Rheinische Friedrich-Wilhelms-Universität Bonn
Mathematisch-Naturwissenschaftliche Fakultät

Algorithmen und Berechnungskomplexität I

Skript WS 2017

**Nach Aufzeichnungen von Rolf Klein,
Elmar Langetepe und Heiko Röglin**

Bonn, Oktober 2017

Inhaltsverzeichnis

Kapitel 1

Einführung

In dieser Vorlesung wird die Komplexität von Berechnungsproblemen untersucht und wir werden die dafür notwendigen theoretischen Konzepte entwickeln. Für viele Problemstellungen werden Algorithmen (Lösungspläne) vorgestellt und analysiert.

1.1 Algorithmen

Formal gesehen ist ein *Algorithmus* ein Lösungsplan, der für eine *Probleminstanz* eines definierten Berechnungsproblems die enstprechende *Ausgabe* liefert.

Beispiel: **Sortierproblem**

Eingabe: Eine Folge von n reellen Zahlen $(a_1, a_2, \ldots, a_n)$.

Ausgabe: Eine Permutation π, so dass $(a_{\pi(1)}, a_{\pi(2)}, \ldots a_{\pi(n)})$ eine sortierte Folge der Eingabe ergibt, d.h., $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

Konkrete Eingabefolgen werden *Instanzen* genannt.

Bezüglich der Algorithmen und der Problemstellung werden wir uns insgesamt zunächst mit drei Begriffen auseinandersetzen:

- Korrektheit: Wir wollen für die Algorithmen formal beweisen, dass zu jeder Instanz das richtige Ergebnis berechnet wird.
- Laufzeitabschätzung: Wie ist die Laufzeit des Algorithmus in Bezug auf die Eingabegröße n und wie effizient ist der Algorithmus im Vergleich zu anderen möglichen Algorithmen?
- Speicherplatzabschätzung: Wieviel Speicher in Bezug auf die Eingabegröße n benötigt der Algorithmus und wie effizient ist das im Vergleich zu anderen möglichen Algorithmen?

Die hier betrachteten Algorithmen sollen mithilfe eines Computers ihre Aufgabe erfüllen, deshalb werden wir uns an gängigen Programmiersprachen orientieren und dazu *Pseudocode* verwenden.

1.1.1 Anwendungsgebiete

Algorithmen spielen in verschiedenen Anwendungsgebieten eine große Rolle. Die Problemstellungen werden durch mathematische Modellierung so formuliert, dass sie durch einen Rechner

bearbeitet werden können.

- Human Genome Project: Datenanalyse, DNA-Vergleiche, Matchingalgorithmen, ...
- Internet: Routing, Datenhaltung, Suchmaschinen, Routenplaner, Verschlüsselung, ...
- Industrie: Beladen von Containern, Optimieren von Arbeitsabläufen, ...
- Infrastruktur: Optimieren von Netzen, Umwege minimieren, Kürzeste Wege berechnen/approximieren, ...
- ...

Ein Beispiel aus unserer aktuellen algorithmischen Forschung in Zusammenarbeit mit dem FKIE Wachtberg.

Wir betrachten ein Gebiet, in dem es einen Industrie-Unfall gegeben hat und das von einer Menge von Robotern beobachtet werden soll. Das Gebiet ist durch den Menschen nicht mehr betretbar. Die Roboter haben einen beschränkten Senderadius und sollen den Kontakt untereinander und zur Basisstation halten. Von der Basisstation aus sollen die Roboter zu vorher festgelegten Positionen gefahren werden. Die Positionen und die Wege der einzelnen Positionen zueinander konnten berechnet werden. Damit die Roboter den Kontakt untereinander nicht verlieren, ist es manchmal erforderlich, mehrere Roboter von Position A zu Position B zu bewegen. Für eine Bewegung von A nach B sind somit zum Beispiel mindestens 10 Roboter notwendig.

Die Frage lautet, wieviel Roboter brauchen wir, um von einer Basisstation alle gegebenen Positionen zu besetzen und bei der Bewegung zu gewährleisten, dass die Agenten alle in Kontakt bleiben?

Modellbildung durch einen Graphen $G = (V, E)$ mit ganzzahligen Knotengewichten w_v für $v \in V$ und ganzzahligen Kantengewichten w_e für $w \in E$, ein Startknoten v_s; siehe Abbildung 1.1.

Folgende Bedingungen:

- Eine Kante e darf nur mit $k \geq w_e$ Agenten traversiert werden.
- Ein Knoten v darf nur mit $k \geq w_v$ Agenten besucht werden.
- Beim ersten Besuch eines Knotens v bleiben w_v Agenten am Knoten zurück.

