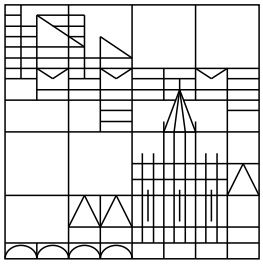

Universität Konstanz

Kurzskript zur Vorlesung

>> Informatik II <<

Karsten Weihe

Konstanzer Schriften in Mathematik und Informatik
71, September 1998

ISSN 1430–3558

Universität Konstanz
Fach D 188, 78457 Konstanz, Germany
Email: preprints@informatik.uni–konstanz.de
WWW: http://www.informatik.uni–konstanz.de/Schriften

Kurzskript zur Vorlesung
>> Informatik II <<

Karsten Weihe*

Lehrstuhl für praktische Informatik I
(Algorithmen und Datenstrukturen)

71/1998
Universität Konstanz
Fakultät für Mathematik und Informatik

Dieses Manuskript ist eine kurze Zusammenfassung der Inhalte der Vorlesung „Informatik II“ im Sommersemester 1998 an der Universität Konstanz (Umfang: drei Semesterwochenstunden). Es dient nicht zum Ersatz, sondern zur Nachbereitung der Vorlesung (bzw. zur Vorbereitung auf die einschlägige Diplomvorprüfung) und ist daher eher systematisch als didaktisch gestaltet.

Die Idee hinter diesem Skript und der Vorlesung war, daß die Vorlesung in erster Linie als „Vorbereitung zur Nachbereitung“ dient: Die Vorlesung versucht eher, Anschauung und Verständnis statt Fakten zu vermitteln, und auf dieser Basis (unterstützt durch häufige Bezugnahmen in der Vorlesung auf die jeweils relevanten Passagen im Skript) sollen die harten Fakten durch selbständiges Skriptstudium nachbereitet werden.

Der entscheidende Punkt in Abschnitt 1 (Dijkstras Algorithmus für kürzeste Wege in Graphen) ist nicht so sehr das darin vermittelte Detailwissen, sondern dieses Kapitel ist vor allem als eine umfangreiche Fallstudie zu verstehen, in der verschiedene Aspekte des Konzepts „Algorithmus an sich“ systematisch beleuchtet werden. Damit sind nicht nur die üblichen Aspekte wie Korrektheit und asymptotische Komplexität gemeint, sondern beispielsweise auch heuristische Beschleunigungstechniken (Abschnitte 1.8–1.10).

Ein meines Erachtens wesentlicher Aspekt von Algorithmen allgemein und von Dijkstras Algorithmus im besonderen ist die *Generizität*: So ziemlich alle nichttrivialen Algorithmen lassen viele Freiheitsgrade offen, die mehr oder weniger geschickt konkretisiert werden können. Diese Art von Generizität ist — immer am konkreten Fallbeispiel — insbesondere Gegenstand der Abschnitte 1.3–1.4.

Die Abschnitte 2 und 3 führen Algorithmen und Datenstrukturen als allgemeine, abstrakte Konzepte ein. Ebenso abstrakt und allgemein werden Themen wie Korrektheitsbeweise darin behandelt. Abschnitt 4 versucht hingegen, an zwei Fallbeispielen das systematische Zusammenwirken von Algorithmen und Datenstrukturen zu illustrieren.

*Universität Konstanz, Fakultät für Mathematik und Informatik, Fach D188, D-78457 Konstanz,
weihe@fmi.uni–konstanz.de,
http://www.fmi.uni–konstanz.de/~weihe

Danksagung: Annegret Liebers sowie etlichen Studierenden für vielfältige Anregungen und das Aufspüren von Fehlern in der ersten Fassung dieses Skripts.

Inhaltsverzeichnis

1 Kürzeste Wege in Graphen

1.1 Problemstellung

Definition 1.1 *Ein* **Graph** G *über einer beliebigen endlichen Menge* V *ist ein Paar* $G = (V, A)$*, wobei* A *eine Menge von geordneten Paaren von* V *ist:*

$$A \subseteq \{(v, w) : v,\, w \in V,\, v \neq w\}\ .$$

Die Elemente von V *heißen die* **Knoten** *von* G *(engl. vertices) und die Elemente von* A *die* **Kanten** *(engl. arcs).*[1]

Definition 1.2 *Eine* **Knotenbewertung** *eines Graphen* G *ist eine Abbildung* $V \to \mathbb{R}$*, und eine* **Kantenbewertung** *entsprechend eine Abbildung* $A \to \mathbb{R}$*. Für eine Kantenbewertung* b *und eine Kante* $a = (v, w)$ *schreiben wir auch* $b(v, w)$ *anstelle von* $b(a)$*.*

Definition 1.3 *Ein* **Pfad** p *mit Knotenzahl* $k \in \mathbb{Z}^+$ *in einem Graphen* $G = (V, A)$ *ist eine geordnete Folge* $p = (v_1, v_2, \ldots, v_k)$ *von Knoten von* G*, so daß* $(v_i, v_{i+1}) \in A$ *für* $i \in \{1, \ldots, k-1\}$ *gilt. Ein solcher Pfad* p *heißt*

- *ein* **einfacher Pfad***, wenn* $v_i \neq v_j$ *für* $i, j \in \{1, \ldots, k\}$ *mit* $i \neq j$ *ist;*
- *ein* **Zykel***, wenn* $k > 1$ *und* $v_1 = v_k$ *ist;*
- *ein* **einfacher Zykel***, wenn* p *ein Zykel ist und* $v_i \neq v_j$ *ist für alle* $i, j \in \{1, \ldots, k\}$ *mit* $i \neq j$ *und* $\{i, j\} \neq \{1, k\}$*.*

[1] Die Terminologie ist in der Literatur nicht ganz einheitlich: Oft wird das, was hier ein Graph genannt wird, etwas genauer ein *gerichteter* Graph genannt, und von einem Graphen spricht man dann nur, wenn die Orientierung jeder Kante (v, w) von v nach w ignoriert, d.h. jede Kante als eine ungeordnete Menge von zwei Knoten angesehen wird.

Wir sagen im weiteren, daß p von v_1 nach v_k über $v_2, \ldots, v_{k-1}$ läuft, oder kürzer, daß p ein (v_1, v_k)-Pfad ist. Die Kanten (v_i, v_{i+1}) für $i \in \{1, \ldots, k-1\}$ nennen wir die Kanten von p.

Definition 1.4 (Kantenlängen) *Für einen Graphen $G = (V, A)$ sei $\ell : A \to \mathbb{R}$ eine Kantenbewertung, so daß $\ell(a) > 0$ für $a \in A$ ist. Dann heißt $\ell(a)$ die* **Länge** *der Kante a bzgl. ℓ und ℓ eine* **Kantenlänge** *von G.*

Definition 1.5 (Pfadlänge) *Die Länge $\ell(p)$ eines Pfades $p = (v_1, v_2, \ldots, v_k)$ bzgl. einer Kantenlänge ℓ ist definiert als die Summe der Längen seiner Kanten:*

$$\ell(p) := \sum_{i=1}^{k-1} \ell(v_i, v_{i+1}) \; .$$

Der Pfad p heißt ein **kürzester** *Pfad von v_1 nach v_k bzgl. ℓ, wenn $\ell(p)$ minimal unter allen Pfaden von v_1 nach v_k ist.*

Definition 1.6 (Shortest-Path-Problem) *Gegeben seien ein Graph $G = (V, A)$, eine Kantenlänge $\ell : A \to \mathbb{R}$ sowie zwei Knoten $s, t \in V$. Gesucht ist ein kürzester (s, t)-Pfad in G bzgl. ℓ.*

Lemma 1.7 *Wenn es überhaupt einen (s, t)-Pfad gibt, so gibt es auch einen kürzesten (s, t)-Pfad, und dann sind alle kürzesten (s, t)-Pfade einfache Pfade.*

Beweis: Die Menge aller Längen von (s, t)-Pfaden ist offensichtlich abgeschlossen und nach unten beschränkt, nimmt also ihr Minimum an. Sei $p = (v_1, \ldots, v_k)$ ein Pfad, der nicht einfach ist. Dann gibt es einen Zykel $(v_i, v_{i+1}, \ldots, v_j)$ in p, und $p' := (v_1, \ldots, v_i, v_{j+1}, \ldots, v_k)$ ist kürzer als p, das heißt, p kann kein kürzester Pfad sein. □

Annahme 1.8 *Soweit nicht explizit anders festgelegt, nehmen wir im folgenden immer an, daß es für jedes Paar s, $t \in V$ einen (s, t)-Pfad in G gibt.*

Beobachtung 1.9 *Sei*

$$p = (v_1, v_2, \ldots, v_k)$$

ein kürzester (v_1, v_k)-Pfad in G bzgl. der Kantenlänge ℓ. Für $i \in \{1, \ldots, k-1\}$ ist dann $(v_1, \ldots, v_i)$ ein kürzester (v_1, v_i)-Pfad.

1.2 Distanzfunktionen

Definition 1.10 (Distanzfunktion) *Sei $G = (V, A)$ ein Graph, $s \in V$ und ℓ eine Kantenlänge von G. Die* **Distanzfunktion** *von G bzgl. s und ℓ ist die Knotenbewertung $\delta : V \to \mathbb{R}$, so daß $\delta(v)$ für $v \in V$ die Länge eines kürzesten (s, v)-Pfades in G bzgl. ℓ ist.*

Insbesondere gilt $\delta(s) = 0$.

Lemma 1.11 *Sei $G = (V, A)$ ein Graph und δ die Distanzfunktion bzgl. $s \in V$ und Kantenlänge ℓ.*

1. *Für $(v, w) \in A$ gilt $\delta(w) - \delta(v) \leq \ell(v, w)$.*
2. *Es gilt $\delta(w) - \delta(v) = \ell(v, w)$ genau dann, wenn (v, w) zu einem kürzesten (s, w)-Pfad gehört.*

Beweis: Da $\delta(w)$ die Länge eines kürzesten (s, w)-Pfades und $\delta(v) + \ell(v, w)$ die Länge eines gewissen (s, w)-Pfades ist, folgt sofort $\delta(w) - \delta(v) \leq \ell(v, w)$. Wenn (v, w) die letzte Kante eines kürzesten (s, w)-Pfades $p = (s = v_1, \ldots, v_{k-1} = v, v_k = w)$ ist, folgt Gleichheit aus Beobachtung 1.9. Für die Umkehrung sei $p = (s = v_1, \ldots, v_k = v)$ ein kürzester (s, v)-Pfad. Falls $\delta(w) - \delta(v) = \ell(v, w)$ ist, gilt

$$\delta(w) = \sum_{i=1}^{k-1} \ell(v_i, v_{i+1}) + \ell(v, w) \, ,$$

das heißt, die Länge des (s,w)-Pfades, der aus p durch Anhängen von (v,w) entsteht, ist gleich $\delta(w)$. □

Korollar 1.12 *Für jeden Knoten $v \in V \setminus \{s\}$ gibt es eine Kante $(u,v) \in A$ mit*

$$\delta(u) + \ell(u,v) = \delta(v) \ .$$

Satz 1.13 *Um einen kürzesten Pfad von $s \in V$ nach $t \in V$ bzgl. der Kantenlänge ℓ zu konstruieren, reicht es, zuerst die Distanzfunktion δ bzgl. s und ℓ zu berechnen und dann algorithmisch wie folgt vorzugehen:*

1. *Setze $w_1 := t$.*
2. *Für $i := 1, 2, 3, \ldots$ führe aus:*
   (a) *Falls $w_i = s$, gib*
   $$p = (w_i, w_{i-1}, \ldots, w_1)$$
   als kürzesten (s,t)-Pfad aus und beende die Prozedur.
   (b) *Ansonsten wähle einen beliebigen Knoten $w_{i+1} \in V$, so daß gilt: $(w_{i+1}, w_i) \in A$ und*
   $$\delta(w_{i+1}) + \ell(w_{i+1}, w_i) = \delta(w_i) \ .$$

Beweis: Korollar 1.12 garantiert, daß es für jeden Knoten $w_i \neq s$ einen Knoten w_{i+1} mit $(w_{i+1}, w_i) \in A$ und $\delta(w_{i+1}) + \ell(w_{i+1}, w_i) = \delta(w_i)$ gibt. Somit sind alle Schritte des Algorithmus wohldefiniert. Da die Länge jeder Kante strikt positiv ist, gilt $\delta(w_1) > \delta(w_2) > \delta(w_3) > \cdots$. Also kann die Schleife höchstens $(n-1)$-mal durchlaufen werden. Insbesondere terminiert der Algorithmus nach endlich vielen Schritten und liefert einen (s,t)-Pfad p.

Es bleibt zu zeigen, daß p ein kürzester Pfad ist. Sei k der Wert von i am Ende des Algorithmus, das heißt, $s = w_k$. Dann gilt

$$\begin{aligned} \ell(p) &= \sum_{i=1}^{k-1} \ell(w_{i+1}, w_i) \\ &= \sum_{i=1}^{k-1} \Big[\delta(w_i) - \delta(w_{i+1})\Big] \\ &= \delta(w_1) - \delta(w_k) = \delta(t) \ . \end{aligned}$$ □

1.3 Ein allgemeines Berechnungsschema für Distanzfunktionen

Lemma 1.14 *Sei $G = (V,A)$ ein Graph, ℓ eine Kantenlänge und $s \in V$. Die Distanzfunktion δ in G bzgl. s und ℓ ist die knotenweise maximale Knotenbewertung unter allen Knotenbewertungen π, so daß*

1. *$\pi(s) = 0$ (d.h. $\pi(s) = \delta(s)$) und*
2. *$\pi(w) - \pi(v) \leq \ell(v,w)$ für alle $(v,w) \in A$ gilt.*

Beweis: Sei π eine Knotenbewertung, die diese beiden Eigenschaften erfüllt. Wir nehmen zum Widerspruch an, daß es ein $v \in V$ mit $\pi(v) > \delta(v)$ gibt. Ohne Einschränkung sei v so gewählt, daß $\delta(v)$ minimal unter allen solchen Knoten ist. Die erste Eigenschaft erzwingt $v \neq s$. Nach Korollar 1.12 gibt es also eine Kante $(u,v) \in A$ mit $\delta(u) + \ell(u,v) = \delta(v)$. Da $\delta(u) < \delta(v)$ ist, folgt $\delta(u) \geq \pi(u)$ aus der speziellen Wahl von v, also der Widerspruch

$$\pi(v) - \pi(u) > \delta(v) - \delta(u) = \ell(u,v) \ .$$ □

Korollar 1.15 *Wenn π die beiden Eigenschaften in Lemma 1.14 erfüllt und außerdem $\pi(v) \geq \delta(v)$ für alle $v \in V$ ist, so gilt schon $\pi = \delta$.*

Satz 1.16 *Falls er terminiert(!), so liefert der folgende allgemeine Algorithmus für einen Graphen $G = (V, A)$, eine Kantenlänge ℓ und $s \in V$ eine Knotenbewertung π, die identisch mit der Distanzfunktion δ in G bzgl. s und ℓ ist:*

1. *Setze $\pi(s) := 0$.*
2. *Für $v \in V \setminus \{s\}$ setze $\pi(v)$ initial auf einen reellen Wert, der nicht kleiner als $\delta(v)$ ist.*
3. *Solange eine Kante $(v, w) \in A$ mit $\pi(v) + \ell(v, w) < \pi(w)$ gefunden werden kann, setze $\pi(w) := \pi(v) + \ell(v, w)$.*

Bemerkung: Eine einfache Wahlmöglichkeit in Schritt 2 ist

$$\pi(v) := \sum_{a \in A} \ell(a)$$

für alle $v \in V \setminus \{s\}$. □

Beweis von Satz 1.16: Man sieht leicht mit einer einfachen vollständigen Induktion über die Anzahl der Durchläufe der Schleife in Schritt 3, daß $\pi(v) \geq 0$ für alle $v \in V$ nach jedem Schleifendurchlauf gilt: Wenn nämlich nach Induktionsvoraussetzung $\pi(v) \geq 0$ gilt, dann gilt auch $\pi(v) + \ell(v, w) \geq 0$.

Da die π-Werte nie erhöht werden, folgt daraus sofort, daß die erste Eigenschaft aus Lemma 1.14 während des ganzen Algorithmus erfüllt ist.

Eine weitere einfache vollständige Induktion über die Anzahl der Durchläufe der Schleife in Schritt 3 beweist, daß $\pi(v) \geq \delta(v)$ für jeden Knoten $v \in V$ nach jedem Schleifendurchlauf ist: Ist nämlich $\pi(v) \geq \delta(v)$, dann ist offensichtlich auch $\pi(v) + \ell(v, w) \geq \delta(w)$.

Falls die Schleife nach endlich vielen Schritten terminiert, erfüllt das Ergebnis aufgrund der Bedingung für Termination auch die zweite Eigenschaft aus Lemma 1.14, und somit folgt $\pi = \delta$ aus Korollar 1.15. □

1.4 Verfeinerung: der Algorithmus von Dijkstra

1. Verfeinerung: Die erste Verfeinerung des Algorithmus in Satz 1.16 besteht in folgender verfeinerter Auswahlstrategie in der Schleife in Schritt 3:

> Wenn in einem Durchlauf der Schleife in Schritt 3 mehrere Kanten (v, w) mit $\pi(v) + \ell(v, w) < \pi(w)$ zur Auswahl stehen, dann wird eine Kante (v, w) mit minimalem Wert $\pi(v)$ ausgewählt.

Im folgenden bezeichne $a_k = (v_k, w_k)$ die Kante, die im k-ten Durchlauf dieser Schleife behandelt wird. Für $v \in V$ sei außerdem $\pi_k(v)$ der Wert von $\pi(v)$ nach dem k-ten Durchlauf (und $\pi_0(v)$ der Wert vor dem allerersten Durchlauf).

Lemma 1.17 *Für die erste Verfeinerung des Algorithmus gilt $\pi_{k-1}(v_{k-1}) \leq \pi_k(v_k)$ für alle $k > 1$.*

Beweis: Wir betrachten zwei Fälle:

1. Falls $w_{k-1} = v_k$ ist, gilt insbesondere $w_{k-1} \neq w_k$ und somit $\pi_{k-1}(w_{k-1}) = \pi_k(w_{k-1})$, also

$$\pi_k(v_k) = \pi_k(w_{k-1}) = \pi_{k-1}(w_{k-1})$$

$$= \pi_{k-1}(v_{k-1}) + \ell(a_{k-1}) > \pi_{k-1}(v_{k-1}) \; .$$

2. Falls hingegen $w_{k-1} \neq v_k$ ist, so gilt $\pi_{k-1}(v_k) = \pi_{k-2}(v_k)$ und insbesondere

$$\pi_{k-2}(w_k) \geq \pi_{k-1}(w_k) > \pi_{k-1}(v_k) + \ell(a_k)$$

$$= \pi_{k-2}(v_k) + \ell(a_k) \; ,$$

das heißt, a_k kam prinzipiell für die Auswahl von a_{k-1} im (i–1)-ten Durchlauf der Schleife in Schritt 3 in Frage. Die Auswahlregel für a_{k-1} in der 1. Verfeinerung impliziert daher $\pi_{k-2}(v_{k-1}) \leq \pi_{k-2}(v_k)$, und aus den offensichtlichen Tatsachen $\pi_{k-1}(v_{k-1}) \leq \pi_{k-2}(v_{k-1})$ und $\pi_k(v_k) = \pi_{k-1}(v_k)$ folgt nun zusammen mit obiger Beobachtung $\pi_{k-1}(v_k) = \pi_{k-2}(v_k)$ die Behauptung. □

Lemma 1.18 *In der ersten Verfeinerung des Algorithmus wird jede Kante in maximal einem Durchlauf der Schleife in Schritt 3 von Satz 1.16 behandelt. Insbesondere terminiert diese Schleife nach maximal $|A|$ Durchläufen.*

Beweis: Wir nehmen zum Widerspruch an, daß es $i < j$ mit $a_i = a_j$ gibt. Offensichtlich gilt

1. $\pi_{k-1}(v) \geq \pi_k(v)$ für alle $k \in \mathbb{Z}^+$ und $v \in V$ sowie
2. $\pi_{k-1}(w_k) > \pi_k(w_k)$ für $k \in \mathbb{Z}^+$.

