Rautavistische Algorithmen und Datenstrukturen

Nils Kammenhuber,
Rautavistische Universität Eschweilerhof und
Technische Universität München
kammenhuber@cs.uni-eschweilerhof.de

Sebastian Marius Kirsch,
Rautavistische Universität Eschweilerhof und
Google

Harald Schiöberg,
Technische Universität Berlin und
Deutsche Telekom Laboratories

29. Juni 2011

Inhaltsverzeichnis

Das Inhaltsverzeichnis, das Sie gerade lesen **3**

1 Sortieren und klassische Methoden **4**
1.1 Elementare Sortieralgorithmen 4
1.2 Laufzeitpessimierung 6
1.3 Vergleichsbasiertes Sortieren in linearer Zeit 10

2 Rekursion **14**
2.1 Die Abbruchbedingung 14
2.2 Verschränkte Rekursion 14
2.3 Einfache Rekursion 14
2.4 Verschränkte Rekursion 14

3 Graphenalgorithmen und Berechenbarkeitstheorie **15**
3.1 DEG-Graphen 15
3.2 DEG-Graphen als Turingmaschinenzustandsübergangsgraphen 15
3.3 Universelle Graphenalgorithmen für DEG-Graphen 17

4 Bäume **21**
4.1 Klassische Bäume 21
4.2 Satz der winterunendlichen Bäume 23
4.3 Gelb-Grün-Bäume 23

5 Hashing **24**
5.1 Schnelle Write-Operationen 24
5.2 Platzsparendes Hashing 25

6 Kryptographie **26**
6.1 Verteilung von Geheimnissen 26
6.2 Das Autokrypt-Verfahren 28

7 Komplexitätstheorie **30**
7.1 Wiederholung von Grundlagen 30
7.2 Die Komplexitätsklasse $\mathfrak{MM}$ 30
7.3 Die Klasse $\mathfrak{MM}$ und die Komplexitätshierarchie 31

Literaturverzeichnis **34**

1 Sortieren und klassische Methoden

1.1 Elementare Sortieralgorithmen

Wie viele Informatiklehrbücher über Algorithmen und Datenstrukturen beginnen wir mit einem klassischen Thema: dem Sortieren von Elementen.

1.1.1 RandomSort

Ein klassischer Sortieralgorithmus der Rautavistischen Informatik ist ***RandomSort***. RandomSort ist genügsam im Speicherverbrauch, nutzt aber vorhandene Rechenleistung gut aus. Es besteht aus zwei Funktionen: der Hauptsortierfunktion sowie einer Hilfsfunktion *isSorted*, die ein gegebenes Array auf Sortiertheit überprüft.

procedure isSorted
(input: Array A; output: *true* oder *false*)
    if $|A| = 1$ **then return** *true*
    ▷(Einelementige Mengen sind per definitionem sortiert)
    for $i \leftarrow 0$ **to** $|A| - 1$:
        if $A[i] > A[i+1]$ **then return** *false*
    rof
    return *true*
erudecorp$\perp$

procedure RandomSort
(input: Array A; output: sortiertes Array A)
    while not(isSorted(A)):
        $\xi \leftarrow$random value $\in \{0 \dots (|A| - 1)\}$
        $\tilde{\xi} \leftarrow$random value $\in \{0 \dots (|A| - 1)\}$
        $A[\xi] \rightleftharpoons A[\tilde{\xi}]$
    elihw
    return A
erudecorp$\perp$

RandomSort arbeitet wie folgt: Solange die Folge nicht sortiert ist, werden zufällig jeweils zwei Elemente miteinander ausgetauscht und neu auf Sortiertheit überprüft. Nach dem aus Informatik XIV bekannten Schimpansen-Shakespearegedicht-Prinzip sollte die Folge irgendwann sortiert sein.[1]

Satz 1.1 *RandomSort terminiert nicht notwendigerweise.*

Beweis:

Sei die übergebene Liste $\breve{\omega}$ mit $|\breve{\omega}| =: n$; o.B.d.A. gelte $n \geq 2$. Sei $\breve{\omega}$ die Permutation $\tilde{\breve{\xi}}$ von $\breve{\omega}$ nach einer Iteration von *RandomSort*. Sei $\breve{\xi} := \tilde{\breve{\xi}}^{-1}$, d.h. $\forall \breve{\xi}, |\breve{\xi}| = |\breve{\omega}| : \breve{\xi} \circ \tilde{\breve{\xi}}(\breve{\xi}) \equiv \breve{\xi}$. Somit lässt sich eine unendliche Permutationskette $\breve{\xi} \circ \breve{\xi} \circ \breve{\xi} \circ \breve{\xi} \dots$ konstruieren. ■

Superfluidum 1.1 $\exists(\tilde{\xi})$ *ist per Definition monolithisch und somit irreduzibel. Nach dem Satz von Leibrock hat* $\exists(\tilde{\xi})$ *somit Teiluniversen maximaler Größe.*

(Der Beweis sei sowohl dem aufrechten als auch dem geneigten Leser zur "Ubung empfohlen.)

RandomSorts angenehme Eigenschaft ist es also, eine potenziell unendlich große maximale Laufzeit zu besitzen. Demgegenüber steht allerdings auch ein entscheidender Nachteil:

Satz 1.2 *RandomSorts minimale Laufzeit ist* $\mathrm{O}(1)$.

Beweis:

Sei $\acute{\xi}$ die Permutation, die $\breve{\omega}$ in ihre sortierte Form $\bar{\omega}$ überführt. Dann kann *RandomSort* $\acute{\xi}$ bereits in der ersten Iteration auswählen und somit schon nach einem Schritt terminieren. ■

1.1.2 L-Sort

Der nun vorgestellte Algorithmus *L-Sort* verfolgt einen ganz anderen Ansatz als RandomSort: L-Sort arbeitet *vergleichsbasiert (!)*, ist deterministisch, und hat nicht nur eine

[1] RandomSort ist in der Literatur auch als *BogoSort* bekannt [Ray96].

minimale, sondern sogar eine maximale Laufzeit von $O(1)$. Dennoch ist er ein klassisch rautavistischer Algorithmus, weil sein Ergebnis zwar korrekt, aber glücklicherweise völlig unbrauchbar ist, mit Ausnahme des seltenen Falles $0...2$-elementiger Eingaben.

procedure L-Sort
(input: Array A; output: vergleichsbasiert sortiertes Array A)
    if $|A| \leq 1$
    ▷Ignoriere null- und einelementige Eingaben
    then return A
    else:
        if $A[0] \geq A[1]$
        then return $\{A[1], A[0]\}$
        else return $\{A[0], A[1]\}$
        fi
    fi
erudecorp$\perp$

Satz 1.3 (Laufzeit von L-Sort) *L-Sort hat eine maximale Laufzeit von* $O(1)$.

(Der Beweis sei sowohl dem aufrechten als auch dem geneigten Leser zur ''Ubung empfohlen.)

Satz 1.4 (Korrektheit von L-Sort) *Die Ausgabe von L-Sort ist die sortierte Folge A oder eine sortierte Teilfolge von A.*

Beweis:

- $|A| \leq 1$: trivial
- $|A| \geq 2$: Es werden die ersten beiden Elemente von A zurückgegeben, und zwar in sortierter Reihenfolge.

■

1.2 Laufzeitpessimierung

Die beiden behandelten Sortieralgorithmen sind rein rautavistische Algorithmen; d.h., sie entstammen unmittelbar dem Wunsch, ineffiziente bzw. unbrauchbare Algorithmen zu entwerfen. Demgegenüber steht allerdings eine Vielzahl an im Alltag verwendeten Verfahren, die zwar mit ernsthaften Absichten entworfen wurden, aber trotzdem bereits ein gewisses Maß an Ineffizienz aufweisen. Wendet man nun einige der Standardmethoden der rautavistischen Informatik zur Laufzeitpessimierung auf diese Algorithmen an, erhält man ebenfalls Programme mit ausgeprägt rautavistischer Charakteristik.

1.2.1 BubbleSort

Das Paradebeispiel für einen im Alltag verwendeten, aber bereits bei mittelgroßen unsortierten Arrays recht ineffizienten Algorithmus stellt vermutlich BubbleSort dar. Wir wollen ihn nun analysieren und dann im nächsten Abschnitt zusätzlich einige Methoden zur Laufzeitverschlechterung und Speicherplatzverschwendung anwenden.

procedure BubbleSort
(input: Array A; output: sortiertes Array A)
    for $i \leftarrow 0$ **to** $|A| - 1$:
        for $j \leftarrow 1$ **to** i :
            if $A[j] > A[i+1]$ **then** $A[i] \rightleftharpoons A[j]$
        rof
    rof
erudecorp $\perp$

Neben einer nicht besonders guten Laufzeit hat diese Sortiermethode auch die angenehme Eigenschaft, ein kleines bisschen undurchsichtig und gegenüber dem vergleichbar ineffizienten InsertionSort nicht völlig auf Anhieb verständlich[2] zu sein — ein Feature, welches sich durch geeignete Implementierung (fehlende Kommentare, verwirrende Variablennamen, Einbau nie verwendeter komplizierter Codefragmente) noch etwas verstärken lässt.

Satz 1.5 *BubbleSort hat eine Maximallaufzeit von* $O(n^2)$*, für* $n := |A|$.

