationsprotokolls

- Markus schickt die Nachricht »Lass uns in die Mensa gehen.«
- Johannes schickt gleichzeitig die Nachricht »Lass uns ins Casino gehen.«
- Markus antwortet »Nein, ich bleibe bei Mensa!«
- Johannes antwortet »Ok.«

Einwände gegen die Problemstellung 20-6

- Warum muss man sich überhaupt einigen?
  Die beiden sollen sich ja im selben Gebäude treffen.
- Warum geht man nicht einfach immer in die Mensa?
  Falls beide lieber in das Casino gehen wollen, dann sollen sie sich darauf einigen.
- Warum entscheidet nicht immer einfach Markus?
  Markus könnte auf einer Konferenz sein. Dann muss Johannes ohne ihn eine Entscheidung fällen.

Das Institut will Mittagessen gehen. 20-7

Problemstellung

- Alle Mitglieder des Instituts wollen Mittagessen gehen.
- Wie jeden Tag gehen sie entweder in das Personalcasino oder in die Mensa.
- Zur Verabredung benutzt man E-Mails mit Kopien an alle.

Widrigkeiten bei der Kommunikation.

- E-Mail ist asynchron, sie kann also in beliebiger Reihenfolge ankommen.
- Institutsmitglieder können *nicht anwesend sein*, weil sie auf einer Konferenz sind.
- Institutsmitglieder können *bei der Arbeit wegen Überlastung einschlafen*. Sie schicken ab diesem Zeitpunkt keine weiteren E-Mails mehr.

20.1.2 Das Konsensproblem

20-8

Das Konsensproblem

Problemstellung

In einem Netzwerk macht jeder Knoten einen *Vorschlag*. Nach einiger Kommunikation sollen sich alle Knoten auf einen der Vorschläge *geeinigt* haben.

Annahmen zum verteilten System

- Die Netztopologie ist ein *zusammenhängender Graph*.
- Die genaue Zahl der Rechner ist *unbekannt*, aber jeder Rechner hat eine *eindeutige ID*.
- Die Kommunikation ist *asynchron*, aber *absolut verlässlich*.
- Rechner und Kommunikation sind *deterministisch*.
- Maximal t Rechner können *abstürzen* (= einschlafen), der Graph muss aber zusammenhängend bleiben.

20-9

Was bedeutet »einigen« genau?

Wir stellen drei Anforderungen an einen Konsensalgorithmen.

Termination Jeder Prozess, der nicht abgestürzt ist, entscheidet sich irgendwann.

Agreement Alle Prozesse, die nicht abgestürzt sind, entscheiden sich für denselben Wert.

Validity Man entscheidet sich für einen der Vorschläge.

20-10

Zwei sehr einfache Aufgaben.

Zur Übung

Geben Sie einen Konsens-Algorithmus an, der Termination und Agreement garantiert, aber nicht unbedingt Validity.

Zur Übung

Geben Sie einen Konsens-Algorithmus an, der Termination und Validity garantiert, aber nicht unbedingt Agreement.

20-11

Ein Unmöglichkeitsresultat.

- Johannes und Markus möchten sich per E-Mail zum Essen verabreden, also asynchron.
- Einer von ihnen kann während der Kommunikation einschlafen.
- Es ist nun ein Protokoll gesucht, dass Termination, Agreement und Validity erfüllt:
  - Falls einer einschläft, soll der andere in endlicher Zeit das entscheiden, was er vorgeschlagen hat.
  - Falls keiner einschläft, sollen sich beide in endlicher Zeit auf einen ihrer beiden Vorschläge einigen.

Zur Übung

Argumentieren Sie, dass es ein solches Protokoll nicht geben kann.

20.2 Konsensalgorithmen

20.2.1 Konsens durch Leader-Election

Konsens und Leader-Election. 20-12

Konsens durch Leader-Election

1. Führe Leader-Election durch.
2. Der Leader entscheidet sich für seinen Vorschlag und teilt dies allen mit.

Leader-Election durch Konsens

1. Jeder Knoten schlägt sich selbst vor.
2. Der Konsens ist der Leader.

20.2.2 Konsens durch atomares Broadcasting

Konsens durch atomares Broadcasting. 20-13

Vorschau

Atomares Broadcasting bedeutet, dass alle (nicht abgestürzten) Prozesse alle Nachrichten in derselben Reihenfolge zustellen.

Konsens durch atomares Broadcasting

1. Jeder Knoten schickt seinen Vorschlag per Atomic-Broadcast an alle (inklusive sich selbst).
2. Jeder Knoten entscheidet sich für den ersten Vorschlag, den er erhält.

Der Algorithmus erfüllt Termination, Validity und Agreement.

20.2.3 Drei-Parteien-Konsens

Zurück zu Konsens. 20-14

- Wir haben schon gesehen: Konsens bei zwei Knoten, von denen einer abstürzen kann, ist nicht möglich.
- Wie steht es mit drei Knoten, von deinen einer abstürzen kann?

Versuch eines Algorithmus. 20-15

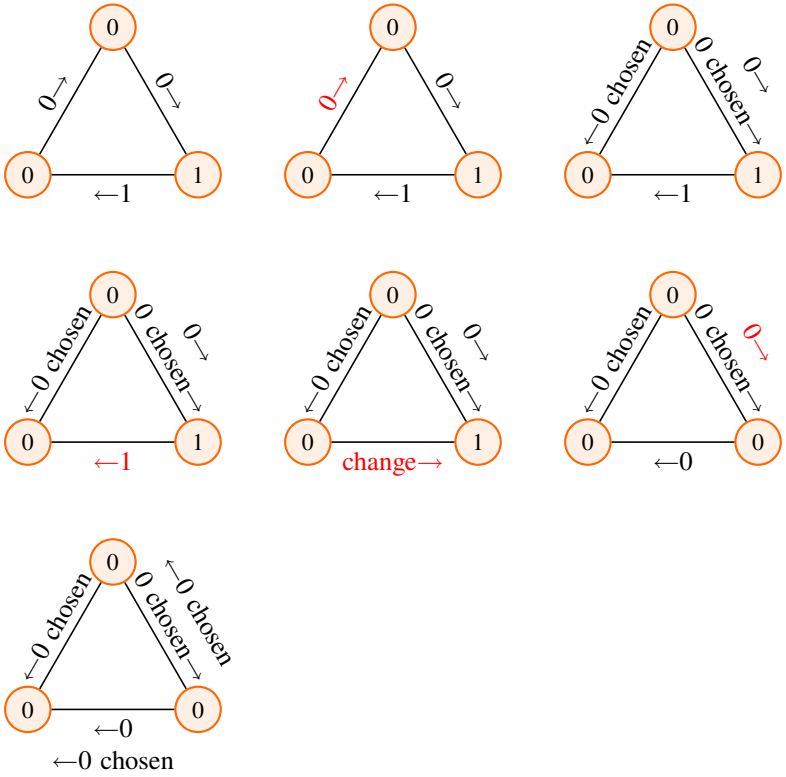
- Wir betrachten drei Knoten, in einem bidirektionalen Ring angeordnet.
- Wir betrachten binären Konsens (es gibt nur die Vorschläge 0 und 1).
- Die Kommunikation ist asynchron und fifo.

Algorithmus

- Jeder Knoten schickt seinen Wert im Ring im Uhrzeigersinn.
- Erhält ein Knoten einen Wert, der gleich seinem eigenen ist, so *entscheidet* er sich für diesen Wert und teilt dies den anderen mit, die sich dann ebenso entscheiden.
- Anderenfalls schickt er eine *Change-Nachricht* rückwärts.
- Erhält ein Knoten eine Change-Nachricht, so ändert er seinen Wert und beginnt wieder von vorne.

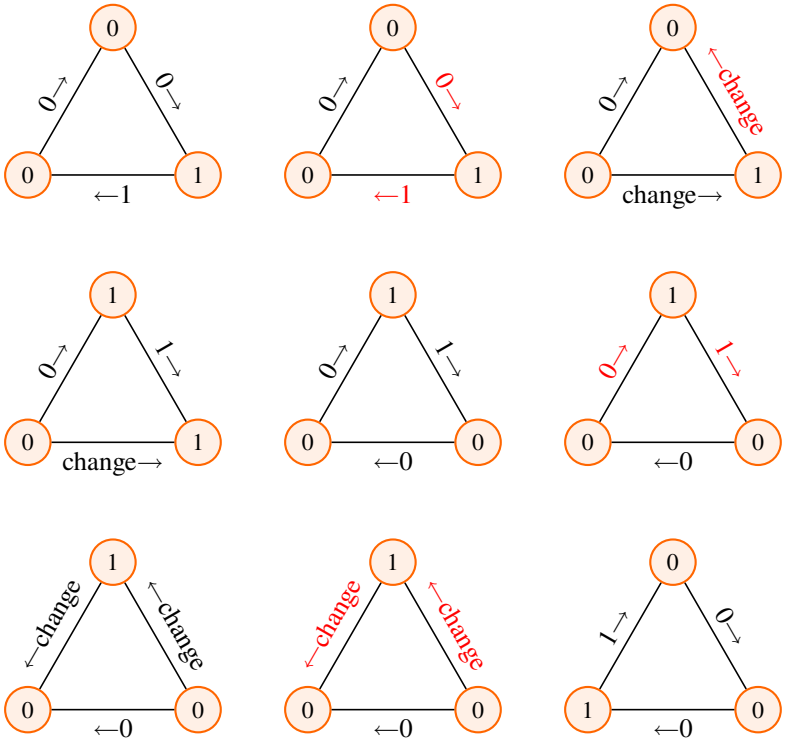
20-16

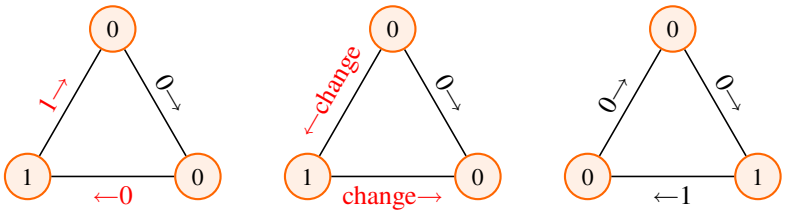
Ablauf einer Konsensbildung.



20-17

Keine Konsensbildung.





Grundeigenschaften des Algorithmus
20-18

Validity

Lemma
Der Algorithmus erfüllt Validity.

Beweis. Gibt es nur einen Vorschlag, so wird dieser sofort von allen Prozessoren angenommen. □

Grundeigenschaften des Algorithmus
20-19

Agreement

Lemma
Der Algorithmus erfüllt Agreement.

Beweis. Nehmen wir an, zwei Prozessoren entscheiden sich gleichzeitig unterschiedlich. Beispielsweise entscheidet sich ein Prozessor für 0 und sein Nachfolger für 1 (der umgekehrte Fall ist symmetrisch).

Der Nachfolger entscheidet sich für 1, weil er gerade ein 1-Token bekommen hat. Dieses ist also als letztes von dem Knoten abgeschickt worden, der sich gerade für 0 entschieden hat. Wenn dieser sich für 0 entscheidet, muss er aber den Wert 0 gespeichert haben und diesen als letztes geschickt haben. □

Grundeigenschaften des Algorithmus
20-20

Keine Termination, aber fast

Zur Übung
Argumentieren Sie, dass der Algorithmus Termination erfüllt, wenn ein Knoten garantiert abstürzt (!).

20.3 Unmöglichkeit von Konsens

20.3.1 Einfaches Unmöglichkeitsresultat

20-21 **Ein einfaches Unmöglichkeitsresultat.**

Satz

Konsens ist asynchron unmöglich, wenn die Hälfte oder mehr Knoten abstürzen können.

20-22 **Ein anschaulicher Widerspruchsbeweis.**

1. Nehmen wir an, Konsens wäre doch möglich.
2. Knoten nennen wir »Institutsmitgliedern«.
3. Wir teilen die Institutsmitgliedern in zwei gleichgroße Gruppen auf (bei ungerader Anzahl hat eine Gruppe ein Mitglied mehr).
4. Die Mitglieder in der einen Gruppe schlagen vor, in die Mensa zu gehen. Die Mitglieder der anderen Gruppe, in das Casino zu gehen.
5. Falls gerade alle Institutsmitglieder, die in das Casino wollten, einschlafen, so müssen sich die Mitglieder der anderen Gruppe darauf einigen, in die Mensa zu gehen.
6. Falls umgekehrt alle Institutsmitglieder, die in die Mensa wollten, einschlafen, so müssen sich die Mitglieder der anderen Gruppe darauf einigen, in das Casino zu gehen.
7. Beide Einigungen finden statt, *ohne dass man eine einzige E-Mail von der anderen Gruppe bekommen hat – die schlafen ja anscheinend.*
8. Falls die E-Mails aber nur *sehr langsam* zugestellt werden, kämen beide Gruppen auch zu ihren Einigungen, so dass Agreement verletzt wäre.

20.3.2 Allgemeines Unmöglichkeitsresultat

20-23 **Ein schwierigeres Unmöglichkeitsresultat.**

Satz

Konsens ist asynchron unmöglich, wenn ein Knoten abstürzen kann.

- Es war längere Zeit unbekannt, ob dieser Satz wirklich gilt.
- Der Beweis ist nicht einfach, aber durchaus machbar.
- Aus dem Satz folgt, dass insbesondere bei dem Drei-Knoten-Problem kein Algorithmus gefunden werden kann.

20.3.3 Folgerungen

20-24 **Was dies atomares Broadcasting und Leader-Election bedeutet.**

- Können wir atomares Broadcasting durchführen, so können wir auch Konsens erzeugen.
- Umkehrschluss:
  Falls Konsens in einem Netz unmöglich ist, so ist auch atomares Broadcasting nicht möglich.
- Also gilt:
  Atomares Broadcasting ist asynchron nicht möglich, falls ein Prozess abstürzen kann.
- Können wir Leader-Election durchführen, so können wir auf Konsens durchführen. Wieder gibt es den Umkehrschluss:
  Falls Konsens in einem Netz unmöglich ist, so ist auch Leader-Election nicht möglich.
- Also gilt:
  Leader-Election ist asynchron nicht möglich, falls ein Prozess abstürzen kann.

Zusammenfassung dieses Kapitels

20-25

1. Konsens ist das Problem, sich auf einen von mehreren vorgeschlagenen Werten zu einigen.
2. Konsens ist genauso schwierig wie Leader-Election und wie atomares Broadcasting.
3. Konsens ist *unmöglich*, wenn ein Prozess abstürzen kann.

21-1

Kapitel 21
Unmöglichkeit von Konsens
Warum man sich nicht per E-Mail verabreden kann

21-2

Lernziele dieses Kapitels

1. Eine formale Modellierung verteilter Systeme kennen
2. Beweis des Unmöglichkeitsresultat für Absturz von einem Prozess kennen

Inhalte dieses Kapitels

Worum es heute geht

Die Behauptung erscheint zunächst weder spektakulär noch sonderlich schwierig: Es gibt keinen Konsensalgorithmus für asynchrone Systeme, bei denen ein Prozessor »ausfallen« kann. Mit anderen Worten: Es soll für die restlichen Prozessoren unmöglich sein, sich zu einigen, wenn sie mit der Möglichkeit rechnen müssen, dass der verbleibende Prozessor ausfällt.

Auf den ersten Blick ist dies nicht schwierig zu zeigen: Da die Kommunikation asynchron verläuft, ist es unmöglich, zwischen »abgestürzt« und »Kommunikation dauert lange« zu unterscheiden. So lange sich als ein Prozessor nicht »meldet«, könnte er durch seine späteren Nachrichten ja »alles noch kaputt machen«.

Doch so einfach kann man nicht argumentieren: Nehmen wir ein System von 1000 Prozessoren, bei denen maximal einer abstürzt, und es gibt nur zwei mögliche Vorschläge: 0 oder 1. Eine Idee für ein Protokoll könnte sein: Alle Prozessoren schicken Ihren Vorschlag für einen Konsens an alle anderen Prozessoren. Da höchstens einer der Prozessoren abstürzt, warten alle Prozessoren, bis sie mindestens 999 Nachrichten empfangen haben. Wenn nun bei den Nachrichten zu 75% der gleiche Vorschlag eintrudelt, dann entscheidet sich der Prozessor für diesen. Wenn beide Vorschläge zu mindestens 25% vorkommen, dann entscheidet man sich für 0.

Ein allgemeiner Beweis, dass Konsens trotzdem nicht möglich ist, muss zeigen, dass das eben skizzierte Protokoll (doch) nicht funktioniert (sehen Sie, weshalb?) und ebenso, dass die (im wahrsten Sinne des Wortes) unendlich vielen anderen möglichen Protokolle auch versagen müssen. Es hat lange gedauert, nämlich bis 1985, bis Fischer, Lynch und Paterson einen solchen Beweis finden konnten.

21.1 Formalisierung eines verteilten Systems

21.1.1 Vorüberlegungen

21-4

Eine kleine Anekdote

Im Sommersemester 2006 lautete eine Übungsaufgaben wie folgt:

»Gegeben sei ein Ring mit n Prozessoren, in dem es genau einen Prozessor mit eindeutiger ID gibt und in dem alle anderen Prozessoren dieselbe ID haben. Zeigen Sie, dass es keinen Algorithmus für Leader-Election gibt, der nur eine Nachricht pro Prozessor versendet.«

- Jedoch stellte sich heraus, dass es doch einen solchen Algorithmus gibt.
- Der vermeintliche »Beweis« der Autoren der Übungsaufgabe war falsch.
- Genauer war er zu ungenau und ging von Voraussetzungen aus, die in subtiler Weise nicht gegeben waren.

21-5

Brauchen wir formale Modellen für verteilte Systeme?

- Bis jetzt haben wir formale Systeme eher »intuitiv« behandelt.
- Der hohe Nichtdeterminismus bei verteilten Systemen macht es aber schwer, sie intuitiv richtig einzuschätzen. Man »vergisst« häufig mögliche Abläufe.
- Detaillierte formale Modelle schaffen hier Abhilfe, sie verstellen aber schnell den Blick auf das Wesentliche.
- Heute nun exemplarisch eine *(recht) saubere Formalisierung* eines verteilten Systems.

21-6

Ein starkes oder ein schwaches Modell?

- Wir wollen ein Unmöglichkeitsresultat beweisen, also eine Aussage der Form: »In jedem System mit Eigenschaft XYZ gilt, dass es keinen Algorithmus zur Lösung des Konsensproblems gibt.«
- Damit unsere Aussage *möglichst »stark«* ist, machen wir
  1. möglichst viele Einschränkungen, die Konsens erleichtern, und
  2. möglichst wenige Einschränkungen, die Konsens erschweren.

  (Dann ist es erstaunlich, dass Konsens trotzdem unmöglich ist.)

21-7

Die Einschränkungen, die wir betrachten werden.

Einschränkungen, die Konsens (prinzipiell) erleichtern

1. Die Topologie ist eine Clique, alle können mit allen reden.
2. Wenn ein Prozessor ausfällt (Crash) muss dies »sauber« geschehen.
3. Die Kommunikation ist absolut verlässlich.
4. Die Prozessoren sind, bis auf den Crash, absolut verlässlich.
5. Wir betrachten nur binären Konsens.

Einschränkungen, die Konsens (prinzipiell) erschweren

1. Es darf ein Prozessor ausfallen, muss es aber nicht.
2. Die Kommunikation ist asynchron.

21.1.2 Prozesse und Nachrichten

Modellierung von Prozessoren und Nachrichten. 21-8

Definition (Prozessoren)
Es sei P eine endliche Mengen von Prozessoren.

Definition (Nachrichten)
Es sei T eine abzählbare Menge von Nachrichtentexten.
Eine *Nachricht* ist ein Paar $(t,p) \in T \times P$, bestehend aus einem Nachrichtentext und einem *Adressaten*. Weiterhin ist auch decide(v) eine Nachricht mit $v \in \{0,1\}$.
Die Menge aller möglichen Nachrichten bezeichnen wir mit M.

Modellierung von Zuständen. 21-9

Definition (Zustände und Zustandsüberführung)
Es sei Q eine abzählbare Menge von (Prozessor-)zuständen. Für jeden Prozessor $p \in P$ gibt es einen *initialen Zustand* $q_0^p \in Q$.
Weiter gibt es für jeden Prozessor eine *Zustandsüberführungsfunktion* δ^p, welche Paare $(q,t) \in Q \times T$ abbildet auf Paare (q',M'). Hierbei ist $q' \in Q$ der neue Zustand und $M' \subseteq M$ eine endliche Menge versendeter Nachrichten.

Definition (Algorithmus)
Ein *Algorithmus* ordnet jedem Prozessor einen initialen Zustand und eine Zustandsüberführungsfunktion zu.

21.1.3 Konfigurationen und Berechnungen

Modellierung von Konfigurationen. 21-10

Wie bei Turingmaschinen oder endlichen Automaten ist eine Konfiguration ein »Snapshot« des Systems zu einem bestimmten Zeitpunkt.

Definition (Konfiguration)
Eine *Konfiguration* c ist ein Paar $(\text{states}_c, \text{msgs}_c)$. Hierbei

- ordnet $\text{states}_c \colon P \to Q$ jedem Prozess seinen Zustand zu; weiter ist
- $\text{msgs}_c \colon M \to \mathbb{N}$ eine Funktion, die angibt, wie oft jede Nachricht gerade unterwegs sind.
  Alternativ kann man msgs_c auch als Multimenge auffassen und wir werden oft $m \in \text{msgs}_c$ schreiben statt $\text{msgs}_c(m) > 0$.

Eine Konfiguration heißt *initial*, wenn states_c jedem Zustand seinen Anfangszustand zuordnet und wenn die Nachrichten»menge« msgs_c für jeden Prozessor p genau eine der beiden Anfangsnachrichten $(\text{initial}(0),p)$ oder $(\text{initial}(1),p)$ enthält.

Modellierung von Schritten. 21-11

Die Konfiguration eines Systems verändert sich nur, wenn es eine Nachricht zustellt.

Definition (Schritte)
Ein *Schritt* (oder auch eine Systemveränderung) ist dasselbe wie eine Nachricht (da nur aufgrund von Nachrichten Systemveränderungen eintreten).

- Ein Schritt m heißt *prinzipiell möglich in der Konfiguration* c, wenn $m \in \text{msgs}_c$.
- Ist ein Schritt m prinzipiell möglich und wir er *genommen*, so entsteht die neue Konfiguration c' wie folgt:
  Sei $c = (\text{states}_c, \text{msgs}_c)$ und $m = (t,p)$ und $\delta^p(\text{states}_c(p),t) = (q',M')$. Dann ist $\text{states}_{c'}$ identisch zu c, außer dass $\text{states}_{c'}(p) = q'$. Weiter ist $\text{msgs}_{c'} = (\text{msgs}_c - \{m\}) \cup M'$.
  Wir schreiben dann $c \to_m c'$.

21-12

Modellierung von Berechnungen.

Definition (Berechnung)

- Eine *Berechnung* ist ein endliche oder unendliche Folge $(c_0, m_0, c_1, m_1, c_2, m_2, \dots)$, wobei für alle i (bis auf das letzte) gilt $c_i \to_{m_i} c_{i+1}$.
- Wenn es eine Berechnung gibt, die mit c_i anfängt und mit c_j endet, so schreiben wir $c_i \Longrightarrow c_j$ (auch für $c_i = c_j$). Dann heißt c_j *erreichbar von* c_i.
- Eine *wohlbegonnene Berechnung* ist eine Berechnung, die mit einer initialen Konfiguration beginnt.
- Eine Konfiguration heißt *erreichbar*, wenn sie in einer wohlbegonnenen Berechnung vorkommt.

21-13

Modellierung von Abstürzen und asynchroner, fairer Zustellung.