Aus diesen beiden Tatsachen und der Widerspruchsannahme $a_i = a_j$ folgt $\pi_i(w_i) > \pi_j(w_j)$. Wegen $\ell(a_i) = \ell(a_j)$, $\pi_i(v_i) + \ell(a_i) = \pi_i(w_i)$ und $\pi_j(v_j) + \ell(a_j) = \pi_j(w_j)$ folgt weiter $\pi_i(v_i) > \pi_j(v_j)$. Das widerspricht aber Lemma 1.17. □

Korollar 1.19 *Die erste Verfeinerung terminiert nach endlich vielen Schritten.*

2. Verfeinerung: Dies ist die erste Verfeinerung mit folgenden Zusatzregeln:

1. In Schritt 2 wird $\pi(v)$ für alle $v \in V \setminus \{s\}$ auf ein und denselben Wert M gesetzt, der nicht kleiner als $\max\{\delta(v) : v \in V\}$ ist.
2. Wenn es für $k \in \mathbb{Z}^+$ einen oder mehrere Knoten $w \in V$ mit $(v_k, w) \in A$ und $\pi_k(v_k) + \ell(v_k, w) < \pi_k(w)$ gibt, so wird $v_{k+1} := v_k$ gesetzt. Insbesondere wird w_{k+1} nur unter diesen Knoten w ausgewählt.

Beachte, daß die zweite Bedingung der zweiten Verfeinerung nicht im Widerspruch zur definierenden Bedingung der ersten Verfeinerung steht, sondern sie im Gegenteil noch verfeinert.

Wir formulieren die zweite Verfeinerung um, indem wir einen „Knotenspeicher“ S einführen. Das ist ein Objekt, das eine Menge von Knoten repräsentiert. Während des Algorithmus können Knoten in S eingefügt und aus S gelöscht werden.

Definition 1.20 *Die folgende Umformulierung der zweiten Verfeinerung ist als der* **Algorithmus von Dijkstra** *in seiner allgemeinen Form bekannt:*

1. *Setze $\pi(s) := 0$ und $S := \{s\}$.*
2. *Für $v \in V \setminus \{s\}$ setze alle $\pi(v)$ initial auf einen gemeinsamen reellen Wert M, der nicht kleiner als $\max\{\delta(v) : v \in V\}$ ist.*
3. *Solange $S \neq \emptyset$, führe aus:*
   (a) *Wähle $v \in S$ mit minimalem Wert $\pi(v)$.*
   (b) *Entferne v aus S.*
   (c) *Für alle $(v, w) \in A$ mit $\pi(v) + \ell(v, w) < \pi(w)$ führe aus:*
      i. *Setze $\pi(w) := \pi(v) + \ell(v, w)$.*
      ii. *Falls $w \notin S$, füge w in S ein.*

Für $k \in \mathbb{Z}^+$ sei im folgenden S_k die Menge S unmittelbar nach dem k–ten Durchlauf der Schleife in Schritt 3 (bzw. S_0 die Menge S unmittelbar vor dem allerersten).

Satz 1.21 *Der Algorithmus von Dijkstra ist tatsächlich eine Umformulierung der zweiten Verfeinerung.*

Beweis: Wir haben zu zeigen, daß der Algorithmus von Dijkstra

1. in das allgemeine Schema von Satz 1.16 paßt (konkret zu zeigen: daß S nicht leer wird, bevor π nicht die zweite Eigenschaft in Lemma 1.14 erfüllt),
2. daß die Bedingung der ersten Verfeinerung dieses Schemas realisiert wird
3. und daß außerdem die beiden Bedingungen realisiert werden, mit denen die zweite Verfeinerung über die erste hinausgeht.

Die dritte Behauptung ergibt sich unmittelbar aus Schritt 2 und 3(c). Die ersten beiden Behauptungen ergeben sich aus der folgenden Hilfsbehauptung:

> *Hilfsbehauptung*: Wenn $\pi(v)+\ell(v,w) < \pi(w)$ für eine Kante $(v,w) \in A$ unmittelbar nach dem k–ten Durchlauf gilt, dann ist $v \in S_k$ (bzw. $v \in S_0$, wenn diese Ungleichung unmittelbar vor dem allerersten Durchlauf gilt).

Daraus folgt die erste der drei zu zeigenden Behauptungen sofort und die zweite Behauptung mit der Auswahlregel für $v \in S$ in Schritt 3(a).

Es ist also nur noch die Hilfsbehauptung zu zeigen. Für $k = 0$ und $v \neq s$ gilt $\pi(v) = M$, also $\pi(v) + \ell(v,w) > \pi(w)$ für alle $w \in V$ mit $(v,w) \in A$, womit die Induktion verankert ist. Sei nun $k > 0$ und $(v,w) \in A$ mit $\pi(v) + \ell(v,w) < \pi(w)$ unmittelbar nach dem k–ten Durchlauf. Wegen Schritt 3(c)i kann v nicht der Knoten sein, der im k–ten Durchlauf aus S_{k-1} in Schritt 3(b) entfernt wurde. Falls $\pi(v)+\ell(v,w) < \pi(w)$ schon unmittelbar vor dem k–ten Durchlauf galt, folgt die Hilsbehauptung also aus der Induktionsvoraussetzung. Ansonsten ist $\pi(v)$ im k–ten Durchlauf in Schritt 3(c)i reduziert worden, und die Hilfsbehauptung folgt aus Schritt 3(c)ii. □

1.5 Implementation von Graphen

Notation 1.22 *Im folgenden gehen wir von beliebigen, aber festen bijektiven Abbildungen $x : V \to \{1, \ldots, |V|\}$ und $y : A \to \{1, \ldots, |A|\}$ aus. Wir nehmen ohne Einschränkung an, daß $x(v)$ bei gegebenem $v \in V$ bzw. $y(a)$ bei gegebenem $a \in A$ in konstanter Zeit (abgekürzt $\mathcal{O}(1)$) berechnet werden kann.*

Notation 1.23 *Für einen Knoten $v \in V$ bezeichnet $A(v)$ im folgenden die Menge der Kanten, die aus v hinauszeigen:*

$$A(v) := \{(v,w) : w \in V, (v,w) \in A\} .$$

Typische Implementationen von Graphen mit Kantenlängen:

1. Mit einer *Adjazenzmatrix*: ein Vektor mit Indexbereich $\{1, \ldots, |V|\}$, dessen Komponenten wiederum Vektoren mit Indexbereich $\{1, \ldots, |V|\}$ und reellwertigen Komponenten sind. Für $(v,w) \in A$ ist die Matrixkomponente mit Indexpaar $(x(v), x(w))$ gleich der Länge der Kante (v,w), und für $(v,w) \notin A$ enthält diese Komponente zum Anzeigen der Nichtexistenz einen „unmöglichen“ Wert M. Beispielsweise könnte man für M einen negativen Wert oder einen „unmöglich großen“ Wert hernehmen.[2]

[2] Vergleiche auch Fußnote 24 auf Seite 27.

2. Mit *Adjazenzlisten*: ein Vektor mit Indexbereich $\{1, \ldots, |V|\}$, dessen Komponenten *Listen* sind. Für $v \in V$ enthält die Liste am Index $x(v)$ ein Element zu jeder Kante $(v, w) \in A(v)$. Der Inhalt dieses Elements besteht aus zwei Teilen:
   - ein Verweis auf w und
   - die Länge der Kante (v, w).

3. *Stern-Repräsentation*: ein Vektor B_1 mit Indexbereich $\{1, \ldots, |V|\}$ und ein weiterer Vektor B_2 mit Indexbereich $\{1, \ldots, |A|\}$. Die Komponenten von B_1 sind vom Typ $\mathbb{Z}$. Für eine Kante $(v, w) \in A$ enthält die Komponente $y(v, w)$ von B_2 zwei Komponenten: die Länge von (v, w) und den Wert $x(w)$.

   Die Kanten sind in B_2 aufsteigend nach den x-Werten der Ausgangsknoten geordnet. Genauer gesagt: Für $(v_1, w_1), (v_2, w_2) \in A$ mit $x(v_1) < x(w_1)$ gilt $y(v_1, w_1) < y(v_2, w_2)$.

   Für $v \in V$ ist $B_1[x(v)]$ der größte Index in B_2, der eine Kante mit Ausgangsknoten v enthält. Durch den oben beschriebenen Zusammenhang zwischen x und y ist also $\{B_1[x(v) - 1] + 1, \ldots, B_1[x(v)]\}$ genau die Menge der Indizes der Kanten, die zu $A(v)$ gehören, bzw. $\{1, \ldots, B_1[1]\}$ im Falle $x(v) = 1$.

   Damit diese Regel auch im Falle $A(v) = \emptyset$ noch korrekt ist, wird in einem solchen Fall gesetzt: $B_1[x(v)] := B_1[x(v) - 1]$.

   Fügt man einen „Dummy-Index“ 0 in B_1 ein und setzt $B_1[0] := 0$, dann ist die obige Ausnahmebehandlung für den Fall $x(v) = 1$ nicht nötig.

Beobachtungen:

1. Der Algorithmus von Dijkstra läßt sich auf allen diesen Implementationen prinzipiell realisieren.

2. Der Speicherplatzaufwand ist
   - $\Theta(|V|^2)$ für Adjazenzmatrizen und
   - $\Theta(|V| + |A|)$ für die anderen beiden Implementationen.

3. Für $v \in V$ ist der Laufzeitaufwand zum Durchlauf von $A(v)$ gleich
   - $\Theta(V)$ für Adjazenzmatrizen und
   - $\Theta(A(v))$ für die anderen beiden Implementationen.

1.6 Der Knotenspeicher

Annahme 1.24 *Im folgenden gehen wir von einer Implementation des Graphen mittels Adjazenzlisten oder Stern-Repräsentation (oder einer in bezug auf asymptotische Laufzeit und asymptotischen Speicherverbrauch äquivalenten Implementation) aus.*

Der Algorithmus von Dijkstra stellt folgende Anforderungen an den Knotenspeicher:

1. Feststellen ob ein bestimmter Knoten momentan enthalten ist (Schritt 3(c)ii).
2. Einfügen eines Knotens, der momentan nicht enthalten ist (Schritt 3(c)ii).
3. Auffinden des Knotens mit minimalem π-Wert in einem nichtleeren Knotenspeicher (Schritt 3(a)).
4. Entfernen dieses Knotens (Schritt 3(b)).

Darüber hinaus wird im weiteren folgende Operation für die Analyse wichtig sein:

5. Eventuelle interne Umorganisation des Knotenspeichers, wenn der π-Wert eines momentan gespeicherten Knotens reduziert wird (Schritt 3(c)i).

Notation 1.25 *Für einen Graphen $G = (V, A)$ seien $T_1(G), \ldots, T_5(G)$ die maximal möglichen Laufzeiten dieser fünf Operationen auf G bzgl. irgendwelcher Kantenlängen und Startknoten.*

Sofern keine Mißverständnisse auftreten können, schreiben wir im weiteren kurz T_i anstelle von $T_i(G)$. Falls die fünfte Operation in einer Implementation des Knotenspeichers nicht auftritt, schreiben wir $T_5 \in \mathcal{O}(1)$ zur Vereinheitlichung der Notation.

Satz 1.26 *Die Laufzeit des Algorithmus von Dijkstra ist abschätzbar durch*

$$\mathcal{O}\Big(|A| \cdot (T_1 + T_2 + T_5) + |V| \cdot (T_3 + T_4)\Big) .$$

Beweis: Als erstes zeigen wir, daß die Schleife in Schritt 3 für jeden Knoten maximal einmal durchlaufen wird. Wir nehmen also zum Widerspruch an, daß $v \in V$ der betrachtete Knoten in den Schleifendurchläufen Nr. i und j ist $(i < j)$. Dann gilt $v \in S_{i-1}$, $v \notin S_i$ und $v \in S_{j-1}$. Bei der Wiedereinfügung von v in S zwischen dem i-ten und j-ten Schritt wird $\pi(v)$ echt vermindert (nämlich in Schritt 3(c)i), was zum Widerspruch mit Lemma 1.17 führt.

Aus der nunmehr bewiesenen Tatsache, daß die Schleife in Schritt 3 für jeden Knoten maximal einmal durchlaufen wird, ergibt sich sofort die Abschätzung für die dritte und vierte Operation. Außerdem folgt aus dieser Tatsache, daß jede Kante (v, w) maximal einmal in Schritt 3(c) betrachtet wird, nämlich in dem Schleifendurchlauf, in dem v abgearbeitet wird. Daraus folgt die Abschätzung für die erste, zweite und fünfte Operation. □

Korollar 1.27 *Man kann für den Algorithmus von Dijkstra Gesamtlaufzeit $\mathcal{O}(|V|^2)$ garantieren, indem man den Knotenspeicher als einen Vektor W von Wahrheitswerten* „wahr"/„falsch" *mit Indexbereich $\{1, \ldots, |V|\}$ und folgender Bedeutung organisiert: Für $v \in V$ ist der Wert in der Komponente $x(v)$ genau dann* „wahr", *wenn v momentan zum Knotenspeicher gehört (vgl. Notation 1.22).*

Beweis: Dann gilt $T_3 \in \mathcal{O}(|V|)$ und T_1, T_2, $T_4 \in \mathcal{O}(1)$. Die fünfte Operation kommt nicht vor, also auch $T_5 \in \mathcal{O}(1)$. Die Behauptung folgt somit aus der allgemeinen Tatsache $|A| \leq |V|^2$. □

1.7 Realisierung des Knotenspeichers durch einen Heap

Definition 1.28 *Seien $n \in \mathbb{Z}^+$ und $m \in \mathbb{Z}_0^+$ mit $m \leq n$, S eine beliebige Menge und $f : S \to \mathbb{R}$. Ein* **(n,m,f)–Heap** *ist gegeben durch einen Vektor H mit Indexbereich $[1 \ldots n]$ und Komponententyp S, so daß die Werte $H[1], \ldots, H[m]$ paarweise verschieden sind und die* **Heap-Eigenschaft** *gilt, das heißt, für $i \in \{1, \ldots, m\}$ muß $f(H[i]) \geq f(H[\lfloor i/2 \rfloor])$ sein. Die Komponenten $1, \ldots, m$ heißen die* **signifikanten Komponenten** *des Heaps.*

Bemerkung: Vergleiche die Behandlung von *Heapsort* in der Informatik I und insbesondere die Interpretation von Heaps als binäre Bäume. Die *Vorgänger* von $i \in \mathbb{Z}^+$ im Baum sind alle positiven ganzen Zahlen, die sich aus i durch wiederholte (mindestens einmalige) Anwendung der Operation $i := \lfloor i/2 \rfloor$ konstruieren lassen. Die Anzahl der Vorgänger entspricht der Höhe $h(i)$ von i im abstrakten Baum (wobei die Wurzel Höhe 0 hat). □

Lemma 1.29 *Die Höhe eines Elements $i \in \{1, \ldots, m\}$ ist $h(i) = \lfloor \log_2 i \rfloor$.*

Beweis: durch vollständige Induktion über $h(i)$. Für $h(i) = 0$ ist $i = 1$ und somit $\log_2 i = 0$. Für $h(i) > 0$ ist $i > 1$ und daher $\lfloor i/2 \rfloor \geq 1$. Da $h(\lfloor i/2 \rfloor) = h(i) - 1$ nach Definition von h ist, folgt der Induktionsschritt sofort aus der Induktionsvoraussetzung angewandt auf $\lfloor i/2 \rfloor$, denn: $h(\lfloor i/2 \rfloor) = \lfloor \log_2(\lfloor i/2 \rfloor) \rfloor = \lfloor \log_2(i/2) \rfloor = \lfloor \log_2 i - 1 \rfloor$.[3] □

Definition 1.30 *Seien n, m, S und f wie in Definition 1.28. Ein Vektor H mit Indexbereich $[1 \ldots n]$ und Komponententyp S ist ein* **(n,m,f)–Pseudoheap**, *wenn H zwar kein (n, m, f)–Heap ist, aber folgendes gilt: Es gibt ein $i \in \{1, \ldots, m\}$ und $c \in \mathbb{R}$, so daß H ein (n, m, f')–Heap in bezug auf die Funktion $f' : S \to \mathbb{R}$ ist, die definiert ist durch*

- $f'(H[i]) := f(H[i]) + c$ *und*
- $f'(r) := f(r)$ *für* $r \in S \setminus \{H[i]\}$.

Der Index i heißt eine **Fehlerstelle** *von H und c eine* **Korrektur** *der Fehlerstelle i.*

Unter einer *Vertauschung* zweier Variablen i und j verstehen wir eine Operation, die in $\mathcal{O}(1)$ Zeit die Inhalte von i und j austauscht.

Lemma 1.31 *Wenn für einen (n, m, f)–Pseudoheap H eine Fehlerstelle i und eine Korrektur c dieser Fehlerstelle bekannt sind, dann kann H durch maximal $h(m) = \lfloor \log_2 m \rfloor$ Vertauschungen von jeweils zwei signifikanten Komponenten in einen (n, m, f)–Heap transformiert werden.*

[3] Die zweite Gleichung ergibt sich aus der generellen Einsicht, daß für $x \in \mathbb{R}^+$ aus $\log_2(x) \in \mathbb{Z}^+$ schon $x \in \mathbb{Z}^+$ folgt, insbesondere gilt also $\lfloor \log_2 x \rfloor = \lfloor \log_2 \lfloor x \rfloor \rfloor$ für jedes $x \in \mathbb{R}^+$.

Beweis: Da H nach Definition kein (n, m, f)–Heap ist, gilt $c \neq 0$. Wir behandeln den Fall $c > 0$ und den Fall $c < 0$ jeweils separat.

- *Fall $c > 0$*: Eine positive Korrektur für i bedeutet, daß $f(H[\lfloor i/2 \rfloor]) > f(H[i])$ ist. Insbesondere muß $H[\lfloor i/2 \rfloor]$ existieren, woraus $h(i) \geq 1$ folgt. Wir zeigen mit vollständiger Induktion über $h(i) \geq 1$, daß H in maximal $h(i)$ Vertauschungen zu einem (n, m, f)–Heap umgeordnet werden kann.

  Die Induktion wird bei $h(i) = 1$ verankert: Aus der Definition von Pseudoheaps folgt leicht, daß die Vertauschung von $H[i]$ und $H[1]$ in diesem Fall aus H einen (n, m, f)–Heap macht.

  Für den Beweis des Induktionsschrittes sei nun $h(i) > 1$. Es folgt wieder leicht aus der Definition von Pseudoheaps, daß die Vertauschung von $H[i]$ und $H[\lfloor i/2 \rfloor]$ aus H entweder einen (n, m, f)–Heap oder einen (n, m, f)–Pseudoheap mit Fehlerstelle $\lfloor i/2 \rfloor$ und positiver Korrektur macht. Auf $\lfloor i/2 \rfloor$ ist die Induktionsvoraussetzung anwendbar, und H läßt sich daher durch insgesamt maximal $h(\lfloor i/2 \rfloor) + 1 = h(i)$ Vertauschungen zu einem (n, m, f)–Heap machen.

- *Fall $c < 0$*: Eine negative Korrektur für i bedeutet, daß $f(H[i]) > f(H[2i])$ oder $f(H[i]) > f(H[2i+1])$ ist.[4] Insbesondere ist $h(m) \geq h(2i) > h(i)$. Wir zeigen mit einer analogen vollständigen Induktion über $h(m) - h(i) \geq 1$, daß H in maximal $h(m) - h(i)$ Schritten zu einem (n, m, f)–Heap umgeordnet werden kann. Die Induktion wird bei $h(m) - h(i) = 1$ wie

[4] Im Spezialfall, daß m eine gerade Zahl und ausgerechnet $i = m/2$ ist, ist $H[2i+1]$ keine signifikante Komponente mehr bzw. existiert im Fall $m = n$ nicht einmal. Dann reduziert sich diese Aussage natürlich zu $f(H[i]) > f(H[2i])$.

folgt verankert: Im Falle, daß m gerade und $i = m/2$ ist, vertauschen wir $H[i]$ mit $H[2i]$. Ansonsten sei $j \in \{2i, 2i+1\}$ so gewählt, daß

$$H[j] = \min\{H[2i], H[2i+1]\}$$

ist. (Im Falle $H[2i] = H[2i+1]$ kann also $j \in \{2i, 2i+1\}$ beliebig gewählt werden.) Dann vertauschen wir $H[i]$ und $H[j]$. Aus der Definition von Pseudoheaps folgt leicht, daß H in beiden Fällen hinterher ein (n, m, f)–Heap ist.