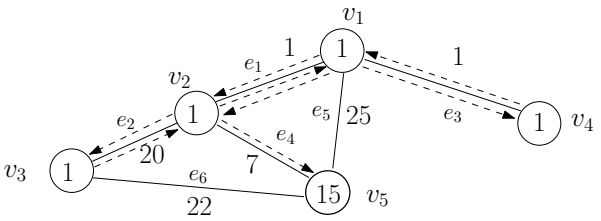


Abbildung 1.1: Falls die Agenten am Knoten v_1 starten, kann eine Tour mit einer minimalen Anzahl an Agenten wie folgt beschrieben werden. 23 Agenten in einer einzelnen Gruppe besuchen Knoten in der Reihenfolge v_1,v_2,v_3,v_4 and v_5. Die Kanten e_5 and e_6 werden nicht besucht.

Frage: Wieviele Agenten werden benötigt, um alle Knoten v mit w_v Agenten zu füllen? Welchen Weg können die Agenten dabei zurücklegen? Lässt sich ein effizienter Algorithmus formulieren, der dieses Problem für beliebige Instanzen löst?

Die Abbildung zeigt ein Beispiel: Vom festgelegten Basisknoten v_1 aus, kommt man mit 23 Agenten aus. Beginnend am Knoten v_1 bleibt zunächst ein Agent zurück und 22 Agenten bewegen sich über die Kante e_1 nach v_2. Hier wird ein Agent abgelegt und mit 21 Agenten geht es zum Knoten v_3 über die Kante e_2. Nachdem wir dort einen Agenten abgelegt haben, können wir mit den verbleibenden 20 Agenten gerade noch über die Kante zurück. Hier stellt sich quasi heraus, dass es mit weniger als 23 gar nicht gehen kann!

Mit den 20 Agenten besuchen wir v_4 (über e_1 und e_3) und lassen einen Agenten dort. Dann geht es mit den verbleibenden über e_3, e_1 nnd e_4 zu v_5. Hier werden 15 Agenten benötigt und wir sind fertig.

Es stellt sich heraus, dass dieses Problem für allgemeine Graphen wirklich *schwer* zu lösen ist. Für einfache Graphen (Bäume) gibt es effiziente Algorithmen, durch die die optimale Lösung in best-möglicher Laufzeit (auch durch die Verwendung effizienter Datenstrukturen (Heaps)) ermittelt werden kann. Daraus resultiert auch eine Approximationslösung für allgemeine Graphen. Aus dem Graphen wird ein Baum mit guter Verbindungseigenschaft generiert und darauf der Baumalgorithmus angewendet.

Ein formaler Beweis zeigt jeweils die Güte der Approximation, die Optimalität des Baumalgorithmus und die schwere des Problems im Allgemeinen. Dieses Beispiel enthält somit viele Aspekte, die in der Vorlesung vermittelt werden sollen.

1.1.2 Maschinenmodell

Die Laufzeitabschätzungen der Algorithmen soll unabhängig von der jeweils aktuellen Hardware-Rechnerausstattung (z.B. Taktfrequenz, Verarbeitungsbreite etc.) sein, deshalb zählen wir die Anzahl der ausgeführten *Elementaroperationen* in unserem jeweiligen Maschinenmodell. Der Einfachheit halber veranschlagen wir für jede Operation in etwa die gleichen Kosten.

Wir verwenden das Modell einer sogenannten **REAL RAM**, eine *Random Access Maschine*, die mit reellen Zahlen rechnet. In diesem Modell werden Anweisungen nacheinander (sequentiell) ausgeführt. Die REAL RAM Modell orientiert sich an realen Rechnen und hat folgende Spezifikation:

- Abzählbar unendlich viele Speicherzellen, die mit den natürlichen Zahlen adressiert werden.
- Jede Speicherzelle kann eine beliebige reelle oder natürliche Zahl enthalten. Direkter oder indirekter Zugriff ist möglich.
- Elementare Rechenoperationen: Addieren, Subtrahieren, Multiplizieren, Dividieren, Restbilden, Abrunden, Aufrunden.
- Elementare Relationen: kleiner gleich, größer gleich, gleich, $\vee$, $\wedge$.
- Kontrollierende Befehle: Verzweigung, Aufruf von Unterroutinen, Rückgabe.
- Datenbewegende Befehle: Laden, Speichern, Kopieren.

Alle angegebenen Operationen können in konstanter Zeit ausgeführt werden.

Wir nehmen an, dass Integer-Zahlen der Größe m mit $c \log m$ Bits für eine Konstante $c \geq 1$ dargestellt werden können. Reelle Zahlen werden gemäß des IEEE Standards im Rechner

durch Binärzahlen approximiert. Innerhalb des Darstellungsbereiches können arithmetische Operationen mit reellen Zahlen im Rechner sehr effizient durchgeführt werden.