Definition (Korrekte Prozessoren)

Gegeben sei eine unendliche Berechnung $(c_0, m_0, c_1, m_1, \dots)$. Ein Prozessor p heißt *korrekt* oder auch *nicht abstürzend*, wenn folgendes gilt:

- Für jedes i für dass in c_i ein Schritt (t, p) prinzipiell möglich ist,
- macht p für ein geeignetes $j > i$ einen Schritt (nicht unbedingt (t, p)).

Definition (Faire Berechnung)

Gegeben sei wieder eine unendliche Berechnung $(c_0, m_0, c_1, m_1, \dots)$. Sie heißt *fair*, wenn für jeden nicht abstürzenden Prozessor gilt: Ist (t, p) prinzipiell möglich in einer Konfiguration c_i, so wird (t, p) zu irgendeinem Zeitpunkt $j \geq i$ auch genommen. (Keine Nachricht bleibt »ewig in der Warteschlange«.)

21-14

Zwischenzusammenfassung

Wir haben bis jetzt Folgendes modelliert:

- Das System ist verteilt, Prozessoren kommunizieren über Nachrichten.
- Die Kommunikation ist asynchron, aber jede verschickte Nachricht kommt irgendwann an.
- Prozessoren können abstürzen, was sich dadurch äußert, dass sie keine Nachrichten mehr verarbeiten oder abschicken.
- Am Anfang hat jeder Prozessor einen initialen Vorschlag und jeder Prozessor kann eine Entscheidung fällen, indem der eine decide-Nachricht verschickt.

Was wir noch nicht formalisiert haben:

- Das Konsensproblem.

21.1.4 Rauteneigenschaft

21-15

Modellierung unabhängiger Berechnungen.

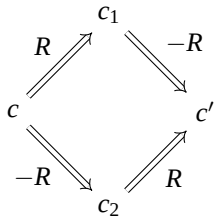
Definition

- Sei $R \subseteq P$. Wir schreiben $c \overset{R}{\Longrightarrow} c'$, falls $c \Longrightarrow c'$ gilt via einer Schrittfolge, in der nur Prozessoren in R Schritte machen.
- Wir schreiben $c \overset{-R}{\Longrightarrow} c'$, falls $c \overset{P-R}{\Longrightarrow} c'$ gilt.
- Wir schreiben $c \overset{p}{\Longrightarrow} c'$ statt $c \overset{\{p\}}{\Longrightarrow} c'$ und entsprechend $c \overset{-p}{\Longrightarrow} c'$ statt $c \overset{-\{p\}}{\Longrightarrow} c'$.

Das Rautenlemma. 21-16

Lemma
Falls $c \xRightarrow{R} c_1$ *und auch* $c \xRightarrow{-R} c_2$, *so existiert ein* c' *mit* $c_1 \xRightarrow{-R} c'$ *und* $c_2 \xRightarrow{R} c'$.

Warum das Lemma so heißt



Zur Übung
Argumentieren Sie (informell), weshalb das Rautenlemma korrekt ist.

21.2 Konsens

21.2.1 Formalisierung von Konsens

Modellierung von Konsens. 21-17

Definition
Ein Algorithmus *löst asynchronen 1-toleranten Konsens*, wenn für jede faire wohlbegonnene Berechnung mit höchstens einem ausgefallenen Prozessor die Eigenschaften Agreement, Validity und Termination gelten. Dies bedeutet:

1. Für Agreement muss mindestens einmal decide(v) verschickt werden und es dürfen nicht sowohl decide(0) als auch decide(1) verschickt werden.
2. Für Validity muss für die initiale Konfiguration, bei der alle Prozessoren die Nachricht initial(0) bekommen, die Nachricht decide(0) verschickt werden. Analog für 1 statt 0.
3. Für Termination muss die Berechnung endlich sein.

21.2.2 Pseudo-Konsens

Ein neues Konzept: Pseudo-Konsens. 21-18

Die *Idee* hinter Pseudo-Konsens ist, dass die Berechnung zwar nicht terminieren muss, es aber *jederzeit könnte*.

Definition (Pseudo-Konsens)
Ein Algorithmus *löst asynchronen 1-toleranten Pseudo-Konsens* unter denselben Bedingung wie bei asynchronem 1-toleranten Konsens, nur dass die Terminationsbedingung durch folgende Bedingung ersetzt ist:

3′. Für jede erreichbare Konfiguration c und jede Prozessormenge R mit $|R| = |P| - 1$ gilt $c \xRightarrow{R} c'$ für eine Konfiguration c', in der eine Decide-Nachricht verschickt wurde.

Offenbar ist Pseudo-Konsens immer möglich, wenn Konsens möglich ist.

21.2.3 Beweis der Unmöglichkeit

21-19

Der zu beweisende Satz und der Beweisplan.

Satz (Fischer, Lynch, Paterson, 1985)
Es gibt keinen Algorithmus, der asynchronen 1-toleranten Konsens löst.

Beweisplan nach Völzer, 2004.

- Wir führen einen Widerspruchsbeweis. Nehmen wir an, es gibt einen solchen Algorithmus. Dann ist dies auch ein 1-toleranter Pseudo-Konsens-Algorithmus.
- Im Folgenden zeigen wir, dass jeder 1-toleranter Pseudo-Konsens-Algorithmus eine unendliche Berechnung zulässt.
- Dazu führen wir das Konzept der *Uniformität* ein, einer *Vorbedingung* für Entscheidung.
- Wir zeigen dann, dass eine Berechnung am Anfang nicht uniform ist und dass man diesen Zustand ewig aufrecht erhalten kann. □

21-20

Stille Entscheidungen und Uniformität.

Definition
Sei c eine erreichbare Konfiguration und p ein Prozessor.

- c heißt *v-entschieden*, wenn $\text{decide}(v) \in \text{msgs}_c$.
- Ein Wert $v \in \{0,1\}$ heißt ein *p-stummer Entscheidungswert* von c, wenn ein v-entschiedenes c' existiert mit $c \overset{-p}{\Longrightarrow} c'$. (Das System kann sich für v entscheiden, wenn es von p nichts mehr hört.)
- Die Menge der p-stummen Entscheidungswerte von c bezeichnen wir mit decisions_c^{-p}.
- c heißt *v-uniform*, wenn $\text{decisions}_c^{-p} = \{v\}$ für alle $p \in P$ gilt. (Wenn ein Prozessor ausfällt, so entscheidet man sich auf jeden Fall für v.)

- Da wir 1-tolerant sind, muss gelten $\text{decisions}_c^{-p} \neq \emptyset$ für alle erreichbaren Konfigurationen c.
- Da Agreement gilt, ist jede v-entschiedene Konfiguration auch v-uniform.

21-21

Am Anfang sind wir nicht uniform.

Lemma (Anfangslemma)
Es gibt eine nicht-uniforme initiale Konfiguration.

Beweis. Sei $P = \{p_1, \ldots, p_n\}$ und sei c_i die Konfiguration in der Prozessoren p_1 bis p_{i-1} den Startnachrichtentext $\text{initial}(0)$ haben und die anderen $\text{initial}(1)$.

1. c_1 ist 0-uniform und c_{n+1} ist 1-uniform wegen der Validity-Bedingung.
2. Sei i minimal, so dass c_i nicht mehr 0-uniform ist.
3. Da c_{i-1} noch 0-uniform ist, gibt es eine 0-entschiedene Konfiguration c für die $c_{i-1} \overset{-p}{\Longrightarrow} c$.
4. Da c_i und c_{i-1} sich nur in Bezug auf p unterscheiden, gilt auch $c_i \overset{-p}{\Longrightarrow} c$.
5. Also ist $0 \in \text{decisions}_{c_i}^{-p}$ und somit ist c_i nicht 1-uniform. Da es auch nicht 0-uniform ist, ist es nicht uniform. □

21-22

Wir bleiben nicht uniform.

Lemma (Fortsetzungslemma)
Sei c nicht uniform und p ein Prozess. Dann gibt es ein c' mit $c \Longrightarrow c'$ und $\text{decisions}_{c'}^{-p} = \{0,1\}$.

Beweis.

- Falls $\text{decisions}_c^{-p} = \{0,1\}$, so leistet $c = c'$ das Gewünschte. Sei also $\text{decisions}_c^{-p} = \{0\}$ (der andere Fall ist symmetrisch). Dies bedeutet: *Hören wir von p nichts mehr, so entscheiden wir uns für* 0.

- Da c nicht uniform ist, muss es ein q geben, so dass $\text{decisions}_c^{-q} = \{1\}$. (Hören wir also von q nichts mehr, so entscheiden wir und für 1.)
- Insbesondere gibt es also eine 1-entschiedene Konfiguration c_1 mit $c \Longrightarrow c_1$ und so $\text{decisions}_{c_1}^{-p} = \{1\}$.
- Nach dem gleich noch zu beweisenden Standhaftigkeitslemma folgt daraus, dass es ein c' gibt mit $c \Longrightarrow c'$ und $\text{decisions}_{c'}^{-p} = \{0,1\}$. □

Wir wechseln nicht plötzlich unsere Meinung.

21-23

Lemma (Standhaftigkeitslemma)
Sei $c \to_m c'$ und sei q ein Prozessor. Falls $\text{decisions}_c^{-q} = \{0\}$, so gilt $\text{decisions}_{c'}^{-q} \neq \{1\}$.

Beweis. Sei $m = (t,p)$.

- Falls $p \neq q$, so gilt weiterhin $\text{decisions}_{c'}^{-q} = \{0\}$.
- Falls $p = q$, so gilt $0 \in \text{decisions}_{c'}^{-q}$. Dies überlegt man sich leicht direkt oder man benutzt das Rautenlemma. □

Hauptbeweis: Jeder 1-tolerante Pseudo-Konsens-Algorithmus erlaubt eine unendliche faire Berechnung.

21-24

1. Beginne mit einer nicht uniformen Anfangskonfiguration.
   Wir nennen diese die *aktuelle Konfiguration* c.
2. Wiederhole nun folgendes:
   - 2.1 Betrachte den am längsten schon aktivierten Schritt (t,p).
   - 2.2 Setze c nach dem Fortsetzungslemma zu c' fort mit $\text{decisions}_{c'}^{-p} = \{0,1\}$.
   - 2.3 Ist in c' die Nachricht (t,p) bereits zugestellt worden, so mache c' zur aktuellen Konfiguration.
   - 2.4 Sonst stelle die Nachricht (t,p) zu, was die neue aktuelle Konfiguration c'' ergibt.
   - 2.5 Dann gilt $\text{decisions}_{c''}^{-p} = \{0,1\}$, denn wie beim Standhaftigkeitslemma kann sich die Menge der Entscheidungen nicht verkleinern.

- Die obige Berechnung ist *fair*, denn jede Nachricht wird früher oder später zugestellt.
- Die Berechnung ist wohlbegonnen und unendlich.
- Es stürzt kein Prozessor ab.

Zusammenfassung dieses Kapitels

21-25

1. Die Formalisierung eines verteilten Systems ist relativ aufwändig.
2. Asynchroner 1-toleranter Konsens ist unmöglich.
3. Der Beweis erfolgt, indem man zeigt, dass jeder asynchrone 1-tolerante Pseudo-Konsens-Algorithmus einen unendliche Berechnung enthält.

22-1

Kapitel 22
Koordinatorwahl
Von der Schwierigkeit, einen Kanzler zu wählen

22-2

Lernziele dieses Kapitels

1. Das Koordinatorwahl-Problem kennen
2. Verteilte Algorithmen zur Koordinatorwahl in Ringen kennen

Inhalte dieses Kapitels

Worum es heute geht

Wenn mehrere Leute etwas gemeinsam arbeiten wollen (oder müssen), sollte diese Arbeit in irgendeiner Weise koordiniert werden. Die Person, die dies macht, nennt man Koordinator oder neudeutsch Manager. Da Computer auch nur Menschen sind, stellt sich bei verteilten System dasselbe Probleme: Bei vielen Problemen, die ein verteiltes System lösen muss, kommt man um die Benennung eines Koordinators nicht herum.

In diesem und dem nächsten Kapitel geht es um verteilte Algorithmen zur Bestimmung eines Koordinators. Beginnen werden wir mit einigen grundsätzlichen Überlegungen, wann eine Koordinatorwahl überhaupt möglich ist, dann schauen wir uns erst einen ganz einfachen Algorithmus an. Der Hirschberg-Sinclair-Algorithmus ist dann eine raffiniertere Version des einfachen Algorithmus; sein Hauptvorteil liegt darin, dass viele der beim einfachen Algorithmus unnötigerweise verschickte Nachrichten wegfallen.

In diesem Kapitel wird die Netztopologie immer ein Ring sein. Dies ist eine reichlich künstliche Annahme, auch wenn sie in der Realität durchaus manchmal erfüllt ist. Wesentlich befriedigender wird der Algorithmus im nächsten Kapitel sein, der für beliebige zusammenhänge Graphen funktioniert. Jedoch basieren die Ideen auch dieses Algorithmus auf den im Folgenden dargestellten Überlegungen, weshalb wir mit diesen beginnen werden.

22.1 Einführung

22.1.1 Problemstellung

Wiederholung: Einen Koordinator wählen.

22-4

Definition (Koordinatorwahl)

Unter allen Rechner soll *einer ausgewählt werden*.

- Das Problem ist trivial, wenn die Rechnermenge fest ist.
- Finden sich aber die Rechner »spontan zusammen«, so ist nicht klar, welchen Rechner man wählen sollte.
- Der gewählte Rechner, *Koordinator* oder *Leader* genannt, kann im Folgenden Berechnung und anderes *koordinieren*.

Koordinatorwahl in Ringen.

22-5

Erste Problemstellung.

Problemstellung

- Wir wollen Koordinatorwahl durchführen.
  - Die Rechner dürfen eine Zeitlang Nachrichten austauschen.
  - Am Ende muss sich *genau ein Rechner* zum Leader erklären.
  - *Alle anderen Rechner* müssen sich zu Gefolgsleuten erklären.
- Die Netztopologie ist ein *unidirektionaler Ring*.
- Die genau Zahl der Rechner ist *unbekannt*.
- Rechner und Kommunikation sind *absolut verlässlich*.
- Rechner und Kommunikation sind *deterministisch*.

Synchrone und asynchrone Kommunikation.

22-6

Verschiedene Stufen von Synchronität:

1. Bei *absolut synchroner Kommunikation* nehmen wir an, dass jede Nachricht in genau einem Zeitschritt zugestellt wird.
   Weiterhin wissen wir genau, wie viele Zeitschritte jede Berechnung dauert.
2. Bei *fast synchroner Kommunikation* nehmen wir an, dass jede Nachricht in einem Zeitschritt plus oder minus einem kleinen ε zugestellt wird.
3. Bei *Kommunikation mit maximaler Zustelldauer* nehmen wir an, dass jede Nachricht innerhalb einer maximalen Anzahl a von Zeitschritten zugestellt wird.
4. Bei *asynchroner Kommunikation* nehmen wir an, dass jede Nachricht nach einer endlichen Anzahl von Zeitschritten zugestellt wird.

22.1.2 Unmöglichkeitsresultate

Koordinatorwahl in Ringen.

22-7

Wiederholung 1: Ein Unmöglichkeitsresultat.

Satz

Deterministische Koordinatorwahl ist unmöglich, wenn alle Rechner im gleichen Zustand starten.

Beweis. Egal, welchen Algorithmus das System verwendet, gilt: Wenn die Kommunikation (eventuell zufälligerweise) absolut synchron abläuft, dann sind immer alle Rechner im gleichen Zustand.
Würde sich ein Rechner zum Leader erklären, so auch gleichzeitig alle anderen. □

Wir benötigen offenbar *Symmetriebrechung*.

Koordinatorwahl in Ringen.

22-8

Wiederholung 2: Ein Unmöglichkeitsresultat.

Satz

Deterministische Koordinatorwahl ist asynchron unmöglich, wenn ein Prozessor ausfallen kann.

Beweis. Wäre dies möglich, so wäre auch Konsensus unter diesen Umständen möglich (der Koordinator entscheidet). Wir haben aber schon gezeigt, dass dies unmöglich ist. □

22.1.3 Einfacher Algorithmus

22-9

Koordinatorwahl in Ringen.
Ein einfacher Algorithmus.

Satz
In einem Ring mit n Prozessor verfüge jeder Prozessor über eine eindeutige Kennung durch eine Zahl. Dann ist Koordinatorwahl asynchron möglich mit $O(n^2)$ Nachrichten. Im absolut synchronen Fall beträgt die Zeit $O(n)$.

Beweis. Der Algorithmus arbeitet wie folgt:

1. Jeder Prozessor schickt initial seine Kennung an seinen Nachfolger. Diese Nachricht wird *Token* genannt.
2. Jeder Prozessor schickt ein Token weiter, wenn die ID des Token kleiner als seine eigene ist (besser: kleiner als die kleinste bis jetzt gesehene).
3. Erhält ein Prozessor seine eigene ID geschickt, so erklärt er sich zum Leader. □

22-10

Zeitbeschränkungen bei verschiedenen Synchronitäten

Zur Diskussion
Wie lange braucht der Algorithmus bei

1. fast synchroner Kommunikation
2. Kommunikation mit maximaler Zustelldauer a?

22.2 Hirschberg-Sinclair-Algorithmus

22.2.1 Problemstellung

22-11

Eine neue Problemstellung.

Problemstellung

- Wir wollen wieder Koordinatorwahl in einem Ring durchführen.
- Der Unterschied zu vorher ist, dass der Ring nun *bidirektional* ist.
- Wir erhoffen uns dadurch, die *Nachrichtenanzahl* zu verringern.

Im Folgenden wird der (leicht modifizierte) Hirschberg-Sinclair-Algorithmus vorgestellt.

22.2.2 Algorithmus

22-12

Grundlegende Ideen des Hirschberg-Sinclair-Algorithmus
Wie beim einfachen Koordinatorwahl

- werden *Tokens* verschickt,
- werden Tokens *initial generiert* und später *ständig weitergeschickt*,
- werden Tokens *weggeschmissen*, falls die Token-ID größer als die eigene ID ist.

Neu ist, dass

- Tokens *möglichst schnell* weggeschmissen werden,
- indem Token zunächst »lokal die Umgebung testen«.
- Dazu werden Tokens *einige Schritte in jede Richtung* geschickt und dann wieder *zurückgeschickt*.
- Wenn sie wieder zurückkommen (sich also durchsetzen), werden wieder Tokens verschickt, dann aber in eine größere Entfernung.

Aufbau der Tokens.

22-13

Ein Token besteht aus:

- Der *Prozessornummer* des Prozessors, der es erzeugt hat.
- Einem Richtungsindikator, der *outbound* oder *inbound* sein kann.
- Einem Entfernungszähler d.
  - Dieser Zähler wird runtergezählt, solange das Token *outbound* vom Sender wegläuft.
  - Fällt er auf Null, so »dreht das Token um«, wechselt also auf *inbound*.

Der Hirschberg-Sinclair-Algorithmus

22-14

Erste Aktionen jedes Prozessors:

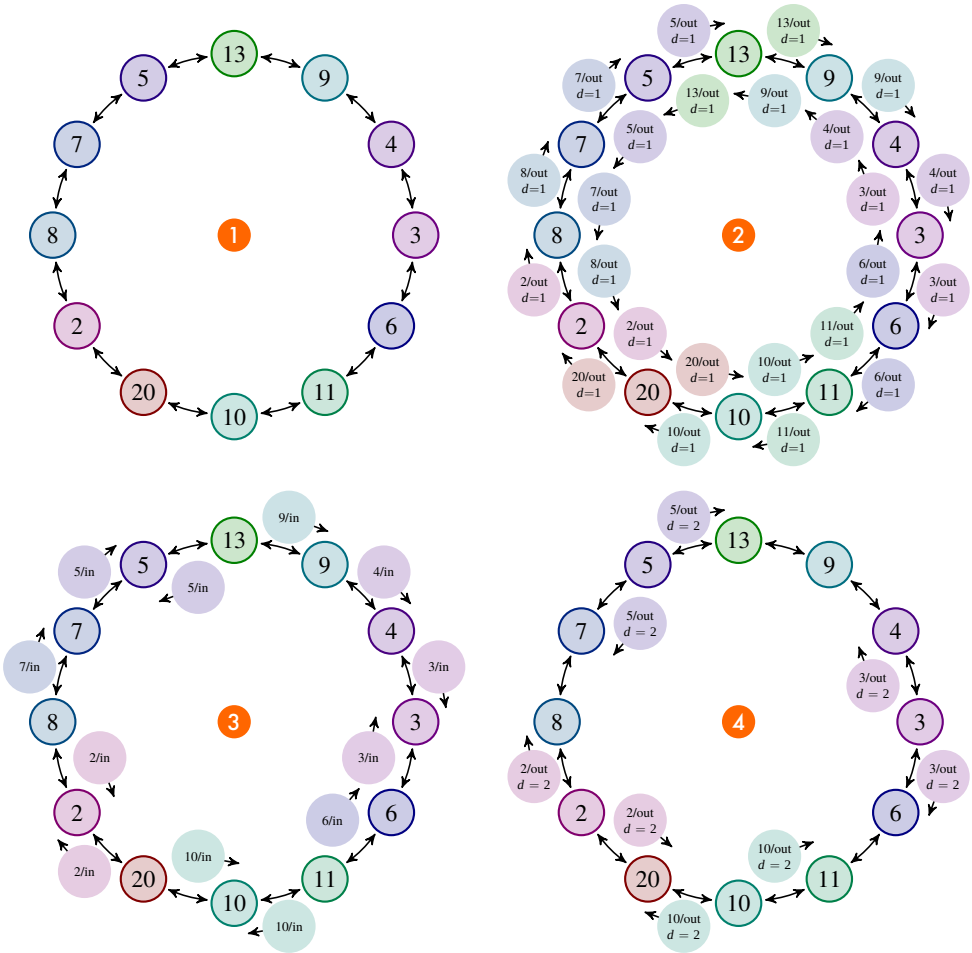
1. Erzeuge zwei *Outbound-Token* mit der *eigenen Prozessornummer* und $d = 1$.
2. Schicke sie *in entgegengesetzte Richtungen herum*.

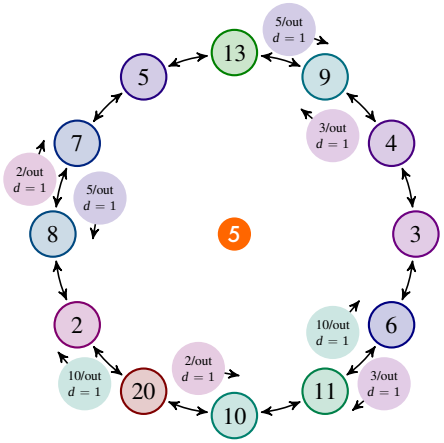
Anschließende Aktionen jedes Prozessors:

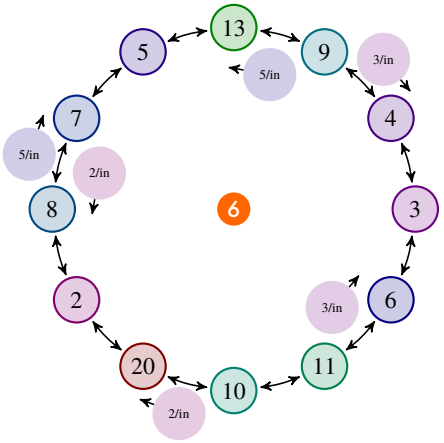
1. Warte auf ein Token von einem der Nachbarn.
2. Falls es *eine fremde Prozessornummer hat*:
   - Ist es inbound, schicke es weiter.
   - Ist die Prozessornummer größer als die eigene, verwerfe es.
   - Sonst erniedrige d um eins. Für $d > 0$ schicke das Token weiter, sonst mache es inbound und schicke es zurück.
3. Falls es *die eigene Prozessornummer hat*:
   - Ist es outbound, so erklärt sich der Prozessor zum Leader.
   - Ist es inbound und ist das andere Token auch schon angekommen, so schicke zwei neue Outbound-Token los mit *verdoppeltem initialen* d.