Für den Beweis des Induktionsschrittes sei nun $h(m) - h(i) > 1$. Dann existiert auch $H[2i+1]$. Wir definieren $j \in \{2i, 2i+1\}$ wieder durch

$$H[j] = \min\{H[2i], H[2i+1]\}$$

und vertauschen $H[i]$ mit $H[j]$. Aus der Definition von Pseudoheaps folgt leicht, daß H danach entweder ein (n, m, f)–Heap oder ein (n, m, f)–Pseudoheap mit Fehlerstelle $H[j]$ und negativer Korrektur ist. Auf j ist die Induktionsvoraussetzung anwendbar und liefert analog das gewünschte Resultat durch insgesamt maximal $h(m) - h(j) + 1 = h(m) - h(i)$ Vertauschungen. □

Korollar 1.32 *Der Gesamtaufwand für den Algorithmus von Lemma 1.31 ist in* $\mathcal{O}(\log m)$.

Für $v \in V$ sei $\tilde{\pi}_k(v)$ der Wert $\pi(v)$ am Ende des k-ten Durchlaufs durch die Schleife in Schritt 3 des Algorithmus von Dijkstra[5] (entsprechend wieder $\tilde{\pi}_0(v)$ für den Wert unmittelbar vor dem ersten Durchlauf). Die Mengen S_0, S_1, S_2, ... werden mittels zweier Vektoren H und W mit Indexbereich $\{1, \ldots, |V|\}$ repräsentiert:

- Nach dem k-ten Durchlauf der Schleife in Schritt 3 des Algorithmus von Dijkstra ist H ein $(|V|, S_k, \tilde{\pi}_k)$–Heap (entsprechend ein $(|V|, S_0, \tilde{\pi}_0)$–Heap unmittelbar vor dem ersten Durchlauf).
- W ist wie in Korollar 1.27 definiert.

Satz 1.33 *Für diese Realisierung des Knotenspeichers gilt*

- $T_1 \in \mathcal{O}(1)$,
- $T_2 \in \mathcal{O}(\log |V|)$,
- $T_3 \in \mathcal{O}(1)$,
- $T_4 \in \mathcal{O}(\log |V|)$,
- $T_5 \in \mathcal{O}(\log |V|)$.

Korollar 1.34 *Der Algorithmus von Dijkstra kann so implementiert werden, daß seine Laufzeit* $\mathcal{O}(|A| \log |V|)$ *ist.*

Beweis von Satz 1.33: Wir betrachten die einzelnen Operationen nacheinander.

- *1. Operation*: simples Nachschauen in W.
- *2. Operation*: Um einen neuen Knoten v in einen $(|V|, m, f)$–Heap mit $m < |V|$ einzufügen, wird zunächst $H[m+1] := x(v)$ und $W[x(v)] := \mathit{wahr}$ gesetzt. Falls das Ergebnis kein $(n, m+1, f)$–Heap ist, so ist es ein $(n, m+1, f)$–Pseudoheap mit Fehlerstelle $m+1$ und Korrektur

  $$f(H[\lfloor (m+1)/2 \rfloor]) - f(H[m+1]) \ .$$

  Somit ist Lemma 1.31 anwendbar.
- *3. Operation*: $H[1]$ ist das gesuchte Element.
- *4. Operation*: Zuerst wird $H[1] := H[m]$ und $W[H[m]] := \mathit{falsch}$ gesetzt. Falls das Ergebnis kein $(n, m-1, f)$–Heap ist, so

[5] Beachte den Unterschied zum Wert $\pi_k(v)$, der auf Seite 5 im Hinblick auf den allgemeinen Algorithmus in Satz 1.16 definiert wurde. Das heißt: Es gilt im allgemeinen $\tilde{\pi}_k \neq \pi_k$, aber $\tilde{\pi}_k = \pi_i$ für ein gewisses $i \geq k$.

ist es ein $(n, m-1, f)$–Pseudoheap mit Fehlerstelle 1 und Korrektur

$$\min\{f(H[2]), f(H[3])\} - f(H[1]) \ .$$

Somit ist Lemma 1.31 wieder anwendbar.[6]

- *5. Operation*: Diese Operation kann auch so formuliert werden: Die Abbildung f wird durch eine neue Abbildung $f' : V \to \mathbb{R}$ ersetzt, so daß es ein $i \in \{1, \ldots, m\}$ mit $f'(r) = f(r)$ für $r \in V \setminus \{H[i]\}$ gibt. Offensichtlich ist ein (n, m, f)–Heap unter dieser Einschränkung auch ein (n, m, f')–Heap oder ein (n, m, f')–Pseudoheap mit Fehlerstelle i und Korrektur $f(H[i]) - f'(H[i])$, so daß wieder Lemma 1.31 anwendbar ist. □

Bemerkung: Mit einer raffinierten Variation von Heaps als Knotenspeicher, den sogenannten *Fibonacci–Heaps*, läßt sich sogar $\mathcal{O}(|A| + |V| \log |V|)$ Gesamtaufwand für den Algorithmus von Dijkstra erzielen. □

1.8 Heuristische Beschleunigungstechniken

Unter einer Beschleunigungstechnik verstehen wir die Verfeinerung eines allgemeinen algorithmischen Ansatzes zur Verringerung der Laufzeit. Zum Beispiel war die Entwicklung eines effizienten Knotenspeichers in Abschnitt 1.7 ein Beispiel für eine Beschleunigungstechnik für den Algorithmus von Dijkstra. Diese Technik war nicht heuristisch.

„Heuristisch“ bedeutet nämlich, daß die Beschleunigung, die aus der Anwendung einer solchen Technik resultiert, nicht „beweisbar“ ist, ja nicht einmal für alle möglichen Eingaben stimmen muß. Möglicherweise können sogar Beispiele existieren, bei denen die Anwendung einer „Beschleunigungs“technik zu einer *Verlangsamung* führt. Das macht nichts, solange diese Beispiele nicht (oder nicht zu oft) in der Praxis auftreten.

1. Beschleunigungstechnik: Beende den Algorithmus, sobald t zum ersten Mal aus dem Knotenspeicher entfernt wird. □

Bemerkungen:

- Diese spezifische Beschleunigungstechnik mag potentiell durch die zusätzliche Abfrage $v = t$ in jedem Durchlauf durch Schritt 3 zu einer Verlangsamung führen können. Aber das dürfte vernachlässigbar sein, und im allgemeinen wird diese Technik vermutlich zu einer spürbaren Beschleunigung führen.
- Bei Einsatz dieser Beschleunigungstechnik ist das Endergebnis π des Algorithmus von Dijkstra im allgemeinen natürlich nicht mehr identisch mit δ.

Lemma 1.35 *Wenn diese Beschleunigungstechnik im Algorithmus von Dijkstra eingesetzt wird, liefert der Algorithmus aus Satz 1.13 angewandt auf das Resultat des Algorithmus immer noch einen kürzesten (s, t)–Pfad.*

Beweis: Im Beweis von Satz 1.26 haben wir gesehen, daß jeder Knoten maximal einmal in den Knotenspeicher aufgenommen wird. Wenn t aus dem Knotenspeicher entfernt wird, kann es also auch beim Algorithmus von Dijkstra in seiner ursprünglichen Form aus Definition 1.20 nicht vorkommen, daß $\pi(t)$ danach noch einmal in Schritt 3(c)i des Algorithmus von Dijkstra verringert wird. Das gleiche

[6] Falls $m = 2$ ist, lautet die Korrektur natürlich einfach $f(H[2]) - f(H[1])$.

gilt wegen Lemma 1.17 auch für alle Knoten $v \in V$ mit $\pi(v) < \pi(t)$. Das heißt also, es gilt zu diesem Zeitpunkt sowohl $\pi(t) = \delta(t)$ als auch $\pi(v) = \delta(v)$ für alle $v \in V$ mit $\delta(v) < \delta(t)$. Dazu gehören aber offensichtlich alle Knoten auf allen kürzesten (s,t)–Pfaden.

Andererseits gilt $\pi \geq \delta$ knotenweise (das wurde im Beweis von Satz 1.16 angesprochen), das heißt, der Algorithmus aus Satz 1.13 kann nicht durch Knoten v, für die fälschlich $\pi(v) < \delta(v)$ gilt, in die Irre geführt werden. (Er kann natürlich generell nicht durch Knoten mit $\pi(v) > \delta(v)$ in die Irre geführt werden, weil ja dann nach obiger Beobachtung $\pi(v) \geq \pi(t)$ gilt, v also nie als nächster Knoten beim Rückwärtsgehen nach s im Algorithmus von Satz 1.13 in Betracht kommt.)

Durch Anwendung des Algorithmus aus Satz 1.13 erhalten wir also tatsächlich einen kürzesten (s,t)–Pfad, auch wenn der Algorithmus von Dijkstra vorher nicht solange gelaufen ist, bis tatsächlich $\pi \equiv \delta$ ist. □

1.9 Berechnungen von mehreren kürzesten Pfaden in Folge

Die folgende heuristische Beschleunigungstechnik ist in einem speziellen Szenario anwendbar: Gegeben sind zunächst nur ein Graph $G = (V, A)$ und eine Kantenlänge $\ell : A \to \mathbb{R}$. Das Programm wartet auf Eingaben. Eine Eingabe besteht aus zwei Knoten, s, $t \in V$, und das Programm soll daraufhin einen kürzesten (s,t)–Pfad berechnen und ausgeben. Dieser Frage-/Antwortzyklus kann sich beliebig oft wiederholen.

Bemerkung: Das ist das typische Szenario etwa eines Verkehrsauskunftssystems. □

Wird die erste heuristische Beschleunigungstechnik aus Abschnitt 1.8 angewandt, so berührt Schritt 3 des Algorithmus von Dijkstra nicht alle Knoten und Kanten, sondern nur einen Ausschnitt des Graphen. Grob gesprochen kann man erwarten, daß dieser Ausschnitt um so kleiner ist, je „näher“ s und t beieinanderliegen. Man kann also mit einigem Recht erwarten, daß die Gesamtzahl der Durchläufe in Schritt 3(c) des Algorithmus von Dijkstra[7] „im Durchschnitt“ deutlich kleiner als $|A|$ ist, sogar *asymptotisch* kleiner.

Formal kann man das so fassen: Sei $N(G, \ell, s, t)$ die Gesamtzahl der Durchläufe durch die Schleife in Schritt 3(c) unter Einsatz der ersten Beschleunigungstechnik aus Abschnitt 1.8. Für eine endliche Folge $F = (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ von Paaren von Knoten in V ist

$$D(G, \ell, F) := \frac{1}{k} \sum_{i=1}^{k} N(G, \ell, s_i, t_i)$$

die durchschnittliche Zahl der Durchläufe. Dann kann man in vielen konkreten Anwendungen erwarten, daß $D(G, \ell, F) \in o(|A|)$ ist, und zwar asymptotisch *deutlich* kleiner als $|A|$.

Wenn $D(G, \ell, F) \in o(|V|/\log|V|)$ ist, dann ist nicht mehr Schritt 3, sondern Schritt 2 des Algorithmus von Dijkstra der „Flaschenhals“ bei asymptotischen Laufzeitbetrachtungen, denn dieser Schritt hat Laufzeit $\Theta(|V|)$. In vielen Anwendungen kann man $|A| \in \mathcal{O}(|V|)$ voraussetzen,[8] das heißt, damit Schritt 2 die asymptotische Laufzeit dominiert, muß in einem solchen Fall nur der relative Anteil der

[7] D.h. zusammengezählt über alle Durchläufe der äußeren Schleife in Schritt 3.

[8] Man kann zum Beispiel mathematisch für ein Straßennetz mit maximal k übereinander verlaufenden Ebenen (d.h. an keiner Stelle passieren sich mehr als k Straßen mittels Brücken und Tunnel übereinander ohne Kreuzungen) beweisen, daß $|A| < 3k \cdot |V|$ sein muß. Siehe dazu Hauptstudiumsveranstaltungen zum Thema Algorithmen und Datenstrukturen, speziell Graphenalgorithmen.

Kanten, die insgesamt in Schritt 3(c) betrachtet werden, asymptotisch kleiner als $1 - 1/\log|A|$ sein.

Die folgende einfache Technik reduziert die durchschnittliche Laufzeit von Schritt 2 über der Folge F auf $\mathcal{O}(D(G,\ell,F))$:

- Es wird eine weitere Knotenbewertung $t : V \to \mathbb{Z}_0^+$ definiert. Für $v \in V$ heißt der Wert $t(v)$ der *Zeitstempel* von v. Diese Werte werden vor der allerersten Berechnung eines kürzesten Pfades auf 0 gesetzt.
- Zusätzlich wird ein *globaler Zeitzähler* g eingerichtet. Diese Variable wird vor der allerersten Berechnung eines kürzesten Pfades auf 1 gesetzt und nach jeder Berechnung eines kürzesten Pfades um 1 erhöht.
- Schritt 2 des Algorithmus von Dijkstra wird fallengelassen.
- Wann immer ein Knoten v in Schritt 3 berührt wird, wird
  - zuerst sein Zeitstempel gelesen. Falls $t(v) \neq g$, wird der tatsächlich in $\pi(v)$ stehende Wert ignoriert.

    Statt dessen behandelt der Algorithmus v so, als ob $\pi(v) = M$ wäre.[9]
  - *Danach* wird $t(v) := g$ gesetzt.

Offensichtlich gilt $t(v) \neq g$ beim Berühren von v genau dann, wenn v in der momentanen Berechnung eines kürzesten Pfades noch nicht berührt wurde. Das ist aber genau der Fall, bei dem in der ursprünglichen Variante des Algorithmus von Dijkstra aus Definition 1.20 ein Knoten v mit $\pi(v) = M$ berührt würde.

Bemerkung: Spätestens, wenn g gleich der größten darstellbaren Zahl im gewählten Datentyp für ganze Zahlen wird, muß die Initialisierung von t und g wiederholt werden, um einen Überlauf zu vermeiden. □

[9] M muß in diesem Szenario natürlich einer strengeren Bedingung als in Abschnitt 1.4 genügen: M darf nun nicht kleiner als die Länge eines kürzesten (v,w)–Pfades zwischen *irgend* zwei Knoten $v, w \in V$ sein. Die Definition aus der Bemerkung unmittelbar nach Satz 1.16 wäre wieder geeignet.

1.10 Zielgerichtete Suchstrategien

Im folgenden sei $G = (V, A)$ ein Graph und $\ell : A \to \mathbb{R}$ eine Kantenlänge. Sei außerdem $h : V \to \mathbb{R}$ eine beliebige Knotenbewertung, und $\ell_h : A \to \mathbb{R}$ sei für $(v,w) \in A$ definiert durch $\ell_h(v,w) := \ell(v,w) + h(w) - h(v)$. Die Länge eines Pfades p bzgl. ℓ_h wird mit $\ell_h(p)$ abgekürzt.

Lemma 1.36 *Für $s, t \in V$ und für jeden (s,t)–Pfad p gilt $\ell_h(p) = \ell(p) + h(t) - h(s)$.*

Beweis: Sei $p = (s{=}v_1, v_2, \ldots, v_k{=}t)$. Dann gilt

$$\ell_h(p) = \sum_{i=1}^{k-1} \ell_h(v_i, v_{i+1})$$

$$= \sum_{i=1}^{k-1} \ell(v_i, v_{i+1}) + \sum_{i=1}^{k-1} \Big(h(v_{i+1}) - h(v_i)\Big)$$

$$= \sum_{i=1}^{k-1} \ell(v_i, v_{i+1}) + h(v_k) - h(v_1)\,. \qquad \square$$

Korollar 1.37 *Insbesondere ist jeder kürzeste (s,t)–Pfad bzgl. ℓ ein kürzester (s,t)–Pfad bzgl. ℓ_h und umgekehrt.*

Definition 1.38 *Eine Knotenbewertung $h : V \to \mathbb{R}$ heißt* **zulässig** *bzgl. ℓ, wenn $\ell(v,w) > h(v) - h(w)$ für $(v,w) \in A$ gilt.*

Mit anderen Worten: ℓ_h ist genau dann eine Kantenlänge gemäß Definition 1.4, wenn h eine zulässige Knotenbewertung ist.

Definition 1.39 *Eine nichtnegative Knotenbewertung $h : V \to \mathbb{R}_0^+$ ist eine* **Zielunterschätzung** *bzgl. ℓ und t, wenn für jeden Knoten $v \in V$ der Wert $h(v)$ nicht größer als die Länge eines kürzesten (v,t)–Pfades bzgl. ℓ ist.*

Insbesondere ist $h(t) = 0$.

Definition 1.40 *Wir sprechen von einer* **zielgerichteten Suche** *eines kürzesten Pfades, wenn auf den Algorithmus von Dijkstra (Definition 1.20)*

1. *die erste Beschleunigungstechnik aus Abschnitt 1.8 angewandt wird und zugleich*
2. *die Kantenlänge ℓ_h zu einer zulässigen Zielunterschätzung h anstelle der gegebenen Kantenlänge ℓ verwendet wird.*

Wegen Lemma 1.35 und Korollar 1.37 liefert der Algorithmus aus Satz 1.13 einen kürzesten (s,t)–Pfad bzgl. ℓ, wenn er auf das Ergebnis einer zielgerichteten Suche angewandt wird.

Korollar 1.41 *Zielgerichtete Suche kann so implementiert werden, daß ihre Laufzeit in $\mathcal{O}(|A| \log |V|)$ ist.*

Bemerkung: Die intuitive Einsicht hinter diesem Verfahren ist, daß die Knoten im Knotenspeicher so etwas wie eine „Wellenfront“ bilden, die sich Schritt für Schritt von s aus in alle Richtungen im Graphen „wegbewegt“. Die erste heuristische Beschleunigungstechnik aus Abschnitt 1.8 „friert“ die Wellenfront in dem Moment ein, wenn t überschritten ist. Grob gesprochen breitet sich die Wellenfront in alle Richtungen ungefähr gleich schnell aus, wenn man die Distanz der im Knotenspeicher enthaltenen Knoten von s aus als Maß für die „zurückgelegte Entfernung“ nimmt.[10] Durch den Übergang von ℓ nach ℓ_h soll erreicht werden, daß die Wellenfront sich statt dessen asymmetrisch in Richtung t schneller ausbreitet und in die entgegensetzte Richtung entsprechend langsamer, so daß der Moment des Einfrierens in weniger Schritten erreicht wird.

Dazu ist es sinnvoll, h so zu wählen, daß h möglichst weitgehend in folgendem Sinne monoton ist: Falls für $v, w \in V$ der kürzeste (v,t)–Pfad kürzer als der kürzeste (w,t)–Pfad ist, dann soll auch $h(v) < h(w)$ sein. □

Beispiel: Wir nehmen an, daß G ein Straßenverkehrsnetz darstellt, das heißt, die Knoten sind Kreuzungspunkte und die Kanten Straßen. Ohne Einschränkung identifizieren wir die Knoten mit ihren Koordinaten. Ein Knoten wird also aufgefaßt als ein Element von $\mathbb{R}^2$. Der Wert $\ell(v,w)$ der Kante $(v,w) \in A$ soll die Länge der Straße von v nach w sein und ist daher größer oder gleich dem Euklidischen Abstand $\|v - w\|_2$.

Für festes $\varepsilon > 0$ ist die Knotenbewertung h_ε, die durch $h_\varepsilon(v) := (1-\varepsilon) \cdot \|v - t\|_2$ für $v \in V$ definiert ist, aufgrund der Dreiecksungleichung für $\|\cdot\|_2$ offensichtlich eine zulässige Zielunterschätzung. Auch aus praktischer Sicht sollte dies bei kleinem ε eine geeignete Wahl sein. Diese intuitive Vermutung hat sich bei „Routenplanern“ für den Autoverkehr in der Praxis durchaus bestätigt. □

[10] Genauer: Man kann beweisen, daß die Differenz der Werte $\delta(v)$ und $\delta(w)$ zweier gleichzeitig im Knotenspeicher enthaltenen Knoten v und w nicht größer als $\max\{\ell(u,u') : (u,u') \in A\}$ sein kann.

Zusammen mit der ersten heuristischen Beschleunigungstechnik aus Abschnitt 1.8 bestehen also in praktischen Anwendungen gute Aussichten, ein geeignetes h zu finden, das für eine echte Verbesserung der Laufzeit sorgt. Es bleibt aber eine *heuristische* Beschleunigungstechnik.

1.11 Abschließende Bemerkung

Man kann beweisen, daß der Algorithmus von Dijkstra nicht nur auf strikt positiven Kantenlängen wie in Definition 1.4 gefordert funktioniert, sondern auch auf nichtnegativen Kantenlängen.[11] Falls bezüglich dieser nichtnegativen Kantenlänge immer noch jeder Zykel im Graphen strikt positive Länge hat, funktioniert auch der Algorithmus aus Satz 1.13 weiterhin. Ansonsten muß dieser Algorithmus etwas verkompliziert werden, damit die Suche von t nach s zurück nicht in eine Endlosschleife entlang eines Zykels der Länge 0 geraten kann.