1.1.3 Algorithmische Paradigmen

Algorithmen lassen sich häufig bezüglich ihrer Heransgehensweise an das jeweilige Problem kategorisieren. Zunächst seien hier einige davon kurz erwähnt. Es geht hier im wesentlichen darum, wie das Problem gelöst wird.

- Brute-Force (Naive Methode)
- Inkrementelle Konstruktion (Schrittweise Vergrößerung der Eingabe)
- Divide-and-Conquer (Aufteilen und Zusammenfügen)
- Greedy (*gierig*, schnelle Verbesserungen)
- Dynamische Programmierung (Tabellarische Auflistung und Verwertung von Teillösungen)
- Sweep (geometrische Probleme, Dimensionsreduktion)
- ...

Die Methoden können auch kombiniert auftreten, zum Beispiel ein Divide-and-Conquer der im Merge-Schritt einen Sweep verwendet. Auf die einzelnen Paradigmen werden wir später gezielt für konkrete Problemstellungen eingehen.

1.1.4 Analysemethode

Sei P eine Probleminstanz eines Problems Π und sei A ein Algorithmus mit RAM-Anweisungen, der für jedes Problem $P \in \Pi$ eine Lösung liefert.

Die Anzahl der Elementaroperationen eines Algorithmus A bei einer Eingabegröße $|P| = n \in \mathbb{N}$ wird durch eine *Kostenfunktion* $T_A : \mathbb{N} \mapsto \mathbb{R}^+$ beschrieben. Die Instanz P könnte beispielsweise eine Menge von n Zahlen sein.

Die exakte Bestimmung dieser Funktion kann vernachlässigt werden, insbesondere, da auf unterschiedlichen Rechnern verschiedene Konstanten für die Elementaroperationen beachtet werden müssten. Also sind wir bei der Analyse nur an der Größenordung der Funktion T_A in Abhängigkeit der Eingabegröße $|P| = n$ interessiert. Ebenso verhält es sich mit der Eingabegröße selbst, ob wir es mit n Punkten in der Ebene zu tun haben oder mit $2n$ Koordinaten, soll beispielsweise keine Rolle bei der Beschreibung der Größenordnung spielen.

Die Anzahl der Elementaroperationen eines Algorithmus mit Eingabegröße $n \in \mathbb{N}$ kann auch von der Zusammenstellung der Eingabe selbst abhängen. Deshalb können wir verschiedene Kennzahlen verwenden:

- Worst-Case: Maximal mögliche Anzahl an Elementaroperationen
- Average-Case: Mittlere Anzahl der Elementaroperationen, gemittelt über alle möglichen Eingabefolgen der entsprechenden Größenordnung

Die Analyse kleiner Eingabegrößen ist generell wenig aussagekräftig, da wir in diesem Fall die Summe aller Elementaroperationen unter einer großen Konstante subsumieren können. Deshalb untersuchen wir das *asymptotische Verhalten* der Kostenfunktion.

Das führt zu folgenden Notationen für Klassen von Funktionen:

Definition 1 (O-,Ω-,Θ-Notation)

$$\begin{aligned} O(f) &= \{g| \text{ es existiert ein } n_0 \geq 0 \text{ und ein } C > 0, \\ &\qquad \text{so dass } g(n) \leq C\, f(n) \text{ für alle } n \geq n_0\} \end{aligned}$$

$$\begin{aligned} g \in \Omega(f) &:\Leftrightarrow f \in O(g) \\ &:\Leftrightarrow \text{ es existiert ein } n_0 \geq 0 \text{ und ein } C > 0, \\ &\qquad \text{so dass } f(n) \leq C\, g(n) \text{ für alle } n \geq n_0. \end{aligned}$$

$$g \in \Theta(f) :\Leftrightarrow g \in O(f) \text{ und } g \in \Omega(f)$$

Wir erlauben also mit $n \geq n_0$ endliche viele Ausnahmestellen.

Die obige Notation wird in dem Sinne gebraucht, dass $f \in O(g)$ eine *obere Schranke* an die Funktion f durch g liefert, währenddessen $f \in \Omega(g)$, eine *untere Schranke* für f durch g darstellt.

Beispiele:

$3n+2 \in O(n)$, da für alle $n \geq 2$ und für $C = 4$ gilt: $3n+2 \leq 4n$. Desgleichen gilt $3n+2 \in \Omega(n)$, da für alle $n \geq 0$ bereits $n \leq 3n + 2$ gilt. Somit gilt $3n + 2 \in \Theta(n)$.