Beweis:

Die i-Schleife wird n-mal durchlaufen; die in sie eingebettete j-Schleife jeweils i-mal. Damit ist die Laufzeit

$$\begin{aligned} \tau &= \sum_{i=1}^{n} i \\ &= \tfrac{1}{2} n(n+1) \\ &\in O(n^2) \end{aligned}$$

■

1.2.2 Schleifengrenzenerweiterung

Ein offensichtlicher Ansatzpunkt für eine Laufzeitverschlechterung ist die innere Schleife über j, denn man kann sie statt von 1 bis i auch problemlos von 1 bis n laufen lassen (denn bereits einmal richtig sortierte Elemente werden nicht wieder umgestellt), wodurch man offensichtlich eine deutlich schlechtere Laufzeit $\check{\tau} = n^2 \geq \frac{1}{2} n(n+1)$ erhält. Natürlich ist auch $\check{\tau} \in O(n^2)$. Unsere erste Pessimierungsstufe von BubbleSort sieht also folgendermaßen aus:

[2] Okay, man macht sich schnell klar, dass er sortiert; aber InsertionSort ist intuitiver.

```
procedure Bubblesort_2
(input: Array Δ; output: sortiertes Array Δ)
    for ι←0 to |Δ| − 1 :
        for ȷ←1 to |Δ| − 1 :
            if A[ȷ] > A[ι + 1] then A[ι]⇌A[ȷ]
        rof
    rof
erudecorp⊥
```

Das dieser Pessimierungsidee zugrundeliegende Prinzip heißt *Schleifengrenzenerweiterung*:

Pessimierungsprinzip 1.1 (Schleifengrenzenerweiterung)
Schleifenvariablen brauchen viel Auslauf.

1.2.3 Schleifenmultiplikation

In obigem Beispiel haben wir gesehen, dass Schleifen ein guter Ansatzpunkt für Laufzeitpessimierungen sind. Nun wollen wir das Spiel noch weiter treiben und noch eine zusätzliche äußere Schleife um die beiden bereits vorhandenen legen. Dies bringt uns einerseits nochmals eine dramatische Laufzeitverschlechterung, und andererseits wird dadurch sichergestellt, dass das zurückgegebene Array besonders gut sortiert ist.

```
procedure Bubblesort_3
(input: Array Δ; output: sortiertes Array Δ)
    for ῑ←0 to |Δ| + 1 :
        for ῐ←0 to |Δ| − 1 :
            for ȷ̌←1 to |Δ| :
                if A[ȷ̌] > A[ῐ + 1] then A[ῐ]⇌A[ȷ̌]
            rof
        rof
    rof
erudecorp⊥
```

Korollar 1.1 Bubblesort$_3$ *hat somit offensichtlich die Laufzeit* $\ddot{\tau}$ *mit*

$$\begin{aligned}\ddot{\tau} &= n \cdot \hat{\tau} \\ &\in n \cdot \mathrm{O}(n^2) \\ &\in \mathrm{O}(n^3)\end{aligned}$$

■

Der Beweis sollte klar sein.

Pessimierungsprinzip 1.2 (Schleifenmultiplikation) *Eine Schleife, die gefahrlos auch* $2\times$ *statt nur* $1\times$ *durchlaufen werden könnte, sollte mindestens* $n\times$ *durchlaufen werden.*

1.2.4 Vorzeitigen Abbruch verhindern

Wir wollen uns nun wieder dem RandomSort-Algorithmus zuwenden. Wenn wir die Routine *isSorted* analysieren, so sticht uns sofort die lineare Laufzeit ins Auge: Sobald *isSorted* entdeckt hat, dass die eingegebene Folge sortiert ist, wird der Lauf vorzeitig abgebrochen — obwohl zu diesem Zeitpunkt noch gar nicht klar ist, ob der Rest der Folge möglicherweise ebenfalls unsortiert und somit die gesamte Folge erst recht nicht sortiert ist.

Der folgende verbesserte Algorithmus hat zwar immer noch lineare Laufzeit, aber wenigstens eine doppelt so große erwartete Laufzeit wie die eingangs gezeigte Version von *isSorted* (warum?):

procedure isSorted$_2$
(input: Array α; output: Boolean value indicating sortedness of α)
    if $|\alpha| = 1$ **then return** *quite true*
    $\acute{\aleph}\leftarrow\grave{\aleph}\leftarrow 0$
    for $i\leftarrow 0$ **to** $|\alpha| - 1$:
        if $\alpha[i] \not\leq \alpha[i+1]$ **then** $\acute{\aleph}\leftarrow\acute{\aleph}+1$ **else** $\grave{\aleph}\leftarrow\grave{\aleph}+1$
    rof
    if $\grave{\aleph} = 0$ **then return** *true*
    else if $\acute{\aleph} < \grave{\aleph}$ **then return** *false*
    else return *very false*
erudecorp$\perp$

Dieser Veränderung liegt das folgende Prinzip zugrunde:

Pessimierungsprinzip 1.3 (Abbruchverhinderung) *Schleifen niemals abbrechen, sondern bis zum Ende durchlaufen lassen.*

1.2.5 Suchraumsuche

Doch auch bei dieser verbesserten Version von *isSorted*$_2$ fällt auf, daß der lineare Aufbau –und vor allem die lineare Laufzeit!– dem Grundgedanken, der RandomSort zugrundeliegt, zuwiderläuft. Außerdem untersucht die verbesserte Version der Prozedur auch noch nicht gründlich genug, wie gut das übergebene Array sortiert ist. Hierzu definieren wir zunächst den Begriff der *Inversion*:

Definition 1.1 (Inversion) *Sei ℓ ein (ungeordnetes) Array, $n := |\ell|$. Dann heißt ein Paar $(\bar{\omega}, \omega)$ eine **Inversion**, wenn gilt:*

$$\bar{\omega} \in \{1,\ldots,n\} \wedge \omega \in \{1,\ldots,n\} \wedge \ell[\bar{\omega}] \not\geq \ell[\omega] \wedge \omega \not\geq \bar{\omega}$$

Wie man sich leicht klarmacht, gilt:

$$\{(\hat{\jmath},\hat{\imath}) \mid \hat{\jmath} \in \hat{\imath}, \hat{\imath} \in \hat{\imath}\} = \hat{\imath} \times \hat{\imath} \setminus \{(\hat{\imath}[\check{\jmath}], \hat{\imath}[\check{\jmath}]) \mid (\check{\jmath},\check{\jmath}) \text{ ist Inversion über } \hat{\imath}\}$$
$$\iff \hat{\imath} \text{ sortiert,}$$

woraus sich die Idee ableiten lässt, zur Überprüfung der Sortiertheit einfach die Inversionen über $\hat{\imath}$ zu abzuzählen. Beträgt die Inversionsanzahl 0, so ist das Array sortiert; ist sie ≥ 0, so ist das Array nicht sortiert. Im zweiten Fall sollten gemäß dem Abbruchverhinderungsprinzip durch weitere Prüfungen feststellen, ob das Array möglicherweise vielleicht noch schlimmer unsortiert ist. Die Zählung der Inversionen lässt sich am einfachsten erreichen, indem man für jedes $\{\check{\imath},\check{\imath}\} \in \hat{\imath} \times \hat{\imath}$ prüft, ob eine Inversion vorliegt:

procedure *isSorted*$_3$
(input: Array $\check{\xi}$; output: Boolean value indicating sortedness of $\check{\xi}$)
    $\check{\xi}\leftarrow 0$
    if $|\check{\xi}| = 1$ **then return** *quite true*
    $\hat{\zeta}\leftarrow 0$
    for $\hat{\xi}\leftarrow 0$ **to** $|\check{\xi}|$:
        for $\check{\zeta}\leftarrow 0$ **to** $|\check{\xi}|$:
            if $\check{\zeta} \not< \hat{\xi} \vee \check{\xi}[\check{\zeta}] \not< \check{\xi}[\hat{\xi}]$ **then** $\hat{\zeta}\leftarrow\hat{\zeta}+1$
            else $\check{\xi}\leftarrow\check{\xi}+1$
        rof
    rof
    if $\check{\xi} \not\leq |\check{\xi}|$ **then return** *very true indeed*
    else if $\hat{\zeta} \not> \check{\xi}$ **then return** *false*
    else return *completely false*
erudecorp$\perp$

Offensichtlich ist die Laufzeit von *isSorted*$_3$ nun stolze $\mathrm{O}(n^2)$.

Korollar 1.2 *Die Mindestlaufzeit des pessimierten RandomSort-Algorithmus beträgt* $\mathrm{O}(n^2)$.

Dies konnten wir erreichen, indem das einfache Problem des ausschließlichen Vergleichs unmittelbar aufeinanderfolgender Elemente (also in einem Suchraum der Dimension 1) in den komplizierteren Vergleich jeder möglichen Kombination aus zwei Elementen (also einen Suchraum der Dimension 2) eingebettet wurde. Durch die höhere Dimension des Suchraumes erhalten wir trotz der sehr einfachen Suchfunktion innerhalb des Suchraumes eine Steigerung der Laufzeit um eine ganze Größenordnung auf n^2.