2 Algorithmen aus abstrakter Sicht

2.1 Einleitung

Im folgenden bezeichnet $\mathcal{P}(S)$ die *Potenzmenge* der Menge S, das heißt, die Menge aller Teilmengen von S.

Definition 2.1 *Ein* **algorithmisches Problem** *ist spezifiziert durch $m, n \in \mathbb{Z}^+$, beliebige Mengen $\mathcal{I}_1, \ldots, \mathcal{I}_m$ und $\mathcal{A}_1, \ldots, \mathcal{A}_n$, eine Teilmenge $\mathcal{I} \subseteq \mathcal{I}_1 \times \cdots \times \mathcal{I}_m$ sowie eine Funktion $f : \mathcal{I} \to \mathcal{P}(\mathcal{A}_1 \times \cdots \times \mathcal{A}_n)$. Die Elemente von $\mathcal{I}$ sind die* **zulässigen Eingaben**.

[11] Insbesondere reicht auch die Forderung $\ell(v,w) \geq h(v) - h(w)$ in Definition 1.38, und im Beispiel am Ende von Abschnitt 1.10 kann der Faktor $(1-\varepsilon)$ eigentlich fallengelassen werden.

Für $i := (i_1, \ldots, i_m) \in \mathcal{I}$ schreiben wir auch $f(i_1, \ldots, i_m)$ anstelle von $f(i)$.

Bemerkung: Definition 2.1 erscheint komplizierter als nötig. Aber diese „Verkomplizierung“ entspricht der typischen Struktur algorithmischer Probleme und ihrer Umsetzung in Programmiersprachen (z.B. als Funktionen, Prozeduren oder — wie in Java — Methoden). Dadurch wird die Formulierung eines konkreten algorithmischen Problems in diesem Modell einfacher.

Die Mengen $\mathcal{I}_1, \ldots, \mathcal{I}_m$ repräsentieren m Eingabeparameter, und entsprechend stehen $\mathcal{A}_1, \ldots, \mathcal{A}_n$ für n Ausgabeparameter. Die Menge $\mathcal{I}$ muß eingeführt werden, weil im allgemeinen nicht jede *mögliche* Eingabe auch eine *zulässige* Eingabe ist. Die Funktion f bildet jedes $i \in \mathcal{I}$ auf die Menge aller *korrekten Ausgaben* bzgl. i ab, das heißt, die Elemente von $f(i)$ sind genau die Ausgaben, die das algorithmische Problem für Eingabe i lösen.

Mit anderen Worten: Das algorithmische Problem besteht letztendlich darin, zu beliebigem $i \in \mathcal{I}$ entweder ein beliebiges Element aus $f(i)$ zu bestimmen oder festzustellen, daß $f(i) = \emptyset$ ist. □

Beispiele:

- Quadratwurzel:
  - $m := n := 1$;
  - $\mathcal{I}_1 := \mathbb{R}$;
  - $\mathcal{A}_1 := \mathbb{R}$;
  - $\mathcal{I} := \mathbb{R}_0^+$;
  - $f(x) := \{\sqrt{x}\}$ für $x \in \mathcal{I}$.
- Potenzen:
  - $m := 2$ und $n := 1$;
  - $\mathcal{I}_1 := \mathcal{I}_2 := \mathbb{R}$;
  - $\mathcal{A}_1 := \mathbb{R}$;

- $\mathcal{I} := \{(x, y) : x,\ y \in \mathbb{R},$
  $x > 0$
  $\vee\ (x = 0 \wedge y \in \mathbb{Z}^+)$
  $\vee\ (x < 0 \wedge y \in \mathbb{Z})\}$;[12]
- $f(x, y) := \{x^y\}$ für $(x, y) \in \mathcal{I}$.

- Nullstellen von Polynomen:
  - $m := n := 1$;
  - $\mathcal{I}_1$ ist die Menge aller reellen Polynome;
  - $\mathcal{A}_1 := \mathbb{R}$;
  - $\mathcal{I} := \mathcal{I}_1$;
  - $f(p) := \{x : x \in \mathbb{R},\ p(x) = 0\}$ für jedes Polynom p.

- Lineare Gleichungssysteme:
  - $m := 2$ und $n := 1$;
  - $\mathcal{I}_1$ ist die Menge aller reellen Matrizen und $\mathcal{I}_2$ die Menge aller reellen Vektoren;
  - $\mathcal{I}$ ist die Menge aller Paare $(B, b) \in \mathcal{I}_1 \times \mathcal{I}_2$, so daß die Dimension von b gleich der Zeilenzahl von B ist;
  - $\mathcal{A}_1 := \bigcup_{d \geq 0} \mathbb{R}^d$;
  - $f(B, b) := \{x : x \in \mathbb{R}^k,\ Bx = b\}$ für $(B, b) \in \mathcal{I}$, wobei k die Spaltenzahl von B ist.

Ein **Algorithmus** zu einem gegebenen algorithmischen Problem ist eine Handlungsvorschrift, wie dieses algorithmische Problem in endlich vielen *elementaren Schritten* zu lösen ist, das heißt, wie zu einer gegebenen Eingabe $i \in \mathcal{I}$ eine korrekte Ausgabe $a \in f(i)$ berechnet werden kann. Damit ein Computer eine solche Handlungsvorschrift ausführen kann, muß sie in einer Programmiersprache abgefaßt sein.

Was unter einem elementaren Schritt zu verstehen ist, hängt von der gewählten Sprache ab: In einer höheren Programmiersprache (z.B. Java) kann ein einzelner elementarer Schritt sehr viel mehr umfassen als etwa in einer maschinennahen Programmiersprache (Assembler). Durch Unterprogramme (Funktionen, Prozeduren, Methoden) kann eine beliebig große Zahl von elementaren Schritten zu einem neuen elementaren Schritt — Aufruf des Unterprogramms — zusammengefaßt werden

Bemerkung: Wir unterscheiden zwischen *Programmanweisungen* und *Schritten*: Jede elementare Programmanweisung ergibt einen oder mehrere elementare Schritte jedesmal, wenn sie abgearbeitet wird. □

Bei Programmanweisungen, in denen komplizierte mathematische Formeln o.ä. ausgewertet werden, zählt man im allgemeinen die Abarbeitung jeder einzelnen Operation als einen elementaren Schritt. Solange keine Schleifen oder Unterprogramme ins Spiel kommen, ergibt also jede Programmanweisung A eine feste Zahl $n(A)$ von elementaren Schritten, die unmittelbar nacheinander ausgeführt werden.

Bei einer Programmanweisung, die in einer Schleife oder einem Unterprogramm steht, ergibt sich als Gesamtzahl induzierter elementarer Schritte natürlich das Produkt aus $n(A)$ und der Gesamtzahl der Durchläufe durch die Schleife bzw. Aufrufe des Unterprogramms. Es werden weiterhin nur jeweils $n(A)$ von diesen elementaren Schritten als eine Kette unmittelbar aufeinanderfolgend ausgeführt.

[12] Zur Potenzbildung gibt es in Java die Klassenmethode `java.lang.Math.pow`. Die Definition der zulässigen Eingaben (x, y) für diese Methode ist bis auf die Behandlung arithmetischer Überläufe identisch mit dieser Definition von $\mathcal{I}$.

2.2 Korrektheit von Algorithmen

Unter einer *Spezifikation* eines algorithmischen Problems verstehen wir eine Beschreibung gemäß Definition 2.1. In Abschnitt 2.2 gehen wir von einer beliebigen, aber festen Spezifikation eines algorithmischen Problems aus und verwenden dafür die Notation aus Definition 2.1.

Definition 2.2 *Ein Algorithmus heißt* **korrekt** *bzgl. einer Spezifikation eines algorithmischen Problems, wenn er für jede Eingabe $i \in \mathcal{I}$ folgende Bedingungen erfüllt:*

- *Alle elementaren Schritte sind wohldefiniert, der Algorithmus terminiert nach endlich vielen elementaren Schritten, und er liefert eine Ausgabe aus der Menge $\mathcal{A}_1 \times \cdots \times \mathcal{A}_n$.*
- *Zu jedem $i \in \mathcal{I}$ ist die Ausgabe des Algorithmus ein Element aus $f(i)$.*

Siehe auch das zugehörige Arbeitsblatt auf Seite 39 ff.

Bemerkungen:

- Die erste Bedingung bedeutet: Der Algorithmus darf weder „abstürzen“ noch in eine Endlosschleife geraten. Arithmetischer Überlauf oder ein Zugriff auf eine nicht existierende Vektorkomponente, der nicht vom Programm abgefangen wird, sind Beispiele für elementare Schritte, die nicht wohldefiniert sind.[13]
- Die Forderung, daß die Eingabe aus $\mathcal{I}$ entnommen sein muß, heißt im allgemeinen auch die *Vorbedingung* des Algorithmus, und die Forderung, daß das Ergebnis für Eingabe i zu $f(i)$ gehören muß, die *Nachbedingung*.
- Das Entwurfsprinzip „Design-by-Contract“ besagt folgendes (speziell angewandt auf Algorithmen): Ein Algorithmus und sein „Benutzer“ schließen einen Vertrag (engl. *contract*). Inhalt: Wenn der Benutzer die Vorbedingung erfüllt, muß der Algorithmus die Nachbedingung erfüllen, ansonsten darf der Algorithmus beliebig auf die Eingabe reagieren (also auch abstürzen oder in eine Endlosschleife geraten).

  Ein solcher Vertrag regelt im Prinzip die Verteilung von Verantwortlichkeiten zwischen Algorithmus und Aufrufer. Die Verantwortung für die Einhaltung der Vorbedingungen wird also bei Design-by-Contract voll und ganz dem Aufrufer aufgebürdet. (Alternativ könnte auch der Algorithmus die Korrektheit der Eingabe prüfen und ggf. eine Fehlermeldung ausgeben und den Dienst verweigern.)

Wenn reelle Zahlen in einem algorithmischen Problem vorkommen, ergibt sich das Problem, daß die numerischen Berechnungen im allgemeinen nicht mehr exakt sein können.

Definition 2.3 *Sei $\varepsilon > 0$ und $j \in \{1, \ldots, n\}$, so daß $\mathcal{A}_j \subseteq \mathbb{R}$. Ein Algorithmus heißt* **absolut ε-korrekt** *im j-ten Ausgabeparameter, wenn folgendes gilt: Zu jedem $i \in \mathcal{I}$ und der zugehörigen j-ten Ausgabe $a_j(i) \in A_j$ des Algorithmus gibt es ein $x = (x_1, \ldots, x_n) \in f(i)$ mit $|x_j - a_j(i)| \leq \varepsilon$. Analog heißt der Algorithmus* **relativ ε-korrekt**, *wenn statt dessen $|x_j - a_j(i)| \leq \varepsilon \cdot |x_j|$ gilt.*

[13] Es gibt in Java ein programmiersprachliches Konstrukt, die sogenannten *Exceptions*, eigens um den Lauf eines Programms noch irgendwie halbwegs vernünftig fortsetzen zu können, nachdem das Programm doch in einen solchen Fehler gelaufen ist.

Bemerkung: Diese beiden Definitionen fordern also, daß der absolute bzw. relative Fehler bei der Berechnung des j-ten Ausgabeparameters durch ε beschränkt ist. Die Betrachtung anderer Fehlerdefinitionen ist ebenfalls manchmal sinnvoll. Natürlich läßt sich Definition 2.3 prinzipiell auf jede beliebige Menge $\mathcal{A}_j$ übertragen, auf der eine Norm definiert ist (bzw. eine Metrik reicht für absolute Korrektheit).

Auf diese numerische Fehlerrechnung gehen wir hier nicht weiter ein. Veranstaltungen zur Numerischen Mathematik vertiefen das Thema. □

2.3 Korrektheitsbeweise für Algorithmen

Um Korrektheitsaussagen über Algorithmen zu beweisen, bietet es sich an, den Algorithmus in einzelne Unteralgorithmen zu zerlegen, die Korrektheit dieser Unteralgorithmen jeweils für sich zu beweisen und daraus dann die Korrektheit des Algorithmus als Ganzes zu folgern. Diese Zerlegung wird so weit getrieben, bis die Korrektheit der einzelnen verbliebenen Unteralgorithmen unmittelbar einsichtig ist. Im Zweifelsfall muß bis auf elementare Programmanweisungen zerlegt werden.

Bemerkung: Eine solche Zerlegung findet im Prinzip „in Gedanken“ statt. Es ist aber durchaus ratsam darauf zu achten, daß die Struktur der Formulierung des Algorithmus möglichst mit der Struktur dieser Zerlegung zusammenfällt, so daß die Korrektheit leichter nachzuvollziehen ist. Am besten hält man sich für beides an die Strukturierungskonstrukte, die die Programmiersprache selbst bietet, insbesondere Verzweigungs- und Schleifenstrukturen sowie Funktionen/Prozeduren/Methoden. □

Um die Korrektheit eines ganzen Algorithmus aus der Korrektheit seiner einzelnen Unteralgorithmen zu folgern, kann man nach folgendem Zerlegungsschema gemäß „Design-by-Contract“ vorgehen: Für jeden Unteralgorithmus wird wie für den „ganzen“ Algorithmus eine Vor- und eine Nachbedingung formuliert. Es müssen dann „nur noch“ folgende Aussagen bewiesen werden:

- Für jeden Unteralgorithmus U gilt: Wenn die Vorbedingung von U unmittelbar vor Aufruf von U erfüllt ist, dann ist unmittelbar nach Abarbeitung von U die Nachbedingung von U erfüllt (nicht mehr notwendigerweise die Vorbedingung von U).
- Wenn ein Unteralgorithmus U_1 unmittelbar vor einem Unteralgorithmus U_2 ausgeführt wird, so impliziert die Nachbedingung von U_1 die Vorbedingung von U_2.
- Die Vorbedingung des eigentlichen Algorithmus impliziert die Vorbedingung des allerersten Unteralgorithmus.
- Die Nachbedingung des allerletzten Unteralgorithmus impliziert die Nachbedingung des eigentlichen Algorithmus.

Oft ist die Nachbedingung eines Unteralgorithmus U_1 (oder ein Teil dieser Nachbedingung) nicht relevant für den unmittelbar danach abgearbeiteten Unteralgorithmus U_2, sondern erst für den unmittelbar nach U_2 abgearbeiteten Unteralgorithmus U_3 (oder sogar einen noch späteren Unteralgorithmus). In diesem Fall muß zusätzlich gezeigt werden, daß die Ausführung von U_2 die Nachbedingung von U_1 nicht zerstört, das heißt, daß diese Bedingung beim Aufruf von U_3 immer noch gültig ist.

Dazu wird der für U_3 relevante Teil der Nachbedingung von U_1 in die Vorbedingung *und*

die Nachbedingung von U_2 aufgenommen. Durch diesen Kniff kann U_2 grundsätzlich nur dann korrekt sein, wenn der Aufruf von U_2 die Vorbedingung für U_3 nicht zerstört.

Beispiel: der Algorithmus aus Satz 1.13.

Die Vorbedingung *VB* des gesamten Algorithmus ist, daß δ die korrekten Distanzen der Knoten von s aus angibt.[14] Als Nachbedingung *NB* wird gefordert, daß das Ergebnis $(w_i, w_{i-1}, \ldots, w_1)$ ein kürzester (s,t)-Pfad ist.

Wir zerlegen den Algorithmus gemäß der Einteilung 1, 2(a) und 2(b) in Unteralgorithmen, zerlegen diese Unteralgorithmen aber nicht weiter, sondern behandeln sie als Einheiten, deren Korrektheit auch ohne weitere Zerlegung einsichtig gemacht werden kann.

VB ist erst für den Unteralgorithmus 2 relevant, so daß *VB* zur Vor- und Nachbedingung von Unteralgorithmus 1 hinzugenommen wird. Ansonsten hat Unteralgorithmus 1 keine Vorbedingung, und seine Nachbedingung insgesamt lautet: (*VB* und $w_1 = t$). Die Korrektheit von Unteralgorithmus 1 folgt trivialerweise.[15]

Eine Schleife wie in Unteralgorithmus 2 kann als ein Unteralgorithmus aufgefaßt werden, der mehrfach hintereinander (mit verschiedenen Variablenwerten) aufgerufen wird. Abgesehen von *VB* gehört zur Vorbedingung von Unteralgorithmus 2(a), daß $(w_i, \ldots, w_1)$ ein (w_i, t)-Pfad ist und daß $\ell(w_j, w_{j-1}) = \delta(w_{j-1}) - \delta(w_j)$ für alle $j \in \{2, \ldots, i\}$ ist. Die Vorbedingung von Unteralgorithmus 2(a) folgt offenbar für $i = 1$ aus der Nachbedingung von Algorithmus 1.

VB und diese beiden Zusatzbedingungen gehören auch zur Nachbedingung von Unteralgorithmus 2(a). Darüber hinaus wird in der Nachbedingung von 2(a) eine Fallunterscheidung gemacht: Falls es sich um den allerletzten Durchlauf durch die Schleife handelt, kommt noch die Bedingung $w_i = s$ zur Nachbedingung von Unteralgorithmus 2(a) dazu. Daraus folgt die Nachbedingung des gesamten Algorithmus: Der zweite Absatz im Beweis von Satz 1.13 ist nämlich ein Beweis dafür, daß die Nachbedingung von Unteralgorithmus 2(a) im Fall $w_i = s$ tatsächlich die Nachbedingung des Algorithmus impliziert.

Andernfalls kommt entsprechend die Bedingung $w_i \neq s$ hinzu. Die Vorbedingung von Unteralgorithmus 2(b) ist genau die Vorbedingung von Unteralgorithmus 2(a) plus der Bedingung $w_i \neq s$. Die Nachbedingung von Unteralgorithmus 2(b) besteht aus vier Teilen:

1. VB,
2. w_{i+1} ist wohldefiniert,
3. $(w_{i+1}, w_i, \ldots, w_1)$ ist ein (w_{i+1}, t)-Pfad, und
4. $\ell(w_j, w_{j-1}) = \delta(w_{j-1}) - \delta(w_j)$ gilt nun für alle $j \in \{2, \ldots, i+1\}$.

Mit anderen Worten: Die Nachbedingung am Ende eines Schleifendurchlaufs ist genau die Vorbedingung für den nächsten Durchlauf. Daß die Teile 1, 3 und 4 der Nachbedingung von Unteralgorithmus 2(b) unmittelbar nach Abarbeitung von Unteralgorithmus 2(b) erfüllt sind, ist offensichtlich (natürlich immer vorausgesetzt, daß die Vorbedingung ebenfalls erfüllt war). Teil 2 folgt aus Korollar 1.12, da die Vorbedingung von Unteralgorithmus 2(b) die Vorbedingung von Korollar 1.12 impliziert.

[14] Lemma 1.35 zeigt, daß auch schon eine schwächere Vorbedingung ausreichen würde.

[15] Wenn der Algorithmus in eine Programmiersprache umgesetzt wird, muß natürlich noch geprüft werden, ob die Umsetzung von Unteralgorithmus 1 korrekt ist. Solche Prüfungen sind nicht immer so trivial wie in diesem Beispiel!

Es ist notwendig, daß *VB* als Bedingung auch durch die Vor- und Nachbedingung von 2(a) „hindurchgereicht“ wird, weil die Korrektheit von Unteralgorithmus 2(b) darauf beruht. *VB* gehört auch zur Nachbedingung von 2(b), da sonst im nächsten Schleifendurchlauf die Vorbedingung für 2(a) nicht garantiert ist. □

Bemerkungen:

- Auch wenn nicht explizit so formuliert, basiert ein Korrektheitsbeweis für eine Schleife (oder eine Rekursion) eigentlich immer auf einer vollständigen Induktion über die Anzahl der Durchläufe (bzw. Rekursionstiefe).
- Eine Bedingung, die zugleich Nachbedingung eines Schleifendurchlaufs und Vorbedingung des nächsten Durchlaufs wie oben im Beispiel ist, nennt man *Schleifeninvariante*.
- Es stellt sich die Frage, ob Korrektheitsbeweise nach obigem Zerlegungsschema nicht automatisierbar, das heißt durch ein Computerprogramm ausführbar sind. Man kann mathematisch beweisen, daß kein Programm dies in voller Allgemeinheit leisten kann. (Das wird in Informatik III bewiesen.)

  Das Hauptproblem ist, daß Schleifeninvarianten nicht vollautomatisch gefunden werden können. Ein Zweig der Informatik erforscht, ob solche Korrektheitsbeweise nicht doch automatisiert werden können, sofern Programmtexte durch explizites Hinschreiben von Invarianten und anderen Zusatzinformationen (*Annotationen*) erweitert werden.