Für $g(n) = 2n^3 - 18n^2 - \sin(n) + 16n + 3$ führt zunächst die grobe Abschätzung

$$\begin{aligned} g(n) &\leq 2n^3 + 16n + 3 \\ &\leq (2 + 16 + 3)n^3 \text{ für alle } n \geq 1 \end{aligned}$$

zur Aussage $g \in O(n^3)$. Es gilt aber auch

$$\begin{aligned} g(n) &\geq 2n^3 - 18n^2 \\ &= n^3 + (n - 18)n^2 \\ &\geq n^3 \text{ für alle } n \geq 15 \end{aligned}$$

und deshalb gilt hier ebenfalls $g \in \Omega(n^3)$ und $g \in \Theta(n^3)$. Wir sagen auch, dass die Funktion g in der *Größenordnung* n^3 wächst.

Eine weitere hilfreiche Notationkonvention ist, dass wir die obigen Symbole der Einfachheit halber bei einer asymptotischen Betrachtung auch in Formeln verwenden möchten. So drückt $2n^2+3n+1 = 2n^2+\Theta(n)$ aus, dass wir uns um die Funktion $3n+1$ in der Größenordnung $\Theta(n)$ nicht genau kümmern möchten. Diese Konvention kann hilfreich sein, wenn beispielsweise eine Laufzeitabschätzung *rekursiv* durch eine Formel $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ beschrieben werden darf. Wahlweise kann hier auch $T(n) = 2T\left(\frac{n}{2}\right) + C \cdot n$ für ein nicht näher bestimmtes C verwendet werden.

Außerdem kann es sinnvoll sein, die Analyse der Laufzeitem *ausgabesensitiv* vorzunehmen, um die Ausgabekomplexität nicht der algorithmischen Komplexität anzulasten. Falls wir beispielsweise die Menge aller Schnittpunkte von n Liniensegmenten suchen, so ist klar, dass es

im worst-case $\Omega(n^2)$ viele Schnittpunkte geben kann, jeder Algorithmus also mindestens diese Laufzeit im worst-case benötigt. Eine ausgabesensitive Analyse könnte zu einem Ergebnis von $O((n+k)\log n)$ führen, wobei k die Anzahl der tatächlichen Schnittpunkte angibt.

1.2 Ein einfaches Beispiel: Insertionsort

Die obigen Begriffe wollen wir zunächst auf das einfache Beispiel des Sortierens anwenden. Wir nehmen an, die n zu sortierenden Zahlen sind in einem *Array* A gespeichert.

Eingabe: n reelle Zahlen $A[1], A[2], \ldots, A[n]$.

Ausgabe: Ein Array B der Zahlen aus A mit $B[1] \leq B[2] \leq \cdots \leq B[n]$.

Beachte, dass wir hier eine etwas andere Ausgabe als zu Beginn erwarten (Permutation π). Die Aufgabenstellung ist aber äquivalent.

Eine *Brute-Force* Vorgehensweise wäre, das kleinste Element des Arrays zu ermitteln, dieses zu entfernen und mit dem restlichen Array fortzufahren.

Wir betrachten einen *inkrementellen* Ansatz. Sukzessive wird ein immer größerer Teil des Arrays A in B sortiert. Anders ausgedrückt, falls wir die ersten $j-1$ Elemente des Arrays A in B bereits sortiert haben, nehmen wir uns danach das j-te Element vor und sortieren es in B ein.

Algorithm 1 InsertionSort(A)
Array A (nichtleer) sortiert in Array B ausgeben

```
1: B[1] := A[1];
2: for j = 2 to length(A) do
3:    key := A[j]; i := j − 1;
4:    while i > 0 und B[i] > key do
5:       B[i + 1] := B[i]; i := i − 1;
6:    end while
7:    B[i + 1] := key;
8: end for
9: RETURN B
```

Die Korrektheit des Algorithmus wird durch eine *Schleifeninvariante* gezeigt. Wir zeigen formal, dass für jeden Beginn der FOR-Schleife die fogende Invariante gilt: *Für den aktuellen Wert j gilt, dass B bis zum Index $j-1$ eine sortierte Folge der ersten $j-1$ Einträge von A ist.*

Formal gesehen, handelt es sich hier um eine Induktion, die bei einem bestimmten Parameter terminiert.

Induktionsanfang: $j = 2$. Die Eigenschaft ist erfüllt, durch die erste Anweisung.

Induktionsschritt: Wir nehmen an, dass die Aussage bereits für $j-1 \geq 2$ gilt. Jetzt werden in der WHILE-Schleife im Array B solange die Werte um einen Index nach hinten geschoben, bis das Element $key := A[j]$ kleiner gleich dem i-ten Element von B ist. Dann wird key als $(i+1)$-tes Element in B eingetragen. Falls das bis $i = 0$ nicht eintritt, wird key als 1-tes Element von B eingetragen. Insgesamt ist B danach bis zum Index j ein sortiertes Array der ersten j-Elemente von A.