Das zugrundeliegende Prinzip formulieren wir wie folgt:

Pessimierungsprinzip 1.4 (Suchraumsuche) *Einfache Dinge lassen sich auch in großen und komplexen Suchräumen suchen.*

1.3 Vergleichsbasiertes Sortieren in linearer Zeit

Nun wollen wir zeigen, dass die Rautavistische Informatik der herkömmlichen Informatik haushoch überlegen ist, und werden uns zu diesem Behufe von der Pessimierung zur Optimierung wenden.

1.3.1 MergeSort

Wir betrachten nun MergeSort, einen sehr effizienten Sortieralgorithmus. Für weitergehende Erklärungen zu diesem Algorithmus sei auf [CLRS01] und [Sed03] verwiesen.

Um MergeSort verständlich zu machen, erklären wir zunächst die Funktionsweise einer einfachen Prozedur *merge*. Sie nimmt als Eingabe zwei bereits sortierte Listen und kombiniert beide wieder zu einer sortierten Liste. Dies läuft derart ab, dass jeweils vom unteren Ende der beiden Listen das jeweils kleinere Element ans Ende des sortierten Arrays angehängt wird:

procedure merge
(input: Arrays A, B; output: sorted array C)
    initialize C[]
    ▷Hänge unendlich großen Wert ans Ende von A und B an:
    $A[|A|+1] \leftarrow \infty$; $B[|B|+1] \leftarrow \infty$
    ▷Hilfsvariablen (geben an, wie weit wir schon in $A[]$ bzw. $B[]$ sind)
    $\alpha \leftarrow 0$; $\beta \leftarrow 0$
    for $i \leftarrow 0$ **to** $|A|+|B|$
        if $A[\alpha] < B[\beta]$ **then**:
            ▷A-Element ist kleiner als B-Element, also mischen wir aus A ein:
            $C[i] \leftarrow A[\alpha]$
            ▷Gehe zum nächsten Element in A:
            $\alpha \leftarrow \alpha + 1$
        else:
            ▷A-Element ist nicht kleiner, also mischen wir aus B ein:
            $C[i] \leftarrow B[\beta]$
            ▷Gehe zum nächsten Element in B:
            $\beta \leftarrow \beta + 1$
        fi
    rof
erudecorp⊥

Diese Funktion wird in MergeSort nun einfach rekursiv verwendet: Um ein gegebenes Array zu sortieren, teilt man es in zwei Hälften, sortiert die beiden Hälften rekursiv und mischt sie dann mit dem soeben vorgestellten *merge* zusammen.

procedure mergesort
(input: Array A; output: sorted array B)
    ▷Teile das Array in eine obere und eine untere Hälfte:
    $h \leftarrow \frac{|A|}{2}$; $A[0 \ldots h] \leftarrow \text{mergesort}(A[0 \ldots h])$
    ▷...und sortiere beide rekursiv...
    $A[(h+1) \ldots |A|] \leftarrow \text{mergesort}(A[(h+1) \ldots |A|])$
    ▷Anschließend mischen wir die beiden sortierten Arrays zusammen:
    $B[\,] \leftarrow \text{merge}(A[0 \ldots h], A[(h+1) \ldots |A|])$
    return $B[\,]$

erudecorp⊥

Satz 1.6 *Die Laufzeit von MergeSort beträgt* $O(n \log(n))$.

Der entsprechende Beweis findet sich in jedem herkömmlichen Lehrbuch über Datenstrukturen und Algorithmen. Er soll hier nur kurz skizziert werden.

Beweis:

Das rekursive Aufspalten läuft offensichtlich in $\log(n)$ Zeit ab (da man dadurch einen binären Rekursionsaufrufbaum der Tiefe $\log(n)$ erhält. Das Mischen ist linear mit $|A| + |B|$. Da auf jeder Rekursionsstufe $2n\frac{1}{2^{\text{Rekursionstiefe}}}$ Daten gemischt werden, dies aber auch entsprechend oft geschieht wird, erhalten wir als Laufzeit

$$\sum_{i=1}^{\log n} 2n \frac{1}{2^{\text{Rekursionstiefe}}} 2^{\text{Rekusionstiefe}} = \sum_{i=1}^{\log n} 2n,$$

was offensichtlich in $O(n \log n)$ liegt. ■

1.3.2 *k*-way-MergeSort

Aber es geht noch besser: Bislang haben wir das Array nur in zwei Hälften aufgespalten und rekursiv sortiert. Der Algorithmus *k-way-Mergesort* verfolgt die Idee, das Array nicht in zwei, sondern in eine –vom Benutzer festgelegte– Anzahl k gleicher Teile aufzuspalten und diese rekursiv zu sortieren. Die Funktion *merge* muss natürlich entsprechend zu einer Funktion *k-way-Merge* umgestaltet werden, dass sie mehr als zwei Eingabearrays verarbeiten kann; das Anfertigen dieser trivialen Funktion sei dem Leser zur Übung empfohlen.

procedure k-way-mergesort
(input: Array A; output: sorted array B)
  ▷Teile das Array in k gleiche Teile und sortiere sie:
  for($i \leftarrow 0$ **to** $k-1$):
    ▷Bestimme Ober- und Untergrenze:
    $\alpha \leftarrow \frac{i}{k}|A|$
    $\beta \leftarrow \frac{i+1}{k}|A|$
    ▷Sortiere rekursiv:
    $A[\alpha \ldots \beta] \leftarrow$ k-way-mergesort$(A[\alpha \ldots \beta])$
  rof
  ▷Zuletzt mischen wir die k sortierten Arrays wieder zusammen:
  $B[\,] \leftarrow$ k-way-merge$(A[0, \ldots], \ldots, A[\ldots, |A|]))$
  return $B[\,]$
erudecorp⊥

Auch *k-way-Mergesort* läuft in Zeit $O(n \log n)$ — allerdings ist der Logarithmus hier zur Basis k.

Satz 1.7 *Die Laufzeit von k-way-mergesort beträgt* $O(n \log_k(n))$.

Beweis:

Der Beweis funktioniert analog zum Beweis beim normalen Mergesort. Das rekursive Aufspalten läuft offensichtlich in $\log_k(n)$ Zeit ab, da man dadurch einen Rekursionsaufrufbaum erhält, der an jedem Rekursionsknoten k Kinder hat und somit als maximale Tiefe $\log_k(n)$ aufweist. ■

1.3.3 Linearzeit

Das Schöne an einer Laufzeit $O(n \log_k n)$ ist, dass $\log_k(n)$ für große k nur sehr, sehr langsam ansteigt und man es somit als nahezu konstant ansehen kann — jedoch nur nahezu.

Die klassische Informatik vermochte es allerdings trotz jahrelanger Forschung nicht, hieraus den Schluss zu ziehen, k zu maximieren. Dieser Erfolg wurde erst der Rautavistischen Informatik zuteil.

Der sowohl naheliegende wie auch einfache Gedanke ist, k mit n gleichzusetzen, so dass man schließlich n-way-Mergesort erhält:

procedure n-way-mergesort
(input: Array A; output: sorted array B)
    ▷Die folgende Zeile ist der **einzige** Unterschied zu k-way-Mergesort:
    $k \leftarrow (|A| - 1)$
    ▷Der Rest wie in k-way-Mergesort:
    for($i \leftarrow 0$ **to** $k - 1$):
        $\alpha \leftarrow \frac{i}{k}|A|$
        $\beta \leftarrow \frac{i+1}{k}|A|$
        $A[\alpha \ldots \beta] \leftarrow$ k-way-mergesort$(A[\alpha \ldots \beta])$
    rof
    $B[\,] \leftarrow$ k-way-merge$(A[0, \ldots], \ldots, A[\ldots, |A|]))$
    return $B[\,]$
erudecorp$\perp$

Satz 1.8 *Der vergleichsbasierte Sortieralgorithmus n-way-MergeSort läuft in linearer Zeit (also in* $O(n)$*).*

Beweis:

n-way-MergeSort ist einfach k-way-MergeSort, bei dem $k := n$ gesetzt wurde. Somit erhalten wir als Laufzeit

$$O(n \log_k n) = O(n \log_n n) = O(n \cdot 1),$$

also eine lineare Laufzeit für vergleichsbasiertes Sortieren. ■

2 Rekursion

Die Rekursion ist ein für die Informatik sehr zentrales Thema. In diesem Kapitel wird versucht, dem Leser das Prinzip der Rekursion mit Hilfe neuartiger didaktischer Methoden auf anschauliche Art und Weise zu vermitteln. Dies trägt – hoffentlich – zu einem tieferen Verständnis dieses fundamentalen Prinzips bei.

2.1 Die Abbruchbedingung

Leser, die mit dem Prinzip der Rekursion vertraut sind, können einzelne Abschnitte dieses Kapitels (bzw. auch gleich das gesamte Kapitel) überspringen und zum nächsten übergehen.

2.2 Verschränkte Rekursion

Das Prinzip der verschränkten Rekursion werden wir später im Skript in Abschnitt 2.4 genauer behandeln. Zunächst wollen wir uns jedoch der einfachen Rekursion zuwenden.

2.3 Einfache Rekursion

Auf das Prinzip der einfachen Rekursion gehen wir genauer in Abschnitt 2.3 ein. Da zu erwarten ist, dass das Verständnis dieses Abschnittes einiges an Grundlagenwissen erfordert, möchten wir den Leser dazu anregen, während der Lektüre gelegentlich Abschnitt 2.1 zu konsultieren.