3 Datenstrukturen

3.1 Abstrakte Definition

Eine *Datenstruktur* D läßt sich formal beschreiben durch eine *Zustandsmenge* Z und Operationen $M_1, \ldots, M_k$, den *Methoden* von D. Ein *Objekt* o vom Typ D ist eine Variable, deren Wertemenge Z ist. Eine Methode M_i hat neben den gewöhnlichen Ein- und Ausgabeparametern von Unterprogrammen zusätzlich ein Objekt o vom Typ D als Parameter.[16]

Beispiel: Der Knotenspeicher gemäß Abschnitt 1.6 ist eine solche Datenstruktur. Die Zustandsmenge Z ist die Menge aller endlichen Mengen von Knoten von Graphen. Ein konkretes Objekt dieser Datenstruktur ist mit einem Graphen $G = (V, A)$ assoziiert, und dieses Objekt kann nur Teilmengen von V als Zustände haben. Die vier Anforderungen am Anfang von Abschnitt 1.6 ergeben die Methoden der Datenstruktur (zuzüglich der unerwähnt gebliebenen, aber notwendigen Methode „Einrichten eines leeren Knotenspeichers für G“). □

Wir sagen, daß der Heap aus Abschnitt 1.7 die Datenstruktur Knotenspeicher *implementiert*. Das heißt, Teilmengen von V werden auf Zustände des Heap-Objektes abgebildet und die Methoden des Knotenspeichers auf entsprechende Operationen auf dem Heap.

Heaps können ebenfalls als eine Datenstruktur aufgefaßt werden. Ein Heap-Objekt ist assoziiert mit einer Menge S, einer Funktion $f : S \to \mathbb{R}$ sowie $n \in \mathbb{Z}^+$. Die möglichen Zustände dieses Objekts sind geordnete Fol-

[16] In sogenannten *objektorientierten* Programmiersprachen wie Java spiegelt sich diese Sonderrolle des Objekts o in der Syntax wider: $o.M_i(p_1, \ldots, p_r)$, wobei $p_1, \ldots, p_r$ die gewöhnlichen Ein- und Ausgabeparameter sind.

gen aus S (die in $H[1], \ldots, H[m]$ gespeichert sind),[17] allerdings nicht alle möglichen Folgen, sondern nur Folgen mit maximal n Elementen, wobei Wiederholungen nicht zugelassen sind und zusätzlich die Heap-Eigenschaft aus Definition 1.28 erfüllt sein muß.

Das heißt, eine Datenstruktur kann eine andere implementieren. Mit anderen Worten: Man kann ein und dieselbe Datenstruktur auf verschiedenen Abstraktionsebenen betrachten: Knotenspeicher sind abstrakter als Heaps und können auch anders als durch Heaps implementiert werden. Heaps sind in folgendem Sinne durch den Datentyp von S parametrisiert:

Eine *parametrisierte Datenstruktur PD* ist im Prinzip ein „Muster“ für eine potentiell unendliche Menge von Datenstrukturen, die sich nur in einigen konkreten Typen unterscheiden, die in den Methoden auftreten. Genauer gesagt wird für mindestens einen der gewöhnlichen Parameter mindestens einer der Methoden von *PD* der konkrete Typ in der Definition von *PD* offengehalten und nur durch ein *formales Typargument* benannt. Erst durch Einsetzen eines richtigen Typs für jeden formalen Typ ergibt sich eine „richtige“ Datenstruktur.

In Pizza, der Spracherweiterung von Java, die in den Übungen behandelt wird, gibt es dafür *Typparameter* als Sprachkonstrukt. Wir verwenden im folgenden die Pizza-Schreibweise $<\ldots>$ zur Parametrisierung von Datenstrukturen. Von besonderer Bedeutung sind folgende parametrisierte Datenstrukturen:

- Stack $<T>$: Die Zustandsmenge eines (unbeschränkten) Stacks ist die Menge aller geordneten Folgen von T, und die Methoden sind:
  - *construct()*: Richtet einen Stack mit Zustand () ein.[18]
  - *push(t)*: Transformiert Zustand $(t_1, \ldots, t_n)$ in Zustand $(t_1, \ldots, t_n, t)$.
  - *top()*: Hat zur Vorbedingung, daß der Zustand nicht gleich () ist, und liefert t_n für den momentanen Zustand $(t_1, \ldots, t_n)$.
  - *pop()*: Hat zur Vorbedingung, daß der Zustand nicht gleich () ist, und transformiert Zustand $(t_1, \ldots, t_n)$ in Zustand $(t_1, \ldots, t_{n-1})$.
  - *size()*: Liefert die Länge der Folge, die den Zustand darstellt, also n für $(t_1, \ldots, t_n)$ und 0 für ().
- Queue $<T>$: Wie beim Stack, nur daß *pop* das Ergebnis $(t_2, \ldots, t_n)$ hat.
- DynamicDictionary $<T_1, T_2>$: Die Zustände sind die Mengen M von Paaren (t_1, t_2) mit $t_1 \in T_1$ und $t_2 \in T_2$, mit der Einschränkung, daß kein t_1 mehrfach in M auftreten darf (t_1 ist der *Schlüssel* und t_2 die zugehörige *Information*). Die Methoden sind:
  - *construct()*: wie bei Stacks.
  - *included(t_1)*: Liefert *wahr* zurück, wenn es ein Paar (t_1, t_2) mit irgendeinem $t_2 \in T_2$ gibt, sonst *falsch*.
  - *info(t_1)*: Hat *included(t_1)=wahr* als Vorbedingung und liefert in diesem Fall das zugehörige t_2.
  - *insert(t_1,t_2)*: Hat *included(t_1)* = *falsch* zur Vorbedingung und fügt das Paar (t_1, t_2) in die momentane Menge M ein.

[17] Solche Formulierungen schließen auch im weiteren immer die leere Folge () ein, hier also den Fall $m = 0$.

[18] In Java oder Pizza realisiert man eine solche Methode *construct* natürlich als einen Konstruktor.

  - *overwrite(t_1,t_2)*: Hat *included(t_1)* = *wahr* als Vorbedingung und überschreibt die momentan mit t_1 assoziierte Information durch t_2.
  - *remove(t_1)*: Hat *included(t_1)* = *wahr* als Vorbedingung und löscht das Paar, das t_1 enthält.

- StaticDictionary $< T_1, T_2 >$: Diese Variante ist weitgehend analog zu DynamicDictionary, abgesehen von den folgenden Ausnahmen:

  - Der Zustand wird nicht nur durch eine Menge von Paaren beschrieben, sondern zusätzlich noch durch einen binären Modus „im Aufbau"/„aufgebaut".
  - Methode *construct* setzt den Modus auf „im Aufbau".
  - Eine zusätzliche Methode namens *ready()* setzt den Modus auf „aufgebaut".
  - Methode *remove* gehört nicht zur Datenstruktur.
  - Methoden *included*, *info* und *overwrite* haben Modus „aufgebaut" als zusätzliche Vorbedingung, während Methode *insert* Modus „im Aufbau" als Vorbedingung hat.

Bemerkungen:

- Die obigen Definitionen von Datenstrukturen wie Stacks, Queues und Dictionaries sind nur beispielhaft zu verstehen. Es gibt keine Standardisierungen „nach DIN-Norm" o.ä. Aus programmiertechnischer Sicht sind die obigen Definitionen auch nicht sonderlich geschickt, sondern eher daraufhin konzipiert, daß nach Möglichkeit die fundamentale Konzeption hinter den einzelnen Datenstrukturen klar wird.

- Eine formale Beschreibung der Zustandsmenge S einer Datenstruktur D wie oben in den einzelnen Beispielen verwendet wird auch eine *Darstellungsinvariante* genannt. Wenn Datenstruktur D_1 durch Datenstruktur D_2 implementiert wird, dann nennt man eine Darstellungsinvariante von D_2 auch eine *Implementationsinvariante* von D_1 in der Implementation durch D_2.

  Ein Beispiel dafür wäre etwa die Heap-Eigenschaft aus Definition 1.28: Das ist eine Darstellungsinvariante für die Heap-Datenstruktur D_2, aber eine Implementationsinvariante für die Datenstruktur *Knotenspeicher*, wenn der Knotenspeicher als Heap implementiert ist.

- In vielen Anwendungen von Dictionaries sind die Elemente von T_1 eher klein, während die Elemente von T_2 beliebig groß werden können (z.B. wenn ein Dictionary — wie es der Begriff sagt — zu einer Menge von Stichwörtern jeweils einen langen, erklärenden Text abspeichert). Dann werden die Elemente von T_2 typischerweise nicht selbst im Dictionary gehalten, sondern nur Verweise darauf, und die Elemente von T_2 werden in einem Hintergrundspeicher (z.B. Festplatte) gehalten. Eine solche Organisation eines Dictionaries nennt man *externe Speicherung*.

Dictionaries sind fundamental für alle Arten von Datenverwaltung, insbesondere Datenbanksysteme, Dokumentenverwaltungssysteme u.ä. In den folgenden beiden Unterabschnitten betrachten wir die wichtigsten Implementationen für dynamische und statische Dictionaries: *B-Bäume* (dynamisch) und *Hashes* (statisch).

3.2 B–Bäume

Unter einer *total geordneten Menge* $(M, <)$ verstehen wir eine Menge M, auf der eine binäre, irreflexive, transitive Operation „$<$“ definiert ist. Im folgenden schreiben wir kurz M anstelle von $(M, <)$.

Definition 3.1 *Seien T_1 und T_2 beliebige Mengen. Eine Menge $M \subseteq T_1 \times T_2$ heißt* **zulässig**, *wenn $t_1 \neq t_1'$ für alle (t_1, t_2), $(t_1', t_2') \in M$ gilt.*[19]

Definition 3.2 *Sei $k \in \mathbb{Z}^+$ gerade mit $k \geq 4$, T_1 eine total geordnete Menge und $M \subseteq T_1 \times T_2$ zulässig. Ein B–Baum der Ordnung k für M ist ein Baum mit folgenden Eigenschaften:*

- *Alle Blätter des Baumes haben die gleiche Höhe.*
- *Jedem Knoten v des Baumes ist eine Teilmenge $M(v) \subseteq M$ so zugeordnet, daß jedes Element von M in genau einer solchen Menge $M(v)$ enthalten ist.*
- *Die Anzahl der Elemente von M in einem Baumknoten ist höchstens k.*
- *Falls $M \neq \emptyset$, enthält die Wurzel mindestens ein Element von M und jeder andere Knoten mindestens $k/2$ Elemente.*
- *Für einen Knoten v sei $m(v) := |M(v)|$ die Anzahl von Elementen von M, die in v enthalten sind. Wenn v kein Blatt ist, dann hat v genau $m(v) + 1$ Nachfolger.*
- *Für einen Baumknoten v und $\ell \in \{1, \ldots, m(v)+1\}$ sei w der ℓ–te unmittelbare Nachfolger von v, und sei $(t_1, t_2) \in M(w)$. Seien $(t_1', t_1''), \ldots, (t_{m(v)}', t_{m(v)}'')$ die Elemente von $M(v)$, so daß $t_i' < t_{i+1}'$ für $i \in \{1, \ldots, m(v) - 1\}$ gilt. Dann ist $t_i' < t_1$ für $i < \ell$ und $t_i' > t_1$ für $i \geq \ell$.*

[19] In mathematischer Terminologie formuliert: M ist eine linkseindeutige binäre Relation.

Typischerweise ist jeder Knoten v durch folgende Komponenten realisiert:

- Eine ganzzahlige Variable $m(v) \in \{0, \ldots, n\}$, wie in Definition 3.2 beschrieben.
- Einen Vektor $[1, \ldots, k]$ von Paaren aus $T_1 \times T_2$. Die tatsächlich gespeicherten Paare sind in den Komponenten $[1, \ldots, m(v)]$ abgelegt.
- Einen Vektor $[1, \ldots, k + 1]$ von Verweisen auf Nachfolger. Die tatsächlichen Nachfolger sind in den Komponenten $[1, \ldots, m(v) + 1]$ abgelegt.

Intention: B–Bäume werden in Datenbanksystemen und anderen Softwarepaketen eingesetzt, um große Datenmengen auf dem Hintergrundspeicher (in erster Linie auf Festplatten) so zu organisieren, daß der Zugriff möglichst effizient ist. Der „Flaschenhals“ ist der Zugriff auf den langsamen Hintergrundspeicher. Ist eine Information erst einmal in den Hauptspeicher des Computers geladen worden, dann ist jeder Zugriff um Größenordnungen schneller.[20]

Aus diesem Grund ist die Hardware typischerweise so organisiert, daß bei jedem Zugriff auf die Festplatte nicht nur ein einzelnes Bit, Byte oder Maschinenwort, sondern ein ganzer Block (*Seite* genannt) eingelesen wird. Das geht praktisch genauso schnell, als würde nur ein Bit eingelesen. Die Größe einer solchen Seite ist plattformspezifisch, ist aber für jede Hardwareplattform konstant.[21]

[20] Auf heutigen Computern ist der Zugriff auf eine Information im Hauptspeicher um den Faktor $10^5 - 10^8$ schneller als das Einladen der Information in den Hauptspeicher.

[21] Auf UNIX–Computern wie unseren Suns kann man sich die Größe einer Seite mit dem Kommando *pagesize* in Bytes ausgeben lassen.

Die Idee speziell hinter B–Bäumen ist nun, eine verzeigerte Datenstruktur aufzubauen, so daß jeder einzelne Knoten in dieser Struktur nicht nur ein Element enthält, sondern so viele Elemente, wie in eine Seite hineinpassen. Mit anderen Worten: k ist so zu wählen, daß k Paare aus $T_1 \times T_2$, $k+1$ Verweise auf Nachfolger sowie die Zahl $m(v)$ zusammen auf eine Seite passen.[22] Die Zeit für die Auswertung einer Seite im Hauptspeicher, das heißt, ob das gesuchte Element in der eingelesenen Seite zu finden ist bzw. (falls nicht) welche der $m(v)+1$ Nachfolgerseiten als nächstes einzulesen ist, ist auch bei ungeschickter Organisation des Inhalts eines Knotens vernachlässigbar im Vergleich zum Zugriff auf die Festplatte.

Damit der Zugriff auf ein Element auch im schlimmsten Fall insgesamt nicht zu viele Zugriffe auf die Festplatte erfordert, muß die Höhe des Baumes beschränkt werden (siehe Satz 3.4 weiter unten). B–Bäume werden in Definition 3.2 so kompliziert definiert, damit auch Einfügen und Löschen asymptotisch nur linear in der Höhe und nicht etwa linear in der Anzahl Knoten des Baumes wächst. □

Die i–te *Schicht* eines Baumes umfaßt die Knoten, die Höhe i haben (wobei die Wurzel wieder Höhe 0 hat).

Lemma 3.3 *Für $M \subseteq T_1 \times T_2$ und $i > 0$ liegt die Gesamtzahl aller Elemente von M in der i–ten Schicht eines B–Baumes der Ordnung k zusammengezählt im abgeschlossenen Intervall*

$$\left[\; k \cdot \left(\frac{k}{2}+1\right)^{i-1} \ldots \;\; k \cdot (k+1)^i \;\right] .$$

Beweis: Die Wurzel enthält mindestens ein Element und jeder andere Knoten mindestens $k/2$ Elemente. Insbesondere hat die Wurzel mindestens zwei und jeder andere Knoten, der kein Blatt ist, mindestens $k/2+1$ unmittelbare Nachfolger. Zusammen mit der Forderung in Definition 3.2, daß alle Blätter auf gleicher Höhe h sind, ergibt sich mit einer trivialen vollständigen Induktion über h die untere Schranke

$$2 \cdot \left(\frac{k}{2}+1\right)^{i-1}$$

für die Anzahl der Knoten in der i–ten Schicht ($i \geq 1$), woraus mit $m(v) \geq k/2$ die untere Schranke in Lemma 3.3 folgt. Analog ergibt sich die obere Schranke daraus, daß jeder Knoten maximal k Elemente enthält und somit maximal $k+1$ unmittelbare Nachfolger hat. □

Satz 3.4 *Die Höhe eines B–Baumes für $M \subseteq T_1 \times T_2$ ist in $\mathcal{O}(\log |M|)$.*

Beweis: Sei h die Höhe. Aus Lemma 3.3 folgt

$$|M| \;\geq\; 1 \;+\; k \cdot \sum_{i=1}^{h} \left(\frac{k}{2}+1\right)^{i-1} .$$

Die Darstellung der geometrischen Summe in der üblichen geschlossenen Form ergibt mit ein paar Umformungen:

$$|M| \;\geq\; 2 \cdot \left(\frac{k}{2}+1\right)^{h} - 1 .$$

Durch Umstellen nach h folgt die Behauptung. □

[22] Bei externer Speicherung (siehe dritte Bemerkung auf Seite 23) braucht man natürlich keinen Platz für Elemente aus $T_1 \times T_2$, sondern für Paare, die aus einem Element von T_1 und einem Verweis auf ein Element aus T_2 bestehen.

Korollar 3.5 *Auffinden eines Elements von M in einem B–Baum erfordert im schlimmsten Fall den Zugriff auf $\mathcal{O}(\log |M|)$ Knoten des B–Baumes.*

Entscheidend ist, daß auch Einfügen eines neuen Elements und Löschen eines gespeicherten Elements nur Zugriff auf $\mathcal{O}(\log |M|)$ Knoten des B–Baumes erfordern. Diese Operationen sind ziemlich komplex, und wir werden sie hier nur kurz skizzieren.

Wenn ein neues Element (t_1, t_2) eingefügt werden soll, wird zunächst das Blatt v gesucht, in dem das neue Element zu finden wäre, wenn es schon enthalten wäre. Falls $m(v) < k$, wird das neue Element in $M(v)$ eingefügt, und die Einfügeoperation ist beendet.

Ansonsten wird v in zwei neue Knoten v' und v'' zerlegt. Dabei wird $M(v) \cup \{(t_1, t_2)\}$ je zur Hälfte auf $M(v')$ und $M(v'')$ verteilt, außer daß das Medianelement $(t'_1, t'_2) \in M(v) \cup \{(t_1, t_2)\}$[23] statt dessen in den unmittelbaren Vorgängerknoten w von v im B–Baum transferiert wird. Falls schon $|M(w)| = k$ galt, wird w analog in w' und w'' zerlegt und das Medianelement von $M(w) \cup \{(t'_1, t'_2)\}$ wiederum in den unmittelbaren Vorgänger von w transferiert. Das wird aufsteigend im Baum solange wiederholt, bis ein Knoten u mit $|M(u)| < k$ oder die Wurzel erreicht ist. Falls die Wurzel erreicht wird und auch schon voll war, wird sie ebenfalls zerlegt, und eine neue Wurzel wird über den beiden neuen Knoten eingerichtet.

Das Löschen eines Elements aus einem Knoten des B–Baumes ist wesentlich komplizierter. Hier müssen potentiell Elemente zwischen einem Knoten und zweien seiner unmittelbaren Nachfolger ausgetauscht werden, und wenn ein Knoten weniger als $k/2$ Elemente enthält, kann es in Umkehrung zur Zerlegungsoperation beim Einfügen passieren, daß der Knoten hinterher mit einem „Geschwisterknoten“ (d.h. einem Knoten mit gleichem unmittelbaren Vorgänger) verschmolzen werden muß. Wenn die Wurzel ihr letztes Element verliert, wird sie aus dem Baum entfernt. Unmittelbar davor hatte die Wurzel noch genau ein Element und somit genau zwei Nachfolger. Diese beiden Nachfolger werden in diesem Fall verschmolzen und bilden die neue Wurzel.

[23] Für eine endliche geordnete Menge $X = \{x_1, \ldots, x_r\}$ ist der *Median* $x \in X$ dadurch definiert, daß $y < x$ für genau $\lfloor |X|/2 \rfloor$ viele Werte $y \in X$ gilt. Hier bezieht sich diese Definition nur auf die Ordnung auf der ersten Komponente der Paare aus $T_1 \times T_2$.