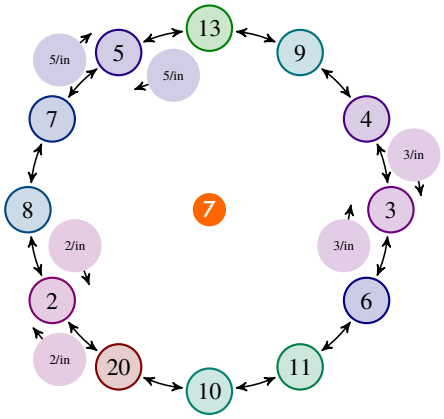
Beispielablauf des Hirschberg-Sinclair-Algorithmus

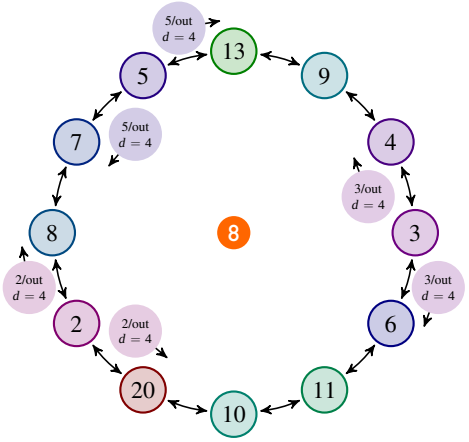
22-15

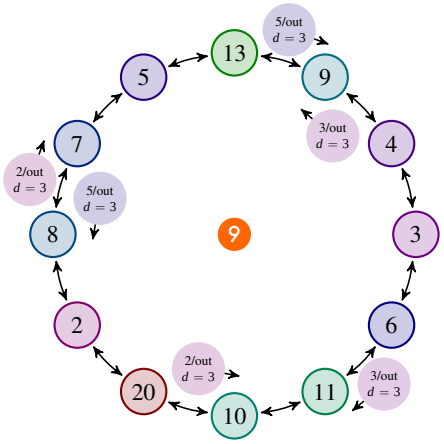


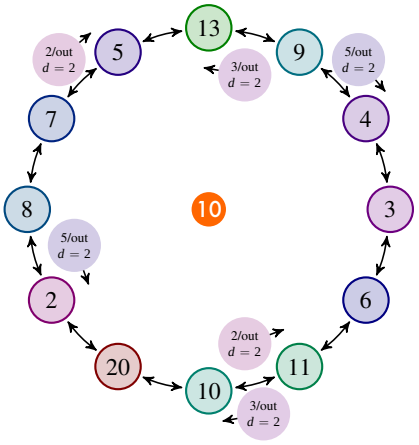

5/out
$d = 1$
13
5
9
7
3/out
$d = 1$
4
2/out
$d = 1$
5/out
$d = 1$
8
3
5
2
10/out
$d = 1$
6
2/out
$d = 1$
10/out
$d = 1$
20
11
3/out
$d = 1$
10

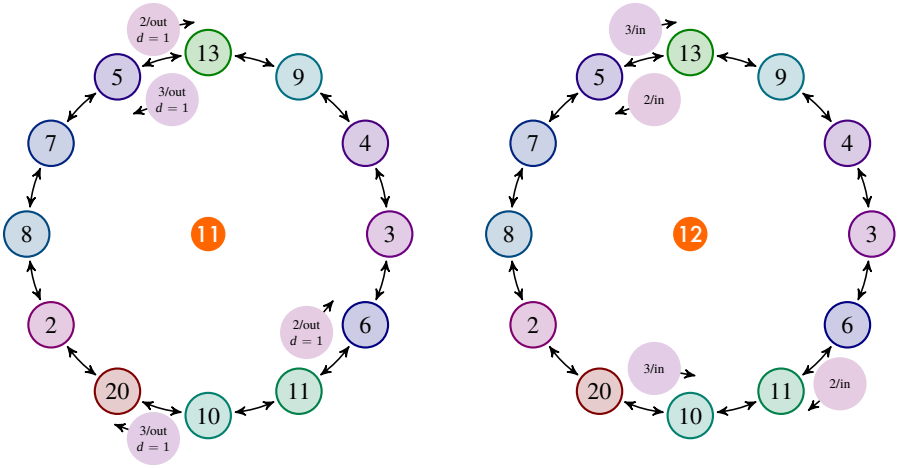

13
5
9
5/in
3/in
7
4
5/in
2/in
8
3
6
2
3/in
20
11
10
2/in


13
5/in
5
9
5/in
7
4
3/in
8
7
3
2/in
3/in
2
6
2/in
20
11
10


5/out
$d = 4$
13
5
9
7
5/out
$d = 4$
4
3/out
$d = 4$
8
8
3
2/out
$d = 4$
3/out
$d = 4$
2
2/out
$d = 4$
6
20
11
10


5/out
$d = 3$
13
5
9
7
3/out
$d = 3$
4
2/out
$d = 3$
5/out
$d = 3$
8
9
3
2
6
2/out
$d = 3$
20
11
3/out
$d = 3$
10


13
2/out
$d = 2$
5
9
5/out
$d = 2$
3/out
$d = 2$
7
4
8
10
3
5/out
$d = 2$
2
6
2/out
$d = 2$
20
11
10
3/out
$d = 2$



22.2.3 Analyse

Analyse des Algorithmus

22-16

Satz

Der Hirschberg-Sinclair-Algorithmus ist ein Koordinatorwahl Algorithmus. Es werden $O(n \log n)$ Nachrichten verschickt. Im absolut synchronen Fall braucht der Algorithmus Zeit $O(n)$.

Beweis.

1. Der Algorithmus ist *korrekt*, da die Tokens des Prozessors mit der kleinsten Nummer *immer weitergereicht* werden. Sie kommen also irgendwann als *Outbound-Token* wieder an.
2. Der Algorithmus braucht im *absolut synchronen* Fall Zeit $O(n)$, da der Leader seine Tokens nach $\sum_{i=1}^{\log n} 2 \cdot 2^i = O(n)$ Runden wiedersieht.
3. Wir werden gleich argumentieren, dass für jedes $d = 2^i$ höchstens $2n/d$ Nachrichten verschickt werden.
4. Dies macht insgesamt $\sum_{i=1}^{\log n} 2n/d \cdot 2d = 4n \log n$ Nachrichten. □

Zur Übung

22-17

Begründen Sie folgende Behauptungen:

- Es werden genau $2n$ Tokens mit einem initialen $d = 1$ ausgeschickt.
- Es werden höchstens n Tokens mit einem initialen $d = 2$ ausgeschickt.
- Es werden höchstens $n/2$ Tokens mit einem initialen $d = 4$ ausgeschickt.
- Es werden höchstens $n/4$ Tokens mit einem initialen $d = 8$ ausgeschickt.
- Und so weiter.

Für Profis

Geben Sie ein Beispiel an, wo für zwei unterschiedliche d gleich viele Nachrichten verschickt werden.

22.3 Nachrichtenarmer Algorithmus

22-18

Der Zeitscheibentrick.

- Wir wollen nun die Nachrichtenzahl auf $O(n)$ drücken.
- Dies ist *nicht immer möglich*, beispielsweise dann nicht, wenn man Tokens *nur vergleichen darf* (was wir bisher auch nur getan haben).

Betrachten wir nun aber folgendes Modell:

- Prozessornummern sind *kleine Zahlen*.
- Eine *Oberschranke n für die Prozessoranzahl* ist bekannt.
- Die Kommunikation ist *absolut synchron*.

Das gibt es folgenden Trick:

- Benutze den einfachen Koordinatorwahl-Algorithmus.
- Prozessor mit der ID i sendet sein Token erst in Runde $i \cdot n$ los.
- Dann wird überhaupt *nur* das Token des Prozessors mit der kleinsten Nummer je versendet.

Zusammenfassung dieses Kapitels

22-19

1. Koordinatorwahl ist das Problem, einen Prozessor auszuwählen.
2. Wir haben Algorithmen hierfür für *Ringe* betrachtet.
3. Der *einfache Algorithmus* schickt einfach Tokens im Kreis herum und braucht $O(n^2)$ Nachrichten.
4. Der *Hirschberg-Sinclair-Algorithmus* testet immer größer werdende *Umgebungen* und braucht $O(n \log n)$ Nachrichten.
5. Das *Zeitscheibenverfahren* braucht nur $O(n)$ Nachrichten.

Übungen zu diesem Kapitel

Übung 22.1 (Leader-Election ohne eindeutige IDs, mittel)

In einem unidirektionalen Ring soll Leader-Election durchgeführt werden. Die IDs der Prozessoren im Ring müssen nicht eindeutig sein, das heißt zwei Prozessoren dürfen dieselbe ID haben. Jedoch ist (a) die Anzahl n der Prozessoren bekannt und es ist (b) bekannt, dass mindestens ein Prozessor eine eindeutige ID hat. Geben Sie einen Algorithmus für diese Situation an; eine Beschreibung der Funktionsweise genügt, es ist kein Pseudo-Code nötig.

Übung 22.2 (Ein Unmöglichkeitsresultat, schwer)

Beweisen Sie, dass die vorherige Aufgabe nicht lösbar ist, wenn die Anzahl n der Prozessoren doch nicht bekannt ist.

Tipp: Führen Sie einen Widerspruchsbeweis. Betrachten Sie einen geeigneten Graphen, für den der Algorithmus funktioniert. Bauen Sie dann einen neuen Graphen, auf dem der Algorithmus nicht funktioniert. Diesen neuen Graphen erhalten Sie durch ein geeignetes »Ausrollen« des ursprünglichen Graphen.

Übung 22.3 (Ringmaximum, mittel)

Die Topologie eines Netzes sei ein bidirektionaler Ring. Die Prozessoren haben *keine* eindeutige IDs, sie kennen aber ihren linken und rechten Nachbarn. Jeder Prozessor im Ring hat eine Zahl gespeichert. Mehrere Prozessoren im Ring initiieren nun eine Berechnung; sie möchten das Maximum aller Zahlen im Ring bestimmen. Am Ende soll dann jeder der Prozessoren, die die Berechnung initiiert hat, dieses Maximum kennen.

1. Argumentieren Sie, dass es keinen Algorithmus für dieses Problem gibt, wenn die Anzahl der Prozessoren unbekannt ist. Sie dürfen die Aussage von Übung 22.2 benutzen.
2. Es sei nun allen Prozessoren eine Zahl n' bekannt, die eine obere Schranke für die Anzahl der Prozessoren darstellt. Geben Sie nun einen Algorithmus für das Problem an.

Übung 22.4 (Koordinatorwahl im Ring, mittel)

In einem unidirektionalen Ring mit asynchroner Kommunikation und n Prozessoren soll eine Koordinatorwahl durchgeführt werden. Genau ein Prozessor im Ring hat eine eindeutige ID alle anderen Prozessoren haben die gleiche ID. Geben Sie einen Algorithus für Leader-Election an, der möglichst wenige Nachrichten versendet. Behandeln Sie gegebenenfalls die Fälle $n = 1$ und $n = 2$ besonders.

23-1

Kapitel 23
Koordinatorwahl in allgemeinen Graphen

Wie man einen Graphen flutet

23-2

Lernziele dieses Kapitels

1. Das Gerüstproblem wiederholen.
2. Verhältnis von Leader-Election und Gerüstproblem kennen.
3. Wellen-Algorithmus zur Leader-Election kennen.

Inhalte dieses Kapitels

Worum es heute geht

Die Algorithmen dieses Kapitels haben eine maritimen Charakter. Wir werden Graphen fluten mit Hilfe einer ausgehenden Welle, welche dann in Form einer eingehenden Welle wieder zurückschwappt. Hinfortgespühlt werden alle Möchtegernkoordinatoren.

Die Grundidee des Koordinatorwahl-Algorithmus für allgemeine Graphen ist ähnlich wie bei Ringen, nämlich das Verschicken von Tokens derart, dass nur die Tokens »überleben«, die vom Prozessor mit der kleinsten ID verschickt wurden. Da die Graphtopologie nun (im Gegensatz zum vorherigen Kapitel) beliebig sein kann, muss man sich darüber Gedanken machen, wie man ein »ewiges Kreisen« von Tokens verhindert, was aber recht einfach gelingt. Der eigentlich schwierige Teil ist, herauszufinden, wer »gewonnen« hat. Anders als in Ringen kommen die eigenen Tokens nämlich nicht »von alleine« zum Initiator zurück. Hier wird die eingehende Welle helfen.

Wellen-Algorithmen eignen sich nicht nur für die Koordinatorwahl. Sie kann man auch zur Konstruktion von Gerüsten (englisch *spanning trees*) einsetzen, zum Broadcasting oder für spezialisiertere Aufgaben.

23.1 Gerüste

23.1.1 Motivation

23-4

Wiederholung: Gerüste

Definition
Ein *Gerüst* (englisch *spanning tree*, nicht ganz glücklich übersetzt als *Spannbaum*) eines Graphen ist ein Teilbaum, der alle Knoten enthält.

Bemerkungen:

- Ein Gerüst entsteht aus einem Graphen, indem man solange Kanten löscht, bis nurnoch ein Baum übrig bleibt.
- Offenbar hat ein Graph genau dann ein Gerüst, wenn er *zusammenhängend* ist.

23-5

Gewurzelte Gerüste.

Definition
Wir nennen einen Baum *gerichtet* oder auch *gewurzelt*, wenn

- eine Wurzel ausgezeichnet ist und
- jede Kante mit einer Richtung versehen ist, die zur Wurzel hinzeigt.

Ein *gerichtetes Gerüst* eines ungerichteten Graphen ist ein gerichteter Baum, der ein Gerüst des Graphen ist, wenn man die Kantenrichtungen vergisst.

Bemerkungen:

- Auch wenn ein Gerüst gerichtet ist, kann man trotzdem Nachrichten »entgegen der Richtung« verschicken.
- Die Richtung hat also lediglich »informativen Charakter«.

23-6

Das Problem, verteilt ein Gerüst zu bestimmen.

Problemstellung
In einem Netzwerk soll ein Gerüst für den Netzgraphen bestimmt werden. Am Ende soll jeder Prozessor seine Nachbarn in diesem Gerüst kennen.

Annahmen zum verteilten System

- Die Netztopologie ist ein *zusammenhängender Graph*.
- Die genaue Zahl der Rechner ist *unbekannt*, aber jeder Rechner hat eine *eindeutige ID*.
- Rechner und Kommunikation sind *absolut verlässlich*.
- Rechner und Kommunikation sind *deterministisch*.
- Die Kommunikation ist *asynchron*.

23.1.2 Anwendungen von Gerüsten

23-7

Anwendungen von Gerüsten.

- Nehmen wir an, wir hätten einen gerichtetes Gerüst eines Netzes errechnet.
- Dann können wir *Broadcasting* implementieren, indem wir Nachrichten nur entlang der Kanten im Gerüst schicken.
  - Dies spart offenbar viele überflüssige Nachrichten ein.
  - Dafür ist das Broadcasting aber auch *sehr störanfällig* im Falle von *ausfallenden Kanten*.
- Weiter können wir die *Koordinatorwahl* implementieren, indem sich *einfach die Wurzel zum Leader* erklärt.

23.1.3 Algorithmus

Ein erster Gerüstalgorithmus. 23-8

Der Algorithmus verläuft in drei Schritten:

1. Wir führen einen *Koordinatorwahl-Algorithmus* durch (dazu gleich mehr).
2. Der Leader l erklärt sich zur *Wurzel* und schickt an alle seine Nachbarn die Nachricht »ich bin dein Elternknoten« oder kürzer parent(l).
3. Auf allen Knoten k läuft ein Hintergrundprozess, der *schläft*, bis er eine Parent-Nachricht bekommt. Dann wacht er auf und macht folgendes:
   - Ist dies nicht das erste Mal, dass eine solche Nachricht ankommt, wird sie verworfen.
   - Sonst speichert der Knoten den Absender der Nachricht als Elternknoten.
   - Dann sendet er die Nachricht parent(k) an alle anderen Nachbarn.

Bemerkungen zum Gerüstalgorithmus. 23-9

- Der Algorithmus ist *wenig robust*.
- Der Leader *weiß nicht*, wann das Gerüst fertig ist.
- Das Fluten des Netzes mit den Parent-Nachrichten werden wir auch *die ausgehende halbe Welle* nennen.

23.2 Koordinatorwahl

23.2.1 Problemstellung

Das allgemeine Koordinatorwahl-Problem 23-10

Problemstellung

Wir wollen eine Koordinatorwahl durchführen.

Annahmen zum verteilten System

- Die Netztopologie ist ein *zusammenhängender Graph*.
- Die genaue Zahl der Rechner ist *unbekannt*, aber jeder Rechner hat eine *eindeutige ID*.
- Rechner und Kommunikation sind *absolut verlässlich*.
- Rechner und Kommunikation sind *deterministisch*.
- Die Kommunikation ist *asynchron*.

Eine erste Idee. 23-11

Vorschlag für einen Algorithmus zur Koordinatorwahl

- Jeder Knoten führt ein Broadcast mit seiner Prozess-ID aus.
- Gewählt wird der Prozess mit der kleinsten Nummer.

Zur Übung

Woran scheitert der obige Algorithmus?

23.2.2 Von wachsenden Bäumen...

23-12

Ideen zum Koordinatorwahl-Algorithmus.

- Wir *kombinieren* die Idee des *Koordinatorwahl-Algorithmus für Ringe* mit dem *Gerüstalgorithmus*.
- Jeder Knoten versucht, ein Gerüst aufzubauen mit sich selbst als Wurzel.
- Jedoch werden parent(k) Nachrichten *gelöscht*, wenn sie ursprünglich von einem Knoten ausgingen mit größerer Prozess-ID als der eigenen.

23-13

Wie startet überhaupt die Koordinatorwahl?

- Jeder Prozessor kann eine Methode *Initiate-LE* aufrufen, um eine Koordinatorwahl zu starten.
- Dies können mehrere Prozesse gleichzeitig machen (weshalb es überhaupt etwas zu tun gibt – sonst könnte sich der initiierende Prozess gleich zum Leader erklären).
- Die Methode beginnt nun damit, ein Gerüst zu erstellen, mit dem Initiator als Wurzel.

Die Methode *Initiate-LE*

Knoten k tue: Für alle ausgehenden Kanten e rufe $send(e, \text{parent}(k), \text{root}(k))$ auf.

23-14

Der Hintergrundprozess auf allen Knoten.

Ausgehende halbe Welle.

Die Variablen des Hintergrundprozess:

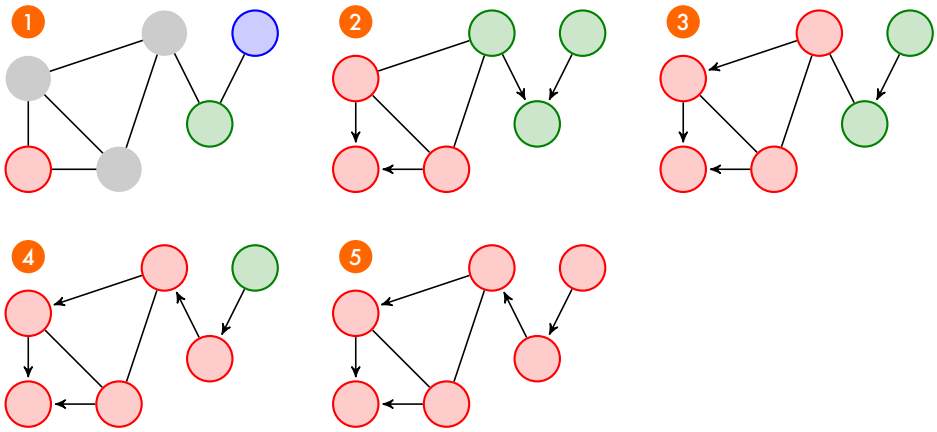
- p speichert den Elternknoten im aktuellen Gerüst.
- r speichert die Wurzel des aktuellen Gerüsts.

Der Hintergrundprozess *schläft*, bis ein Receive-Ereignis eintritt.

1. Sei $receive(e, \text{parent}(p'), \text{root}(r'))$ das Ereignis.
2. Falls r leer oder $r' < r$:
   - $p \leftarrow p'$ und $r \leftarrow r'$.
   - Rufe $send(e', \text{parent}(p'), \text{root}(r'))$ für alle Kanten außer e auf, die vom eigenen Knoten ausgehen.

23-15

Beispiel der ausgehenden halben Welle.



23-16

Zur Übung

Im absolut synchronen Fall: Von welchen Parametern hängt die Zeit ab, dies es dauert, bis der Graph monochromatisch ist (nur eine Farbe hat):

1. Durchmesser
2. Anfangsposition des Leaders
3. Anfangsposition der anderen Initiatoren

23.2.3 . . . und Wellen

Wie findet man heraus, wer der Leader ist? 23-17

- Der Algorithmus sorgt dafür, dass *am Ende* alle Knoten die Nummer des Leader gespeichert haben.
- Leider wissen wir nicht, *wann das Ende erreicht ist*.
- Wir benutzen deshalb eine *eingehende halbe Welle*.

Idee zur eingehenden halben Welle. 23-18

- Wir wollen überprüfen, ob der Graph monochromatisch ist.
- Dazu *schneiden wir immer wieder Blätter ab*, bis nur noch die Wurzel überbleibt.
- Wir schneiden aber ein Blatt nur ab, wenn wir *uns sicher sind*, dass es sich um ein Blatt handelt.
- Abschneiden bedeutet lediglich, dass wir ein Flag setzten, dass diese Blatt abgeschnitten wurde.

Definition

Ein *Blatt* ist ein Knoten mit folgenden Eigenschaften:

- Alle Nachbarn haben dieselbe Farbe.
- Kein (noch vorhandener) Nachbar hat das Blatt als Elternknoten gespeichert.

Der Hintergrundprozess auf allen Knoten. 23-19

Eingehende halbe Welle – von unentschlossen zu entschlossen.

Wir erweitern den Hintergrundprozess wie folgt:

1. Immer, wenn die Farbe des Knoten geändert wird (wenn sich also r ändert) tue:
   - Gehe in den Zustand *unentschlossen* über.
   - Schicke an alle Nachbarn die Nachricht »Bitte benachrichtige mich, wenn du die Farbe r hast (also dieselbe wie ich).« und sammle im Hintergrund die Antworten.
   - Schicke entsprechend Benachrichtigungen aus, wenn sie von anderen Knoten angefordert werden.
2. Wenn alle Nachbarn die Nachricht geschickt habe, dass sie dieselbe Farbe haben, gehe in den Zustand *entschlossen*.