Terminierung: Falls $j = \text{length}(A) + 1$ wird, gilt zunächst die Invariante, dass für den aktuellen Wert j gilt, dass B bis zum Index $j - 1 = \text{length}(A)$ eine sortierte Folge der ersten $\text{length}(A)$ Einträge von A ist. Die FOR-Schleife bricht ab und das Ergebnis wird korrekt in B ausgegeben.

Streng genommen muss die WHILE-Schleife genauso analysiert werden. Welche Schleifeninvariante muss hier gewählt werden?

Für eine Laufzeitabschätzung analysieren wir die einzelnen Programmschritte. Dabei sei t_j die Anzahl der Durchläufe der WHILE-Schleife wenn die FOR-Schleife für j aufgerufen wird. Sei $\text{length}(A) = n$.

Es ergibt sich die folgende Tabelle:

Anweisungen in Algorithmus 1	Kosten	Häufigkeit
1:	c_0	1
3:	c_1	$n-1$
5:	c_3	$\sum_{j=2}^{n} t_j$
7:	c_4	$n-1$

Insgesamt ergibt sich nach unseren Konventionen eine Laufzeit $T(n) = c_3 \sum_{j=2}^{n} t_j + O(n)$.

Die Laufzeit hängt also von der Größe von t_j ab. Im schlechtesten Fall ist das Array A absteigend sortiert, alle Elemente aus B müssen verschoben werden und das j-te Element wird vorne eingefügt. In diesem Fall gilt $t_j = j$ und wir haben.

$$T(n) \leq c_3 \sum_{j=1}^{n} j + O(n) = c_3 \frac{n(n+1)}{2} + O(n) \in O(n^2)$$

Die worst-case Analyse ist im allgemeinen sinnvoll, weil ...

- ... wir eine Laufzeit für jede beliebige Eingabe garantieren.
- ... der worst-case in vielen Anwendungsfeldern relativ häufig vorkommen kann.
- ... die Analyse der mittleren Laufzeit nicht unbedingt bessere Ergebnisse erzielt.

Beispielsweise können wir bei der Analyse einer mittleren Laufzeit die Frage stellen, an welcher Stelle im Mittel das j-te Element eingeordnet werden muss. Im Mittel ist die Hälfte der Elemente größer als $A[j]$ und die Hälfte der Elemente kleiner als $A[j]$ und somit könnten wir $t_j = \frac{j}{2}$ betrachten. Das führt asymptotisch gesehen exakt zur gleichen Laufzeit.

Anders kann es aussehen, wenn wir aus der Eingabe durch *Würfeln* zufällig gleichverteilt die Elemente aus A ziehen.

Solche *probabilistische Analysen* werden wir später für sogenannte *randomisierte* Algorithmen durchführen. Im Gegensatz dazu haben wir hier zunächst einen *deterministischen* Algorithmus betrachtet. Der Lösungsplan ist von vorneherein vollständig festgelegt und nicht vom Zufall abhängig.

Kapitel 2

Divide-and-Conquer

In diesem Kapitel behandeln wird die Entwurfsmethode *Teile-und-Herrsche* und eine zugehörige allgemeine Analysemethode. Gleichzeitig lernen wir dadurch das Prinzip der *Rekursion* kennen, das heißt wir rufen bestimmte Routinen unseres Algorithmus immer wieder auf, um Teilprobleme zu lösen. Am Ende werden die Lösungen der Teilprobleme zur Lösung des Gesamtproblems zusammengefügt bzw. zusammengemischt.

Das allgemeine Schema der Methode für ein Problem P der Größe n funktioniert wie folgt:

1. Divide	$\begin{cases} n > c & : \text{ Teile das Problem in } k \geq 2 \text{ Teilprobleme} \\ & \quad \text{der Größen } n_1, n_2, \ldots, n_k, \text{ gehe zu 2.} \\ n \leq c & : \text{ Löse das Problem direkt} \end{cases}$
2. Conquer	Löse die k Teilprobleme auf dieselbe Art (rekursiv)
3. Merge	Füge die k berechneten Teillösungen zu einer Gesamtlösung zusammen

In diesem Kapitel werden wir zunächst das Beispiel des Sortierens von Zahlenketten betrachten und später ein allgemeines nützliches Theorem zur Lösung von Rekursionsgleichungen vorstellen.