Bemerkungen:

- Oft wird eine Variation von B-Bäumen, die sogenannten B*–Bäume, angewendet. In dieser Variante werden *alle* Elemente von M in den Blättern gehalten, und die internen Knoten des B–Baumes speichern nur Werte aus T_1. Genauer gesagt: Sei v ein innerer Knoten des B-Baumes, seien $t_1 < t_2 < \cdots < t_{m(v)}$ die Werte aus T_1 gespeichert in v, und seien $w_1, \ldots, w_{m(v)+1}$ die unmittelbaren Nachfolger von v. Für $\ell \in \{1, \ldots, m(v)\}$ ist t_ℓ so gewählt, daß die Elemente aus M in den Blättern, die an $w_1, \ldots, w_\ell$ hängen, kleiner als t_ℓ sind und die Werte in den übrigen Blättern größer oder gleich t_ℓ.
- Die Bereichssuche, das heißt, die Berechnung aller Elemente $(t_1, t_2) \in M$ mit $x \leq t_1 \leq y$ für vorgegebene $x, y \in T_1$, ist ebenfalls eine wichtige Operation in Datenbank- und Informationssystemen. In B-Bäumen braucht man zur Bestimmung des kleinsten und größten Wertes in $M \cap [x, y]$ nur $\mathcal{O}(\log |M|)$ Seitenzugriffe, nämlich indem man nach x und y im Baum sucht und entweder das jeweils gesuchte Element findet oder — falls nicht vorhanden — zumindest die Seite, auf der die Menge $M \cap [x, y]$ beginnt bzw. endet.

Wenn jeder Baumknoten noch zusätzlich einen Verweis auf den unmittelbaren Vorgängerknoten im Baum enthält, sind nach dieser Eingrenzung offensichtlich nur noch ungefähr n/k Festplattenzugriffe nötig, wobei n die Anzahl der Elemente von M im Bereich $[x, y]$ ist.

3.3 Hashes

Eine Menge T heißt *erweitert*, wenn sie einen „Nichtwert“ λ enthält, der einfach das Fehlen eines „richtigen“ Wertes aus T anzeigt. Wenn beispielsweise T eigentlich die Menge der positiven ganzen Zahlen darstellen soll, bietet sich $\lambda := 0$ als zusätzlicher Nichtwert an, der als Inhalt einer Variable anzeigt, daß diese Variable momentan keinen „richtigen“ Wert besitzt.[24]

Im folgenden gehen wir von einem Vektor H aus, dessen Komponenten Paare aus einer Menge $T_1 \times T_2$ sind, wobei T_1 eine erweiterte Menge ist. Wir schreiben $H[i].x$ und $H[i].y$, um den Zugriff auf den ersten bzw. zweiten Teil des Paares, das in der i–ten Komponente von H gespeichert ist, anzudeuten. (Insbesondere ist also $H[i].x \in T_1$ und $H[i].y \in T_2$.) Im Falle $H[i].x \neq \lambda$ heißt Index i durch $H[i].x$ *belegt*, sonst heißt er *frei*.

[24] Dieses Konzept ist für Klassentypen in der Sprache Java direkt eingebaut: $\lambda :=$ `null`.

Falls T_1 sich nicht so einfach durch einen solchen Nichtwert λ erweitern läßt (z.B. im Fall $T_1 = \mathbb{Z}$), kann man zum selben Effekt kommen, indem man T_1 durch $T_1' := T_1 \times \{wahr, falsch\}$ ersetzt mit der Bedeutung, daß $t \in T_1'$ genau dann als identisch mit λ aufzufassen ist, wenn die zweite Komponente von t gleich *falsch* ist.

Definition 3.6 *Seien T_1 eine erweiterte und T_2 eine beliebige Menge, $n \in \mathbb{Z}^+$, $F = (f_1, f_2, f_3, \ldots)$ eine unendliche Folge von Funktionen $f_i : T_1 \to \{0, \ldots, n-1\}$ und $m \in \mathbb{Z}_0^+$ mit $m \leq n$. Ein* **(n,m,F)–Hash** *ist gegeben durch einen Vektor H (die* **Hashtabelle***) mit Indexbereich $[0, \ldots, n-1]$, dessen Komponenten Paare $(t_1, t_2) \in T_1 \times T_2$ sind. Es muß $t_1 \neq \lambda$ für genau m der in H gespeicherten Paare $(t_1, t_2) \in T_1 \times T_2$ gelten, und diese m Werte $t_1 \neq \lambda$ müssen paarweise verschieden sein. Für jeden belegten Index $i \in \{0, \ldots, n-1\}$ gibt es ein $h \geq 1$, so daß $f_h(H[i].x) = i$ ist und $f_j(H[i].x) \neq \lambda$ für alle $j \in \{1, \ldots, h-1\}$ gilt.*

Hashes realisieren statische Dictionaries. Beim Einfügen eines neuen Elements $(t_1, t_2) \in T_1 \times T_2$ (Phase „im Aufbau“) wird zunächst geschaut, ob $f_1(t_1)$ frei ist. Falls ja, wird $f_1(t_1)$ durch t_1 belegt. Ansonsten wird geschaut, ob $f_2(t_1)$ frei ist usw. Die Suche bricht nach genau i Schritten ab, wenn $f_j(t_1)$ für $j \in \{1, \ldots, i-1\}$ belegt, $f_i(t_1)$ aber frei ist. Dann wird $f_i(t_1)$ durch t_1 belegt.

Beim Suchen nach einem Schlüssel $t_1 \in T_1$ (Phase „aufgebaut“nach Aufruf von Methode *ready*) schaut man analog in H an den Indizes $f_1(t_1)$, $f_2(t_1)$ usw. nach, bis man zu einem Index $f_i(t_1)$ kommt, der entweder frei oder durch t_1 belegt ist. Falls $f_i(t_1)$ frei ist, kann t_1 unmöglich im Hash gespeichert sein. Ist $f_i(t_1)$ hingegen durch t_1 belegt, so ist $H[f_i(t_1)].y$ die zu t_1 gespeicherte Information.

Die Funktionen f_i werden üblicherweise durch eine einfache Vorschrift aus einer endlichen Zahl von Funktionen konstruiert.

Beispiel: Gegeben sind zwei Funktionen, $f, g : T_1 \to \mathbb{Z}^+$, und für $i \in \mathbb{Z}^+$ ist f_i definiert als $f_i := (f + (i-1) \cdot g) \bmod n$. □

Zunächst betrachten wir nur f_1. Die Idee für f_1 ist, daß die m zu speichernden Werte so durch f_1 über die n Komponenten „gestreut“ (engl. *hashed*) werden, daß bei der Suche eines Elements *im Durchschnitt* möglichst wenige einzelne Suchschritte nötig sind. Das heißt formal: Wenn die Paare $(t_1, t'_1), \ldots, (t_m, t'_m)$ im Hash gespeichert sind und zum Zugriff auf (t_i, t'_i) die Funktionen $f_1, \ldots, f_{r(i)}$ auszuwerten sind, dann sollte folgender Wert möglichst klein gehalten werden:

$$\frac{1}{m} \sum_{i=1}^{m} r(i) \,.$$

Grob gesprochen ist es günstig, f_1 so zu definieren, daß jede mögliche Auswahl von m Elementen aus T_1 möglichst gleichmäßig auf $\{1, \ldots, n\}$ verteilt wird. (Diese Faustregel wird in Satz 3.9 fundiert.) Jeder Satz realer Daten aus einer Anwendung statischer Dictionaries ist eine Auswahl in diesem Sinne. Typischerweise sind naheliegende Definitionen von f_1 eher *ungünstig* im Sinne dieser Zielsetzung.

Beispiel: Wenn T_1 die Menge aller Strings ist, so ist es beispielsweise für f_1 naheliegend, die ASCII–Werte der einzelnen Zeichen aufzusummieren und durch Modulo–Bildung mit n auf den Indexbereich $\{0, \ldots, n-1\}$ von H abzubilden. Wenn die Auswahl, die im Hash zu speichern ist, die Stichwörter eines normalen Wörterbuchs sind, ergibt sich ein starkes Ungleichgewicht in den Häufigkeiten einzelner Buchstaben, aber auch ganzer Silben und sogar noch größerer Buchstabengruppierungen. Dadurch dürfte eine so definierte Funktion f eine solche Auswahl signifikant ungleichgewichtig verteilen. □

Im Idealfall verteilt f_1 jede Auswahl so auf $\{1, \ldots, n\}$, daß das Ergebnis dem Ergebnis einer Zufallsverteilung zum Verwechseln ähnlich sieht. Mit Zufallsverteilung ist hier gemeint, daß jedem Element von T_1 (unabhängig voneinander) einer der Werte $1, \ldots, n$ jeweils mit Wahrscheinlichkeit $1/n$ zugewiesen wird.[25]

Der Einfacheit halber beschränken wir uns im Rest von Abschnitt 3.3 auf $T_1 := \mathbb{Z}_0^+$ mit $\lambda := 0$. Wir schreiben aber weiterhin T_1 und λ, um Schlüsselwerte von anderen positiven ganzen Zahlen mit anderer Bedeutung notationell abzugrenzen.

Beispiele:

- *Erweiterte Divisionsmethode*: Man wählt feste Zahlen $a, b \in \mathbb{Z}^+$ und definiert $f_1(j) := (a \cdot j + b) \bmod n$ für $j \in T_1$.

  Die Zahlen a und b sollten nach Möglichkeit so groß sein, daß $a \cdot j + b$ für jedes $j \neq \lambda$ sehr viel größer als n ist. Die besten empirischen Erfahrungen hat man allgemein mit paarweise verschiedenen Primzahlen a, b und n gemacht.

- *Multiplikationsmethode*: Man wählt eine feste Zahl $z \in (0,1)$ und definiert $f_1(j) := \lfloor n \cdot ((j \cdot z) \bmod 1) \rfloor$.[26]

  Es gibt sowohl mathematische als auch empirische Indizien dafür, daß der *Goldene Schnitt*

[25] **Wichtige allgemeine Warnung:** Es ist ein weit verbreiteter Irrglaube, daß eine solche Zufallsverteilung von m Elementen auf n Indizes dafür sorgt, daß jedem $i \in \{0, \ldots, n-1\}$ mit großer Wahrscheinlichkeit ziemlich genau m/n Elemente zugeordnet werden. Für die zufällige Anzahl $a(i) \in \{0, \ldots, m\}$ der Elemente, die einem einzelnen, festen Index $i \in \{0, \ldots, n-1\}$ zugeordnet werden, nähert sich $a(i)/m$ zwar für wachsendes m mit Wahrscheinlichkeit 1 immer mehr $1/n$ an (nach dem sogenannten *Gesetz der großen Zahlen*). Aber $a(i)$ liegt dennoch **nicht** notwendig nahe bei m/n, sondern $|a(i) - m/n|$ wächst sogar mit Wahrscheinlichkeit 1 auf die Dauer ins Unendliche. Siehe Vorlesungen über Stochastik (Wahrscheinlichkeitstheorie) für genauere Erklärungen.

[26] Für $c \in \mathbb{R}^+$ ist der *fraktionale Teil* von c definiert als $c - \lfloor c \rfloor \in [0,1)$ und üblicherweise abkürzend notiert in der Form $c - \lfloor c \rfloor =: c \bmod 1$.

$$\frac{\sqrt{5}-1}{2} \approx 0.618$$

generell eine gute Wahl für z ist.

Im Rest von Abschnitt 3.3 betrachten wir die Definition von f_i mit $i > 1$. Die intuitive Zielsetzung dahinter ist: Für $i_1, i_2 \in \mathbb{Z}^+$ und $j_1, j_2 \in T_1$ mit $f_{i_1}(j_1) = f_{i_1}(j_2)$ und $f_{i_2}(j_1) = f_{i_2}(j_2)$ sollten i_1 und i_2 möglichst weit auseinander liegen, so daß Kollisionen von j_1 mit demselben Wert j_2 möglichst selten auftreten.

Beispiele:

- *Lineares Sondieren*: Es wird definiert: $f_i(j) := (f_1(j) + (i-1)) \bmod n$ für $i > 1$ und $j \in T_1$.
- *Double Hashing*: Es wird eine weitere Hash–Funktion $g : T_1 \to \{1, \ldots, n\}$ analog zu f_1 gewählt und definiert: $f_i(j) := (f_1(j) + (i-1) \cdot g(j)) \bmod n$ für $i \geq 1$ und $j \in T_1$.

Lineares Sondieren ist ein sehr einfaches Verfahren, im Lichte der obigen intuitiven Zielsetzung aber als ausgesprochen schlecht zu bewerten, denn gilt erst einmal $f_{i_1}(j_1) = f_{i_1}(j_2)$ für ein $i_1 \in \mathbb{Z}^+$ und $j_1, j_2 \in T_1$, so gilt auch $f_{i_2}(j_1) = f_{i_2}(j_2)$ für alle $i_2 > i_1$.

Es ist daher nicht überraschend, daß auch die empirischen Erfahrungen mit linearem Sondieren vergleichsweise schlecht sind, weil man oft beobachten kann, daß sich lange Intervalle von belegten Indizes bilden. Wann immer ein $f_i(j)$ in einem solchen Intervall liegt, müssen nämlich beim Einfügen von j erst alle nachfolgenden Indizes dieses Intervalls durchlaufen werden, bevor der erste freie Index gefunden ist. Dadurch wächst das Intervall natürlich noch weiter. Von einer Streuung der Werte kann also keine Rede sein!

Lineares Sondieren hat allerdings den Vorteil, daß offensichtlich für jeden Schlüssel $j \in T_1$ immer alle Indizes in $\{0, \ldots, n-1\}$ erreicht werden, das heißt, solange $m < n$ ist, kann jeder neue Schlüssel eingefügt werden. Das ist beim Double Hashing nicht mehr unbedingt der Fall. Bei der Wahl der Funktion g muß also darauf geachtet werden, daß für alle $j \in \mathbb{Z}^+$ gilt: $\{f_i(j) : i \in \mathbb{Z}^+\} = \{0, \ldots, n-1\}$, bzw. dazu äquivalent: $\{f_i(j) : i \in \{1, \ldots, n\}\} = \{0, \ldots, n-1\}$.

Beobachtung 3.7 *Für $j \in T_1$ und eine Folge $F = (f_1, f_2, \ldots)$, die durch $f, g : T_1 \to \mathbb{Z}_0^+$ und*

$$f_i := (f + (i-1) \cdot g) \bmod n$$

für $i \in \mathbb{Z}^+$ definiert ist, gilt

$$\{f_i(j) : i \in \mathbb{Z}^+\} = \{0, \ldots, n-1\}$$

genau dann, wenn $\operatorname{ggt}(g(j), n) = 1$ *ist.*

Beispiel: n ist eine Potenz von 2, und $g(j)$ ist für jedes $j \in T_1$ ungerade. □

Bei gut gewählten Funktionen f und g kommt Double Hashing dem Ideal des *uniformen Hashings* schon recht nahe. Uniformes Hashing ist ein idealisiertes Modell für Hashing und geht von zufällig definierten Funktionen f_1, f_2, $f_3 \ldots$ aus. Die empirische Einsicht dahinter ist, daß eine Folge F von Hash–Funktionen eine um so kleinere durchschnittliche Anzahl von notwendigen Suchschritten bei der Suche nach einem Element ergibt, je „zufälliger“ die Werte $f_i(j)$ aussehen.[27] Naturgemäß ist eine rein zufällige Wahl der $f_i(j)$ ein Ideal, an das kein deterministisches Hash–Verfahren wirklich heranreichen kann.

[27] In Anlehnung an die Thermodynamik führt man in der *Informationstheorie*, einem Zweig der Theoretischen Informatik, den Begriff *Entropie* als eine Art Maß für die Zufälligkeit einer Folge von Werten ein.

Für $i \in \mathbb{Z}^+$, $j \in T_1$ und $\ell \in \{0, \ldots, n-1\}$ sei $P(i,j,\ell)$ die Wahrscheinlichkeit (engl. *probability*), daß $f_i(j) = \ell$ ist. Dann bedeutet uniformes Hashing, daß die folgende Bedingung erfüllt ist: $P(i,j,\ell)$

$$= \begin{cases} 0, & \text{falls } i > n; \\ 0, & \text{falls } \ell \in \{f_1(j), \ldots, f_{i-1}(j)\}; \\ \frac{1}{n-i+1} & \text{sonst.} \end{cases}$$

Mit einer *erfolgreichen* Suche ist im folgenden gemeint, daß nach einem Element gesucht wird, das tatsächlich im Hash enthalten ist. Entsprechend heißt eine Suche nach einem nichtenthaltenen Element erfolg*los*.

Für $i \in \mathbb{Z}^+$ seien p_i und q_i die Wahrscheinlichkeiten, daß man bei einer erfolgreichen bzw. erfolglosen Suche genau i Suchschritte braucht. Die durchschnittliche Anzahl benötigter Suchschritte für eine erfolgreiche oder erfolglose Suche (der sogenannte *Erwartungswert*) ist

$$\sum_{i=1}^{\infty} i \cdot p_i \quad \text{bzw.} \quad \sum_{i=1}^{\infty} i \cdot q_i \,.$$

Da $p_i = q_i = 0$ für $i > n$ ist, konvergieren beide Reihen trivialerweise. Seien p'_i und q'_i die Wahrscheinlichkeiten, daß man *mindestens* i Suchschritte für eine erfolgreiche bzw. erfolglose Suche benötigt, also

$$p'_i \;=\; \sum_{h=i}^{\infty} p_h \quad \text{bzw.} \quad q'_i \;=\; \sum_{h=i}^{\infty} q_h \,.$$

Hilfslemma 3.8 *Es gilt*

$$\sum_{i=1}^{\infty} i \cdot p_i \;=\; \sum_{i=1}^{\infty} p'_i \quad \textit{und} \quad \sum_{i=1}^{\infty} i \cdot q_i \;=\; \sum_{i=1}^{\infty} q'_i \,.$$

Beweis: Jedes p_i taucht insgesamt i–fach in der Reihe

$$\sum_{h=1}^{\infty} p'_h \;=\; \sum_{h=1}^{\infty} \sum_{\ell=h}^{\infty} p_\ell$$

auf. Da die Reihe nichtnegativ und konvergent ist, darf man umordnen (analog q_i). □

Grob gesprochen besagt der folgende Satz 3.9, daß bei jedem Hash–Verfahren, das sich „ungefähr“ wie uniformes Hashing verhält, im Durchschnitt eine konstante Anzahl Suchschritte notwendig ist, wenn das Verhältnis m/n konstant ist. Je kleiner m/n ist, um so geringer ist diese Durchschnittszahl (was man ja auch erwarten würde).

Satz 3.9 *Unter der Annahme, daß jeder der m Schlüsselwerte in der Hashtabelle mit gleicher Wahrscheinlichkeit gesucht wird, ist die* **durchschnittliche** *Anzahl der Suchschritte bei uniformem Hashing nach oben beschränkt durch*

$$\frac{1}{1 - \frac{m}{n+1}}$$

bei erfolgreichen Suchen, und bei erfolgloser Suche ist die durchschnittliche Anzahl Suchschritte gleich

$$\frac{1}{1 - \frac{m+1}{n+1}} \,.$$

Beweis: Wir konzentrieren uns zuerst auf erfolgreiche Suche.

Natürlich gilt $p'_1 = 1$. Es sind genau dann mindestens zwei Suchschritte notwendig, wenn der erste Suchschritt auf einen Index gestoßen ist, der zwar durch ein Element belegt ist, aber dies nicht das gesuchte Element ist. Die Wahrscheinlichkeit hierfür ist

$$p'_2 \;=\; \frac{m-1}{n} \,.$$

Analog sind mindestens drei Suchschritte notwendig, wenn die ersten beiden Suchschritte zu „falsch belegten“ Indizes geführt haben. Da $f_1(j) \neq f_2(j)$ für $j \in T_1$ ist, ist beim zweiten Suchschritt ein Index von vornherein aus dem Spiel, und zwar einer der m belegten Indizes.

Daher ist die Wahrscheinlichkeit dafür, daß der zweite Suchschritt auf einen falsch belegten Index stößt, nicht mehr $(m-1)/n$ wie beim ersten, sondern $(m-2)/(n-1)$. Die Wahrscheinlichkeit, daß sowohl der erste als auch der zweite Suchschritt auf einen falsch belegten Index stößt, ist daher [28]

$$p'_3 \;=\; \frac{m-1}{n} \cdot \frac{m-2}{n-1}\,.$$

Diese Argumentation kann man fortsetzen und erhält für $i \in \mathbb{Z}^+$:

$$p'_i \;=\; \prod_{h=1}^{i-1} \frac{m-h}{n-h+1}\,.$$

Damit folgt leicht

$$p'_i \;\leq\; \left(\frac{m}{n+1}\right)^{i-1}$$

mit Hilfe von $m \leq n+1$. Weiter ergibt sich

$$\sum_{i=1}^{\infty} p'_i \;\leq \sum_{i=0}^{\infty} \left(\frac{m}{n+1}\right)^{i}$$

$$= \;\frac{1}{1-\frac{m}{n+1}}$$

mit der üblichen Darstellung der geometrischen Reihe in geschlossener Form.