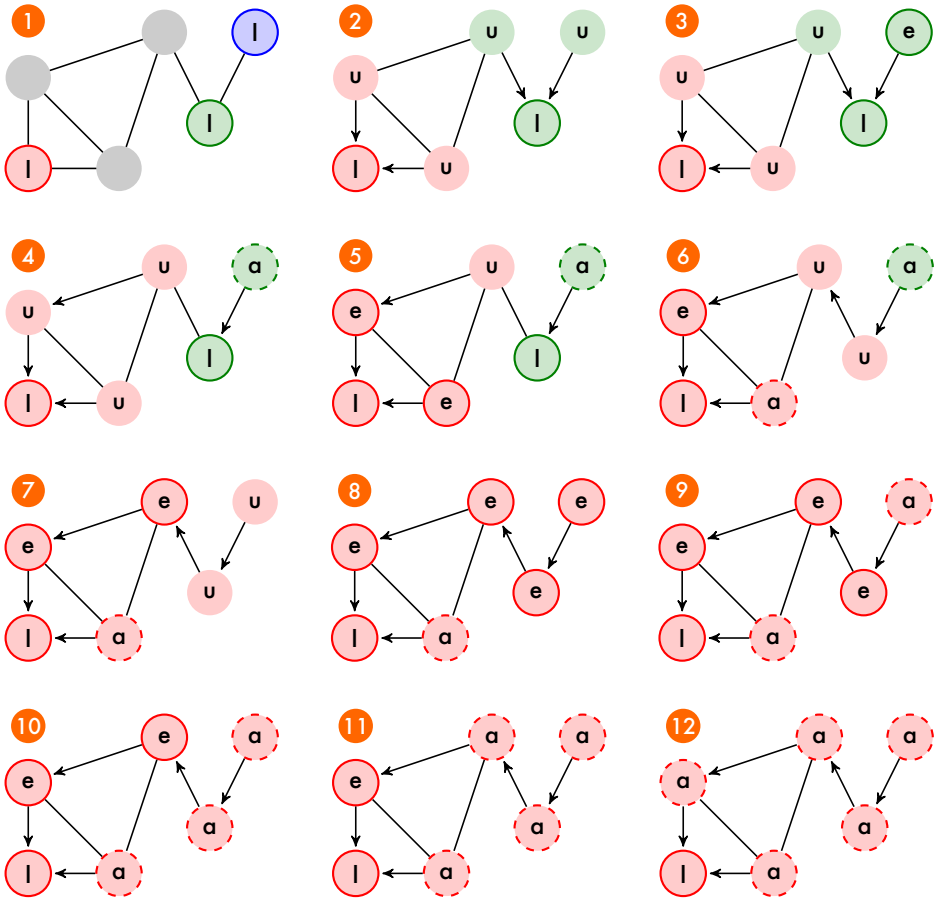
Der Hintergrundprozess auf allen Knoten. 23-20

Eingehende halbe Welle – von entschlossen zu abgeschnitten.

1. Sobald der Knoten *entschlossen* ist, tue:
   - Schicke an alle Nachbarn die Nachricht »Bitte benachrichtige mich, wenn ich nicht dein Elternknoten bin oder du abgeschnitten wurdest.« und sammle im Hintergrund die Antworten.
   - Schicke entsprechende Benachrichtigungen aus, wenn sie von anderen Knoten angefordert werden.
2. Wenn alle Nachbarn antworten, dass sie abgeschnitten wurden oder der eigenen Knoten nicht ihr Elternknoten ist, setze den eigenen Zustand auf *abgeschnitten*.
3. Schicke gegebenenfalls eine Benachrichtigung aus.

23-21

Beispiel der eingehenden halben Welle.



- I Initiator
- u unentschlossen
- e entschlossen
- a abgeschnitten

23-22

Der fertige Algorithmus.

- Auf allen Knoten laufen die Hintergrundprozesse *ausgehende halbe Welle* und *eingehende halbe Welle*.
- Sobald ein Initiator von allen seinen Nachbarn die Nachricht bekommt, dass sie abgeschnitten wurden, erklärt er sich zum Leader.

Satz
Der Wellen-Algorithmus ist korrekt.

Beweis.

- Es sollte klar sein, dass die ausgehende Welle den Graphen mit der ID des zukünftigen Leaders flutet.
- Die eingehende Welle:
  - Eine eingehende Welle kann nicht komplett zurückkommen, wenn noch nicht alle Knoten im Graphen gefärbt sind.
  - Sind sie alle gefärbt, so kommt sie auch zurück.

□

Zusammenfassung dieses Kapitels

1. Die *Konstruktion von Gerüsten* und die *Koordinatorwahl* sind gleich schwierig.
2. Zur Koordinatorwahl benutzt man einen *Wellen-Algorithmus*.
3. In der *ausgehenden Halbwelle* flutet der zukünftige Leader das Netz mit seiner ID.
4. In der *eingehenden Halbwelle* melden die Knoten von den Blättern her, dass sie gefärbt wurden.

Übungen zu diesem Kapitel

Übung 23.1 (Leader-Election in Bäumen, mittel)

In einem ungerichteten, gefärbten Baum soll eine Leader-Election durchgeführt werden. Jedem Knoten des Baumes sind lediglich seine Nachbarn bekannt. Jeder Knoten hat eine Farbe (eine positive ganze Zahl), die anders ist als die Farben aller seiner Nachbarn – dieselbe Farbe kann aber ansonsten mehrfach auftauchen.

Geben Sie Pseudo-Code für die beteiligten Prozesse eines Leader-Election-Algorithmus hierfür an. Ihr Algorithmus kann sich beispielsweise an den Eingehende-Welle-Algorithmus anlehnen (die Färbung des Graphen brauchen Sie nur ganz am Ende, um bei der letzten Kante einen »Tie-Break« durchzuführen).

Übung 23.2 (Größen-/Durchmesserbestimmung, leicht)

Die Größe eines Netzwerks mit unbekannter Topologie soll bestimmt werden, wobei alle Prozessoren eine eindeutige ID haben. Genau ein Prozessor initiiert eine Berechnung; am Ende soll dieser Prozessor die Anzahl n an Prozessoren im Netzwerkgraphen kennen. Beschreiben Sie einen Algorithmus hierfür; Sie dürfen bekannte Algorithmen aufrufen.

Wenn Ihnen dies zu einfach ist, bestimmen Sie statt der Größe den Durchmesser (die maximale Entfernung zwischen zwei Knoten) des Netzgraphen.

24-1

Kapitel 24
Broadcasting-Algorithmen
Wie gleicht man Konten und News-Server ab?

24-2

Lernziele dieses Kapitels

1. Das Broadcasting-Problem kennen.
2. Unterschiede zwischen verlässlichem, Fifo-, kausalem und atomarem Broadcast kennen.
3. Algorithmen für Broadcasting kennen.

Inhalte dieses Kapitels

Worum es heute geht

Das Verbreiten von Nachrichten ist eine Wissenschaft für sich, wie schon Douglas Adams sehr richtig erkannte. Die ersten Absätze seines im Großen und Ganzen sonst eher pessimistisch gestimmten Buches *Mostly Harmless* (immerhin wird am Ende des Buches das Universum zerstört) fassen die Hauptprobleme des Broadcastings elegant zusammen:

Beginn von *Mostly Harmless* von Douglas Adams

> Anything that happens, happens.
>
> Anything that, in happening, causes something else to happen, causes something else to happen.
>
> Anything that, in happening, causes itself to happen again, happens again.
>
> It doesn't necessarily do it in chronological order, though.
>
> The history of the Galaxy has got a little muddled, for a number of reasons: partly because those who are trying to keep track of it have got a little muddled, but also because some very muddling things have been happening anyway.
>
> One of the problems has to do with the speed of light and the difficulties involved in trying to exceed it. You can't. Nothing travels faster than the speed of light with the possible exception of bad news, which obeys its own special laws. The Hingefreel people of Arkintoofle Minor did try to build spaceships that were powered by bad news but they didn't work particularly well

and were so extremely unwelcome whenever they arrived anywhere that there wasn't really any point in being there.

So, by and large, the peoples of the Galaxy tended to languish in their own local muddles and the history of the Galaxy itself was, for a long time, largely cosmological.

Which is not to say that people weren't trying. They tried sending off fleets of spaceships to do battle or business in distant parts, but these usually took thousands of years to get anywhere. By the time they eventually arrived, other forms of travel had been discovered which made use of hyperspace to circumvent the speed of light, so that whatever battles it was that the slower-than-light fleets had been sent to fight had already been taken care of centuries earlier by the time they actually got there.

This didn't, of course, deter their crews from wanting to fight the battles anyway. They were trained, they were ready, they'd had a couple of thousand years' sleep, they'd come a long way to do a tough job and by Zarquon they were going to do it.

This was when the first major muddles of Galactic history set in, with battles continually re-erupting centuries after the issues they had been fought over had supposedly been settled. However, these muddles were as nothing to the ones which historians had to try and unravel once time-travel was discovered and battles started pre-erupting hundreds of years before the issues even arose. When the Infinite Improbability Drive arrived and whole planets started turning unexpectedly into banana fruitcake, the great history faculty of the University of MaxiMegalon finally gave up, closed itself down and surrendered its buildings to the rapidly growing joint faculty of Divinity and Water Polo, which had been after them for years.

24.1 Einführung

24.1.1 Motivation

Das allgemeine Broadcasting-Problem. 24-4

Problemstellung

Ein Rechner hat eine Nachricht m, die er allen anderen Rechnern im Netz mitteilen möchte.

- Offenbar ist dies nur möglich, wenn das Netz zusammenhängend ist, was wir deshalb heute immer annehmen.
- Algorithmus hierfür sollten sicherlich folgende Eigenschaften haben:
  1. Er sollte *korrekt* sein.
  2. Er sollte *möglichst wenige* Nachrichten verschicken.
  3. Er sollte *möglichst schnell* sein.
  4. Er sollte *möglichst robust* sein.
- Wir werden noch sehen, dass es noch weniger offensichtlich Eigenschaften gibt, die auch noch erstrebenswert sind.

Beispiel: Broadcasting bei einer Bank. 24-5

Das Netz einer Großbank sei wie folgt aufgebaut:

- Jeder Server hält (vielleicht nicht alle) Kundendaten vorrätig.
- Dieselben Kundendaten können auf mehreren Servern liegen, zum Beispiel auf den Servern der Immobilienabteilung und nochmal auf den Servern der Giroabteilung.

Beispiel-Ereignisse, die allen Servern mitgeteilt werden müssen:

- Ein Kunde zahlt *100 Euro auf sein Konto ein*.
- Ein Kunde hebt *100 Euro von seinem Konto ab*.
- Der *Zinssatz* ändert sich.

Beispiel: Broadcasting bei News-Servern. 24-6

Ein News-Dienst sei wie folgt aufgebaut:

- Viele Server und Rechner sind Teil des Netzes.
- Jeder Rechner kennt einen oder mehrere Server, über die er Nachrichten schicken kann.
- Die Server halten alle Nachrichten in Diskussionsthread angeordnet vor.

Beispiel-Ereignisse, die allen Servern mitgeteilt werden müssen:

- Ein Diskussionsbeitrag, der auf einen Server geschickt wurde.

24.1.2 Modell

24-7

Modell für das Broadcasting-Problem.

Problemstellung

- Wir wollen (eventuell mehrere) Broadcastings in einem Netz durchführen.
  - Jedes Broadcasting beginnt damit, dass ein Knoten r die Methode $broadcast(m)$ für eine Nachricht m aufruft.
  - Nach einiger Kommunikation soll dann irgendwann bei jedem Knoten einmal das Ereignis $deliver(m)$ eintreten.
- Die Netztopologie ist ein *zusammenhängender Graph*.
- Die genaue Zahl der Rechner ist *unbekannt*, aber jeder Rechner hat eine *eindeutige ID*.
- Rechner und Kommunikation sind *absolut verlässlich*.
- Rechner und Kommunikation sind *deterministisch*.
- Die Kommunikation ist *asynchron*.

24.1.3 Methoden versus Ereignisse

24-8

Send-Methode versus Receive-Ereignis.

Zur Send-Methode:

- Das Betriebssystem stellt eine Methode *send* zur Verfügung.
- Diese bekommt als Parameter eine *Kante e*, die mit dem Knoten inzident ist, und eine *Nachricht m*.
- Das Betriebssystem schickt dann diese Nachricht über die Kante und kehrt sofort zurück.

Zum Receive-Ereignis:

- Das Betriebssystem des Knoten am anderen Ende der Kante *erzeugt einen Interrupt*, wenn eine Nachricht ankommt.
- Dann wird ein *Receive-Ereignis* vom Betriebssystem erzeugt und in die Systemwarteschlange eingereiht.

24-9

Broadcast-Methode versus Deliver-Ereignis.

Zur Broadcast-Methode:

- Wir werden *verschiedene Implementation* dieser Methode kennenlernen, die dann Namen wie fifo-broadcast haben.
- *Innerhalb* dieser Methode wird die *Kommunikation angestoßen*.
- Dazu rufen wir unter anderem die *send-Methode* auf.

Zum Deliver-Event:

- Deliver-Events sollen ähnlich wie das Receive-Event in die Systemwarteschlange eingereiht werden.
- *Ausgelöst* werden sie aber nicht vom Betriebssystem, sondern von einem *eigenen Prozess*, der Teil des Broadcasting-Algorithmus ist.

24-10

Die zwei Teile eines Broadcasting-Algorithmus.

Die Broadcasting-Algorithmen werden aus zwei Teilen bestehen:

1. Dem Code der *Broadcasting-Methode*. Dieser Code wird nur bei dem Knoten ausgeführt, der das Broadcasting initiiert.
2. Dem Code eines *Hintergrundprozesses*, der auf Receive-Events wartet und sich dann um zwei Dinge kümmert:
   - 2.1 Er erzeugt Deliver-Ereignisse.
   - 2.2 Er leitet Nachrichten weiter.

24.2 Verlässliches Broadcast

24.2.1 Algorithmus

Unser erstes Ziel.

24-11

- Wir wollen einen einfachen Algorithmus für Broadcast angeben.
- Sei m die Nachricht und r der Knoten, der das Broadcast initiiert.
- Der Algorithmus soll *verlässlich* sein, was bedeutet:
  1. Alle Knoten erzeugen irgendwann das Ereignis *deliver*(m).
  2. Jeder Knoten erzeugt dieses Ereignis pro Broadcast nur einmal.

Der Reliable-Broadcast-Algorithmus.

24-12

Grobe Idee
Flute das Netz mit der Nachricht.

Die Methode *reliable-broadcast*(m)
Für alle Kanten e, die vom eigenen Knoten ausgehen, rufe *send*(e,m) auf.

Der Hintergrundprozess auf allen Knoten
Der Hintergrundprozess *schläft*, bis ein Receive-Ereignis eintritt. Wenn dies eintritt, passiert folgendes:

1. Sei *receive*(e,m) das Ereignis.
2. Falls m zum ersten Mal ankommt:
   - Erzeuge das Ereignis *reliable-deliver*(m).
   - Merke für die Zukunft, dass m erhalten wurde.
   - Rufe *send*(e',m) für alle Kanten außer e auf, die vom eigenen Knoten ausgehen.

Zur Übung

24-13

Nehmen wir nun an, Prozessoren können während der Algorithmus läuft ausfallen (durch einen Crash).
Falls der Graph ohne die ausgefallen Knoten noch zusammenhängend ist, funktioniert der er dann trotz der Ausfälle?

Zur Übung
Wie viele Nachrichten werden durch den Algorithmus verschickt? Bei absolut synchroner Kommunikation: Wie lange dauert es, bis alle Prozessoren die Nachricht zugestellt haben?

24.3 Fifo-Broadcast

24.3.1 Problematik

24-14

Ein mögliches Problem bei verlässlichem Broadcast.

- Nehmen wir an, die Großbank bekommt einen neuen Kunden und dieser zahlt 100 Euro auf sein neues Konto ein.
- Dann werden vom Computer der Filiale, wo das Konto eröffnet wurde, folgende Nachrichten versandt:
  1. Es gibt ein neues Konto 123 für Kunde X.
  2. Auf Konto 123 werden 100 Euro eingezahlt.
- Unser Broadcasting-Algorithmus *garantiert aber nicht*, dass diese Nachrichten in dieser Reihenfolge ankommen. Ein Server könnte also folgende Nachrichten bekommen:
  1. Auf Konto 123 werden 100 Euro eingezahlt.
  2. Es gibt ein neues Konto 123 für Kunde X.
- Da der Server beim Erhalt der ersten Nachricht noch kein Konto 123 kennt, liegt ein Fehler vor.

24-15

Neue Anforderung an das Broadcast: Fifo

Wir wollen einen Algorithmus für Broadcast angeben, der *verlässlich* ist und *noch folgende* Eigenschaft hat:

Fifo-Eigenschaft

Ruft ein Knoten erst Broadcast für eine Nachricht m_1 auf und danach Broadcast für eine Nachricht m_2, so soll bei allen Knoten das Deliver-Ereignis für m_1 vor dem Deliver-Ereignis für m_2 eintreten.

24.3.2 Algorithmus

24-16

Vorüberlegungen zum Fifo-Broadcast.

- Wir können *nicht verhindern*, dass Nachrichten in der falschen Reihenfolge ankommen.
- Der Hintergrundprozess kann aber das Deliver-Ereignis *hinauszögern*.
- Genauer werden wir *zwei Hintergrundprozesse* benutzen:
  1. Den alten Hintergrundprozess des verlässlichen Broadcast.
  2. Einen neuen Hintergrundprozess, der auf Reliable-Deliver-Ereignisse wartet und dann (eventuell verzögert) Fifo-Deliver-Ereignisse erzeugt.
- Wir *benutzen* also den Reliable-Broadcast-Algorithmus und bauen den Fifo-Broadcast-Algorithmus *darüber*.

24-17

Nachrichten-IDs und zustellbare Nachrichten.

- Es ist problematisch, dass Nachrichten mehrfach auftreten können.
- Ebenso ist es problematisch, dass man nicht weiß, wer eine Nachricht gesandt hat.
- Deshalb vereinbaren wir, dass jede Nachricht zusätzlich die Knoten-ID enthält und eine laufende Nummer (lokal für den Knoten).

Definition

Sei m eine Nachricht mit Absender-Adresse k und Message-ID i. Diese Nachricht heißt auf einem Knoten k' *Fifo-zustellbar*, falls

- $i = 0$ (erste Nachricht von k) oder
- bei k' das letzte Fifo-Deliver-Ereignis mit Absender k die Message-ID $i-1$ hatte.

Der Fifo-Broadcast-Algorithmus. 24-18

Grobe Idee
Führe ein Reliable-Broadcast durch, verzögere aber Deliver bis alle vorherigen Nachrichten des Absenders ausgeliefert wurden.

Die Methode *fifo-broadcast*(m)
Führe *reliable-broadcast*(m) aus, wobei m Knoten- und Message-ID enthält.

Der zweite Hintergrundprozess auf allen Knoten
Der Hintergrundprozess schläft, bis ein *Reliable-Deliver-Ereignis* eintritt.

1. Sei *reliable-deliver*(m) das Ereignis.
2. Füge m in den *Message-Bag* ein.
3. Solange er eine fifo-zustellbare Nachricht m' enthält:
   - Erzeuge das Ereignis *fifo-deliver*(m').
   - Entferne m' aus dem Bag.

24.4 Kausaler Broadcast

24.4.1 Problematik

Ein mögliches Problem bei Fifo-Broadcast. 24-19

- Server X verschickt einen *Diskussionsbeitrag* A an alle.
- Auf Server Y sieht jemand diesen und schreibt daraufhin eine *Erwiderung* B, die Y nun an alle verschickt.
- Bei einem dritten Server Z kann nun folgendes passieren:
  1. Das ursprüngliche Broadcast des *Beitrags* A durch X *verzögert* sich.
  2. Das Broadcast der *Erwiderung* B durch Y geht hingegen *schnell*.
  3. Also erhält Z *zuerst* B, *dann* A.
  4. Dies ist *kein Verstoß* gegen das Fifo-Prinzip, da A und B von unterschiedlichen Prozessen verschickt wurden.

Neue Anforderung an das Broadcast: Kausalität 24-20

Wir wollen einen Algorithmus für Broadcast angeben, der folgende Eigenschaft hat:

Kausalitätseigenschaft
Erhält ein Knoten eine Nachricht m_1 und führt ruft danach Broadcast für eine Nachricht m_2 auf, so soll bei allen Knoten das Deliver-Ereignis für m_1 vor dem Deliver-Ereignis für m_2 eintreten.

24.4.2 Algorithmus

Vorüberlegungen zum kausalen Broadcast. 24-21

- Wir benutzen nun Fifo-Broadcast, um darauf kausalen Broadcast aufzubauen.
- Um die Kausalität zu garantieren, schicken wir einfach *alle Message-IDs mit* von Nachrichten, die vor dieser Nachricht zugestellt werden müssen.

Definition
Sei m eine Nachricht mit Absender-Adresse k und c_1 bis c_n die IDs von weiteren Nachrichten, die vor m ausgeliefert werden müssen. Dann heißt die Nachricht auf einem Knoten k' *kausal-zustellbar*, falls

- alle vorherigen Nachrichten von k auf diesem Knoten schon zugestellt wurden und
- alle Nachrichten c_i auf diesem Knoten schon zugestellt wurden.

24-22

Der kausale Broadcast-Algorithmus.

Grobe Idee
Schicke mit jeder Nachricht alle Bedingungen mit, wann diese Nachricht zugestellt werden darf.

Die Methode *causal-broadcast*(m)
Führe *fifo-broadcast*$(m; c_1; c_2; \ldots; c_n)$ aus, wobei die c_i alle Nachrichten sind, die seit dem letzten Aufruf dieser Methode zugestellt wurden.

Der dritte Hintergrundprozess auf allen Knoten
Der Hintergrundprozess *schläft*, bis ein Fifo-Deliver-Ereignis eintritt. Wenn dies eintritt, passiert folgendes:

1. Sei *fifo-deliver*$(m; c_1; c_2; \ldots; c_n)$ das Ereignis.
2. Füge $m; c_1; c_2; \ldots; c_n$ in den *Message-Bag* ein.
3. Solange er ein kausal-zustellbares m' enthält:
   - Erzeuge das Ereignis *causal-deliver*(m').
   - Entferne m' aus dem Bag.

24-23

Korrektheit des Algorithmus

Satz
Der kausale Broadcast-Algorithmus arbeitet korrekt.

Beweis.

- Der Algorithmus stellt keine Nachricht in kausal falscher Reihenfolge zu: Keine Nachricht wird zugestellt, bis alle beim Absender vorher eingegangenen Nachrichten zugestellt wurden.
- Der Algorithmus stellt alle Nachrichten zu, da die Kausalitätsbedingung einen DAG auf allen Nachrichten definiert, der von den Quellen her abgearbeitet wird. □

24.5 Atomarer Broadcast

24.5.1 Problematik

24-24

Ein mögliches Problem bei kausalem Broadcast.

- Ein Kunde *zahlt 100 Euro* bei seiner Filiale auf sein Konto ein. Dies wird per Broadcast allen mitgeteilt.
- Der *Zinssatz wird erhöht*, was auch per Broadcast mitgeteilt wird.
- Bei zwei verschiedenen Konto-Servern X und Y kann nun folgendes passieren:
  1. Die Ereignisse der Einzahlung und der Zinzsatzänderung sind *kausal nicht verbunden*.
  2. Deshalb können sie in *unterschiedlichen Reihenfolgen* eintreffen.
  3. Dann werden aber die *Kontostände* auf den Servern *unterschiedlich* sein.

24-25

Neue Anforderung an das Broadcast: Unteilbarkeit

Wir wollen einen Algorithmus für Broadcast angeben, der folgende Eigenschaft hat:

Unteilbarkeit (atomicity)
Alle Nachrichten werden bei allen Knoten in derselben Reihenfolge zugestellt.

24.5.2 Algorithmus

Asynchrones atomares Broadcasting mittels eines Koordinators. 24-26

Die Methode *atomic-broadcast*(m)

- Sende Nachricht an den Koordinator (Leader) mit der bitte um eine neue Message-ID. (Diese werden vom Koordinator fortlaufend mit 0 beginnend ausgestellt.)
- Warte auf den Erhalt einer ID i.
- Führe *reliable-broadcast*$(m;i)$ aus.

Der Hintergrundprozess auf allen Knoten

Ein ID-Zähler j ist mit 0 initialisiert.

1. Warte auf ein *reliable-deliver*$(m;i)$ Ereignis und füge $(m;i)$ in den *Message-Bag* ein.
2. Solange er ein Paar $(m';j)$ enthält:
   - Erzeuge das Ereignis *atomic-deliver*(m').
   - Entferne $(m';j)$ aus dem Bag und inkrementiere j.