2.1 Sortieren von Zahlenketten, Mergesort

Hier betrachten wir $k = 2$ und $n_1, n_2 \approx \frac{n}{2}$. Für eine konkrete Zahlenfolge ergibt sich das folgendes Aufrufschema eines Divide-And-Conquer Algorithmus:

Eingabe	:	$(3,2,1,7,3,4,9,2)$
Divide I.	:	$(3,2,1,7)(3,4,9,2)$
(Conquer)Divide II.	:	$(3,2)(1,7)(3,4),(9,2)$
(Conquer)Divide III.	:	$(3)(2)(1)(7)(3)(4)(9)(2)$
Merge III.	:	(2,3)(1,7)(3,4)(2,9)
Merge II.	:	(1,2,3,7)(2,3,4,9)
Merge I.	:	(1,2,2,3,3,4,7,9)

Im Pseudocode und unter Verwendung von Arrays könnte eine Implementierung wie folgt aussehen. Wir verwenden ein nichtleeres Array A von Index 1 bis $n = \text{length}(A)$ und rufen zur vollständigen Sortierung MergeSort$(A, 1, n)$ auf.

Procedure 2 MergeSort(A,p,r)
Array A wird zwischen Index p und r sortiert

```
1: if p < r then
2:    q := ⌊(p + r)/2⌋;
3:    MergeSort(A, p, q);
4:    MergeSort(A, q + 1, r);
5:    Merge(A, p, q, r);
6: end if
```

Die eigentliche Arbeit (bis auf die Anzahl der rekursiven Aufrufe) wird im Merge-Schritt vollzogen. Dafür kopieren wir die bereits von Index p bis q und von Index $q+1$ bis r sortierten Abschnitte in zwei Arrays L und R und fügen diese dann wieder gemeinsam sortiert in A von Index p bis r ein. Für ein Abbruchkriterium des Mischens fügen wir einen Dummywert ∞ am Ende von L und R ein.

Danach werden die beiden Teilarrays gemischt, indem in einem gleichzeitigen Durchlauf jeweils das kleinste momentane *Kopfelement* von L und R in A eingefügt wird.

Procedure 3 Merge(A,p,q,r)
Array A vorsortiert zwischen Index p und q und Index $q + 1$ und r wird zwischen p und r sortiert

```
1: n1 := q − p + 1; n2 := r − q;
2: for i = 1 to n1 do
3:    L[i] := A[p + i − 1];
4: end for
5: for j = 1 to n2 do
6:    R[j] := A[q + j];
7: end for   // erzeuge Arrays L[1 . . . (n1 + 1)] und R[1 . . . (n2 + 1)]
8: L[n1 + 1] := ∞; R[n2 + 1] := ∞;
9: i := 1; j := 1;
10: for k = p to r do
11:    if L[i] ≤ R[j] then
12:       A[k] := L[i]; i := i + 1;
13:    else
14:       A[k] := R[j]; j := j + 1;
15:    end if
16: end for
```

2.1.1 Laufzeitanalyse

Für einen einzelnen Merge-Schritt entsteht offensichtlich ein Aufwand von $O(|L| + |R|)$; siehe Prozedur 3. In jedem Schritt der finalen FOR-Schleife wird ein Element von L oder R nicht mehr betrachtet. Deshalb kann die Schleife nicht mehr als $(|L| + |R|)$ Mal aufgerufen werden.

Zur Laufzeitanalyse verwenden wir eine Rekursionsgleichung. Eine klassische Vereinfachung ist dabei, dass wir annehmen, dass n eine Zeierpotenz ist, also $n = 2^k$. Wir können die Kostenfunktion dann für den obigen Algorithmus wie folgt beschreiben.

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{falls } n > 1 \end{cases} \tag{2.1}$$

In der obigen Gleichung haben wir eine einzelne Konstante c verwendet. Das ist prinzipiell erlaubt, da wir für die Konstanten für das Problem der Größe 1 und den Merge-Schritt einfach die größte der beiden Konstanten wählen können. Die asymptotische Laufzeit ändert sich nicht.

Beachten Sie, dass in den Kosten cn auch das Aufteilen der Aufgabenstellung in zwei Teilprobleme subsumiert ist. Hier musste lediglich ein Index in der Mitte in konstanter Zeit berechnet werden.

Die Rekursionsgleichung läßt sich nun leicht inbesondere auch wegen der Annahme $n = 2^k$ durch einen Rekursionsbaum abschätzen, siehe Abbildung 2.1. Die allerunterste Ebene des Baumes enthält exakt n Aufrufe für die Arrays der Größe 1. In jeder darüberliegenden Ebene werden jeweils zwei Mengen der darunterliegenden Ebene im Merge-Schritt vereinigt. Die Kosten sind jeweils *linear* zur Summe der Größen der darunterliegenden Arrays. Somit sind die Kosten auf jeder Ebene linear zur Gesamtzahl der Elemente überhaupt. Der Baum hat $k = \log_2 n$ viele Ebenen, da sich die Anzahl der Knoten von unten nach oben jeweils halbiert. Wir können also $T(n) = O(n \log n)$ und sogar $T(n) = \Theta(n \log n)$ folgern. Streng genommen