2.4 Verschränkte Rekursion

Das Prinzip der verschränkten Rekursion wird in Abschnitt 2.2 eingehend behandelt. Auch hier möchten wir den Leser dazu animieren, während der Lektüre gelegentlich Abschnitt 2.1 zu konsultieren.

3 Graphenalgorithmen und Berechenbarkeitstheorie

Mit Hilfe von Graphen lassen sich sehr viele Berechnungsprobleme modellieren und lösen. Graphenalgorithmen sind daher ein sehr wichtiges Forschungsgebiet der Informatik. Auch die Rautavistische Informatik hat mit der Theorie der DEG-Graphen einen wichtigen Beitrag geleistet, da sich sehr viele praktische Probleme extrem effizient lösen lassen, wenn man sie als DEG-Graph formuliert.

3.1 DEG-Graphen

Wir wollen zunächst den für die rautavistische Graphentheorie sehr zentralen Begriff einführen, nämlich den sog. *DEG-Graphen*.

Definition 3.1 (Deutlich endlich großer Graph) *Ein Graph $G = (V, E)$ heißt deutlich endlich groß (oder auch DEG-Graph), wenn alle nachfolgend aufgeführten Bedingungen erfüllt sind:*

1. $|E| \geq |V| \geq 1$
2. *G ist stark zusammenhängend.*
   Formal: $\forall v_1, v_2 \in V : \exists \text{Pfad } v_1 \to \ldots \to v_2$
3. $(\exists \textit{Pfad } v_1 \to v_2 \to \ldots \to v_r \wedge \exists \text{Pfad } v_r \to v_{r+1} \to \ldots \to v_i \to v_1)$
   $\Rightarrow (v_1 = v_r)$

Die dritte Bedingung für DEG-Graphen nennt man auch die sog. *R-Bedingung*. Im folgenden Abschnitt sehen wir nun eine Anwendung von DEG-Graphen.

3.2 DEG-Graphen als Turingmaschinenzustandsübergangsgraphen

Wir wollen nun zeigen, dass das Halteproblem für Turingmaschinen –unter gewissen Randbedingungen– entscheidbar ist.

Satz 3.1 *Sei $G = (V, E)$ der Zustandsübergangsgraph einer Turingmaschine $\mathcal{M}$. Wenn G ein DEG-Graph ist, dann ist das Halteproblem für $\mathcal{M}$ entscheidbar.*

Beweis:

Der Beweis dieses Satzes ist dreiteilig.

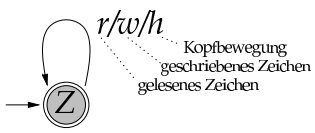

Abbildung 3.1: Die Struktur $\dot{\pi}$. (Z ist ein Endzustand.)

Lemma 3.1 (Zustandsübergangsgraphenendschleifendeterminiertheit) *Der Zustandsübergangsgraph einer Maschine $\mathcal{M}$ enthalte die in Abbildung 3.1 gezeigte Struktur $\dot{\pi}$. Wenn der beschriebene Zustand Z erreichbar und der Bandinhalt bekannt ist, dann ist das Halteproblem (ab Erreichen des Zustandes Z) für $\mathcal{M}$ entscheidbar.*

Unterbeweis:

Das gelesene Zeichen sei x. Wir unterscheiden nun folgende Fälle:

$r \neq x$ In diesem Fall terminiert $\mathcal{M}$ sofort, da wir uns in einem Endzustand befinden.

$r = x$ Hier müssen wir je nach Kopfbewegung verschiedene Unterfälle betrachten:

- $h =$ **„links“** Da keine andere Kopfbewegung als nach links möglich ist, muss beim ersten nicht-passenden Zeichen die Maschine terminieren. Da die Eingabe von $\mathcal{M}$ zwar beliebig groß, aber endlich ist, und da wir den Zustand Z innerhalb einer endlichen Anzahl an Schritten erreicht haben, kann das Band auch nur endlich viele Zeichen enthalten. $\mathcal{M}$ muss in diesem Fall also irgendwann zwangsläufig terminieren.
- $h =$ **„rechts“** analog zu „links“
- $h =$ **„bleibt“** Falls $r \neq w$, dann terminiert $\mathcal{M}$ nie, da auf ewige Zeiten ein falsches Zeichen an die Kopfposition geschrieben wird und sich der Kopf nicht mehr bewegen kann. Falls hinwegen $r = w$, dann terminiert $\mathcal{M}$ unmittelbar im nächsten Schritt (s. o.).

Unabhängig davon, welcher der Fälle eintritt, können wir also stets entscheiden, ob $\mathcal{M}$ terminiert oder nicht. □

Lemma 3.2 *Jeder DEG-Graph G enthält genau eine Struktur $\dot{\pi}$.*
(Ohne Beweis.)

Bevor wir nun mit dem dritten Lemma fortfahren, wollen wir jedoch aus didaktischen Gründen über die Struktur eines typischen DEG-Graphen sprechen. Wir wollen nun den Graphen aus Abbildung 3.2 schrittweise in einen DEG-Graphen umformen.

Zunächst stellen wir fest, dass wir beispielsweise von Knoten D ausgehend den Knoten B nicht erreichen können, was der Bedingung des starken Zusammenhangs (Bedingung 2) widerspricht. Daher fügen wir eine Kante $D \to B$ ein (Abbildung 3.3). Nun wenden wir uns der *R-Bedingung* zu (Bedingung 3). Da ein Pfad $B \rightsquigarrow C$ existiert (via $B \to A \to C$, und gleichzeitig auch ein Pfad $C \rightsquigarrow B$ (via $C \to D \to B$), muss $B = C$ gelten. Das Ergebnis ist in Abbildung 3.4 zu sehen. Auch bei der Betrachtung von A und D fällt auf, dass $A \rightsquigarrow D$ (via $A \to B/C \to D$) und $B \to A$, also müssen auch diese beiden Knoten unifiziert werden

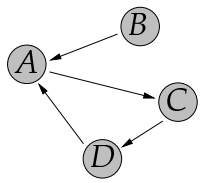

Abbildung 3.2: Unser Ausgangsgraph.

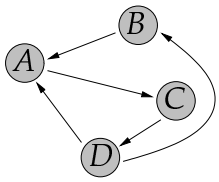

Abbildung 3.3: Herstellung des starken Zusammenhangs.

(Abbildung 3.5). Wir wenden nun die R-Bedingung ein letztes Mal an wegen $A/D \leftrightarrow B/C$ und erhalten so einen DEG-Graphen (Abbildung 3.6).

Korollar 3.1 *Das vorhergehende Lemma muss ergänzt werden durch den Nachsatz „. . . und besteht auch aus nix anderem sonst“.*
(Ohne Beweis.)

■

3.3 Universelle Graphenalgorithmen für DEG-Graphen

Bislang haben wir deutlich endlich große Graphen nur im Zusammenhang mit der Berechenbarkeitstheorie kennengelernt. Nun wollen zeigen, dass sie nicht nur theoretisch, sondern auch praktische Anwendung finden.

Wir betrachten zunächst das Programm *first-vertex*:

procedure first-vertex
(input: Knotenmenge V eines DEG-Graphen; output: ein Array A von Knoten)
    $A[1] \leftarrow$ erstes Element von V
    return A
erudecorp$\perp$

Man macht sich leicht klar, dass der Algorithmus *first-vertex* universell verwendbar ist. Beispielsweise lassen sich damit viele Lösungen für sowohl in der Praxis als auch in Theorie häufig auftretende Graphenprobleme zumindest auf DEG-Graphen sehr schnell berechnen.

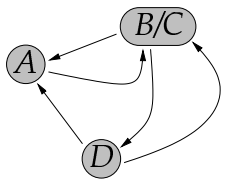

Abbildung 3.4: Anwendung der R-Bedingung.

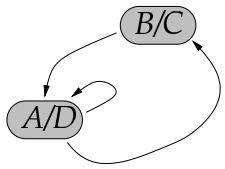

Abbildung 3.5: Nochmalige Anwendung der R-Bedingung.