Zusammenfassung dieses Kapitels

1. *Broadcast* ist das Problem, eine Nachricht an alle Knoten eines Netzes zu verteilen. 24-27
2. Dieses Problem gibt es in verschiedenen Varianten:
   - verlässliches Broadcast
   - Fifo-Broadcast
   - kausaler Broadcast
   - atomarer Broadcast

Übungen zu diesem Kapitel

Übung 24.1 (Broadcast im $n \times n$-Gitter, mittel)

Gegeben sei ein $n \times n$-Gitter von Prozessoren, die den Wert von n allerdings nicht kennen.

1. Geben Sie ein Protokoll für Broadcast an, in dem möglichst wenige Nachrichten versendet werden. (Wie immer ist die Kommunikation asynchron.)
2. Geben Sie nun an, wie lange es in Ihrem Protokoll bei absolut *synchroner* Kommunikation dauert, bis jeder Prozessor den Broadcast erhalten hat und wie viele Nachrichten insgesamt versendet werden.

Teil V
Spezifikation verteilter Systeme

Der vorherige Teil dieser Vorlesung dürfte eine Sache klargemacht haben: Verteilte Systeme sind schwierig zu beherrschen. Man kann schon froh sein, wenn ein verteiltes System mit sich selbst klarkommt – dass es auch noch etwas Sinnvolles macht ist schon Luxus. Unter diesen Umständen stellt sich eine Frage dringlicher als dies bei sequentiellen und auch bei parallelen Systemen der Fall ist: Wie beschreibt man, was ein verteiltes System machen soll und wie es arbeiten soll? Etwas wissenschaftlicher formuliert: Wie spezifiziert man ein verteiltes System?

Viele Methoden, die bei nichtverteilten Systemen ganz natürlich funktionieren, versagen jämmerlich bei verteilten Systemen. Ein einfaches Beispiel: Wir wollen Summen von Zahlen berechnen. Eine Spezifikation könnte lauten: »Eingabe: Ein Array n von Zahlen. Ausgabe: Die Summe der Zahlen.« Bei einem verteilten System klappt eine solche Eingabe-Ausgabe-Spezifikation nicht so recht. Man muss schon sagen wo die Eingabe am Anfang ist und wo am Ende die Ausgabe sein soll und was überhaupt »am Anfang« und »am Ende« bedeuten sollen und was passieren soll, wenn etwas schiefgeht und so weiter. Ebenso unklar ist, wie man die gewünschte *Arbeitsweise* eines verteilten Systems geschickt beschreiben kann. Wir haben uns bis jetzt mit Pseudo-Code beholfen, in dem etwas nebulös auf »Betriebssystemereignisse« gewartet wird, oder das Problem ganz einfach ignoriert.

Im letzten Teil dieser Veranstaltung soll es darum gehen, wie man das gewünschte Verhalten eines verteilte Systems beschreiben kann. Dazu werden wir exemplarisch nur einen Formalismus betrachten, nämlich Petrinetze, diesen dafür aber etwas ausführlicher. (Für einen ebenfalls gut in diese Vorlesung passenden anderen Ansatz, die Temporallogiken, war leider kein »Platz« mehr.)

Petrinetze, ein von Adam Petri in seiner Dissertation eingeführter Formalismus – auch wenn man in seiner Arbeit der Namen »Petrinetz« natürlich vergeblich suchen wird und auch wenn die dort eingeführten Netz auf den ersten Blick nicht viel mit den »modernen« Petrinetzen zu tun zu haben scheinen –, gibt es mittlerweile in vielen verschiedenen Formen: Es gibt die Grundform des Petrinetzes, etwas despektierlich »Low-Level-Netze« genannt; es gibt »bunte« Petrinetze (colored nets); und es gibt High-Level-Netze, die alles können, aber dafür leider nicht mehr vernünftig zu analysieren sind. Wir werden uns auf die Low-Level-Netze konzentrieren. Diese sind erstens schön einfach zu verstehen und zu formalisieren. Zweitens, und das ist der Hauptgrund, weshalb Petrinetze auch heute noch in realen Projekten eingesetzt werden, lassen sich Low-Level-Netze erstaunlich gut *automatisch* analysieren.

Mit Low-Level-Netze verhält es sich ein wenig wie mit regulären Ausdrücken: Man kann sie nicht immer benutzen, denn die meisten Sprachen sind nicht regulär ebenso wie die meisten verteilten Probleme sich nicht mit Low-Level-Netzen beschreiben lassen. Wenn aber zumindest Teile eines Problems mit ihnen beschreibbar sind, dann gibt es mächtige theoretische und praktische Werkzeuge für die Analyse: Hinter reguläre Ausdrücke steht die geballte Theorie-Macht der regulären Sprachen und sie werden in ganz realer Software ständig eingesetzt, hinter Low-Level-Netzen steht die Theorie der altehrwürdigen Linearen Algebra und sie werden in ganz realen Systemen auch benutzt.

Man kommt mit Low-Level-Netzen nicht immer aus, weshalb man zu High-Level-Netzen greifen kann. Aber ähnlich wie beim Schritt von regulären Ausdrücken zu allgemeinen Grammatiken verliert man hierdurch alle automatisierten Entscheidungsverfahren und alle interessanten Fragestellungen werden unentscheidbar. Wir werden deshalb High-Level-Netze am Ende nur recht kurz streifen.

25-1

Kapitel 25
Petrinetze
Visualisierung und maschinelle Analyse verteilter Systeme

25-2

Lernziele dieses Kapitels

1. Konzept des Petrinetz kennen
2. Formalisierung von Petrinetzen kennen
3. Problemstellungen mit Petrinetzen beschreiben können

Inhalte dieses Kapitels

Worum es heute geht

Petrinetze sind zu aller erst ein Formalismus mit dem sich das Verhalten eines (gerne auch verteilten) Systems über die Zeit hinweg beschreiben lässt. Statt »Zeit« sollte es bei den einfachen Systemen eher »Kausalität« heißen, denn es geht um Fragen wie »Wenn dieses passiert, kann dann prinzipiell auch jenes passieren ohne dass auch jenes andere passiert?« In Erweiterungen von Petrinetzen werden aber auch echte Timingprobleme angepackt.

Petrinetze halten elegant die Waage zwischen einfacher Formalisierung, Universalität und Analysierbarkeit. Wie wir in diesem Kapitel sehen werden ist die mathematische Formalisierung wirklich recht einfach, kaum komplizierter als beispielsweise endliche Automaten. Wir werden ebenfalls in diesem Kapitel beginnen uns Beispiele anzuschauen, bei denen unterschiedlichste Problemstellungen mit dem Formalismus der Petrinetze modelliert werden – weitere Beispiel werden in späteren Kapiteln und in den Übungen folgen. Die für Petrinetze zur Verfügung stehenden Analysemethoden sind vielfältig wie wir aber erst in den nachfolgenden Kapiteln sehen werden.

25.1 Konzept des Petrinetz

25.1.1 Das Versprechen der Petrinetze

Zur Geschichte der Petrinetze. 25-4

- Petrinetze wurden von *Carl Adam Petri* in seiner Dissertation eingeführt.
- Zunächst wurden nur *Low-Level-Netze* sowie *Stellen-/Transitionssysteme* untersucht (was wir auch tun werden).
- Zu Spezifikationszwecken sind die S/T-Systeme oft zu einfach. Deshalb wurden *immer mächtigere* Netze eingeführt.
- Mächtige Netze wie *algebraische High-Level-Netze* sind aber nicht mehr automatisch analysierbar.

Die Versprechen zu Petrinetzen. 25-5

- *Einfache Beschreibungssprache.* Petrinetze lassen sich leicht verstehen, leicht zeichnen und leicht benutzen.
- *Universalität.* Man kann »alles und jedes« mit Petrinetzen beschreiben.
- *Analysierbarkeit.* Low-Level-Netze sind mathematisch einfach genug, dass man viele Eigenschaften automatisch überprüfen kann.
- *Vergleichbarkeit.* Low-Level-Netze sind mathematisch einfach genug, dass man sie mit anderen Formalismen (wie State-Charts) automatisch vergleichen kann.

25.1.2 Aufbau von Petrinetzen

Die Bestandteile eines Petrinetzes. 25-6

Die zentralen Bestandteile von Petrinetzen sind:

- *Marken.* Sie repräsentieren Prozesse oder Ressourcen. Bei Low-Level-Netzen haben die Marken keine weitere innere Struktur, insbesondere haben sie keine Identität.
- *Stellen.* Sie sind die Zustände in denen Marken (also Prozesse oder Ressourcen) seien können.
  Ein Netz kann mehrere Marken gleichzeitig enthalten und diese können auf verschiedenen Stellen sein, also verschiedene Zuständen haben.
  Im Vergleich dazu ist ein endlicher Automat immer nur in einem Zustand.
- *Transitionen.* Sie erlauben Übergänge von Marken von einem Zustand in einen anderen.
  Da ein Petrinetz mehrere Marken enthalten kann, die dann verschiedene Transitionen benutzen können, lassen sich Dinge wie Konflikte oder Synchronisation modellieren.

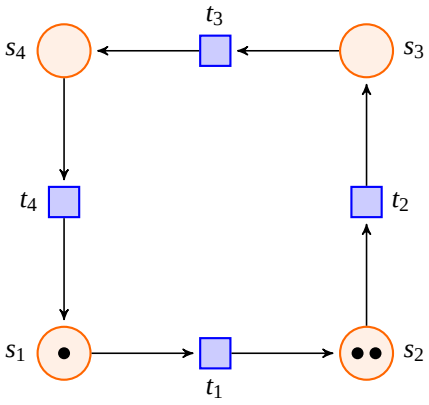
Zur Benutzung von Petrinetzen. 25-7

Wie man sie aufschreibt.

- Stellen werden durch große offene Kreise dargestellt.
- Marken werden durch kleine Kreise dargestellt, die in die Stellen gezeichnet werden.
- Transitionen werden als Rechtecke dargestellt, manchmal eher als Balken, manchmal eher als Quadrate.
- Gerichtete Kanten führen von Stellen zu Transitionen und von Transitionen zu Stellen.
- Kanten und Stellen können gewichtet sein.

25-8

Beispiel eines Petrinetzes



25-9

Zur Benutzung von Petrinetzen.

Wie man sie benutzt.

- Ist ein Netz mit eingezeichneten Marken gegeben, so kann man beginnen, *Transitionen zu schalten*:
  1. Man wählt eine Transition aus, für die für jede eingehende Kante auf der Stelle, von der die Kante kommt, eine Marke vorliegt (bei höher gewichteten Kanten entsprechend mehrere Marken).
  2. Diese vorliegenden Marken werden entfernt.
  3. Zu jeder Stelle, zu der eine von der Transition ausführende Kanten zeigt, fügt man eine Marke hinzu (bei höher gewichteten Kanten auch mehrere).
- Alle möglichen Arten, wie sich das Netz durch Schaltfolgen entwickeln kann, bilden die *Semantik* des Netzes.

25.1.3 Beispiel: Semaphore

25-10

Problemstellung: Mutex

Problemstellung

- Wie wollen ein Mutex mit Hilfe eines Petrinetzes modellieren.
- Gegeben sind zwei Prozesse p_1 und p_2, die jeweils aus mehreren Threads bestehen.
- Es gibt zwei *kritische Abschnitte* c_1 und c_2.
- Wir wollen durch ein Mutex garantieren, dass immer höchstens ein Thread in einem der beiden kritischen Abschnitte ist.

25-11

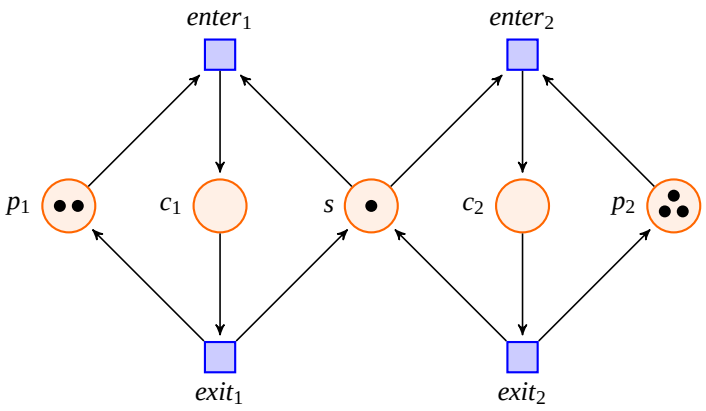
Modellierung des Mutex.

Modellierung

- Threads werden durch Marken repräsentiert.
- Threads können in zwei Zuständen sein: Normal (p_1 und p_2) oder kritisch (c_1 und c_2).
- Der Mutex ist eine Ressource, also eine Marke.
- Es gibt nur einen Zustand für diese Marke, nämlich »vorhanden«, also gibt es nur eine Stelle.

Beispiel eines Petrinetzes, das ein Mutex simuliert.

25-12



Belegt man s mit mehr als einer Marke vor, so erhält man eine Semaphore statt eines Mutex.

Zur Übung

25-13

Im Netz verschwindet die Mutex-Ressource und taucht wieder.
Dies soll nun anders modelliert werden, so dass die Mutex-Ressource zwischen zwei Zuständen »jemand im kritischen Bereich« und »niemand im kritischen Bereich« wechselt.
Wie sieht das Netz hierzu aus?

Zur Übung

Verändern Sie das Netz, so dass bis zu fünf Threads eines Prozesses im kritischen Bereich seien können, aber nie gleichzeitig zwei Threads von unterschiedlichen Prozessen in den beiden kritischen Bereichen seien können.

25.1.4 Beispiel: Scheduling

Problemstellung: Scheduling

25-14

Problemstellung

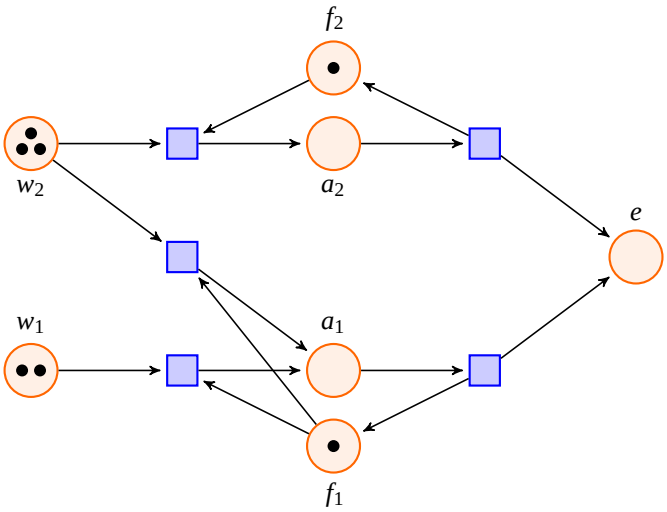
- Ein Scheduler soll Prozesse auf einer Zwei-Core-Maschine auf die Prozessorkerne verteilen.
- Manche Prozesse können nur auf dem ersten Core laufen, andere auf beiden.

Modellierung

- Prozesse werden durch Marken repräsentiert.
- Prozesse können in folgenden Zuständen sein: Wartend auf Core 1, wartend auf einen beliebigen Core, ausgeführt auf Core 1, ausgeführt auf Core 2, beendet.
- Damit höchstens ein Prozess gleichzeitig auf einem Core ausgeführt werden kann, führen wir noch eine Stelle mit einer »Core-ist-frei-Marke« ein.

25-15

Beispiel des Schedulers.



25.1.5 Ziele der Petrinetztheorie

25-16

Ziele der Petrinetztheorie

Ziel der *Theorie der Petrinetze* ist

- das *Verhalten* von Petrinetzen *vorherzusagen*,
- durch *automatisierbare Analysen*.

Beispielsweise möchte man *mittels eines Computerprogramms* Fragen beantworten können wie

- »Ist es wahr, dass auf Stellen c_1 und c_2 insgesamt immer höchstens eine Marke liegt?«
- »Ist es wahr, dass früher oder später alle Marken auf der Stelle e liegen?«
- »Ist es wahr, dass irgendeine Transition immer schalten kann?«
- »Ist es wahr, dass eine bestimmte Situation nie eintreten kann?«

25.2 Formalisierung von Petrinetzen

25.2.1 Netze

25-17

Die formale Definition von Netzen.

Definition

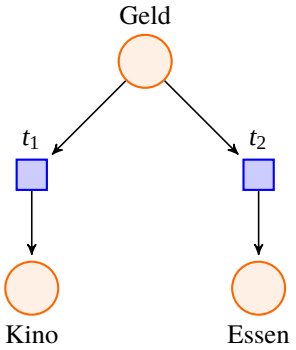
Ein *(Petri-)Netz* $N = (S, T, F)$ besteht aus

- einer Menge S von *Stellen*
- einer Menge T von *Transitionen* (disjunkt zu S)
- einer *Flussrelation* $F \subseteq (S \times T) \cup (T \times S)$.

Die Elemente von $S \cup T$ heißen *Elemente* von N.

Zur Übung 25-18

Geben Sie die Mengen S, T und die Flussrelation von folgendem Netz an:



25.2.2 Vor- und Nachbereich

Vor- und Nachbereiche von Stellen und Transitionen. 25-19

Definition

Sei $N = (S,T,F)$ ein Netz und $x \in S \cup T$ ein Element.

- Der *Vorbereich* von x ist die Menge ${}^\bullet x = \{y \in S \cup T \mid (y,x) \in F\}$.
- Der *Nachbereich* von x ist die Menge $x^\bullet = \{y \in S \cup T \mid (x,y) \in F\}$.

Für eine Menge $X \subseteq S \cup T$ ist ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$ und $X^\bullet = \bigcup_{x \in X} x^\bullet$.

Zur Übung

Geben Sie $t_1^\bullet$ und ${}^\bullet t_2$ sowie Geld$^\bullet$ an.

25.2.3 Markierungen

Das Konzept der Markierung. 25-20

Definition

Eine *Markierung* eines Netzes $N = (S,T,F)$ ist eine Funktion $m\colon S \to \mathbb{N}$.

Definition (Operationen auf Markierungen)

Seien m_1 und m_2 Markierungen.

- Die Markierung $m_1 + m_2$ ist definiert durch $(m_1 + m_2)(s) = m_1(s) + m_2(s)$.
- Wir schreiben $m_1 \leq m_2$, falls $m_1(s) \leq m_2(s)$ für alle $s \in S$ gilt.
- Wir schreiben $m_1 < m_2$, falls $m_1 \leq m_2$ und $m_1 \neq m_2$ gilt.

Zur Diskussion

Was bedeuten $m_1 + m_2$ und $m_1 \leq m_2$ anschaulich?

25.2.4 S/T-Systeme

25-21

Die formale Definition von S/T-Systemen.

Definition

Ein *S/T-System* besteht aus

- einem Netz $N = (S, T, F)$.
- einer Kantengewichtung $W\colon (S \times T) \cup (T \times S) \to \mathbb{N}$. Dabei soll $W(x,y) = 0$ genau dann gelten, wenn $(x,y) \notin F$.
- einer Anfangsmarkierung m_0.

Genauer sind dies S/T-Systeme ohne Kapazitäten.

25.2.5 Erreichbarkeitsgraph

25-22

Aktivierung und Schalten von Transitionen.

Definition

Sei Σ ein S/T-System und sei m eine Markierung und t eine Transition.

- t heißt *aktiviert für* m, falls für alle $s \in {}^\bullet t$ gilt
$$m(s) \geq W(s,t).$$
- Ist t aktiviert für m, so kann t *von* m *nach* m' *schalten* mit
$$m'(s) = m(s) + W(t,s) - W(s,t).$$

Wir schreiben dann $m \to_t m'$.

25-23

Erreichbarkeitsgraph eines S/T-Systems.

Definition

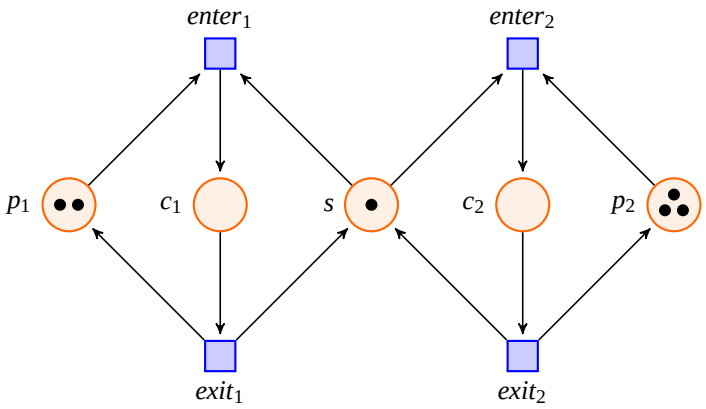
Sei Σ ein S/T-System.

- Der *Markierungsgraph* ist der unendliche Graph, der alle Markierungen als Knoten hat und der eine Kanten von einer Markierung m zu einer Markierung m' mit dem Label t hat, wenn $m \to_t m'$ gilt.
- Der *Erreichbarkeitsgraph* von Σ ist die Einschränkung des Markierungsgraphen auf die Menge aller Knoten (Markierungen), die von der Startmarkierung m_0 erreichbar sind.
  Die Menge der Knoten im Erreichbarkeitsgraph bezeichnet man auch mit $[m_0\rangle$.

25-24

Zur Übung

Geben Sie den Erreichbarkeitsgraph des Netzes an:



Zusammenfassung dieses Kapitels

1. Petrinetze nutzt man zur *Modellierung* von Systemen mit mehreren Prozessen und Ressourcen.
2. *Petrinetze* bestehen aus Stellen, Transitionen und einer Flussrelation.
3. Ein S/T-System ist ein Petrinetz zusammen mit einer Anfangsmarkierung und Kantengewichten.
4. Der *Erreichbarkeitsgraph* beschreibt das mögliche Verhalten des Netzes.

Übungen zu diesem Kapitel

Übung 25.1 (Modellierung von Sicherheitseigenschaften, leicht)

Zwei sich kreuzende Einbahnstraßen sind mit jeweils einer Ampel ausgestattet sind. Jede Ampel durchläuft periodisch die Zustände *rot, gelb, grün, gelb*. Modellieren Sie die Kreuzung als S/T-System, wobei folgende Bedingungen zu erfüllen sind:

1. Die beiden Ampeln dürfen niemals gleichzeitig *grün* sein.
2. Die beiden Ampeln dürfen niemals gleichzeitig *rot* sein.

Analysieren Sie die Korrektheit ihres S/T-Systems, indem Sie seinen Erreichbarkeitsgraphen konstruieren.

Übung 25.2 (Modellierung von Langtons Ameisen, mittel)

Langtons Ameise lebt auf einem unendlichen Gitter (von dem wir nur einen endlichen Ausschnitt betrachten). Auf ihrem Rücken befindet sich ein unendlich großer Rucksack, der unendliche viele kleine Steine enthält. Anfänglich ist das Gitter leer und die Ameise sitzt auf einem Feld und schaut nach Norden. Nun wiederholt sie folgendes: Sie läuft in das angrenzende Feld in der Richtung, in die sie gerade schaut. Ist dieses Feld leer, so legt sie dort einen Stein aus ihrem Rucksack ab und dreht sich nach rechts. Ist auf dem Feld bereits ein Stein, so verfrachtet sie den Stein in ihren Rucksack und dreht sich nach links.

Modellierung Sie Langtons Ameise für eine feste Gittergröße als ein S/T-System.