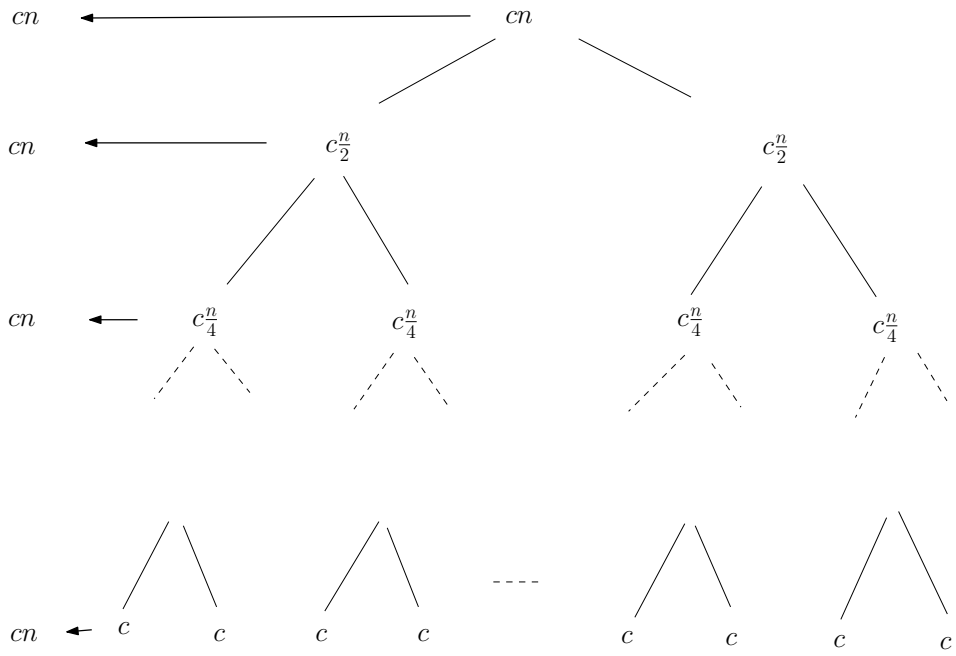


Abbildung 2.1: Auf jeder Ebene des Rekursionsbaumes fällt ein Aufwand von cn für die einzelnen Merge-Schritte an. Nach $k = \log_2 n$ Ebenen sind nur noch einzelne Elemente vorhanden.

hätten wir die Rekursionsgleichnung (2.1) wie folgt formulieren müssen.

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + cn & \text{falls } n > 1 \end{cases} \tag{2.2}$$

In jedem Fall müssen wir uns beim Aufstellen und Lösen der Rekursionsformeln Gedanken darüber machen, ob die Einschränkung $n = 2^k$ nicht das asysmptotischen Verhalten der Laufzeit beinflusst. Im obigen Fall ist das offensichtlich nicht der Fall, der zugehörige Rekursionsbaum wäre zwar nicht so ausgeglichen, wie in Abbildung 2.1. Für jedes n könnten wir aber beispielsweise die nächste Zweierpotenz oberhalb von n nehmen, um eine obere Schranke zu erzielen, sowie die nächste Zweierpotenz unterhalb von n für eine untere Schranke der Laufzeit. Der Algorithmus hat dann entsprechend mehr oder entsprechend weniger Aufrufe zu absolvieren. Darüber müssen wir uns aber Gedanken machen.

Beispielsweise kann eine Rekursionsformel der Form

$$T(n) = \begin{cases} n & \text{falls } n \text{ gerade} \\ n^2 & \text{falls } n \text{ ungerade} \end{cases} \tag{2.3}$$

nicht mit dem Ansatz $n = 2^k$ betrachtet werden.

2.1.2 Korrektheit

Die Korrektheit des obigen Divide-and-Conquer Algorithmus ergibt sich direkt aus der Korrektheit der Merge Prozedur. Sobald diese gemäß ihrer Spezifikation funktioniert, erhalten wir die richtige Zwischenergebnisse rekursiv für weitere Aufrufe.

Zunächst ist klar, dass die Merge-Prozedur durch die ersten beiden FOR-Schleifen die Einträge des Arrays A von Index p bis Index q und von Index $q+1$ bis r in die Kopien L und R der Größen $n_1 := q-p+1$ $n_2 := r-q$ einträgt. Streng genommen müssen auch diese Behauptungen durch eine Schleifeninvariante bewiesen werden.

Wir beschränken uns hier auf die dritte FOR-Schleife und formulieren eine geeignete Schleifeninvariante.