Nun betrachten wir die erfolglose Suche. Natürlich gilt auch $q'_1 = p'_1 = 1$. Aber es gilt $p'_2 \neq q'_2$, denn nun sind *alle* m belegten Indizes durch Elemente ungleich dem gesuchten belegt, also $q'_2 = m/n$. Mit einer zum erfolgreichen Fall analogen Argumentation folgt

$$q'_i \;=\; \prod_{h=1}^{i-1} \frac{m-h+1}{n-h+1}$$

für $i \in \mathbb{Z}^+$. Der weitere Beweis für erfolglose Suche ist völlig analog zur erfolgreichen Suche. □

Bemerkungen:

- Aussagen über uniformes Hashing können nicht in einem mathematisch strengen Sinne auf Double Hashing oder andere Hash–Verfahren übertragen werden. Satz 3.9 erfüllt eher eine ähnliche Funktion wie beispielsweise Sätze in der Physik: Sie dienen dem menschlichen Verständnis der Vorgänge und der (allerdings unsicheren und ungenauen) Vorhersage empirischer Beobachtungen.

- Hash–Verfahren können auch dahingehend verallgemeinert werden, daß sie sogar *dynamische* Dictionaries realisieren, aber die Effizienz leidet massiv — so massiv, daß andere Realisierungen wie B–Bäume meist besser sein dürften.

- Man kann die Güte des Hash–Verfahrens empirisch verbessern, indem man nicht nur eine einzige Folge $F = (f_1, f_2, \ldots)$ definiert, sondern mehrere: $F_1, \ldots, F_r$. Methode *insert* benutzt F_1. Aber Methode *ready* des statischen Dictionaries wiederholt das Einfügen aller Elemente noch einmal probeweise mit $F_2, \ldots, F_r$

[28] **Allgemeine wichtige Warnung:** Im allgemeinen ist es nicht korrekt, die Wahrscheinlichkeiten für das Eintreten zweier Tatsachen miteinander zu multiplizieren, um die Wahrscheinlichkeit für das gemeinsame Eintreten beider Tatsachen zu berechnen. Das vorliegende Szenario ist sogar besonders problematisch, weil die Ereignisse, die mit den Wahrscheinlichkeiten $P(i,j,\ell)$, p_i, p'_i, q_i und q'_i belegt sind, krass voneinander abhängig sind. Nur die Wahrscheinlichkeiten von absolut unabhängig voneinander eintretenden Ereignissen dürfen ohne weitere Begründung multipliziert werden! Aus Gründen, die in Vorlesungen über Stochastik ausführlich thematisiert werden, ist es allerdings korrekt, die Wahrscheinlichkeiten dafür, daß der erste und der zweite Suchschritt auf falsch belegte Indizes treffen, zu multiplizieren, obwohl das zweite Ereignis natürlich davon abhängt, ob das erste Ereignis überhaupt eintritt.

und wählt dann die beste Folge F_h unter $F_1, \ldots, F_r$ zum endgültigen Aufbau der Hashtabelle. Zur Implementationsinvariante der Hashtabelle gehört dann auch die Nummer h der gewählten Folge F_h. Diese Variante von Hashing heißt *universelles Hashing*.[29]

- Die Restriktion $m \leq n$ kann umgangen werden, indem jede Komponente von H nicht mehr ein einzelnes Paar aus $T_1 \times T_2$ enthält, sondern eine beliebig große Menge solcher Paare (z.B. organisiert als eine Liste). Das heißt, um auf ein Element zuzugreifen, muß man diese Liste durchsuchen und erhält aus Satz 3.9 keine noch so vage Garantie für konstante durchschnittliche Zugriffszeit mehr.

  In dieser Variante braucht man überhaupt keine Funktionen f_2, $f_3, \ldots$: Jedes Paar $(t_1, t_2) \in T_1 \times T_2$ wird einfach in die Menge $H[f_1(t_1)]$ eingefügt. Diese Variante heißt *Chaining* (engl. Verkettung).

- Eine gut streuende Hash–Funktion für Strings (siehe obiges Beispiel) wäre beispielsweise durch eine Menge $\pi_0, \ldots, \pi_{r-1}$ von möglichst großen, paarweise verschiedenen Primzahlen definiert: Sei S ein String, das heißt, ein Vektor von Zeichen mit Indexbereich $[1, \ldots, n(S)]$. Sei a_i die ASCII–Nummer des i–ten Zeichens. Dann definieren wir

$$f_1(S) := \left(\sum_{i=1}^{n(S)} \pi_{(i \bmod r)} \cdot a_i \right) \bmod n\,.$$

  Allerdings ergibt sich bei großen Primzahlen und langen Strings schnell ein arithmetischer Überlauf. Das kann man ohne Verschlechterung der Streueigenschaft vermeiden, indem man jeweils nach einer gewissen Zahl von Additionen das momentane Zwischenergebnis durch Modulo–Bildung mit einer weiteren Primzahl wieder auf eine „ungefährliche“ Größenordnung reduziert.

[29] Die Terminologie in der Literatur ist hier nicht ganz einheitlich.

4 Algorithmen und Datenstrukturen

Während in Abschnitt 2 nur Algorithmen für sich und in Abschnitt 3 nur Datenstrukturen für sich betrachtet wurden, sollen hier wie in Abschnitt 1 zwei weitere Fallbeispiele für das Zusammenspiel von Algorithmen mit Datenstrukturen betrachtet werden.

4.1 Graphensuche mit Hash

Gegeben ist ein Graph $G = (V, A)$ und ein *Startknoten* $s \in V$. Es sollen alle Knoten $v \in V$ „besucht“ werden, für die es einen (s, v)–Pfad in G gibt.

Anwendungsbeispiel (stark vereinfacht!): Mit sogenannten *Internet–Suchmaschinen*,[30] kann man sich alle WWW–Seiten auflisten lassen, die eine bestimmte Kombination von Wörtern (oder Teilen von Wörtern) enthalten. Eine solche Suchmaschine basiert auf einem statischen Dictionary (beispielsweise ein Hash), in dem diese einzelnen Wörter die Schlüssel sind. Die Information zu einem Wort ist die Menge der WWW–Seiten, die dieses Wort enthalten.

Dieses Dictionary wird regelmäßig durch einen Algorithmus aktualisiert, der von einer zentralen Startseite (etwa eine weltweite Liste aller zentralen WWW–Knotenpunkte)

[30] Über die URL http://www.informatik.uni–konstanz.de/Allgemeines#suche kann man auf diverse Suchmaschinen zugreifen.

ausgeht und alle WWW–Seiten durchsucht, die von dieser Seite aus direkt oder indirekt über Querverweise erreichbar sind.

Das WWW kann als ein Graph $G = (V, A)$ aufgefaßt werden, dessen Knoten die einzelnen WWW–Seiten sind. Eine mögliche Kante (v, w) gehört genau dann zu A, wenn es mindestens einen Verweis von der WWW–Seite v zur WWW–Seite w gibt. □

Der folgende Algorithmus, die sogenannte *Breitensuche* (*breadth–first search*), ist ein allgemeines Verfahren zum Durchlauf aller Knoten im Graphen von einem gegebenen Startknoten s aus. Hier wird der Hash H allerdings nicht als ein statisches Dictionary benutzt, weil Einfüge- und Suchoperationen in beliebiger Reihenfolge aufeinanderfolgen (und somit die Unterscheidung zwischen Modus „im Aufbau“ und Modus „aufgebaut“ wegfällt). Technisch ist das natürlich kein Problem; nur die Laufzeitanalyse aus Satz 3.9 wird komplizierter.[31]

In der Breitensuche interessiert uns die Information aus T_2, die zu einem Schlüsselwert aus T_1 gemäß Definition in Abschnitt 3.1 mitgespeichert wird, nicht: Ob ein Schlüssel im Hash gespeichert ist oder nicht, ist die allein interessierende Information. Das wird im folgenden Pseudocode mit einem „$\cdot$“ für den Typ T_2 und Objekte dieses Typs angedeutet.

Breadth–First Search:

1. Richte eine $Queue<T_1>$ Q ein.
2. Q.construct().
3. Q.push(s).
4. Richte einen $Hash<T_1, \cdot>$ H ein.
5. H.construct().
6. H.insert($s, \cdot$).
7. Solange Q.size() > 0, führe aus:
   (a) $v := Q$.top().
   (b) Q.pop().
   (c) Für alle $(v, w) \in A$, für die H.included(w) falsch ist, führe aus:
      i. H.insert($w, \cdot$).
      ii. Q.insert(w).

Offensichtlich ist der Aufwand der Breitensuche linear, genauer: $\mathcal{O}(|A|)$.

Bemerkung: Breitensuche wird in Lehrbüchern üblicherweise nicht auf der Basis eines Hashes eingeführt, sondern statt dessen wird jedem Knoten eine Markierung angehängt, das heißt, eine Variable mit Wertebereich $\{wahr, falsch\}$. Wert *wahr* bedeutet dann, daß der Knoten schon besucht wurde. Insbesondere werden alle diese Markierungen vor Beginn des Algorithmus mit *falsch* initialisiert.

Eine solche Implementation von Breitensuche setzt natürlich voraus, daß alle Knoten des Graphen als Objekte vorliegen, und zwar schon vor Beginn des Algorithmus. Bei vielen Anwendungen von Breitensuche ist das auch der Fall, und dann ist eine Implementation mit Markierungen sicherlich wesentlich

[31] Dies ist ein Beispiel für die erste Bemerkung auf Seite 23: Die abstrakten Definitionen in Abschnitt 3.1 dienen weniger als Vorlage für konkrete Implementationen, sondern dem Verständnis der Grundidee hinter der jeweiligen Datenstruktur. Es wäre beispielsweise aus praktischer Sicht nicht sinnvoll, einen Hash so als Java–Klasse zu realisieren, daß alle Einfügeoperationen allen Suchoperationen vorangehen müssen, denn dann ist eine solche Klasse in Situationen wie der in Abschnitt 4.1 beschriebenen überhaupt nicht anwendbar. Das heißt, die *bereitgestellte* Funktionalität eines Hashes geht über die für ein statisches Dictionary *geforderte* Funktionalität hinaus. Universelles Hashing (Seite 32) ist hingegen ein Beispiel für eine Implementation statischer Dictionaries, bei der die Unterscheidung zwischen Phase „im Aufbau“ und Phase „aufgebaut“ durchaus eine Rolle spielt.

schneller in der Praxis als die Implementation mit einem Hash.

Im obigen Anwendungsbeispiel funktioniert diese Variante aber nicht, denn dazu müßte schon vor Beginn des Algorithmus das WWW einmal durchsucht werden, um eine Durchsuchung des WWW zu ermöglichen! □

4.2 Dynamische Programmierung

In diesem Abschnitt betrachten wir ein allgemeines algorithmisches Prinzip, das sich auf bestimmte algorithmische Problemstellungen anwenden läßt: die sogenannte *dynamische Programmierung.*[32] Das Ziel ist, ein virulentes Effizienzproblem bei rekursiven Algorithmen zu lösen: daß eine „naive" Implementation einer rekursiven Berechnungsvorschrift typischerweise immer wieder unnötigerweise dieselben Werte berechnet.

Beim dynamischen Programmierungsansatz wird die Rekursion so in eine Schleife umgewandelt, daß jeder rekursive Berechnungsschritt nur einmal ausgeführt wird. Jeder Berechnungsschritt wird erst dann ausgeführt, wenn alle rekursiven Schritte, deren Ergebnisse er benötigt, ausgeführt worden sind.

Alle diese Ergebnisse müssen in einer geeignet zu wählenden Datenstruktur solange zur Verfügung gehalten werden, wie sie für weitere Berechnungen benötigt werden, aber auch möglichst nicht länger als nötig, da sich sonst eine viel zu große Datenmenge ansammeln würde. Das ist der Grund, warum dynamische Programmierung ein interessantes Beispiel für Abschnitt 4 ist, der ja das Zusammenspiel zwischen Algorithmen und Datenstrukturen zum Thema hat.

Im folgenden verwenden wir die Notation aus Definition 2.1, und wir gehen davon aus, daß $\mathcal{I}$ endlich oder abzählbar unendlich ist.

Definition 4.1 *Ein algorithmisches Problem heißt* **parametrisiert** *mit $\ell \in \mathbb{Z}^+$ Parametern, wenn jeder Eingabe $x \in \mathcal{I}$ spezifische Parameterwerte $k_1^x, \ldots, k_\ell^x \in \mathbb{Z}_0^+$ zugeordnet sind. (Diese Zuordnung muß weder injektiv noch surjektiv sein.)*

Wir schreiben auch $\mathcal{K}(x) = (k_1^x, \ldots, k_\ell^x)$.

Zur formalen Behandlung erweitern wir Definition 1.1 zu *unendlichen* Graphen. Damit ist gemeint, daß Knoten- und Kantenzahl, aber auch die Anzahl der aus einem Knoten heraus- oder in einen Knoten hineinzeigenden Kanten abzählbar unendlich sein dürfen. Bei der Definition von Pfaden, Zykeln etc. bleibt auch weiterhin alles endlich. Ein Graph gemäß Definition 1.1 wird im folgenden ein *endlicher Graph* genannt.

Definition 4.2 *Ein* **Rekursionsgraph** *zu einem parametrisierten algorithmischen Problem ist ein endlicher oder unendlicher Graph $G = (V, A)$ mit folgenden Eigenschaften:*

1. *G ist* **azyklisch**, *d.h. enthält keine Zykel.*
2. *Es gibt genau einen Knoten $s \in V$, in den keine Kante hineinzeigt (d.h. es gibt kein $v \in V$ mit $(v, s) \in A$).*
3. *Für jeden Knoten $v \in V$ gibt es mindestens einen, aber insgesamt nur endlich viele (s, v)–Pfade in G.*
4. *Zusätzlich ist eine bijektive Abbildung $\varrho : V \to \{\mathcal{K}(x) : x \in \mathcal{I}\}$ definiert.*[33]

Für einen Rekursionsgraph $G = (V, A)$ und $w \in V$ ist $G_w = (V_w, A_w)$ der endliche Graph,

[32] Der Name ist historisch entstanden, und das Wort „dynamisch" hat in unserem Kontext keine weitere Bedeutung.

[33] Beachte, daß dies im allgemeinen keine Bijektion $V \longleftrightarrow \mathcal{I}$ impliziert.

der aus der Vereinigung aller (s,w)–Pfade in G resultiert. Genauer gesagt sind V_w bzw. A_w die Menge aller Knoten bzw. Kanten, die zu mindestens einem (s,w)–Pfad gehören.

Definition 4.3 *Ein parametrisiertes algorithmisches Problem heißt* **rekursiv definiert** *mit Rekursionsgraph $G=(V,A)$, wenn folgendes gilt:*

1. *Für $w \in V$, $x \in \mathcal{I}$ mit $\mathcal{K}(x) = \varrho(w)$ und $v \in V_w$ ist x genau ein Element $r(x,v) \in \mathcal{I}$ mit $\mathcal{K}(r(x,v)) = \varrho(v)$ zugeordnet, wobei $r(x,w) = x$ ist.*
2. *Für u, v, $w \in V$ mit $u \in V_v \subseteq V_w$ und $x \in \mathcal{I}$ mit $\mathcal{K}(x) = \varrho(w)$ gilt $r(x,u) = r(r(x,v),u)$.*
3. *Es gibt eine* **rekursive Berechnungsvorschrift**, *das heißt, ein Algorithmus, der folgendes leistet: Für $w \in V$ seien $(v_1,w),\ldots,(v_k,w) \in A$ die nach w hineinzeigenden Kanten. Sind ein $x \in \mathcal{I}$ mit $\mathcal{K}(x) = \varrho(w)$ und je ein Element aus $f(r(x,v_1)),\ldots,f(r(x,v_k))$ gegeben, dann berechnet diese Vorschrift daraus ein Element von $f(x)$.*[34]

Definition 4.4 *Sei $G=(V,A)$ ein endlicher azyklischer Graph. Eine* **topologische Sortierung** *von G ist eine bijektive Abbildung $\sigma : V \to \{1,\ldots,|V|\}$ so daß für $(v,w) \in A$ gilt $\sigma(v) < \sigma(w)$.*

Bemerkung: Es ist nicht schwer zu beweisen, daß ein endlicher Graph genau dann eine topologische Sortierung besitzt, wenn er azyklisch ist, und daß eine solche topologische Sortierung in linearer Zeit gefunden werden kann (siehe Vorlesungen zu Algorithmen und Datenstrukturen/Graphenalgorithmen). □

[34] Im Fall $w = s$ wird das Element aus $f(x)$ also nur aus dem gegebenen x selbst berechnet.

Der dynamische Programmierungsansatz geht von einem Rekursionsgraphen $G=(V,A)$, einem Knoten $w \in V$, einem $x \in \mathcal{I}$ mit $\mathcal{K}(x) = \varrho(w)$ und einer topologischen Sortierung σ von G_w aus und „hangelt" sich an der topologischen Sortierung entlang, um ein Element von $f(x)$ zu berechnen.

Dynamischer Programmierungsansatz:

> Für $i := 1,\ldots,|V_w|$ berechne ein Element von $f(r(x,\sigma^{-1}(i)))$ nach der rekursiven Berechnungsvorschrift aus Definition 4.3(3) auf Basis der schon für $j = 1,\ldots,i-1$ berechneten Werte. □

Beispiel I (Binomialkoeffizient): Für $n \in \mathbb{Z}^+$ und $k \in \{0,\ldots,n\}$ ist der Binomialkoeffizient bekanntlich definiert durch

$$\binom{n}{k} := \frac{n!}{k!\cdot(n-k)!}\ .$$

Diese Formel taugt nicht zur Berechnung von $\binom{n}{k}$, weil $n!$ schon für kleine Zahlen n zu einem Überlauf führt. Eine andere Idee wäre, die allgemein bekannte rekursive Beziehung für $1 \le k < n$ zur Grundlage einer Implementation zu machen:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}\ .$$

Eine Umsetzung dieser Formel in einen rekursiven Algorithmus führt allerdings dazu, daß $\binom{m}{\ell}$ für sehr kleine Werte m und ℓ exponentiell häufig in n ausgewertet wird.

$\mathcal{I}$ ist die Menge aller Paare (n,k), und das ergibt schon auf natürliche Weise eine Parametrisierung mit $\ell = 2$, nämlich einfach die Identität auf $\mathcal{I}$. (Insbesondere ist die Definition von $r(x,v)$ für $x \in \mathcal{I}$ mit $\mathcal{K}(x) = \varrho(w)$ und $v \in V_w$ eindeutig.)

Der Rekursionsgraph enthält eine Kante (v, w) für jedes Paar der Form $\varrho(v) = (n, k)$ und $\varrho(w) \in \{(n+1, k), (n+1, k+1)\}$. Für $\varrho(w) = (n, k)$ ist $\varrho(V_w)$ also die Menge aller (m, ℓ) mit $0 \leq m \leq n$, $0 \leq \ell \leq k$ und $k - \ell \leq n - m$.[35]

Eine geeignete topologische Sortierung σ von G_w mit $\varrho(w) = \binom{n}{k}$ wäre etwa gegeben durch die Reihenfolge

$$\begin{aligned}
&\binom{0}{0} \to \binom{1}{0} \to \cdots \to \binom{n-k}{0} \\
\to &\binom{1}{1} \to \binom{2}{1} \to \cdots \to \binom{n-k+1}{1} \\
&\qquad\qquad \cdots \\
\to &\binom{k}{k} \to \binom{k+1}{k} \to \cdots \to \binom{n}{k}.
\end{aligned}$$

Es bietet sich bei dieser Wahl von σ an, alle Werte sukzessive in einem Vektor A mit Indexbereich $[0, \ldots, n-k]$ im Rahmen zweier ineinandergeschachtelter Schleifen zu berechnen. Nach dem i–ten Durchlauf durch die äußere Schleife soll $A[\ell] = \binom{\ell+i}{i}$ sein (bzw. $A[\ell] = \binom{\ell}{0}$ vor dem allerersten Durchlauf). Die Initialisierung ist trivial, da $\binom{\ell}{0} = 1$ für alle ℓ ist. Im i–ten Durchlauf werden die Werte $A[1], \ldots, A[n-k]$ *in dieser Reihenfolge* neu berechnet ($A[0] = 1$ bleibt natürlich unverändert). Und zwar ergibt sich der neue Wert von $A[\ell]$ gemäß obiger Rekursionsformel für Binomialkoeffizienten als Summe aus dem neuen Wert von $A[\ell-1]$ und dem alten Wert von $A[\ell]$.