Wir führen hier nur einige Beispiele auf:

- Bestimme die Menge aller Knoten, die von einem gegebenen Knoten v aus erreichbar sind
- Bestimme die Menge aller Knoten des kürzesten Weges $v_1 \rightsquigarrow v_2$ für beliebige $v_1, v_2 \in V$ mit mindestens n Knoten, $n \geq 1$
- Bestimme die Menge aller Knoten des längsten Weges $v_1 \rightsquigarrow v_2$ für beliebige $v_1, v_2 \in V$.
- Bestimme alle Knoten der größten starken Zusammenhangskomponente von G
- Bestimme alle Knoten der kleinsten schwachen Zusammenhangskomponente von G

Satz 3.2 *Die Laufzeit von first-vertex beträgt* $\mathrm{O}(\sin^2 n + \cos^2 n)$ *(n:=Größe des DEG-Graphen).*

Beweis:

Sei $\gamma = (\hat{\xi}, \tilde{\zeta}, \check{\xi}, \check{\zeta}, \bar{\xi})$ die aufspannende $\Gamma^{\rceil_n}$-Grammatik der kategoriellen Summe $\hat{\Sigma}_k$ über der Wurzelbasisinvariantenmatrix $I_{\vee}$ aller Elemente aus E. Nach dem Satz von Leibrock existiert eine aufsteigende Faktorzerlegung $\sigma_1 \circ \sigma_2 \circ ... \circ \sigma^{I_{:}^{0}}_{\nabla^{i}_{\mathrm{dq}}} \preccurlyeq \hat{\Sigma}_k$ derart, dass ein maximales σ_k existiert, welches alle semidefinit beschränkten Superterminale aus $[\check{\xi}]$ dominiert. Nach dem Lemma von Kasimir-Kostonjenko ist damit $(\check{\xi} \curlyvee \check{\zeta})$ irreduzibel und somit das monolithische Dual $\hat{\xi}^{\bar{\xi}^{\check{\zeta}_0} : \check{\xi}_{\zeta}}$ ein linksfreies Halbmonoid über γ. Dadurch lässt sich eine Differentialsperre $\Delta S\prime$ mit einem zu $\widehat{<\gamma>}$ antiparallel rechtsdrehenden Gegenfeld $\overset{\curvearrowleft}{G}$ konstruieren, so dass $\underset{\rightarrow}{\overset{\curvearrowleft}{G}}$ eine Ekliptik ε mit Ξ^n-kompaktem Frühlingspunkt $\curlyvee$ tangiert, dessen isostatische Parallaxenverschiebung δp geriontologisch modulo der strikten Indexsumme $\widehat{\bar{\xi}_{\iota} \prime_J}$ aus defizitärer Präzession π' und stratosphärischer Abberationsalbedo

Abbildung 3.6: Das Endresultat: Ein DEG-Graph.

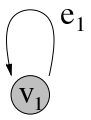

Abbildung 3.7: Ein typischer DEG-Graph.

α_α den Zenith $\top$ auf den Nadir $\bot$ in eine Singularität $\circledcirc$ abbildet, wenn Uranus $\sharp$ in unterer Konjunktion σ zur Sonne $\sun$ oder Saturn $\hbar$ im Jupiter $\jupiter$ steht[1], sogar an beliebigen Weihnachten w^* ohne τ-Wetter und für φ-Zuchtnomaden, die beim Skifahren $\pi \mathrm{st}^N$-Säue sind; aber diese Geschichte soll ein andermal erzählt werden. ■

Der vorgestellte Beweis zeigt übrigens die Beweismethode *Beweis durch extremes Dummschwätzen unter Zuhilfename von Fremdworten und komischen Symbolen*, kombiniert mit der Beweismethode *Verweis auf das Lemma von Kasimir-Kostonjenko* [FKK92].

Zum Abschluss dieses Kapitels betrachten wir noch kurz einen zweiten Universalalgorithmus für DEG-Graphen.

procedure random-edge
(input: Kantenmenge E eines DEG-Graphen; output: eine Kante e)
    $e \leftarrow$ zufällig gewähltes Element aus E
    return e
erudecorp$\bot$

Auch mit diesem Algorithmus lassen sich etliche Graphenprobleme für DEG-Graphen in konstanter Zeit lösen, beispielsweise die Menge aller Kanten, die auf dem kürzesten Weg $v_1 \rightsquigarrow v_2$ liegen.

Satz 3.3 *Für Eingabegröße m beträgt die Laufzeit von random-edge* $\mathrm{O}(1 + \pi^{\sinh^{-1} 1 + \sqrt{\frac{\cos n}{n^3}}})$.

Der Beweis ist nicht sehr kompliziert und funktioniert prinzipiell wie der Beweis zu 3.2, mit dem Unterschied, dass hier nach dem Speziellen Vermischungsprinzip über austrodravidische Agglutinationsklimaten induziert wird, von denen sich zeigen lässt, dass sie aride Gebiete mit ammonitischen Radiolariealveolaren enthalten.

Dem Leser mag die Beschäftigung mit DEG-Graphen noch etwas eintönig und nicht besonders interessant erscheinen, wie die vorhergehende Abbildung 3.7 eines DEG-Graphen möglicherweise vermuten ließe. Doch sollte man sich vor Augen führen, dass es auch sehr viel komplexere DEG-Graphen als den von Abbildung 3.7 gibt, wie uns die folgende Abbildung 3.8 deutlich zeigt.

[1] Dies wurde von Nostradamus bereits 1561 beschrieben; in Versen heißt es bei ihm: *„Soît Saturnus dedans Jupitre/ la Vénus n'importe où/ Terre renverse-t-elle et feux/ célestiels sous nôtres pieds“*. Auch die moderne Astronomie geht stark davon aus, dass es ein großes Feuer geben wird, wenn Saturn im Jupiter steht. Eine untere Konjunktion von Uranus würde die Erde $\sharp$ ebenfalls negativ beeinflussen.

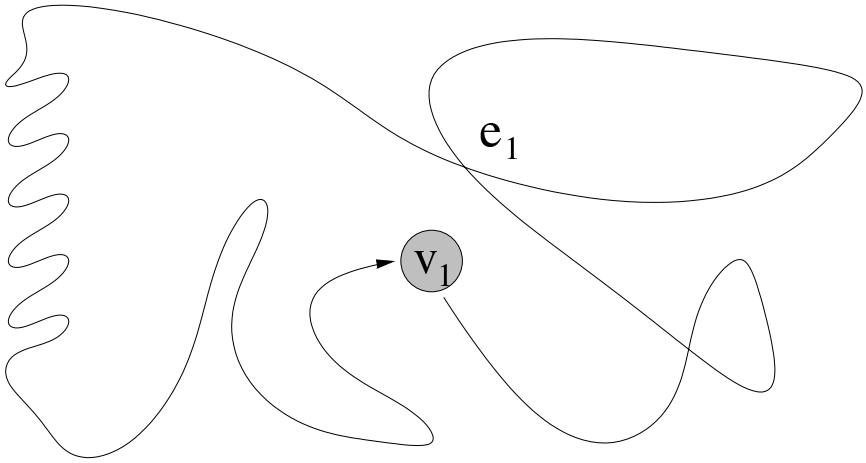

Abbildung 3.8: Ein komplexer DEG-Graph.

4 Bäume

Wie die klassische Informatik befasst sich auch die rautavistische Informatik viel mit einer speziellen Graphenart, nämlich mit Bäumen. Wir wollen in diesem Abschnitt kurz einige klassische Bäume und ihre Eigenschaften vorstellen.

4.1 Klassische Bäume

4.1.1 Buche

Definition 4.1 *Ein Baum $b \in B$ heißt Buche, wenn gilt:*

$$\begin{aligned} & b \text{ ist sommergrün} \\ \wedge\ & \Pr(b \text{ ist hoch}) \gg 0.0 \end{aligned}$$

wobei für die Menge aller Buchen B gelten muss:

$$\begin{aligned} & B \mathrel{\hat{=}} \text{„}\textit{fagus}\text{“} \\ \wedge\ & B \subseteq \textit{Familie der Buchengewächse.} \end{aligned}$$

Die Buche (*fagus*) ist also ein sommergrünes, meist hohes Gewächs aus der Familie der Buchengewächse.

Lemma 4.1 *Von besonderer Bedeutung ist die bis zu $\lambda = 40\text{m}$ hohe Rotbuche R, die in Gebirgsgegenden und auf Kalkböden Mitteleuropas heimisch ist.*
(Ohne Beweis.)

Definition 4.2 *Eine Menge von Früchten $\{f | f \in F(R)\}$ einer Rotbuche R heißt Bucheckern.*

Satz 4.1 *Das mittelharte und recht haltbare Holz $h(R)$ einer Rotbuche R wird meist als Faser-, Werk-, Brenn-, Möbel- und Vorkopfbrettholz genutzt.*

Beweis:

Trivial. ■

4.1.2 Eiche

Eichen und Buchen sind sich definitionsgemäß recht ähnlich:

Definition 4.3 *Ein Baum $\varepsilon \in E$ heißt Eiche, wenn gilt:*

$$\begin{aligned} & \max \text{Alter}(\varepsilon) \not\leqslant 700 \textit{Jahre} \\ \wedge\ & \varepsilon \textit{ ist sommergrün} \\ \wedge\ & \Pr(\varepsilon \textit{ ist immergrün}) \gg 0.0 \\ \wedge\ & \text{Knospen}(\varepsilon) = \textit{vielschuppig} \\ \wedge\ & h(\varepsilon) \subseteq \{\textit{wertvolles Nutzholz}\} \end{aligned}$$

wobei für E gilt, dass

$$\begin{aligned} & E \triangleq \textit{„quercus“} \\ \wedge\ & E \subseteq \textit{Familie der Buchengewächse.} \end{aligned}$$

Die Eiche (*quercus*) ist also eine Gattung der Buchengewächse, deren Holz wertvolles Nutzholz ist. Eichen sind sommer-, zum Teil auch immergrüne Bäume, die bis zu 700 Jahre alt werden können. Sie bilden vielschuppige Knospen.

Definition 4.4 *Die Früchte $\{n \in F(E)\}$ (Nüsse) einer Eiche E heißen Eicheln.*

(Auch andere Dinge werden als Eichel bezeichnet; sie sollen hier jedoch nicht eingeführt werden.)