Übung 25.3 (Petrinetze und reguläre Sprachen, mittel)

Sei $\Sigma = (N, W, m_0)$ ein S/T-System bestehend aus einem Netz $N = (S, T, F)$, einer Kantengewichtung W mit $W(x,y) = 1$ für alle $(x,y) \in F$ und einer Anfangsmarkierung m_0. Das Netz besitzt Stellen $S = \{s_1, \ldots, s_n\}$ und Transitionen $T = \{t_1, \ldots, t_m\}$. Das S/T-System Σ definiert folgende formale Sprache:

$$L(\Sigma) = \{t_{i_1} \ldots t_{i_\ell} \in T^* \mid \exists m_1, \ldots, m_\ell : m_0 \to_{t_{i_1}} m_1 \to_{t_{i_2}} m_2 \to_{t_{i_3}} \cdots \to_{t_{i_\ell}} m_\ell\} \ .$$

Beispiel: Sei $\Sigma = (N, W, m_0)$ ein S/T-System mit $N = (S, T, F)$, $S = \{s_1, s_2\}$, $T = \{\mathtt{a}, \mathtt{b}\}$, $F = \{(s_1, \mathtt{a}), (\mathtt{a}, s_2), (s_2, \mathtt{b}), (\mathtt{b}, s_1)\}$ und $m_0(s_1) = 1$, $m_0(s_2) = 0$. Dann ist die Sprache $L(\Sigma) = \{\mathtt{a}, \mathtt{ab}, \mathtt{aba}, \mathtt{abab}, \ldots\}$.

1. Geben Sie ein S/T-System Σ an, so dass $L(\Sigma)$ keine reguläre Sprache ist. (Wenn Ihnen das zu leicht erscheint, geben Sie ein S/T-System an, so dass $L(\Sigma)$ nicht *kontextfrei* ist.)
2. Sei $\Sigma = (N, W, m_0)$ ein S/T-System, in dem für alle $t \in T$ gilt: $|{}^\bullet t| = |t^\bullet| = 1$. Zeigen Sie: $L(\Sigma)$ ist eine reguläre Sprache.

26-1

Kapitel 26
Modellierung und Analysemethoden
Petersons Algorithmus und Beschränktheit

26-2

Lernziele dieses Kapitels

1. Petersons Algorithmus kennen.
2. Diesen und ähnliche Algorithmen als Petrinetz modellieren können.
3. Die Begriffe Tod, Verklemmung, Konflikt, Lebendigkeit, Erreichbarkeit und Beschränktheit kennen.
4. Entscheidungsalgorithmus für Beschränktheit kennen.

Inhalte dieses Kapitels

Worum es heute geht

Jemand zeigt Ihnen einen kurzen Programmtext und behauptet, das Programm »mache dieses und jenes«. Kann man dies überprüfen, vorzugsweise automatisch? Wenn Sie in Theoretischer Informatik aufgepasst haben, wissen Sie, dass dies nicht automatisch möglich ist (dies ist im Wesentlichen die Aussage des Satzes von Rice). Das heißt aber nicht, dass man nicht für ein kurzes konkretes Programm mit einem geeigneten Formalismus wie dem Hoare-Kalkül doch eine Verifikation durchführen kann; es ist aber etwas anstrengend (wenn Sie dies noch nie selbst getan haben, fragen Sie mal jemanden, der zwei Seiten Programmtext Hoare-verfizieren durfte). Bei Algorithmen für verteilte Systeme sieht die Verifikationsgroßwetterlage zunächst noch viel düsterer aus. In der Tat muss man erstmal darüber nachdenken, wie überhaupt ein Formalismus zur Verifikation verteilter Systeme aussehen soll – der Hoare-Kalkül beispielsweise modelliert die Komplexität einer asynchronen Kommunikation mehrerer Prozessoren nur unzureichend.

Petrinetze bieten einen Ansatz zur Verifikation verteilter Algorithmen. Dabei gibt es vielfältige Einschränkungen, insbesondere dürfen die vorkommenden Variablen in der Regel nur kleine Werte annehmen, jedoch ist es prinzipiell möglich, viele verteilte Algorithmen als Petrinetze zu modellieren. Wir werden in diesem Kapitel für einen einfachen Algorithmus (Petersons Algorithmus), dessen Korrektheit aber nicht offensichtlich ist, ein modellierendes Netz bauen.

Sobald die Beispiel auch nur etwas komplexer werden, werden die Netze schnell sehr groß. Netze mit Tausenden von Stellen und Transitionen sind da keine Seltenheit, automatisch generierte Netze können auch Millionen von Stellen haben. Solche Netze wird man nicht

mehr »per Hand« untersuchen können. Deshalb benötigen wir möglichst gute *automatische Analysemethoden*.

Es ist zunächst nicht klar, ob es automatische Analysemethoden überhaupt gibt. Schließlich könnten Petrinetze Turing-mächtig sein und somit prinzipiell nicht automatisch analysierbar. Dies ist aber nicht der Fall. Petrinetze *können* algorithmisch analysiert werden. Wie dies geht werden wir uns in diesem und in den nächsten beiden Kapiteln genauer anschauen.

26.1 Petersons Algorithmus

26.1.1 Problemstellung

Ein Synchronisationsproblem. 26-4

Problemstellung

- Zwei Prozesse enthalten einen *großen kritischen Bereich*, der gleichzeitig immer nur von einem Prozess bearbeitet werden kann.
- Gesucht ist ein Algorithmus für ein *asynchrones Shared-Memory-Modell*, der dies gewährleistet.

Ein simpler Algorithmus. 26-5

- Wir führen eine Bit-Variable `mutex_available` ein.
- Algorithmus der Prozesse:

```
while (true) {
  do_normal_stuff ();         // Line 1
  await (mutex_available);    // Line 2
  mutex_available = false;    // Line 3
  do_critical_stuff ();       // Line 4
  mutex_available = true;     // Line 5
}
```

- Offenbar müssen Zeilen 2 und 3 als gemeinsamer atomaren Block ausgeführt werden.

26.1.2 Der Algorithmus

Vom einfachen Algorithmus zu Petersons Algorithmus. 26-6

Nachteile des einfachen Algorithmus

- Es ist unschön, dass wir einen »großen atomaren Block« brauchen.
- Der Algorithmus ist *unfair*, weil Prozesse *aushungern* können.

Vorteile von Petersons Algorithmus

- Petersons Algorithmus benötigt keinen großen atomaren Block, sondern nur atomare Zuweisungen.
- Petersons Algorithmus ist fair: Jeder Prozess, der will, kommt garantiert dran.

26-7

Petersons Algorithmus.

- Es gibt drei Variablen: Bits m_1 und m_2, beide initial *false*, und $hold \in \{1,2\}$, initial beliebig.
- Der erste Prozess führt folgenden Code aus:

```
while (true) {
  do_normal_stuff ();          // Line p1
  m_1  = true;                 // Line p2
  hold = 1;                    // Line p3
  await (!m_2 || hold == 2);   // Line p4
  do_critical ();              // Line p5
  m_1  = false;                // Line p6
}
```

- Der zweite Prozess führt folgenden Code aus:

```
while (true) {
  do_normal_stuff ();          // Line q1
  m_2  = true;                 // Line q2
  hold = 2;                    // Line q3
  await (!m_1 || hold == 1);   // Line q4
  do_critical ();              // Line q5
  m_2  = false;                // Line q6
}
```

26.1.3 Modellierung als Petrinetz

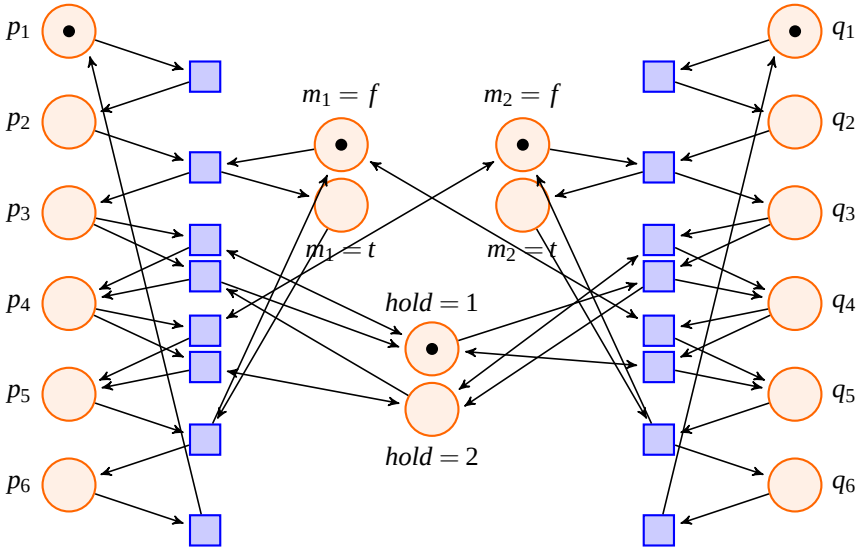
26-8

Vom Algorithmus zum S/T-System.

- Die *Analyse* des Algorithmus ist nicht ganz einfach.
- Wir wollen deshalb den *Algorithmus als Petrinetz* modellieren und dann dieses *automatisch analysieren*.
- Dazu gehen wir wie folgt vor:
  - Jeder Prozess wird durch eine Marke repräsentiert.
  - Die Zeilen, in denen Prozess p sein kann, sind Zustände (also Stellen im Netz).
  - Ebenso für Prozess q und die Zeilen von q.
  - Der Wert jeder Variable wird ebenfalls durch eine Marke repräsentiert, die auf zwei Stellen sein kann.
  - Transitionen sind die Übergänge von einer Zeile zur nächsten.

26-9

Das S/T-System zum Peterson-Algorithmus.



Was kann man über den Algorithmus durch Netzanalyse herausbekommen? 26-10

- Der Erreichbarkeitsgraph ist *endlich*.
- Durch Analyse des Graphen kann man zum Beispiel zeigen:
  1. Es gibt keine Deadlocks.
  2. Der Algorithmus ist *korrekt*: In p_5 und q_5 befindet sich insgesamt immer nur eine Marke.
  3. Der Algorithmus ist *fair*: Jeder Prozess kann jederzeit von p_1 nach p_2 oder von q_1 nach q_2 wechseln und kommt dann nach einiger Zeit *garantiert* dran.

26.2 Grundbegriffe

26.2.1 Die Begriffe

Wichtige Begriffe zu S/T-Systemen. 26-11

Begriffe für Transitionen und Markierungen.

aktiviert Eine *Transition* heißt in einer Markierung m *aktiviert*, wenn sie schalten könnte.

tot Eine *Markierung* ist *tot* oder *verklemmt*, wenn keine Transition in ihr schalten kann.

Konflikt Zwei *Transitionen* stehen *in Konflikt* in einer Markierung m, wenn sie beide aktiviert sind, aber durch Wegnehmen der Marken für eine von ihnen die andere nicht mehr schalten kann:

$$\exists s \in {}^\bullet t_1 \cap {}^\bullet t_2 : m(s) < W(s,t_1) + W(s,t_2)$$

Zur Übung

Geben Sie ein S/T-System an, in dem zwei Transitionen in Konflikt stehen, sie aber direkt nacheinander schalten können.

Wichtige Begriffe zu S/T-Systemen. 26-12

Begriffe für S/T-Systeme.

Sei $\Sigma = (N, W, m_0)$ ein S/T-System.

verklemmungsfrei Σ ist *verklemmungsfrei*, wenn es keine toten/verklemmten erreichbare Markierungen gibt.

lebendig Σ ist *lebendig*, wenn für jede Transition t und jede erreichbare Markierung m eine Markierung m' erreichbar ist, die t aktiviert.
(Jede Transition kann immer wieder aktiviert werden.)

beschränkt Σ heißt *b-beschränkt*, wenn es eine Zahl b gibt, so dass keine erreichbare Markierung mehr als b Marken auf einer Stelle aufweist. Es heißt *beschränkt*, wenn es b-beschränkt ist für irgendein b.

26-13

Zur Übung
Geben Sie S/T-Systeme mit folgenden Eigenschaften an:

1. verklemmungsfrei und nicht lebendig
2. unbeschränkt und nicht lebendig
3. beschränkt und nicht verklemmungsfrei

26.2.2 Beziehungen

26-14

Einfache Beziehungen zwischen den Eigenschaften.

Lemma
Jedes lebendige S/T-System ist verklemmungsfrei.

Lemma
Jedes beschränkte S/T-System ist b-beschränkt für eine Zahl b.

Lemma
Ein S/T-System ist genau dann beschränkt, wenn sein Erreichbarkeitsgraph endlich ist.

26.3 Beschränktheit

26.3.1 Behauptung: Beschränktheit ist entscheidbar

26-15

Ein einfacher Satz.

Satz
Es ist entscheidbar, ob ein S/T-System b-beschränkt ist für ein gegebenes b.

Beweis. Ein b-beschränktes S/T-System mit n Stellen hat nur $(b+1)^n$ verschiedene mögliche Markierungen. Der Algorithmus arbeitet wie folgt:

1. Erstelle den Erreichbarkeitsgraph.
2. Falls dabei herauskommt, dass alle Markierungen in dem Graphen b-beschränkt sind, akzeptiere.
3. Falls eine erreichbare Markierung gefunden wird, die nicht b-beschränkt ist, verwerfe.

Der Algorithmus hat Laufzeit $O\big((b+1)^n\big)$. □

26-16

Beschränktheit ist automatisch analysierbar.

Satz
Es ist entscheidbar, ob ein S/T-System beschränkt ist.

- Man kann nicht wie bei b-Beschränktheit verfahren, da man nicht weiß, wann man aufhören soll.
- Wir werden deshalb ein wenig Theorie brauchen, um ein Kriterium zu erhalten, wann wir abbrechen können.

26.3.2 Beweis: Das Monotonielemma

Ein paar nützliche Notationen.

26-17

- Wir hatten schon definiert, dass wir $m \to_t m'$ schreiben, falls man von m durch Schalten von t nach m' kommt.
- Für eine Folge $\sigma = t_1 \dots t_n$ von Transitionen schreiben wir $m \to_\sigma m'$, falls Markierungen m_i existieren mit

$$m \to_{t_1} m_1 \to_{t_2} m_2 \to_{t_3} \cdots \to_{t_{n-1}} m_{m-1} \to_{t_n} m'$$

  Wie üblich gilt $m \to_\varepsilon m'$ genau dann, wenn $m = m'$.
- Für eine Zahl i schreiben wir $m \to^i m$, falls eine Folge σ der Länge i existiert mit $m \to_\sigma m$.
- Wir schreiben $m \to^* m'$, falls $m \to^i m$ für ein i gilt.

Das Monotonielemma.

26-18

Lemma
Sei Σ ein S/T-System, k eine Markierung und $m \to_\sigma m'$. Dann gilt $m+k \to_\sigma m'+k$.

Beweis. Per Induktion über die Länge von σ. Ist σ leer, so stimmt die Behauptung trivialerweise. Sei nun $\sigma = \sigma' t$ und $m \to_{\sigma'} m'' \to_t m'$. Nach Induktionsvoraussetzung gilt $m+k \to_{\sigma'} m''+k$. Da aus $m'' \to_t m'$ aber $m''+k \to_t m'+k$ folgt, erhalten wir $m+k \to_\sigma m'+k$. □

26.3.3 Beweis: Das Antikettenlemma

Das Antikettenlemma.

26-19

Satz
Sei $n \geq 1$. Dann hat die partielle Ordnung $(\mathbb{N}^n, \leq)$ keine unendliche Antikette (Menge von paarweise unvergleichbaren Elementen).

Zum Beweis siehe Übungsaufgabe 26.2.

26.3.4 Beweis: Charakterisierungslemma

Eine Charakterisierung von Beschränktheit.

26-20

Lemma
Ein S/T-System Σ ist genau dann unbeschränkt, wenn es eine Markierung $k \neq 0$ und eine erreichbare Markierung m gibt mit $m \to^* m+k$.

Beweis: $\Leftarrow$
Es gelte $m \to^* m+k$. Nach dem Monotonielemma gilt dann auch $m+k \to^* m+2k$ und damit $m+2k \to^* m+3k$ und so weiter.
Wir erhalten also die folgende Schaltfolge:

$$m_0 \to^* m \to^* m+k \to^* m+2k \to^* m+3k \to^* \cdots$$

Wegen $k \neq 0$ ist dies eine unendliche Folge von erreichbaren Markierungen. Der Erreichbarkeitsgraph ist also unendlich und das S/T-System somit unbeschränkt.

Beweis: $\Rightarrow$
Sei Σ unbeschränkt. Dann ist der Erreichbarkeitsgraph unendlich und enthält somit einen unendliche Weg

$$m_0 \to m_1 \to m_2 \to m_3 \to \cdots,$$

weshalb die Menge $M = \{m_i \mid i \geq 0\}$ unendlich ist.
In der Menge $M \subseteq \mathbb{N}^n$ muss es zwei Elemente m_i und m_j geben mit $m_i \leq m_j$ und $m_i \neq m_j$.
Dann gilt für $m = m_i$ und $k = m_j - m_i$ das Gewünschte.

26.3.5 Beweis: Entscheidungsalgorithmus

26-21

Der Entscheidungsalgorithmus

Algorithmus für Beschränktheit

1. Baue den Erreichbarkeitsgraphen per Breitensuche auf.
2. Ist er endlich, akzeptiere.
3. Findet man erreichbare Markierungen m und $m+k$ für ein Markierung $k \neq 0$, so verwerfe.

Nach dem Charakterisierungslemma ist der Algorithmus *korrekt und hält immer an*.

Zusammenfassung dieses Kapitels

26-22

1. Bestimmte verteilte Algorithmen kann man mit Hilfe man Petrinetzen *modellieren*.
2. Die *Analyse* von Petrinetzen bedeutet, Eigenschaften wie *Lebendigkeit*, *Verklemmungsfreiheit* oder *Beschränktheit* zu überprüfen.
3. Beschränktheit ist *entscheidbar*.

Übungen zu diesem Kapitel

Übung 26.1 (Beweis des vereinfachten Antikettenlemmas, mittel)

Beweisen Sie das Antikettenlemma (der Satz von Projektion 26-19) für $n = 2$.

Tipps: Machen Sie sich dazu zunächst an Beispielen klar, was die Behauptung genau ist. Dazu empfiehlt es sich, die Teilmengen von $\mathbb{N}^2$ in der Ebene zu zeichnen.

Übung 26.2 (Beweis des Antikettenlemmas, schwer)

Beweisen Sie das Antikettenlemma (den Satz von Projektion 26-19) für alle n.

Tipps: Eine Induktion über n. Für den Induktionsschritt beginnen Sie mit einer unendlichen Teilmenge von $\mathbb{N}^n$. Betrachten Sie die Projektion dieser Menge auf die ersten $n-1$ Koordinaten. Auf diese Projektion können Sie die Induktionsvoraussetzung anwenden. Es ist könnte auch nützlich sein, dass bei jeder Aufteilung einer unendlichen Menge in endlich viele Teilmengen einer der Teilmengen unendlich sein muss.

Übung 26.3 (Modellierung eines Algorithmus, mittel)

Betrachten Sie folgenden Algorithmus zum gegenseitigen Ausschluss, der von mehreren Prozessen parallel ausgeführt wird:

```
while (true) {
  do_normal_stuff ();         // Line 1
  await (mutex_available);    // Line 2
  mutex_available = false;    // Line 3
  do_critical_stuff ();       // Line 4
  mutex_available = true;     // Line 5
}
```

Auf die boolesche Variable `mutex_available` haben alle Prozessoren sowohl Lese- als auch Schreibzugriff. Jeder Prozessor benötigt für die Ausführung jeder der fünf Zeilen eine unbestimmte, veränderliche Zeit.

Modellieren Sie den Algorithmus als S/T-System. Führen Sie dazu fünf Stellen ein, die die verschiedenen Zeilen repräsentieren, in denen ein Prozessor sein kann. Jeder Prozessor wird durch eine Marke repräsentiert. Modellieren Sie die Variable `mutex_available` durch zwei Stellen.

Nachdem Sie das System modelliert haben, zeigen Sie, dass der Algorithmus nicht korrekt ist, indem Sie eine Schaltfolge angeben, die zwei Prozessoren gleichzeitig in den kritischen Bereich führt.

Übung 26.4 (Modellierung eines Algorithmus, mittel)

In einem Producer-Consumer-Szenario kooperieren zwei Prozesse. Der erste Prozess, der Producer, produziert Daten (beispielsweise Messwerte oder auch einen von der Festplatte gelesenen Sektor). Der zweite Prozess, der Consumer, verarbeitet diese Daten.

Die beiden Prozesse sollen durch einen Puffer kommunizieren, der nur ein Datum aufnehmen kann. Der Pseudo-Code der Prozesse sieht wie folgt aus:

```
// Producer
while (true) {
  data = produce_data ();          // Line p1
  puffer_filled = true;            // Line p2
  await (puffer_filled == false); // Line p3
}

// Consumer
while (true) {
  await (puffer_filled == true);  // Line c1
  consume_data (data);            // Line c2
  puffer_filled = false;          // Line c3
}
```

Auf die boolesche Variable `puffer_filled`, die initial `false` ist, haben beide Prozessoren sowohl Lese- als auch Schreibzugriff. Jeder Prozessor benötigt für die Ausführung jeder der Zeilen eine unbestimmte, veränderliche Zeit.

Modellieren Sie den Code als Petrinetz. Führen Sie dazu sechs Stellen ein, die die verschiedenen Zeilen repräsentieren, in denen die Prozessoren sein können. Jeder Prozessor wird durch eine Marke repräsentiert. Modellieren Sie die Variable `puffer_filled` durch zwei Stellen.

Übung 26.5 (Erreichbarkeitsgraph bestimmen, mittel)

Geben Sie den Erreichbarkeitsgraph des Netzes aus Übung 26.4 an.

Übung 26.6 (Modellierung eines komplexeren Algorithmus, schwer)

Wir erweitern nun das Szenario aus der ersten Aufgabe wie folgt: Der Producer produziert Daten, die von zwei Consumern verarbeitet werden. Mit dem Datum gibt der Producer an, welcher Consumer dies sein soll.

```
// Producer
while (true) {
  data = produce_data ();              // Line p1
  puffer_filled = true;                // Line p2
  dispatch_to = data.target;           // Line p3
  await (puffer_filled == false);      // Line p4
}

// Consumer 1
while (true) {
  await (puffer_filled == true);       // Line c1
  if (dispatch_to == 1) {              // Line c2
    consume_data (data);               // Line c3
    puffer_filled = false;             // Line c4
  }
}

// Consumer 2
while (true) {
  await (puffer_filled == true);       // Line d1
  if (dispatch_to == 2) {              // Line d2
    consume_data (data);               // Line d3
    puffer_filled = false;             // Line d4
  }
}
```

Modellieren Sie den Code wieder als Petrinetz. Zeigen Sie dann, dass der Code fehlerhaft ist, da es eine Schaltfolge gibt, in der der falsche Consumer ein Datum verarbeitet. Wie muss man den Code korrigieren?

27-1

Kapitel 27
Stelleninvarianten

Stelleninvariante = konstante gewichtete Summe der Marken

27-2

Lernziele dieses Kapitels

1. Linear-algebraische Darstellung von Petrinetzen kennen
2. Konzept der Stelleninvariante kennen
3. Anwendungen von Invarianten in der Analyse kennen

Inhalte dieses Kapitels

Worum es heute geht

Das Ergebnis vom Ende des letzten Kapitels nach dem die Beschränktheit von Petrinetzen entscheidbar ist, ist sicherlich ein sehr schönes Ergebnis. Leider lässt sich aber auch nicht viel mehr über die Beschränktheit aussagen, als dass sie prinzipiell entscheidbar ist; effizient war unser Algorithmus hierfür nämlich nicht. Um Petrinetze und damit verteilte Systeme algorithmisch tatsächlich zu analysieren, brauchen wir effizientere Hilfsmittel.