Die Laufzeit ist in $\mathcal{O}(k \cdot (n-k)) \subseteq \mathcal{O}(kn)$.[36]

Beispiel II (optimaler statischer Suchbaum): Wenn man ein statisches Dictionary gemäß Abschnitt 3.1 durch einen binären Suchbaum realisieren will und relative Zugriffshäufigkeiten (also Zugriffswahrscheinlichkeiten) für die einzelnen Elemente bekannt sind, kann man den Suchbaum anhand dieser Wahrscheinlichkeiten optimal konstruieren.

Das heißt, für jeden der Werte $t_1, \ldots, t_n \in T_1$, die (jeweils mit einer Information aus T_2) zu speichern sind, gibt es eine Zugriffswahrscheinlichkeit $p_i \geq 0$, inbesondere gilt also $p_1 + \cdots + p_n = 1$. Die durchschnittliche Anzahl von Zugriffsschritten (also der *Erwartungswert* für die Anzahl Schritte) soll minimiert werden: Wenn h_i die Höhe von Element t_i im Baum ist, dann ist also $p_1 \cdot h_1 + \cdots + p_n \cdot h_n$ der zu minimierende Wert. Für diese Zielsetzung reicht es aus, den Vektor $(p_1, \ldots, p_n)$ zu betrachten, und die eigentlichen Werte $t_1, \ldots, t_n$ kann man ignorieren.[37]

[35] Zur Veranschaulichung: Im *Pascalschen Dreieck*, das auch als eine Visualisierung des Rekursionsgraphen aufgefaßt werden kann, sind das gerade die Einträge $\binom{m}{\ell}$, die von dem auf der Spitze stehenden Rechteck mit Ecken $\binom{0}{0}$, $\binom{n-k}{0}$, $\binom{n}{k}$ und $\binom{k}{k}$ eingeschlossen werden.

[36] Speziell bei diesem Beispiel gibt es eine bessere Lösung:

$$\binom{n}{k} = \prod_{i=1}^{k} \frac{n-i+1}{i}.$$

Es ist leicht zu sehen, daß

$$(n-\ell+1) \cdot \prod_{i=1}^{\ell-1} \frac{n-i+1}{i}$$

für jedes $\ell \in \{1, \ldots, n\}$ durch ℓ teilbar ist, so daß man diese Vorschrift mit ganzzahliger Arithmetik (d.h. ohne numerische Fehler) in einen Algorithmus umsetzen kann, der sogar linear in k ist. Leider läßt sich diese Idee nicht verallgemeinern, wohingegen dynamische Programmierung ein sehr allgemeiner Ansatz ist.

[37] In Lehrbüchern wird das Problem üblicherweise in etwas komplizierterer Form betrachtet: Zusätzlich werden Wahrscheinlichkeitswerte $q_0, \ldots, q_n$ eingeführt, so daß q_i für $i \in \{1, \ldots, n-1\}$ die Wahrscheinlichkeit bedeutet, daß ein Wert zwischen t_i und t_{i+1} erfolglos gesucht wird (bzw. ein Wert kleiner als t_1 für q_0 und größer als t_n für q_n). Alles folgende läßt sich auf diesen Fall sinngemäß übertragen, wird aber komplizierter.

Um eine Rekursion zu etablieren, betrachten wir dieses Minimierungsproblem für jedes $i \in \{1, \ldots, n\}$ unter der Zusatzbedingung, daß t_i in der Wurzel gespeichert ist. Unter dieser Zusatzbedingung läßt sich das Problembeispiel $(p_1, \ldots, p_n)$ lösen, indem die Problembeispiele $(p_1, \ldots, p_{i-1})$ und $(p_{i+1}, \ldots, p_n)$ gelöst und die beiden Lösungen als Teilbäume an die Wurzel t_i gehängt werden. Die Rekursion berechnet also für jedes $i \in \{1, \ldots, n\}$ eine optimale Möglichkeit unter dieser Zusatzbedingung und wählt aus diesen n Vorschlägen die beste aus. Eine naive rekursive Implementation durchläuft die alleruntersten Rekursionsschritte wieder exponentiell häufig in n.

Wir verallgemeinern das Problem zunächst rein formal, indem wir den Vektor $(p_1, \ldots, p_n)$ nicht unbedingt mit 1 beginnend numerieren, sondern mit beliebigen Indexbereichen $[m, n]$, $0 < m \leq n$. Zusätzlich definieren wir pro forma den leeren Indexbereich $[m, n] = [1, 0]$. Dann parametrisieren wir das Problem gemäß Definition 4.1 mit den Parametern m und n, wobei $\varrho(s) := (1, 0)$. $V \setminus \{s\}$ enthält also einen Knoten v mit $\varrho(v) = (m, n)$ für jedes Paar $(m, n) \in \mathbb{Z}^+ \times \mathbb{Z}^+$ mit $m \leq n$.

Um obige Rekursionsvorschrift in einen Rekursionsgraphen $G = (V, A)$ umzusetzen, setzen wir fest, daß die Kante (v, w) genau dann existiert, wenn für $(m_1, n_1) := \varrho(v)$ und $(m_2, n_2) := \varrho(w)$ gilt:

1. $v = s$ und $m_2 = n_2$ oder
2. $v \neq s$ und zusätzlich
   (a) entweder $m_2 = m_1$ und $n_1 < n_2$
   (b) oder $m_2 < m_1$ und $n_1 = n_2$.

Für $\varrho(w) = (m, n)$ ist $\varrho(V_w \setminus \{s\})$ daher die Menge aller (k, ℓ) mit $m \leq k \leq \ell \leq n$. Für $x = (p_m, \ldots, p_n)$ und $v = \varrho^{-1}(k, \ell)$ ist also $r(x, v) = (p_k, \ldots, p_\ell)$.

Offensichtlich ist jede Abzählung der Elemente von V_w, in der $n - m$ monoton wächst, eine topologische Sortierung von G_w.

Die Implementation des dynamischen Programmierungsansatzes ist am einfachsten, wenn man eine $(n \times n)$–Matrix M anlegt und in jeder Komponente $M[k][\ell]$ mit $k \leq \ell$ zwei Informationen berechnet und speichert: die optimale durchschnittliche Anzahl $M[k][\ell].x$ von Zugriffsschritten für einen Baum auf $(p_k, \ldots, p_\ell)$ sowie ein Wert $M[k][\ell].y \in \{k, \ldots, \ell\}$, der für mindestens einen optimalen Baum auf $(p_k, \ldots, p_\ell)$ die Wurzel ist. Die Berechnung des Inhalts einer Komponente $M[k][\ell]$ erfordert nur den Zugriff auf $\mathcal{O}(n)$ Elemente von M, und somit ergibt sich $\mathcal{O}(n^3)$ Gesamtaufwand.[38]

Den optimalen Baum kann man nun rückwärts konstruieren: Wenn $M[1][n].y = i$ ist, dann ist i die Wurzel des Baumes, $M[1][i-1].y$ und $M[i+1][n].y$ sind die beiden unmittelbaren Nachfolger usw. □

Beispiel III (Vergleich von DNA–Sequenzen): Bekanntlich tragen DNA–Sequenzen das Erbgut aller Lebewesen. Abstrakt gesehen ist eine DNA–Sequenz ein sehr langer String von vier verschiedenen Zeichen: A, D, T und Z.[39]

Man ist daran interessiert festzustellen, wie unterschiedlich zwei gegebene Sequenzen sind. Diese Frage ist wichtig, weil die Bestimmung von DNA–Sequenzen immer fehlerhaft ist, das heißt, wenn zwei auf verschiedene Weisen bestimmte DNA–Sequenzen „fast“ identisch sind, muß man davon ausgehen, daß sie identisch sind und daß die Unterschiede von

[38] Mit einer weiteren mathematischen Einsicht, die hier nicht betrachtet werden soll, kann der Aufwand sogar auf $\mathcal{O}(n^2)$ reduziert werden.

[39] Das sind die Anfangsbuchstaben der vier chemischen Substanzen, aus denen die Kodierung des Erbguts gebildet wird.

Fehlern bei der Bestimmung herrühren.

Als Maß für die Unterschiedlichkeit von zwei Sequenzen betrachtet man die minimale Anzahl von Fehlern beim Versuch, die beiden Sequenzen „zur Deckung“ zu bringen. Wenn diese Zahl einen gewissen Grenzwert (der von der Länge der Strings abhängt und aufgrund empirischer Erfahrungen definiert wird) nicht überschreitet, sieht man die beiden DNA–Sequenzen als identisch an.

Seien X und Y zwei Strings mit Länge $m \in \mathbb{Z}^+$ bzw. $n \in \mathbb{Z}^+$. Gesucht ist eine *Deckungsrelation*, das heißt, eine Relation

$$R \subseteq \{1, \ldots, m\} \times \{1, \ldots, n\}$$

mit folgenden Eigenschaften:

- Für zwei verschiedene Elemente (i_1, j_1), $(i_2, j_2) \in R$ gilt *sowohl* $i_1 \neq i_2$ *als auch* $j_1 \neq j_2$.
- Für (i_1, j_1), $(i_2, j_2) \in R$ mit $i_1 < i_2$ gilt $j_1 < j_2$.
- Für $(i, j) \in R$ gilt $X[i] = Y[j]$.

Unter diesen Bedingungen soll die Deckungsrelation R so bestimmt werden, daß $|R|$ maximal ist (d.h. maximal auffindbare Übereinstimmung zwischen X und Y). Dieser Maximalwert wird im weiteren mit $C(X, Y)$ bezeichnet.

Zur Etablierung der Rekursion seien $k \in \{0, \ldots, m\}$ und $\ell \in \{0, \ldots, n\}$. Weiter bezeichnen X_k und Y_ℓ die Teilstrings $(X[1], \ldots, X[k])$ und $(Y[1], \ldots, Y[\ell])$.[40] Es gibt vier Fälle zu betrachten:

1. Falls es eine Optimallösung $R_{k,\ell}$ für (X_k, Y_ℓ) gibt mit $(k, \ell) \in R_{k,\ell}$, so ist $C(X_k, Y_\ell) = C(X_{k-1}, Y_{\ell-1}) + 1$.
2. Falls es eine Optimallösung $R_{k,\ell}$ für (X_k, Y_ℓ) gibt mit $k \neq i$ und $\ell \neq j$ für alle $(i, j) \in R_{k,\ell}$, so gilt $C(X_k, Y_\ell) = C(X_{k-1}, Y_{\ell-1})$.
3. Falls es eine Optimallösung $R_{k,\ell}$ mit $(k, j) \in R_{k,\ell}$ und $j \in \{1, \ldots, \ell - 1\}$ gibt, so gilt $C(X_k, Y_\ell) = C(X_k, Y_{\ell-1})$.
4. Falls es eine Optimallösung $R_{k,\ell}$ mit $(i, \ell) \in R_{k,\ell}$ und $i \in \{1, \ldots, k - 1\}$ gibt, so gilt $C(X_k, Y_\ell) = C(X_{k-1}, Y_\ell)$.

Man beachte, daß kein logischer Widerspruch zwischen diesen vier Fällen auftreten kann. Wenn mehr als ein Fall auf ein Paar (X_k, Y_ℓ) zutrifft, kommen alle Fälle übereinstimmend zum selben Ergebnis. (Der offenkundige „Widerspruch“ zwischen den ersten beiden Fällen kann nicht eintreten, da beide Fälle für kein (X_k, Y_ℓ) zugleich zutreffen können.)

Insgesamt ist (k, ℓ) eine geeignete Parametrisierung des Problems gemäß Definition 4.1. Der Rekursionsgraph hat genau dann eine Kante (v, w), wenn für $\varrho(v) = (k, \ell)$ und $\varrho(w) = (m, n) \neq (k, \ell)$ gilt: $m - 1 \leq k \leq m$ und $n - 1 \leq \ell \leq n$. Insbesondere ist $\varrho(V_w)$ die Menge aller (k, ℓ) mit $k \leq m$ und $\ell \leq n$.

Zum Beispiel ist jede „diagonale“ Abzählung von V_w eine topologische Sortierung, das heißt, zuerst alle Paare (k, ℓ) mit $k + \ell = 0$, dann mit $k + \ell = 1$, $k + \ell = 2$ usw.

Eine Möglichkeit, den dynamischen Programmierungsansatz für diese Problemstellung effizient zu implementieren, besteht in einer Schleife über $h = 2, \ldots, m + n$, in der im h–ten Durchlauf eine optimale Lösung für jedes Paar (X_k, Y_ℓ) mit $k + \ell = h$ in einem Vektor A der Größe $m+n$ berechnet wird. Dazu richtet man zwei weitere Vektoren B und C der Größe $m + n$ ein. Nach dem h–ten Durchlauf enthält B bzw. C Optimallösungen für alle Paare (k, ℓ) mit $k+\ell = h-1$ bzw. $k+\ell = h-2$. In jedem Durchlauf wird zuerst b nach C und

[40] Für $i = 0$ bzw. $j = 0$ ist das jeweils wieder der leere String ().

dann A nach B kopiert und danach mit Hilfe der Werte in B und C das Ergebnis dieses Durchlaufs berechnet und in A abgelegt.

Die Gesamtlaufzeit ist in $\mathcal{O}((m+n)^2)$ und der Speicherplatzverbrauch in $\mathcal{O}(m+n)$. □

Anhang: Arbeitsblatt zu Korrektheitsbeweisen

Dieses Arbeitsblatt wurde zum letzten der beiden Vorlesungstermine, in denen Abschnitt 2.2 behandelt wurde, ausgeteilt, und dieser Vorlesungstermin hat sich im wesentlichen damit befaßt, dieses konkrete Beispiel eines Algorithmus Schritt für Schritt zu studieren und anhand dessen die abstrakten Inhalte von Abschnitt 2.2 zu erläutern.

Der Algorithmus `mark_prime_numbers` bestimmt für ein Intervall $[m,n]$ positiver ganzer Zahlen, welche Zahlen in diesem Intervall prim sind, und speichert das Ergebnis im Vektor `result` ab. Um die asymptotische Laufzeit zu drücken, ist der Algorithmus etwas komplexer als der naive Algorithmus, vor allem weil er zum Test, ob eine Zahl k prim ist, im wesentlichen nichts anderes tut als die ungeraden Zahlen zwischen 3 und $\sqrt{k}$ daraufhin zu überprüfen, ob sie k teilen.

Hinter der unschuldig anmutenden Phrase „im wesentlichen“ stehen einige notwendige Spezialfallbehandlungen, bei denen man leicht Fehler machen kann. Dieser Algorithmus ist also nicht nur als ein Beispiel zu verstehen, wie Korrektheitsbeweise funktionieren, sondern auch dafür, daß Korrektheitsbeweise eine sinnvolle Sache sind.

Der Algorithmus ist in der Syntax der Programmiersprache Java formuliert. Es wird allerdings kein Anspruch auf hundertprozentige Konformität mit dem voraussichtlichen ANSI/ISO-Standard für Java oder irgendeinem Java-Dialekt erhoben.

Algorithmus:

```
void mark_prime_numbers
       (int m, int n, boolean result[])
  {
    // Unteralgorithmus 1:
    int k = m;
    // Unteralgorithmus 2:
    if ( k<=2 && n>=2 )
     {
       result[2-m] = true;
       k = 3;
     }
    // Unteralgorithmus 3:
    for ( k=k|1; k<=n; k+=2 )
     {
       // Unteralgorithmus 3(a):
       double upper_bound
          = Math.sqrt(k) + 1;
       // Unteralgorithmus 3(b):
       bool nonprime_proved
          = false;
       // Unteralgorithmus 3(c):
       int i = 3;
       // Unteralgorithmus 3(d):
       while ( i<=upper_bound
            && !nonprime_proved )
        {
          // Unteralgorithmus 3(d)i:
          if ( k%i == 0 )
             nonprime_proved = true;
          // Unteralgorithmus 3(d)ii:
          i += 2;
        }
       // Unteralgorithmus 3(e):
       result[k-m] = !nonprime_proved;
     }
  }
```

Analyse des Algorithmus gemäß Abschnitt 2.2:

Algorithmus:

- Vorbedingung: VB := ($0 < m$; $m \leq n$; alle Zahlen $1, \ldots, n$ sind im Datentyp `double` exakt darstellbar; `result` ist ein Array der Länge $n - m + 1$; alle Komponenten von `result` sind zu `false` initialisiert).
- Nachbedingung: NB := (für $i \in \{m, \ldots, n\}$ ist `result[`i–m`]` = `true` genau dann, wenn i eine Primzahl ist).
- Bemerkung: `mark_prime_numbers` funktioniert auch in manchen Situationen, in denen die Vorbedingung VB nicht voll erfüllt ist.

Unteralgorithmus 1:

- Vorbedingung: VB.
- Nachbedingung: VB sowie Zusatzbedingung NB':

  NB' := (für alle Zahlen $i \in \{m, \ldots, \min\{n, k-1\}\}$ ist `result[`$i-m$`]` = `true` genau dann, wenn i eine Primzahl ist).

Unteralgorithmus 2:

- Vorbedingung: VB; NB'.
- Nachbedingung: Vorbedingung sowie ($k \geq 3$ oder $n < 2$).[41]

Unteralgorithmus 3:

- Vorbedingung: Nachbedingung von Unteralgorithmus 2.
- Nachbedingung: NB'; $k > n$.

[41] Hier und im folgenden ist „oder“ immer inklusiv gemeint.

Unteralgorithmus 3(a):

- Vorbedingung: VB, NB'; $k \geq 3$; $k \leq n$; k ist ungerade.
- Nachbedingung: Vorbedingung; `upper_bound` $\geq \sqrt{k}$.

Unteralgorithmus 3(b):

- Vorbedingung: Nachbedingung von Unteralgorithmus 3(b).
- Nachbedingung: Vorbedingung; `nonprime_proved` = `false`.

Unteralgorithmus 3(c):

- Vorbedingung: Nachbedingung von Unteralgorithmus 3(c).
- Nachbedingung: Vorbedingung; $i = 3$.

Unteralgorithmus 3(d):

- Vorbedingung: Nachbedingung von Unteralgorithmus 3(d).
- Nachbedingung: Vorbedingung; `nonprime_proved` ist `true` genau dann, wenn k Primzahl ist.

Unteralgorithmus 3(d)i:

- Vorbedingung: Vorbedingung von Unteralgorithmus 3(d); $i \geq 3$; $i \leq$ `upper_bound`; $\{3, \ldots, i-2\}$ enthält keinen Teiler von k.
- Nachbedingung: Vorbedingung; `nonprime_proved` ist `true` genau dann, wenn i ein Teiler von k ist.

Unteralgorithmus 3(e):

- Vorbedingung: Nachbedingung von Unteralgorithmus 3(d).
- Nachbedingung: Vorbedingung; `result[`$k - m$`]` ist `true` genau dann, wenn k eine Primzahl ist.

Arbeitsfragen:

1. Ist garantiert, daß der Algorithmus weder abstürzt noch in eine Endlosschleife gerät?

2. Wie groß ist der asymptotische Aufwand?

3. Warum steht in der Vorbedingung des Algorithmus der Ausdruck $n-m+1$ und nicht einfach $n-m$?

4. Wird im Falle $m = 1$ korrekterweise m als nicht–prim markiert?

5. Warum kann man Unteralgorithmus 2 nicht einfach streichen?

6. Warum gehört *VB* zur Vor- und Nachbedingung der einzelnen Unteralgorithmen?

7. Warum gilt ($k \geq 3$ oder $n < 2$) nach Unteralgorithmus 2?

8. Wie geht die Vorbedingung ($k \geq 3$ oder $n < 2$) in die Korrektheit von Unteralgorithmus 3 ein?

9. In der Vorbedingung von Unteralgorithmus 3(a) wird verlangt, daß k ungerade ist.

   (a) Warum ist k unmittelbar vor jeder Abarbeitung von Unteralgorithmus 3(a) tatsächlich ungerade?

   (b) Wofür ist dies notwendig?

10. Warum ist *NB'* unmittelbar vor jeder Abarbeitung von Unteralgorithmus 3(a) erfüllt?

11. In der Vorbedingung des Algorithmus wird gefordert, daß alle Komponenten von `result` als `false` initialisiert sind. Wofür wird das gebraucht?

12. Warum steht $m, \ldots, \min\{n, k-1\}$ in *NB'*? Warum nicht einfach $m, \ldots, k-1$?

13. In der Vorbedingung des Algorithmus selbst wird gefordert, daß die Zahlen $\{1, \ldots, n\}$ im Typ `double` exakt darstellbar sind. Wo wird das benötigt?

    Zusatzfrage: Warum reicht es nicht zu fordern, daß n exakt darstellbar ist?