Von der Eiche gibt es zahlreiche Varianten, beispielsweise die Stieleiche, Traubeneiche, Roteiche, Steineiche, Korkeiche, Sumpfeiche, Libanoneiche, Wasserleiche, Chlorbleiche, Durchreiche, Radspeiche, Tagundnachtgleiche, Ölscheiche — um nur einige zu nennen.

4.2 Satz der winterunendlichen Bäume

In diesem Kapitel werden wir eine äußerst erstaunliche Eigenschaft von Laubbäumen (wir zeigen sie aus didaktischen Gründen nur für Buchen und Eichen) kennenlernen, die sich im sog. „Satz der winterunendlichen Bäume“ niederschlägt:

Satz 4.2 *Buchen und Eichen sind im Winter unendlich groß.*

Dieser durchaus verblüffende Sachverhalt wurde bislang von Biologen offensichtlich übersehen, und wurde eher zufällig von der Rautavistischen Informatik entdeckt. Dabei ist der Beweis relativ einfach:

Beweis:

Es sei $B = (V, E)$ ein beliebiger Baum, wobei $B \in \mathit{Buchen} \cup \mathit{Eichen}$ gelte.

Nun breche der Winter w herein (wobei w frei gewählt werden kann). Da sowohl Buchen als auch Eichen nach Definition o. B. d. A. sommergrün sind, muss B seine Blätter in w abwerfen.

Dass B jetzt keine Blattknoten mehr hat, impliziert, dass für jeden Knoten $v \in V$ im Baum mindestens eine ausgehende Kante $e \in E$ zu einem Nachfolgeknoten $v' \in V$ existieren muss.

Da B ein Baum ist, kann B keine Zyklen enthalten, so dass induktiv jeder $v \in V$ mindestens einen unendlich langen Nachfolgerpfad $v \to v' \to \ldots$ aus paarweise verschiedenen Knoten besitzen muss.

Daraus folgt, dass $|V| = \infty$ und somit auch $|E| = \infty$ (da alle Knoten mit mindestens einem anderen verbunden sind). Somit ist der Baum B für beliebige Winter w unendlich groß. ■

(Zusammengefasst funktioniert der Beweis also so, dass Bäume im Winter ihre Blätter abwerfen, und dass Bäume ohne Blätter unendlich groß sein müssen.)

4.3 Gelb-Grün-Bäume

Gelb-Grün-Bäume sind die kleineren Verwandten der Rot-Schwarz-Bäume, die aus Informatik V bekannt sein sollten. Der einzige Unterschied zu den Rot-Schwarz-Bäumen ist der, dass bei ihnen die roten Knoten grün und die schwarzen Knoten gelb gefärbt sind. (Gleiches gilt auch für die Variante, bei der die Kanten statt der Knoten gefärbt werden.)

In der Realität hat sich jedoch gezeigt, dass Gelb-Grün-Bäume praktisch oft unimplementierbar sind, da man sie erst ab $\alpha \geq 5\%$ benutzen kann.

5 Hashing

Auch Hashing ist ein grundlegendes algorithmisches Feld der Informatik. Physikalisches Hashing ermöglicht Produkte wie z. B. Hackfleischbällchen; Hashing in der Informatik wird hingegen für wichtige Komponenten wie z. B. Lookup-Tabellen o. ä. eingesetzt.

Das wichtigste Problem beim Hashing ist die Wahl einer geeigneten Hashfunktion. Hier konnte die rautavistische Informatik bahnbrechende Erfolge vorweisen, indem einerseits beweisbar kollisionsfreie Hashfunktionen als auch andererseits besonders platzsparende Hashfunktionen entwickelt wurden.

5.1 Schnelle Write-Operationen

Ein generell auftretendes Problem beim Verwenden von Hashes ist die Tatsache, dass Schreib-Operationen lange brauchen können. Dieses Verhalten kann insbesondere in Echtzeitsystemen mit Antwortgarantieen zu Problemen führen. Die rautavistische Informatik hat daher eine sehr einfach zu implementierende Menge an kollisionsfreien Hashfunktionen zur Beschleunigung des Schreibvorgangs entwickelt.

Definition 5.1 (Rautavistische Hashfunktionen) *Die Indexumkehrfunktionen $h(x_i) = f^{-1}(i)$ von in der Menge $M := \{f_p\}$ enthaltenen Funktionen $f_p : V \to K$ mit $V \subseteq \mathbb{N} \setminus \{0\}$ und*

$$f_p(v) = \int \cos v \cdot \left(\sin v \cdot \tan v + \sin\left(v + \frac{\pi}{2}\right)\right) + \cosh v \cdot \left(\sinh' v - \frac{(\cosh' v)^2}{\sinh' v}\right) \; dv$$

heißen rautavistische Hashfunktionen.

Satz 5.1 (Rautavistisches Hashing mit schnellen Write-Operationen) *Die Größe der Hashtabelle H sei $|H| = n$. Die zu hashenden Elemente seien m. B. d. A. in der Reihenfolge der Verarbeitung $(x_1, x_2, \ldots, x_n)$. Werden die Elemente x_i mit einer beliebigen rautavistischen Hashfunktion gehasht, dann treten für beliebige Füllstände $\alpha \leq 1$ der Hashtabelle keinerlei Hashkollisionen auf.*

Beweis:

Der Beweis ist sehr einfach und soll hier nur skizziert werden. Genauer gesagt, erfolgt dies in Abbildung 5.1. Die grundlegende Beweisidee ist die, dass sich zeigen lässt, dass $h(\cdot)$ folgende angenehme mathematische Eigenschaft aufweist: $\forall x_i : h(x_i) = i$. ■

Korollar 5.1 *Aus der Kollisionsfreiheit folgt unmittelbar, dass Hashing mit rautavistischen Hashfunktionen in Zeit $O(1)$ abläuft (da man keinerlei Kollisionsauflösungsstrategie implementieren muss).*

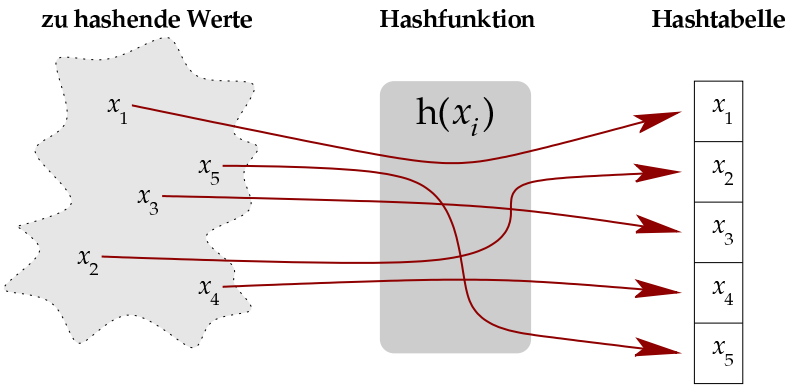

Abbildung 5.1: Rautavistisches Hashing verursacht keine Kollisionen.

5.2 Platzsparendes Hashing

Nun wollen wir eine Weiterentwicklung des rautavistischen Hashings kennenlernen. Hierbei wird bewusst die Garantie der Kollisionsfreiheit gegen einen geringeren Speicherverbrauch eingetauscht.

Definition 5.2 (Rautavistisch nichtinjektive Hashfunktion) *Sei $h_p(\cdot)$ eine beliebige Hashfunktion. $\breve{h}_p$ heißt rautavistisch nichtinjektive Hashfunktion, wenn gilt:* $\breve{h}_p = \frac{h_p}{h_{2p+1}}$

Die Funktionsweise von platzsparendem Hashing unter Verwendung rautavistisch nichtinjektiver Hashfunktionen ist in Abbildung 5.2 schematisch dargestellt.

Satz 5.2 *Rautavistisch nichtinjektives Hashing verbraucht Speicherplatz $O(1)$.*
(Ohne Beweis.)

Aufgrund der nun wieder zugelassenen Kollisionen empfiehlt sich, entweder eine Kollisionsauflösungsstrategie zu verwenden, oder –analog zu Abschnitt 1.1.2– verlustbehaftet zu hashen.

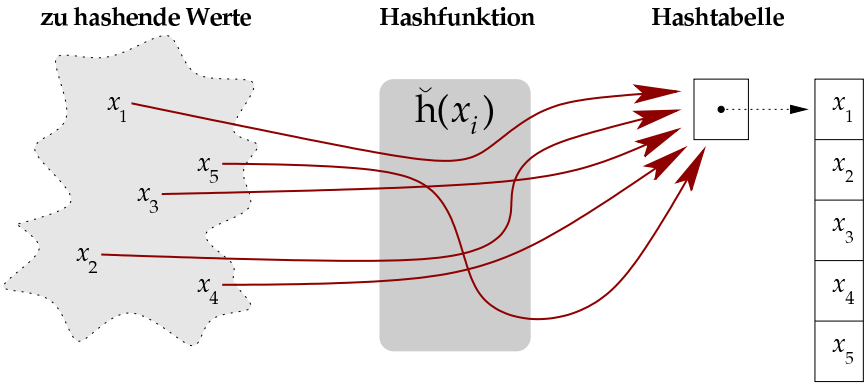

Abbildung 5.2: Rautavistisches Hashing unter Verwendung rautavistisch nichtinjektiver Hashfunktionen erzeugt zwar Kollisionen, benötigt aber nur eine sehr kleine Hashtabelle.