Wenn ein Problem zu schwierig zu lösen scheint, dann sollte man es vereinfachen. Genau dies werden wir in diesem Kapitel bei der Analyse von Petrinetzen machen: Statt der Petrinetze selber werden wir ihre so genannten *Systemmatrizen* untersuchen. Diese enthalten viel, aber nicht alle Information über das Netz. Wie der Name System*matrix* suggeriert können wir auf diese Matrizen die Methoden der Linearen Algebra anwenden – und Sie erinnern sich sicherlich daran, dass die Lineare Algebra mit einem wahrlich reichhaltigen Fundus an Analysemethoden aufwarten kann.

Das Konzept, das diesem Kapitel den Namen gegeben hat, ist ein Beispiel einer Eigenschaft von Petrinetzen, die in der Systemmatrix einer linearalgebraischen Eigenschaft entspricht: Eine Stelleninvariante ist eine gewichtete Summe von Stellen, so dass die Summe bei allen möglichen Schaltvorgängen gleich (sprich: invariant) bleibt. Übersetzt in die Sprache der Linearen Algebra ist eine Stelleninvariante gerade ein Element des Kerns der transponierten Systemmatrix. Wohingegen man nicht leicht sieht, wie man gewichtete Summen von Stellen finden soll, deren Summe invariant unter allen Schaltvorgängen ist, ist es natürlich einfach, den Kern einer Matrix zu berechnen.

27.1 Linear-Algebraische Darstellung

Worum es heute geht. 27-4

- Wir wollen wieder Eigenschaften von Petrinetzen *analysieren*.
- Wir werden dazu Methoden aus der *Linearen Algebra* benutzen.
- Wir werden uns mit *notwendigen* oder auch mit *hinreichenden* Bedingungen zufrieden geben.
  Dies sind Aussagen wie »Wenn XYZ gilt, dann ist das Netz lebendig. Wenn XYZ nicht gilt, dann weiß man nicht, was los ist.«

27.1.1 Crashkurs Lineare Algebra

Zur Erinnerung: Definition von Matrizen. 27-5

Definition (Matrix)
Eine $(n \times m)$-*Matrix* A hat n Zeilen und m Spalten, in denen Zahlen stehen. Wir schreiben $A_{i,j}$ für die Zahl in Zeile i und Spalte j.

Definition (Transposition)
Die *Transponierte* A^{T} einer Matrix A erhält man, indem man Zeilen und Spalten vertauscht.

Definition (Vektor)
Ein *Vektor* ist eine $(1 \times m)$-Matrix (Zeilenvektor) oder eine $(n \times 1)$-Matrix (Spaltenvektor).

Definition (Vergleich von Vektoren)
Wir schreiben $u \leq v$, wenn u in jeder Komponente kleiner oder gleich v ist. Wir schreiben $u < v$ für $u \leq v$ und $u \neq v$.

Zur Erinnerung: Matrixmultiplikation, Kern, Spann. 27-6

Definition (Matrixmultiplikation)
Sei A eine $(n \times m)$-Matrix und B eine $(m \times l)$-Matrix. Dann ist ihr *Produkt* AB eine $(n \times l)$-Matrix mit $(AB)_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j}$.

Definition (Nullraum, Kern)
Der *Nullraum oder Kern* einer Matrix A ist die Menge der Vektoren x mit $Ax = 0$.

Definition (Spann)
Der *Spann* einer $(n \times m)$-Matrix A ist die Menge $\{Ax \mid x \text{ ist ein Vektor passender Größe}\}$.

Zur Erinnerung: Orthogonalraum. 27-7

Definition (Skalarprodukt)
Für zwei n-Spaltenvektor u und v ist $\langle u, v \rangle = u^{\mathrm{T}} v$ ihr *Skalarprodukt*.
Zwei Vektoren *sind orthogonal*, wenn ihr Skalarprodukt 0 ist.

Definition (Orthogonalraum)
Der *Orthogonalraum* $V^{\perp}$ einer Menge V von Vektoren ist die Menge aller Vektoren, die orthogonal zu allen Vektoren in V sind.

Satz
Für jede Matrix A gilt:

$$(\operatorname{kern} A)^{\perp} = \operatorname{span}(A^{\mathrm{T}}).$$

27-8

Zur Erinnerung: Linear-algebraische Berechnungsprobleme.

Definition (Gleichungsprobleme)
Sei A eine Matrix und b ein Vektor. Sei M die Menge aller Lösungen der Gleichung $Ax = b$, bei der alle Komponenten rational sind. Es ergeben sich folgende Probleme:

1. Ist M nicht leer?
2. Enthält M eine ganzzahlige Lösung?
3. Enthält M eine Lösung mit natürlichen Zahlen?

Definition (Ungleichungsprobleme)
Sei A eine Matrix und b ein Vektor. Wir betrachten nun die Menge M aller Lösungen der Gleichung $Ax \leq b$. Wieder ergeben sich ähnliche Probleme:

1. Ist M nicht leer?
2. Enthält M eine ganzzahlige Lösung?
3. Enthält M eine Lösung mit natürlichen Zahlen?

27-9

Was man über die linear-algebraische Berechnungsprobleme weiß.

- Man kann mit dem *Gauß-Verfahren* in polynomieller Zeit entscheiden, ob $Ax = b$ eine rationale Lösung hat.
- Man kann mit der *Ellipsoid-Methode* in polynomieller Zeit entscheiden, ob $Ax \leq b$ eine rationale Lösung hat.
- Alle anderen genannten Problem sind *NP-vollständig*.

27.1.2 Vom Netz zur Matrix

27-10

Vom Netz zur Matrix.

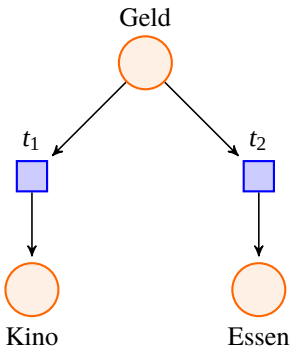
Die Idee.

- Das *Schalten* einer Transition in einem Netz bedeutet, dass
  1. eine feste Anzahl Marken von bestimmten Stellen genommen wird,
  2. eine feste Anzahl Marken zu bestimmten Stellen hinzugefügt wird.
- Fassen wir nun Markierungen m als Vektoren auf, so bedeutet das Schalten $m \to_t m'$ nichts anderes, als $m' = m + \text{token-change}(t)$.
  Hierbei ist token-change(t) ein Vektor, der für jede Stelle angibt, *wie viele Marken durch das Schalten insgesamt zu der Stelle hinzukommen oder entfernt werden*.
- Für ein System Σ fassen wir sämtliche Schaltvektoren in einer *Systemmatrix* N_Σ zusammen. Dies ist eine $(|S| \times |T|)$-Matrix mit

$$(N_\Sigma)_{s,t} = W(t,s) - W(s,t).$$

27-11

Beispiel einer Systemmatrix.



	t_1	t_2
Geld	-1	-1
Kino	1	0
Essen	0	1

Zur Übung 27-12

Geben Sie zwei unterschiedliche Systeme mit der gleichen Systemmatrix an.

27.1.3 Parikh-Vektoren

Schalten mehrerer Transitionen. 27-13

- Das *Schalten* einer Transition bedeutet, eine Spalte der Systemmatrix auf die Markierung zu *addieren*.
- Das *zweifache Schalten* einer Transition bedeutet, eine Spalte der Systemmatrix *doppelt* auf die Markierung zu *addieren*.
- Das *Schalten mehrerer* Transitionen bedeutet, jede Spalte der Systemmatrix *so oft* auf die Markierung zu *addieren, wie sie vorkommt*.

Definition (Parikh-Vektor)

Sei σ ein Schaltwort (Folge von Transitionen). Dann ist der Parikh-Vektor der Spaltenvektor $\vec{\sigma}$ mit

$$\vec{\sigma}_t = \text{Anzahl der Vorkommen von } t \text{ in } \sigma.$$

27.2 Die Markierungsgleichung

27.2.1 Die Gleichung

Eine einfache Beobachtung 27-14

Lemma

Sei Σ ein System und N_Σ seine Systemmatrix. Sei $m \to_\sigma m'$. Dann gilt

$$m' = m + N_\Sigma \vec{\sigma}.$$

Beweis. Eine ganz einfache Induktion. □

Die Markierungsgleichung. 27-15

Definition

Sei $\Sigma = (N, K, m_0)$ ein System. Die *Markierungsgleichung* ist folgende Gleichung:

$$m = m_0 + N_\Sigma x.$$

Hierbei sind m und x Variablen.

27.2.2 Anwendungen in der Analyse

27-16

Notwendiges Kriterium für Erreichbarkeit.

Satz

Sei Σ ein System und m eine Markierung. Falls m erreichbar ist, so hat folgende Gleichung eine Lösung x:

$$m = m_0 + N_\Sigma x$$

Beweis. Dies folgt unmittelbar aus der Definition. □

27-17

Hinreichendes Kriterium für Beschränktheit.

Satz

Sei Σ ein System. Wenn folgendes Optimierungsproblem eine Lösung b hat, so ist das System b-beschränkt:

$$\begin{aligned} &\max \sum_{s \in S} m(s) \\ \text{such that } \quad &m = m_0 + N_\Sigma x \\ &x \geq 0 \end{aligned}$$

Beweis. Ist b eine Lösung, so gilt $b \geq \sum_{s \in S} m(s)$ für alle erreichbaren Markierungen m (denn jede solche ist von der Form $m_0 + N_\Sigma x$ für ein geeignetes $x \geq 0$). Also gilt für jede erreichbare Markierungen m und jede Stelle s auch $b \geq m(s)$. □

27-18

Hinreichendes Kriterium für Verklemmungsfreiheit.

Satz

Sei Σ ein 1-beschränktes System. Wenn folgendes System keine Lösung x hat, so ist das System verklemmungsfrei.

$$\begin{aligned} &m = m_0 + N_\Sigma x \\ &\sum_{s \in {}^\bullet t} m(s) < |{}^\bullet t| \text{ für } t \in T. \end{aligned}$$

Beweis. Durch Kontraposition. Sei m eine tote erreichbare Markierung. Dann gilt $m = m_0 + N_\Sigma x$ für ein x. Da m tot ist, kann es nicht schalten. Da das System 1-beschränkt ist, bedeutet dies gerade $\sum_{s \in {}^\bullet t} m(s) < |{}^\bullet t|$. □

27-19

Zur Übung

Lösen Sie *eine* der folgenden Aufgaben:

1. Geben Sie ein System an und eine unerreichbare Markierung m, so dass $m = m_0 + N_\Sigma x$ eine Lösung hat.
2. Geben Sie ein beschränktes System an, für das das Optimierungsproblem keine Lösung besitzt.
3. Geben Sie ein verklemmungsfreies 1-beschränktes System an, so dass das Gleichungssystem eine Lösung hat.

27.3 Stelleninvarianten

27.3.1 Das Konzept

Die Idee hinter Stelleninvarianten. 27-20

- Während ein System schaltet, *ändern* sich die Markierungen.
- Bestimmte Eigenschaften bleiben aber oft *gleich* oder *invariant*.
- So ist oft die *Summe der Marken* konstant.
- Solche Invarianten haben wichtige Folgen: Ist die Summe der Marken beispielsweise konstant, so ist das System offenbar beschränkt.

Die Definition von Stelleninvarianten. 27-21

Definition

Eine *Stelleninvariante* eines Systems Σ ist ein Spaltenvektor I mit der Eigenschaft

$$N_\Sigma^T I = 0.$$

Beachte: $N_\Sigma^T I = 0$ gilt genau dann, wenn $I^T N_\Sigma = 0$.

Die zentrale Eigenschaft von Stelleninvariante lautet:

Satz

Sei I eine Stelleninvariante eines Systems Σ. Falls $m \to_\sigma m'$, so gilt $\langle I, m' \rangle = \langle I, m \rangle$.

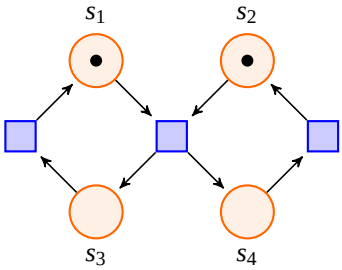
Beweis. $\langle I, m' \rangle = I^T m' = I^T m + \underbrace{I^T N_\Sigma \vec{\sigma}}_{=0} = I^T m = \langle I, m \rangle$. □

Berechnung von Stelleninvarianten. 27-22

- Die Menge der (rationalen) Stelleninvarianten ist gerade der Kern von N_Σ^T.
- Folglich ist die Menge der (rationalen) Stelleninvarianten ein *Vektorraum über dem Körper der rationalen Zahlen*.
- *Eine Basis* dieses Vektorraumes kann durch das Gaußverfahren berechnet werden.
- Es ist aber (leider!) *NP-vollständig*, die *ganzzahligen* Elemente dieses Vektorraumes zu bestimmen (außer dem Nullvektor).

Beispiel zur Berechnung von Stelleninvarianten. 27-23

Das System



Die Systemmatrix

$$\begin{pmatrix} 1 & -1 & 0 \\ 0 & -1 & 1 \\ -1 & 1 & 0 \\ 0 & 1 & -1 \end{pmatrix}$$

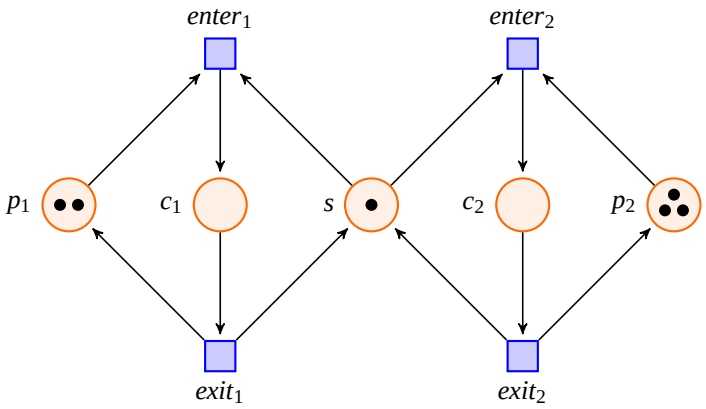
Basis des Kerns von N_Σ^T

$$\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

27-24

Zur Übung

Geben Sie eine S-Invariante des Mutex-Netzes an:



27.3.2 Anwendungen in der Analyse

27-25

Hinreichende Bedingung für Beschränktheit.

Satz

Sei Σ ein System und I eine S-Invariante mit $I(s) > 0$ für alle s. Dann ist Σ beschränkt.

Beweis. Sei m eine erreichbare Markierung. Dann gilt $\langle I, m\rangle = \langle I, m_0\rangle$. Also gilt für jede Stelle s:

$$I(s) \cdot m(s) \leq \langle I, m\rangle = \langle I, m_0\rangle.$$

Dies liefert $m(s) \leq \langle I, m_0\rangle / I(s)$ und der letzte Ausdruck ist eine Konstante. □

27-26

Notwendige Bedingung für Lebendigkeit.

Satz

Sei Σ ein lebendiges System und I eine S-Invariante mit $I \geq 0$ und $I \neq 0$. Weiter sei jede Stelle im Vor- oder Nachbereich wenigstens einer Transition. Dann gilt $\langle I, m_0\rangle > 0$.

Beweis. Sei s eine Stelle mit $I(s) > 0$. Da das System lebendig ist, gibt es eine Markierung m mit $m_0 \to^* m$ und $m(s) > 0$. Also gilt

$$\langle I, m_0\rangle = \langle I, m\rangle \geq I(s)m(s) > 0.$$

□

27-27

Notwendige Bedingung für Erreichbarkeit.

Definition

Seien m und m' Markierungen. Wir schreiben $m \sim m'$, falls $\langle I, m\rangle = \langle I, m'\rangle$ für alle Stelleninvarianten I gilt.

Satz

Sei Σ ein System und m sei erreichbar. Dann gilt $m_0 \sim m$.

Beweis. Dies folgt direkt aus der Definition. □

Satz

Es gilt $m \sim m'$ genau dann, wenn $m' = m + N_\Sigma x$ lösbar ist.

Beweis. $m \sim m'$ ist äquivalent dazu, dass $m' - m$ orthogonal zur Menge der S-Invarianten von Σ ist. Wegen $(\ker N_\Sigma^T)^\perp = \operatorname{span}(N_\Sigma)$ folgt, dass $m' - m$ im Spann von N_Σ liegt, also $m' - m = N_\Sigma x$ für ein geeignetes x gilt. □

Zusammenfassung dieses Kapitels

27-28

1. Man kann Petrinetze (nicht eindeutig) mittels *Matrizen* beschreiben.
2. Die *linear-algebraischen Eigenschaften* dieser Matrizen liefern notwendige oder hinreichende Bedingungen für Beschränktheit, Erreichbarkeit oder Lebendigkeit.
3. Eine *S-Invariante* ist eine gewichtete Summe der Marken, die immer gleich bleibt.

Übungen zu diesem Kapitel

Übung 27.1 (Invarianten berechnen, mittel)

Berechnen Sie eine Basis der Stelleninvarianten des Netzes aus Übung 26.4. Interpretieren Sie Ihr Ergebnis.

28-1

Kapitel 28
Transitionsinvarianten
Von toten Schlössern und Fallen

28-2

Lernziele dieses Kapitels

1. Konzept der Transitionsinvariante kennen
2. Konzept des Deadlocks und der Trap kennen
3. Invarianten, Deadlocks und Traps in der Analyse anwenden können

Inhalte dieses Kapitels

Worum es heute geht

Anders als bei sequenziellen Algorithmen sind Endlosschleifen bei verteilten Systemen nichts Anrüchiges, in der Tat ist man oft froh, wenn das System wieder zu einem Ausgangszustand zurückkehrt.

Transitionsinvarianten, das erste zentrale Thema dieses Kapitels, sind Schaltfolgen, die Endlosschleifen signalisieren: Schaltet man ein Netz gemäß einer Transitionsinvariante, so sieht das Netz hinterher genauso aus wie vorher. Offenbar sind Transitionsinvarianten eng verbunden mit Eigenschaften wie Lebendigkeit und Beschränktheit: Ist ein System lebendig und beschränkt, so muss es eine Endlosschleife enthalten und somit eine (nichttriviale) Transitionsinvariante. Hat umgekehrt ein Netz keine solche Transitionsinvariante, »so muss etwas faul sein« mit dem Netz. Genau wie Stelleninvarianten lassen sich auch Transitionsinvarianten leicht berechnen. Hat man die Stellen- und Transitionsinvarianten eines Netzes bestimmt, so kann man – mit etwas Glück – schon weitreichende Aussagen über die Eigenschaften des Netzes machen.

Leider hat man nicht immer Glück im Leben, weshalb die Stellen- und Transitionsinvarianten nicht immer ausreichen, um bestimmte Netzeigenschaften zu beweisen. In solchen Fällen können *Deadlocks* und *Traps* weiterhelfen. Ähnlich wie bei der Systemmatrix spiegeln auch diese bestimmte Eigenschaften eines Petrinetzes wider, ohne es komplett zu beschreiben. Ein Trap ist eine Teilmenge der Stellen »aus der es kein Entrinnen gibt«: Sobald sich wenigsten eine Marke in einen Trap verirrt hat, bleibt ein Trap nie leer. Umgekehrt verhält es sich bei Deadlocks (ein nicht ganz glücklicher Name), welche auch Teilmengen der Stellen sind: Ist ein Deadlock einmal leer, so wird sich nie wieder eine Marke in ihn aufmachen, er bleibt tot.

28.1 Transitionsinvarianten

28.1.1 Konzept

Die Idee hinter Transitionsinvarianten.

28-4

- Es gibt Schaltfolgen, nach denen »sieht das Netz wieder genauso aus«.
- Es gilt also für eine solche Schaltfolge $m \to_\sigma m$ und somit $m = m + N_\Sigma \vec{\sigma}$, beziehungsweise $N_\Sigma \vec{\sigma} = 0$.
- Allgemein bezeichnen wir ein x mit $N_\Sigma x = 0$ als *Transitionsinvariante*.
- Man beachte, dass die Transitionsinvariante nicht eine Schaltfolge ist, sonder ein Vektor, der angibt, wie oft jede Transition schalten muss, damit man wieder am Anfang ist.

Die Definition von Transitionsinvarianten.

28-5

Definition

Eine *Transitionsinvariante* eines S/T-Systems Σ ist ein Spaltenvektor I mit der Eigenschaft

$$N_\Sigma I = 0.$$

Die zentrale Eigenschaft von Transitionsinvarianten lautet:

Satz

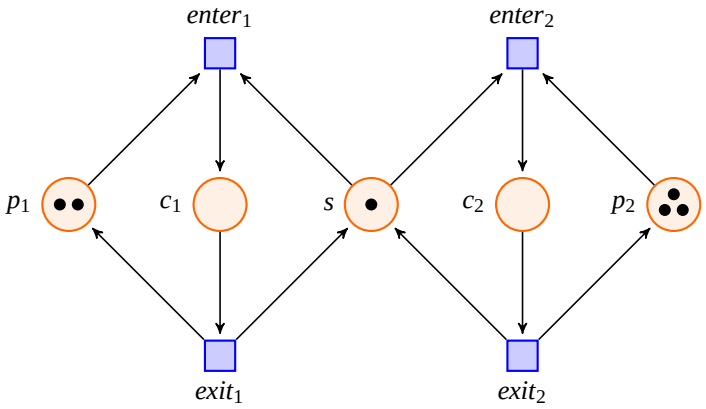
Sei σ eine Schaltfolge, die von m aktiviert wird. Dann ist $\vec{\sigma}$ genau dann eine T-Invariante, wenn $m \to_\sigma m$ gilt.

Beweis. Da m aktiviert ist von σ, gilt $m \to_\sigma m'$ für ein m'. Aus der Markierungsgleichung $m' = m + N_\Sigma \vec{\sigma}$ folgt $m' - m = N_\Sigma \vec{\sigma}$. Also ist $m' = m$ genau dann, wenn $N_\Sigma \vec{\sigma} = 0$. □

28.1.2 Anwendungen

Finden von Transitionsinvarianten

28-6



Zur Übung

Geben Sie drei Transitionsinvarianten des S/T-Systems an.

28-7

Notwendige Bedingung für Lebendigkeit und Beschränktheit.

Satz

Sei Σ ein lebendiges, beschränktes S/T-System. Dann hat es eine T-Invariante I mit $I(t) > 0$ für alle $t \in T$.

Beweis. Das S/T-System ist lebendig. Also gibt es eine *unendliche Folge von Schaltfolgen* σ_i, so dass

1. in jeder Schaltfolge σ_i alle Transitionen aus T vorkommen und
2. $m_0 \to_{\sigma_1} m_1 \to_{\sigma_2} m_2 \to_{\sigma_3} \dots$

Da das S/T-System beschränkt ist, gilt $m_i = m_j$ für geeignete i und j mit $i < j$.
Betrachte die Schaltfolge $\sigma = \sigma_i \sigma_{i+1} \dots \sigma_{j-1}$. Ihr Parikh-Vektor ist eine T-Invariante und er ist in allen Komponenten positiv. □

28-8

Zur Übung

Die Bedingungen waren nur notwendig/hinreichen. Zeigen Sie dies, indem Sie S/T-System ohne komplett positive T-Invariante angeben, das

1. lebendig und nicht beschränkt ist,
2. beschränkt und nicht lebendig ist.

28.2 Deadlocks und Traps

28.2.1 Konzept

28-9

Idee hinter Deadlocks und Traps.

- Bei Stelleninvarianten bleibt eine (gewichtete) Stellensumme konstant. Dies kann man in der Analyse nutzen:
  - Finde eine komplett positive Stelleninvariante.
  - Folgere, dass das S/T-System beschränkt ist.
- Falls nun aber keine komplett positive Invariante existiert, so kann trotzdem vielleicht folgendes gelten:
  - Die Summe über alle Marken wird stets kleiner oder bleibt gleich.
  - Folgere, dass das S/T-System beschränkt ist.
- Ein *Deadlock* ist eine Stellemenge, bei denen die Marken »eher weniger werden«, ein *Trap* ist eine Stellenmenge, bei denen die Marken »eher mehr werden«.