6 Kryptographie

Die Kryptographie[1] ist ein weiteres wichtiges Forschungsgebiet der Informatik. Auch hier konnte die rautavistische Informatik wertvolle Beiträge leisten.

6.1 Verteilung von Geheimnissen

In der Informatik trifft man oft auf das Problem, wie man eine geheime Information möglichst sicher aufbewahrt. Hierbei ist ein beliebter Ansatz, das Geheimnis in n Teile aufzuspalten und diese zu verteilen. Dabei sind die n Teile so gestaltet, dass man mindestens k Teile benötigt, um das Geheimnis wiederherzustellen.

Die klassische Informatik beschäftigt sich mit diesem Problem unter der Nebenbedingung, dass $k \leq n$. In der rautavistischen Informatik betrachten wir hingegen $k > n$.

Ein möglicher Ansatz könnte folgendermaßen aussehen:

procedure *splitSecret*$_1$
(input: Array $\imath$; output: Array ς)
    for $i \leftarrow |\imath|$ **downto** 0:
        $\varsigma[i] \leftarrow 1$;
    rof
    return ς;
erudecorp $\perp$

Erkl"arung:

Für jedes der n Elemente des Eingabearrays $\imath$ wird ein neues Geheimnis („1“) erzeugt und im Ausgabearray ς abgelegt.

Wenn man anschließend alle n Elemente von ς zusammensetzt, kann man trotzdem nicht die ursprüngliche Nachricht rekonstruieren.

Satz 6.1 *Um die mit splitSecret*$_1$ *berechnete Nachricht wiederherzustellen, braucht man mehr als n Geheimnisse*[2].

Wir können bei den folgenden Betrachtungen nun o.B.d.A. davon ausgehen, dass $\imath \neq [1, \ldots, 1]$.

[1] auf deutsch etwa: Kartierung bzw. Bemalung von Kellern in Kirchen
[2] (oder sauviel Schwein)

Beweis:

Gegenannahme: Es sei möglich, auf einfache Art und Weise aus ς die ursprüngliche Information $\imath$ wiederherzustellen.

Lemma 6.1 *Dann ist entweder $\imath = [1, \ldots, 1]$, oder der Rechner wurde gehackt.*

Unterbeweis:

Im ersten Fall ist unsere Annahme falsch, die wir ohne Beschränkung der Allgemeinheit getroffen haben. Da die Allgemeinheit nach allgemeiner Menschenkenntnis allerdings ohnehin immer beschränkt ist, folgt hier nach dem Satz des unlogischen Umkehrschlusses[3], dass die Aussage falsch ist und somit $\imath = [1, \ldots, 1]$ und damit gemäß des Allgemeinen Konfusionsprinzips[4] unsere Annahme nicht stimmt oder vielleicht auch doch, was uns dann aber auch völlig egal sein kann.

Im zweiten Fall hilft uns ein weiteres Lemma:

Lemma 6.2 *Wenn der Rechner gehackt wurde, dann war entweder $\imath$ kein Geheimnis, oder der Angreifer hat ein $(n+1)$tes Geheimnis verwendet.*

Damit ist der Beweis für unser erstes Lemma abgeschlossen. □

Nun müssen wir nur noch unser zweites Lemma beweisen, und wir sind fertig.

Unterbeweis:

Der Angreifer kann o.B.d.A. (siehe oben) den Rechner nur in folgenden drei Fällen hacken:

1. *Der Angreifer wusste das* `root`*-Passwort.*

   In diesem Fall kennt der Angreifer offensichtlich ein $(n+1)$tes Geheimnis: nämlich das `root`-Passwort.

2. *Das* `root`*-Passwort war so bescheuert gewählt, dass der Angreifer es mit einem Brute-Force-Angriff herausbekommen konnte.*

   Einem solchen Rechner vertraut man kein Geheimnis an; $\imath$ kann also kein Geheimnis sein.

3. *Der Rechner verwendet kein Richtiges Betriebssystem*^{TM}.

   Einem solchen Rechner vertraut man gleich drei mal kein Geheimnis an; $\imath$ kann somit weder ein Geheimnis sein, noch ein Geheimnis sein, noch ein Geheimnis sein.

Damit ist auch unser zweites Lemma bewiesen. □

Offensichtlich benötigt man also zum Wiederherstellen der ursprünglichen Information k Geheimnisse, mit $k > |\varsigma| \geq |\imath| \geq n$. ■

Bemerkung 6.1 *Die Schleifenvariable i in unserem Programm zählt nur deshalb von oben herunter, damit der Leser verwirrt wird.*

[3] Siehe Vorlesung „Rautavistische Mathematik“

[4] Bekannt aus „Rautavistik I“

6.2 Das Autokrypt-Verfahren

Viele große Probleme der Kryptographie haben etwas mit Schlüsseln zu tun. Beispielsweise muss die Übermittlung eines gemeinsamen Schlüssels bei der Verwendung symmetrischer Kryptographie über einen sicheren Kanal erfolgen, was zumindest recht aufwändig[5] ist.

Daher hat die rautavistische Informatik das sogenannte *Autokrypt*-Verfahren entwickelt. Es handelt sich hierbei um ein symmetrisches Verfahren, das ohne Schlüssel auskommt, aber trotzdem die Information wirksam vor Angreifern schützt. Die zugrundeliegende Idee ist die, den Text mit sich selbst zu verschlüsseln.

procedure Autokrypt
(input: Array I (entschlüsselt); output: Array O (verschlüsselt))
    for $i \leftarrow 0$ **to** $|I|$:
        $O[i] \leftarrow I[i]$ **xor** $I[i]$
    rof
    return O
erudecorp $\perp$

Der Angreifer sieht, wie man sich leicht klarmacht ($\forall x : x \veebar x = 0$) nur einen Strom aus Nullen, aus dem er lediglich die Länge der ursprünglichen Nachricht ableiten kann. Selbst mit differentieller Kryptanalyse oder ähnlich fortgeschrittenen Methoden kann er den Klartext nicht ableiten; es sei denn, dass er den Klartext (welcher ja gleichzeitig den Schlüssel darstellt!) kennt und damit die Nachricht entschlüsseln kann.

Der rechtmäßige Empfänger der Nachricht kann –und das ist das Besondere am Autokrypt-Verfahren– die Nachricht entschlüsseln, ohne vorher den Schlüssel übermittelt zu bekommen. Hierzu macht er sich die Eigenschaft der verschlüsselten Nachricht, dass sie ihren eigenen Schlüssel enthält, zunutze, indem er folgendes rekursive Entschlüsselungsverfahren namens *Autodekrypt* anwendet:

Zunächst dekodiert er die verschlüsselte Nachricht. Hierdurch erhält er nicht nur den Klartext, sondern gleichzeitig auch den Schlüssel, mit dem er nun wiederum die verschlüsselte Nachricht dekodieren kann. Dies wird für jedes Zeichen iteriert:

procedure Autodekrypt
(input: Array I (verschlüsselt); output: Array O (entschlüsselt))
    for $i \leftarrow 0$ **to** $|I|$:
        $O[i] \leftarrow$ dekrypt-sign$(I[i])$
    rof
    return O
erudecorp $\perp$

Der Algorithmus ruft für jedes zu dekodierende Zeichen also folgende Hilfsfunktion auf, die nach dem beschriebenen rekursiven Verfahren arbeitet:

[5] Die neue Recht Schreibung sieht irgendwie doof aus.

procedure dekrypt-sign
(input: verschlüsseltes Zeichen e; output: entschlüsseltes Zeichen d)
    ▷Schlüssel rekursiv bestimmen:
    $k \leftarrow$ dekrypt-sign(e)
    ▷Mit dem Schlüssel können wir jetzt dekodieren:
    $d \leftarrow e$ **xor** k
    return d
erudecorp⊥

Ein kleiner Nachteil der Autokrypt-Verschlüsselung sei nicht verschwiegen: Ein kompletter Durchlauf von *dekrypt-sign* dauert verhältnismäßig lange.

Beim alternativen Entschlüsselungsverfahren *Presend-Autokrypt* wird daher der Schlüssel vor der Übertragung der verschlüsselten Nachricht auf einem sicheren Kanal dem Empfänger bekanntgemacht. Allerdings erkauft man sich diese schnellere Entschlüsselung durch das Risiko, dass ein Dritter den Schlüsseltext abfangen und damit die Nachricht dekodieren kann.

7 Komplexitätstheorie

7.1 Wiederholung von Grundlagen

In diesem Kapitel gehen wir davon aus, dass der Leser bereits mit dem Konzept von deterministischen und nichtdeterministischen Turingmaschinen vertraut ist. Dennoch sollen einige grundlegende Begriffe aus Gründen der mathematischen Klarheit definiert werden, da sie u. U. in einigen Lehrbüchern in Details anders definiert werden.

Definition 7.1 (Speicherverbrauch einer Turingmaschine) *Eine Turingmaschine $\mathcal{M}$ hat einen Speicherverbrauch s, wenn sie auf s Bandpositionen* ***zusätzlich zur Eingabe*** *zugreift.*

Diese Definition ist sehr universell, da sie die Eingabegröße nicht berücksichtigt und somit auch Algorithmen erlaubt, welche einen Speicherplatzverbrauch aufweisen, welcher sub-linear zur Eingabe ist. Sie ist analog zur Definition in [Sip05]. Ohne das Ignorieren der Eingabegröße wäre beispielsweise eine sinnvolle Definition der Komplexitätsklasse $\mathfrak{L}$ (logarithmischer Speicherverbrauch, siehe [Sip05]) nicht möglich.

Definition 7.2 (Laufzeit einer Turingmaschine) *Eine Turingmaschine $\mathcal{M}$ hat eine Laufzeit $t := t_e - t_0$, wenn sie zum Schritt t_0 zum ersten Mal ein Eingabezeichen vom Band liest, und zum Schritt t_e zum letzten Mal ein Ausgabezeichen auf das Band schreibt.*

Auch diese Definition ist sehr universell, da sie problemlos sowohl auch auf Mehrbandturingmaschinen als auch auf nichtdeterministische Turingmaschinen angewendet werden kann.

7.2 Die Komplexitätsklasse $\mathfrak{NN}$

Wir betrachten den folgenden Algorithmus für eine nichtdeterministische Turingmaschine, welcher sich ausgiebig der Eigenschaft nichtdeterministischer Maschinen bedient, korrekte Lösungen nichtdeterministisch zu erraten:

procedure NN-Universalalgorithmus
(input: I; output: R)
    Rate nichtdeterministisch die korrekte Ausgabezeichenfolge R
    Lies die Eingabezeichenfolge I
    Terminiere
erudecorp$\perp$

Satz 7.1 *Der NN-Universalalgorithmus hat Laufzeit* $O(-1)$.

In anderen Worten: Offensichtlich weist dieser Algorithmus eine negative Laufzeit auf. Der Beweis ergibt sich unmittelbar aus der Laufzeitdefinition.

Definition 7.3 (Komplexitätsklasse $\mathfrak{NN}$) *Alle Probleme, welche sich mit dem NN-Universalalgorithmus lösen lassen, gehören zur Komplexitätsklasse $\mathfrak{NN}$ (**n**ichtdeterministisch, **n**egative Laufzeit).*

Diese Komplexitätsklasse $\mathfrak{NN}$ ist eine sehr mächtige Klasse:

Satz 7.2 (NN-Universalitätssatz) *Alle von einer nichtdeterministischen Turingmaschine berechenbaren Probleme lassen sich mit Hilfe des NN-Universalalgorithmus berechnen.*

(Der Beweis dieses Satzes sei sowohl dem geneigten als auch dem aufrechten Leser zur Übung empfohlen.)

7.3 Die Klasse $\mathfrak{NN}$ und die Komplexitätshierarchie

Neben vergleichsbasiertem verlustlosen Sortieren in linearer Zeit (siehe Abschnitt 1.3.3) konnte die rautavistische Informatik mit Hilfe des NN-Universalitätssatzes (Satz 7.2) einige weitere bahnbrechende Entdeckungen machen, von denen hier die wichtigsten kurz vorgestellt werden:

Satz 7.3 $\mathfrak{P} \neq \mathfrak{coNP}$.

Sowohl der Beweis als auch der Beweis des Satzes setzen etwas Wissen über Komplexitätstheorie voraus, hier insbesondere Zeit- und Platzkomplexitätsklassen. Daher geben wir einen Überblick in Tabelle 7.1. Genauere Details, insbesondere auch über die Beziehungen der Komplexitätsklassen untereinander, kann der interessierte Leser beispielsweise in [Sip05] oder [Wik] nachlesen.

Beweis:

Allgemein ist bekannt, dass $\mathfrak{P} \subseteq \mathfrak{coNP} \subseteq \mathfrak{ExpTime} \subseteq \mathfrak{ExpSpace} \overset{\text{(Savitch)}}{=} \mathfrak{NExpSpace}$. Darüberhinaus ist logischerweise $\mathfrak{P} \subsetneq \mathfrak{ExpTime}$.

Da sich *alle* Probleme mit Hilfe des NN-Universalalgorithmus' in (sub-)linearer Laufzeit auf einer nichtdeterministischen Maschine lösen lassen, gilt offensichtlich $\mathfrak{coNP} = \mathfrak{NExpSpace}$. Somit kollabiert die Komplexitätshierarchie zwischen $\mathfrak{coNP}$ und $\mathfrak{NExpSpace}$. Und da $\mathfrak{P} \subsetneq \mathfrak{ExpTime}$, muss auch $\mathfrak{P} \subsetneq \mathfrak{coNP}$ sein. ■

Abgesehen von dieser bahnbrechenden Erkenntnis lässt sich mit Hilfe der rautavistischen Informatik desweiteren zeigen:

Satz 7.4 $\mathfrak{P} = \mathfrak{NP}$.

Beweis:

Es ist bekannt, dass $\mathfrak{L} \subseteq \mathfrak{P} \subseteq \mathfrak{NP}$. Andererseits ist $\mathfrak{L} \subsetneq \mathfrak{PSpace}$.

Da sich *alle* Probleme mit Hilfe des NN-Universalalgorithmus' in (sub-)linearer Laufzeit auf einer nichtdeterministischen Maschine lösen lassen, gilt offensichtlich $\mathfrak{NL} = \mathfrak{NP}$. Nach dem Satz von Savitch ist allerdings $\mathfrak{L} = \mathfrak{NL}$. Somit kollabiert die Komplexitätshierarchie zwischen $\mathfrak{L}$ und $\mathfrak{NP}$ zu $\mathfrak{NL} = \mathfrak{L} = \mathfrak{P} = \mathfrak{NP}$. ■

Aus $\mathfrak{coNP} \subseteq \mathfrak{PSpace} \subseteq \mathfrak{ExpTime}$ folgt desweiteren:

Korollar 7.1 $\mathfrak{coNP} = \mathfrak{PSpace}$.

Hieraus ergibt sich wiederum ein verblüffender Sachverhalt: Nach Satz 7.4 ist $\mathfrak{NP} = \mathfrak{P}$; andererseits gilt $\mathfrak{NP} \subseteq \mathfrak{PSpace}$. Desweiteren gilt aber nach wie vor $\mathfrak{L} \subsetneq \mathfrak{PSpace}$ und damit nach unserem Korollar 7.1:

Korollar 7.2 $\mathfrak{NP} \subsetneq \mathfrak{coNP}$.

Korollar 7.3 $\mathfrak{NP} \subsetneq \mathfrak{PSpace}$.

Analog zum Beweis, dass vergleichsbasiertes Sortieren in linearer Laufzeit möglich ist (Abschnitt 1.3.3), hat die rautavistische Informatik damit auch gezeigt, dass sich sehr schwierige Probleme effizient lösen lassen (die Frage ist nur noch, wie). Dies hat weitreichende Folgen, u. a. auch für die Kryptographie: Da die meisten kryptographischen Verfahren auf Primfaktorzerlegung beruhen, von welcher man annimmt, dass sie sogar leichter als $\mathfrak{NP}$-vollständige Probleme zu lösen ist, müssen diese ab sofort als sehr leicht zu brechen angesehen werden. Ein möglicher Ausweg wäre hier die Verwendung des besonders sicheren Autokrypt-Verfahrens (Abschnitt 6.2). Somit liefert die rautavistische Informatik zugleich die Lösung dieses folgenschweren Problems.

Komplexitätsklasse	Maschine	Komplexität
$\mathfrak{L}$	deterministisch	logarithmischer Speicherverbrauch
$\mathfrak{NL}$	nichtdeterministisch	logarithmischer Speicherverbrauch
$\mathfrak{P}$	deterministisch	polynomielle Laufzeit
$\mathfrak{NP}$	nichtdeterministisch	polynomielle Laufzeit
$\mathfrak{coNP}$	nichtderministisch	Komplement in Polynomialzeit berechenbar
$\mathfrak{PSpace}$	deterministisch	polynomieller Speicherverbrauch
$\mathfrak{NPSpace}$	nichtdeterministisch	polynomieller Speicherverbrauch
$\mathfrak{ExpTime}$	deterministisch	exponentielle Laufzeit
$\mathfrak{NExpTime}$	nichtdeterministisch	exponentielle Laufzeit
$\mathfrak{NN}$	nichtdeterministisch	negative Laufzeit
$\mathfrak{LMSin}$	nichtdeterministisch	uniforme Kosten, trigonometrische Laufzeit

Tabelle 7.1: Übersicht über verschiedene Komplexitätsklassen.

Literaturverzeichnis

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[FKK92] Michael Fr. . . , Oleg Pawlowitsch Kasimir, and Michail Andrejewitsch Kostonjenko. *Das spontan erfundene Lemma von Kasimir und Kostonjenko*. Notbehelf, Mündliche Prüfung, Freiburg, ca. 1992.

[Par58] Cyril Northcote Parkinson. *Parkinson's Law*. John Murray, 1958.

[Ray96] Eric S. Raymond, editor. *New Hacker's Dictionary*. MIT Press, 3rd edition, 1996.

[Sed03] Robert Sedgewick. *Algorithms in Java*. Addison-Wesley, 3rd edition, 2003.

[Sip05] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2nd edition, 2005.

[Wik] Wikipedia. `http://de.wikipedia.org/`.