28-10

Definition von Deadlocks und Traps.

Definition

Sei Σ ein S/T-System. Eine Menge A von Stellen heißt

1. *Deadlock*, wenn ${}^\bullet A \subseteq A^\bullet$,
2. *Trap*, wenn ${}^\bullet A \supseteq A^\bullet$,

Beobachtungen:

- Für Deadlocks gilt: Tut eine Transition t in A etwas hinein ($t \in {}^\bullet A$), so nimmt t auch etwas heraus ($t \in A^\bullet$).
- Für Traps gilt: Nimmt eine Transition t aus A etwas heraus ($t \in A^\bullet$), so tut t auch etwas hinein ($t \in {}^\bullet A$).

Fundamentalen Eigenschaften.

28-11

Satz

Sei Σ ein S/T-System und $A \subseteq S$.

1. *Ist A ein Trap und in m_0 liegt wenigsten eine Marke auf einer Stelle in A, so liegt auch in allen erreichbaren Markierungen wenigstens auf einer Stelle in A eine Marke.*
2. *Ist A ein Deadlock und in m_0 liegen keine Marke auf Stellen in A, so liegen auch in allen erreichbaren Markierungen keine Marken auf Stellen in A.*

Die Beweise sind ganz einfache Induktionen.

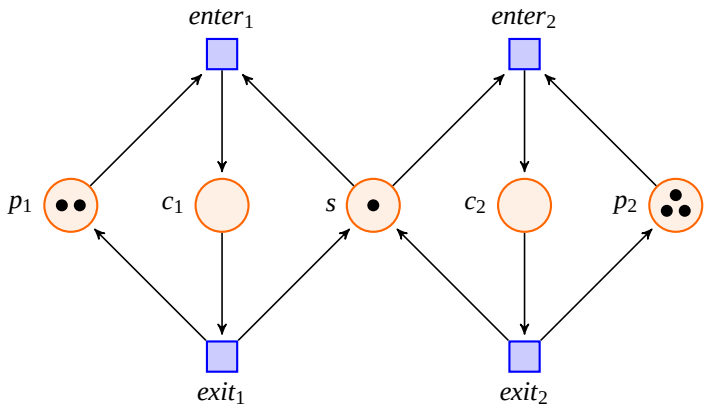
Notationen

- Für eine Trap $A = \{s_1, \ldots, s_n\}$ aus Punkt 1 schreiben wir auch $s_1 + \cdots + s_n \geq 1$.
- Für einen Deadlock $A = \{s_1, \ldots, s_n\}$ aus Punkt 2 schreiben wir auch $s_1 + \cdots + s_n = 0$.

28.2.2 Anwendungen

Finden von Deadlocks und Traps

28-12

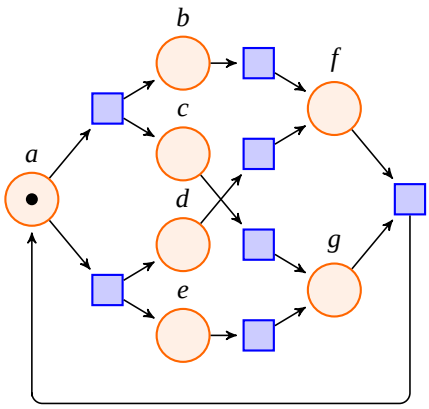


Zur Übung

Welche Deadlocks und Traps hat das S/T-System?

Beispielanwendung von Traps.

28-13



- Wir behaupten, auf c und d kann insgesamt immer höchstens eine Marke sein.
- Dabei nützt die Invariante $2a + b + c + d + e + f + g = 2$ nichts.

28-14

Wie man zeigt, dass sich c und d ausschließen.

- Es gibt folgende zwei Invarianten und folgenden Trap:

$$a+c+e+g=1$$
$$a+b+d+f=1$$
$$a+b+e+f+g\geq 1.$$

- Wäre nun $c \geq 1$ und $d \geq 1$, so impliziert die erste Gleichung $a+e+g=0$ und die zweite $a+b+f=0$.
- Summiert man diese neuen Gleichung, so erhält man $a+b+e+f+g=0$, ein Widerspruch.

28-15

Ein Satz, der den Namen »Deadlock« rechtfertigt.

Satz
Sei Σ ein S/T-System mit mindestens einer Transition und mit Kantengewichten 1. Falls m verklemmt (tot) ist, so bildet die Menge $R = \{s \in S \mid m(s) = 0\}$ einen Deadlock von Σ mit $R \neq \emptyset$.

Beweis. Wir zeigen $R^\bullet = T$, woraus sofort folgt, dass R ein Deadlock ist. Sei $t \in T$. Dann gilt $t \in R^\bullet$, denn wenigstens eine Stelle im Vorbereich von t hat keine Marke. Wäre R leer, so wäre $R^\bullet = \emptyset \neq T$. □

28-16

Hinreichende Bedingung für Verklemmungsfreiheit.

Satz (Deadlock-Trap-Eigenschaft)
Sei Σ ein S/T-System mit mindestens einer Transition und mit Kantengewichten 1. Wenn jeder nicht-leere Deadlock einen Trap enthält, für den in m_0 mindestens eine Stelle eine Marke hat, so ist das S/T-System verklemmungsfrei.

Beweis. Nehmen wir an, das S/T-System sei nicht verklemmungsfrei und m sei die erreichbare tote Markierung. Für m muss die Menge aller Stellen s mit $m(s) = 0$ einen Deadlock bilden. Dies kann aber nicht sein, da jeder Deadlock einen Trap enthält und somit auch eine Marke. □

Zusammenfassung dieses Kapitels

28-17

1. *Transitionsinvarianten* sind die Parikh-Vektor von Schaltfolgen, die Markierungen nicht ändern.
2. Ein *Deadlock* ist eine Menge von Stellen, die, wenn sie erstmal leer ist, leer bleibt.
3. Ein *Trap* ist eine Menge von Stellen, die, wenn sie einmal eine Marke enthält, nie leer wird.

29-1

Kapitel 29

Syntaktische und semantische Erweiterungen von Petrinetzen

Marken mit Graffiti

29-2

Lernziele dieses Kapitels

1. Verfahren zur Entfernung von Kapazitäten kennen
2. Syntaktische Erweiterungen von Petrinetzen kennen
3. Semantische Erweiterungen von Petrinetzen kennen

Inhalte dieses Kapitels

Worum es heute geht

Gute Formalismen lassen sich nicht nur in dem Bereich einsetzen, für den sie ursprünglich gedacht waren. Turingmaschinen haben recht klein angefangen in Alan Turings Aufsatz *On Computable Numbers with an Application to the Entscheidungsproblem*, heute werden in der Theorie Turingmaschinen untersucht, die auf mehrdimensionalen Gittern mit vielen Köpfen arbeiten, dabei Zugriff auf öffentliche und auf private Zufallsbits haben und über Orakelbänder noch mit anderen Turingmaschinen kommunizieren. Ähnlich ist um die schönen einfachen endlichen Automaten ein komplexes Theoriegebäude errichtet worden, mit Potenzreihendarstellung und syntaktischen Monoiden inklusive.

Petrinetze sind ein guter Formalismus, weshalb man sich mit der »Basisvariante« nicht recht zu Frieden gegeben hat. Einige Erweiterungen kann man leicht in Petrinetze einbauen ohne die Analysewerkzeuge, die in den letzten Kapiteln erarbeitet wurden, zu verlieren. Weitergehende Änderungen führen aber zu Netzen, die schnell »zu mächtig« werden als dass man sie noch analysieren könnte. So sind die so genannten High-Level-Netze Turing-mächtig, weshalb es mit der Analysierbarkeit hier vorbei ist.

Neben syntaktischen Erweiterungen kann man auch semantische Varianten betrachten, was sich bei verteilten Systemen besonders anbietet – schließlich sind bei ihnen gerade die verschiedenen möglichen Verhaltensweisen besonders interessant. Die am Ende dieses Kapitels kurz skizzierten Prozessnetze sind ein besonders drolliges Beispiel einer solchen alternativen Semantik, denn bei ihnen ist die Semantik des Netzes ein Netz.

29.1 Syntaktische Erweiterungen

29.1.1 Kantengewichte

29-4

Syntaktische Erweiterung: Kantengewichte

- Ein *Netz* ist ein Tupel (S,T,F) bestehend aus Stellen, Transitionen und einem Fluss.
- Die *Kantengewichte* eines *Netzes* sind gewissenmaßen schon eine *syntaktische* Erweiterung von Petrinetzen.
- Kantengewichte können *die Analyse stören*. So gilt der Deadlock-Trap-Satz nur in Systemen ohne Kantengewichte.

Erste Frage

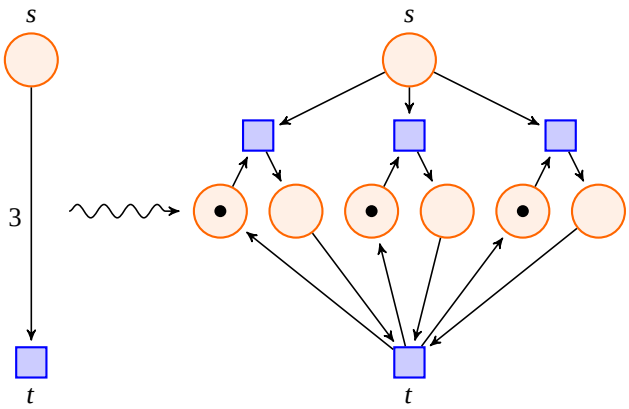
Brauchen wir Kantengewichte wirklich? Anders gefragt: Können wir jedes Netz mit Kantengewichte durch ein äquivalentes ohne ersetzen?

29-5

Wie man Kantengewichte los wird.

Konstruktion zur Vermeidung von Kantengewichten

Wir ersetzen eine mit beispielsweise 3 gewichte Kante von s nach t wie folgt:



29-6

Zentrale Eigenschaften der Konstruktion.

1. Die Markierungsgraphen der Systeme sind offenbar nicht isomorph.
2. Es lassen sich aber sinnvolle Simulationen in beide Richtungen durchführen.

Satz

Sei Σ ein S/T-System mit Kantengewichten und Σ' das S/T-System ohne Kantengewichte, das mittels der Konstruktion entsteht. Für eine Schaltfolge σ' in Σ' sei $\mathrm{strip}(\sigma')$ *die Einschränkung auf die Transitionen in Σ.*

- *Ist σ eine aktivierte Schaltfolge in Σ, so existiert eine in Σ' aktivierte Schaltfolge σ' mit* $\mathrm{strip}(\sigma') = \sigma$.
- *Ist σ' eine aktivierte Schaltfolge in Σ', so ist* $\mathrm{strip}(\sigma')$ *eine aktivierte Schaltfolge in Σ.*

29.1.2 Kapazitäten von Stellen

Syntaktische Erweiterung: Kapazitäten von Stellen

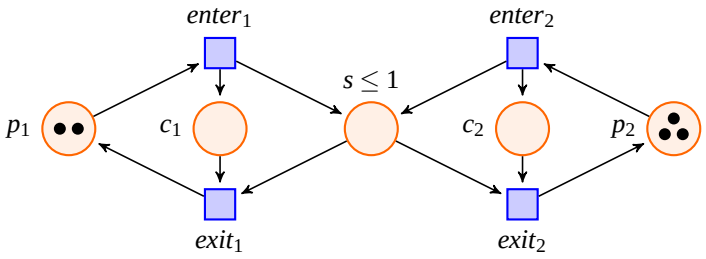
29-7

- Als zweite syntaktische Erweiterung führen wir nun *Kapazitäten für Stellen* ein.
- Dies wird durch eine Funktion $\kappa\colon S \to \mathbb{N} \cup \{\infty\}$ modelliert.
- Eine Transition darf nur schalten, wenn dadurch nicht auf einer Stelle s mehr als $\kappa(s)$ Marken liegen.

Zweite Frage
Brauchen wir Kapazitäten von Stellen wirklich?

Das Mutex-Netz mit Kapazitäten.

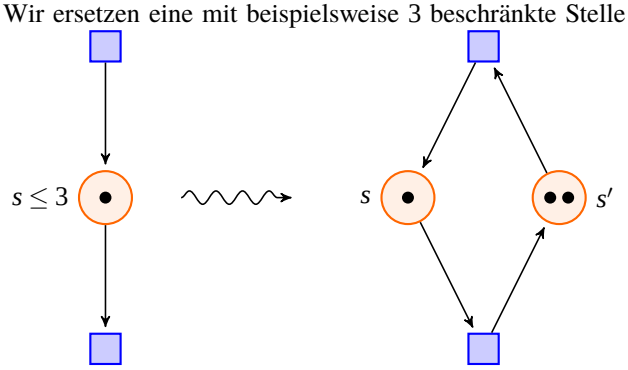
29-8



Wie man Kapazitäten los wird.

29-9

Konstruktion zur Vermeidung von Kapazitäten
Wir ersetzen eine mit beispielsweise 3 beschränkte Stelle s wie folgt:



Zentrale Eigenschaften der Konstruktion.

29-10

Satz
Sei Σ ein S/T-System mit Kapazitäten und Σ' das S/T-System ohne Kapazitäten, das mittels der Konstruktion entsteht. Dann gilt für jede Schaltfolge σ, dass sie in Σ genau dann aktiviert ist, wenn sie es in Σ' ist.

29.1.3 Netze mit gefärbten Marken

Syntaktische Erweiterung: Gefärbte Marken.

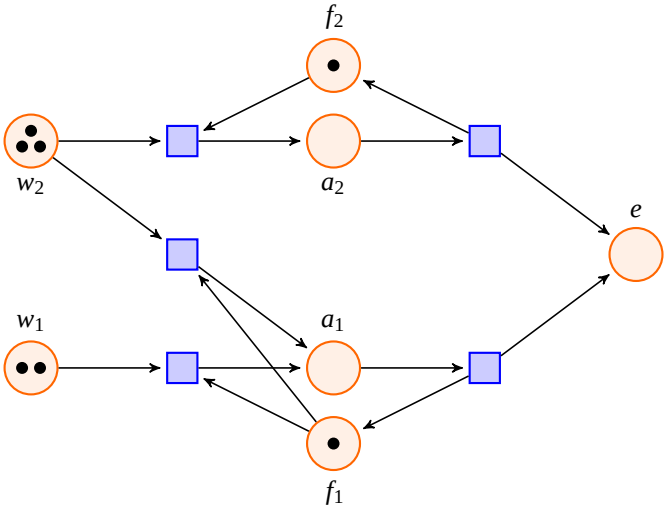
29-11

- Marken haben in einem Petrinetz *keine Identität*. Alle Marken sind gleich.
- Wir führen jetzt *endlich viele Farben* für Marken ein. Eine rote Marke lässt sich nun von einer grünen Marke unterscheiden.
- Transitionen können nun (müssen aber nicht) Bedingungen an die Farbe von Token stellen.

Dritte Frage
Brauchen wir gefärbte Stellen wirklich?

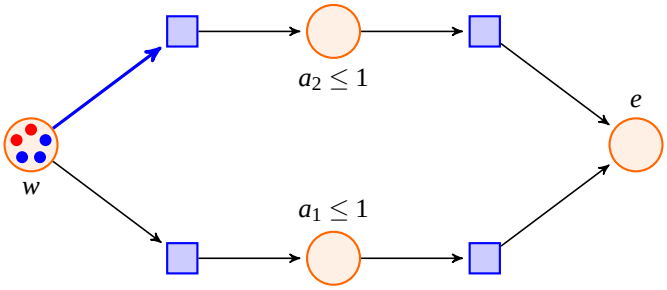
29-12

Das Schedule-Netz.



29-13

Das Schedule-Netz mit gefärbten Marken.



29-14

Simulation von gefärbten Marken.

Konstruktion

Gegeben sei ein Netz mit gefärbten Marken. Seien k Farben möglich. Dann können wir jede Stelle durch k Stellen ersetzen, die die Marken einer Farbe speichern.
Dann passen wir die Transitionen so an, dass nun die korrekten Anzahlen von Marken der jeweiligen »Farbe« weggenommen und hinzugefügt werden.

Beobachtung (ohne Beweis)

Gefärbte Marken sind nützlich, aber nicht nötig.

29.1.4 Netze mit strukturierten Marken (High-Level Netze)

29-15

Syntaktische Erweiterung: Strukturierte Marken.

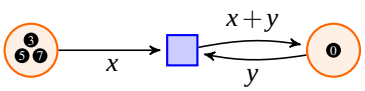
- Wir erlauben nun, dass Marken komplexe Gebilde werden.
- Beispielsweise kann man eine universell-algebraische Signatur und eine Algebra festlegen.
- Marken sind nun Elemente der Algebra.
- Kanten werden mit Termen über der Signatur markiert.
- Eine Kante kann schalten, wenn sich die Terme so belegen lassen, dass sie ausgewertet Marken ergeben.

Vierte Frage

Brauchen wir strukturierte Marken wirklich?

29-16

Ein Zahlen-Summier-Netz.



Strukturierte Marken machen Netze mächtiger. 29-17

- Es ist recht leicht einzusehen, dass Systeme mit strukturierten Marken *Turing-mächtig* sind.
- Für Low-Level-Netze ist aber Erreichbarkeit *entscheidbar*.
- Folglich kann man nicht jedes High-Level-Netz in ein Low-Level-Netz umwandeln.
- Überhaupt kann man nach dem Satz von Rice *keine einzige nichttriviale Eigenschaft von High-Level-Netzen entscheiden*.
- Das ist traurig.

29.2 Semantische Erweiterungen

29.2.1 Erreichbarkeitsgraph

Erste Semantik für Petrinetze: Der Erreichbarkeitsgraph-Semantik. 29-18

Wir hatten bereits folgende Semantik eingeführt:

Erreichbarkeitsgraph-Semantik
Sei Σ ein Petrinetz.

- Die *Erreichbarkeitsgraph-Semantik* ist ein Graph.
- Die Knoten sind alle Markierungen m mit $m_0 \to^* m$.
- Die Kanten sind alle Paare (m, m') mit $m \to_t m'$ für eine Transition $t \in T$.

29.2.2 Nebenläufigkeitsgraph

Kritik an der Erreichbarkeitsgraph-Semantik. 29-19

- Petrinetze dienen unter anderem dazu, verteilte Algorithmen zu analysieren.
- Es ist daher *schade*, dass sich *Nebenläufigkeit* nicht in der Semantik widerspiegelt.
- Beim *Nebenläufigkeitsgraph* nimmt man deshalb auch *paralleles Schalten* in die Semantik auf.

Definition
Ein Schaltwort $\sigma \in T^*$ heißt *parallel schaltbar in* m, wenn für jede Stelle s gilt $\sum_{t \in T} \vec{\sigma}_t \cdot W(s,t) \leq m(s)$.

Zweite Semantik für Petrinetze: Die Nebenläufigkeitsgraph-Semantik. 29-20

Nebenläufigkeitsgraph-Semantik
Sei Σ ein S/T-System.

- Die *Nebenläufigkeitsgraph-Semantik* ist ein Graph.
- Die Knoten sind alle Markierungen m mit $m_0 \to^* m$.
- Die Kanten sind alle Paare (m, m') mit $m \to_\sigma m'$ für ein in m parallel schaltbares $\sigma \in T^*$.

29.2.3 Prozessnetze

29-21

Kritik an der Nebenläufigkeitsgraph-Semantik.

- Paralleles Schalten ist nicht gleich paralleles Schalten.
- Wenn zwei Transitionen in »weit entfernten Teilen« eines Netzes gleichzeitig schalten (könnten), ist dies *weniger spannend* als wenn zwei potentiell in Konflikt stehende dies tun.
- Bei der *Prozessnetz- oder auch Halbordnungssemantik* versucht man, die *Kausalität* von Schaltvorgängen abzubilden.

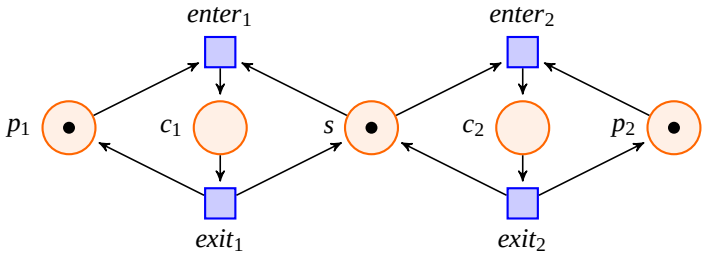
29-22

Begriffe zur Definition von Prozessnetzen.

Ereignis Das Schalten einer Transition in einem Netz heißt ab sofort ein *Ereignis*.

Bedingung Für eine Markierung m bilden wir eine (Multi-)Menge, die für jede Marke die Stelle auf der sie liegt enthält. Diese Menge bezeichnen wir als die *Bedingung* $b(m)$.

Beispiel



Die Bedingung in diesem Netz ist $\{p_1, s, p_2\}$.

29-23

Dritte Semantik für Petrinetze: Die Prozessnetz-Semantik.

Prozessnetz-Semantik

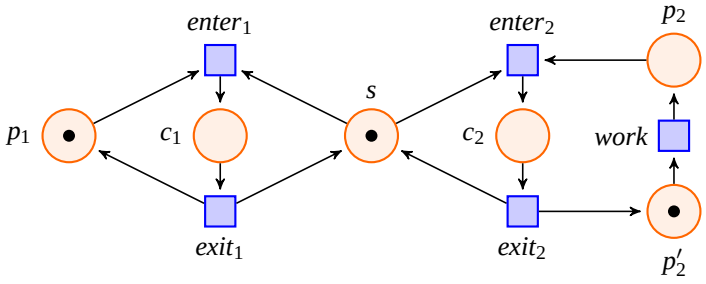
Sei Σ ein Petrinetz und $m_0 \to_{t_1} m_1 \to_{t_2} m_2 \to_{t_3} \cdots \to_{t_n} m_n$ eine Schaltfolge.

- Die *Prozessnetz-Semantik* ist ein Petrinetz P.
- Das Netz ist iterativ definiert.
- Zu jedem Zeitpunkt der Definition ist eine Menge von Stellen ausgezeichnet, die gerade eine Bedingung $b(m_i)$ bilden. Initial finden sich im Netz gerade die Stellen der Bedingung $b(m_0)$.
- Wir fügen ein Ereignis t_i hinzu, indem wir als Vorbereich die Stellen der Bedingung $b(m_{i-1})$ nehmen, von denen etwas weggenommen wird, und als Nachbereich neue Stelle.
- Stellen der alten Bedingung, die nicht vom Ereignis verbraucht wurden, und die neuen Stellen bilden gerade die Bedingung $b(m_i)$.

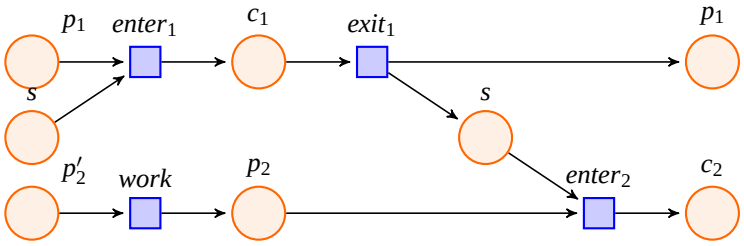
29-24

Beispiel eines Prozessnetzes.

System



Prozessnetz zu $m_0 \to_{enter_1} m_1 \to_{exit_1} m_2 \to_{work} m_3 \to_{enter_2} m_4$



Zusammenfassung dieses Kapitels

1. Es gibt mehrere *syntaktische Erweiterungen* von Petrinetzen.
2. Nur *High-Level-Netze* sind *wirklich mächtiger*.
3. Es gibt verschiedene *Semantiken von Petrinetzen*, die *Nebenläufigkeit besser modellieren*.

29-25