Zu Beginn jeder Iteration der FOR-Schleife enthält das Array $A[p \dots k-1]$ die $(k-p)$ kleinsten Einträge aus L und R in sortierter Reihenfolge. $L[i]$ und $L[j]$ sind die kleinsten Einträge der jeweiligen Arrays, die noch nicht in A zurückkopiert wurden.

Induktionsanfang: $k = p$. Das Array $A[p \dots k-1]$ enthält die $k - p = 0$ kleinsten Elemente von L und R. $L[1]$ und $R[1]$ sind die kleinsten Elemente der Arrays, die noch nicht zurückkopiert wurden.

Induktionsschritt: Wir nehmen an, die Aussage gelte bereits für $k = p + l$ und wollen zeigen, dass die Aussage nach dem Durchlauf der FOR-Schleife für $k' = p + l + 1$ erhalten bleibt. Die Schleife wird mit $k = p + l$ ausgeführt. Es werden die aktuellen Köpfe der beiden Arrays L und R verglichen und das kleinste der beiden Elemente als $k = (p + l)$-tes Element in A eingetragen. Also besteht danach $A[p \dots k'-1]$ für $k' = p+l+1$ aus den $(k'-p)$ kleinsten Einträge aus L und R in sortierter Reihenfolge. Je nachdem, ob $L[i] \leq R[j]$ oder $L[i] > R[j]$ gilt, wird i oder j um eins erhöht und die Aussage gilt auch für $k' = p + l + 1$ und das neue j oder i.

Terminierung: Beim Abbruch gilt $k = r + 1$ und $A[p \dots r]$ enthält die $r + 1 - p$ kleinsten Elemente von L und R in sortierter Reihenfolge. Zusammen enthalten L und R insgesamt $n_1 + n_2 + 2 = r - p + 3$ Elemente. Also muss sowohl i am Index $n_1 + 1$ und j am Index $n_2 + 1$ angekommen sein. Alle Werte außer den Dummywerten ∞ sind in A eingetragen worden.

2.1.3 Untere Schranke von Sortieralgorithmen

Mergesort kann also n Zahlen $(a_1, a_2, \dots, a_n)$ in Zeit $O(n \log n)$ sortieren. Dabei greift der Algorithmus auf die Elemente der Eingabefolge nur durch *paarweise Vergleiche* zu (siehe Zeile 11 der Merge-Prozedur): Es wird getestet, ob zum Beispiel $a_i < a_j$ gilt, und der Ausgang dieses Tests entscheidet, was der Algorithmus als nächstes macht. “Gerechnet” wird mit den a_i aber nicht.

Mit solchen vergleichsbasierten Sortierverfahren kann man deswegen nicht nur Zahlen sortieren, sondern Objekte aus beliebigen vollständig geordneten Mengen, vorausgesetzt, es steht ein Größenvergleichstest zur Verfügung.

Interessanterweise ist Mergesort sogar optimal.

Theorem 2 *Jeder vergleichsbasierte Sortieralgorithmus benötigt im worst-case* $\Omega(n \log n)$ *viele Vergleiche, um* n *Objekte zu sortieren.*

Beweis. Sei A ein vergleichsbasierter Sortieralgorithmus. Wir nehmen der Einfachheit halber an, dass A deterministisch ist; das Theorem gilt aber auch für randomisierte Verfahren. Außerdem setzen wir voraus, dass die a_i paarweise verschieden sind. Weil A den Input $(a_1, a_2, \ldots, a_n)$ zunächst nicht kennt, wird stets derselbe Test $a_i < a_j$? als erster ausgeführt. In Abhängigkeit vom Ergebnis verzweigt der Algorithmus. Für alle Eingaben, bei denen $a_i < a_j$ gilt, wird jetzt stets derselbe Test $a_v < a_w$? als nächster ausgeführt. Ebenso kommt für alle Inputs mit $a_i > a_j$ als nächstes stets derselbe Test $a_r < a_s$? an die Reihe, und so fort. So entsteht der Entscheidungsbaum B_A von Algorithmus A, ein Binärbaum, der mindestens so viele Blätter enthält, wie es Permutationen gibt; denn ein korrekter Sortieralgorithmus darf für unterschiedlich permutierte Eingabefolgen nicht zum selben Ergebnis kommen. Also gilt

$$\begin{aligned} \text{Höhe}(B_A) &\geq \log_2(n!) \geq \log_2\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2}\log_2\left(\frac{n}{2}\right) \\ &\geq \frac{1}{3} n \log_2(n) \end{aligned}$$

für alle $n \geq 8$. Dabei gilt die erste Ungleichung, weil es in $n!$ mindestens $n/2$ viele Faktoren gibt, die mindestens so groß sind wie $n/2$. Es gibt also im Baum B_A mindestens einen Pfad der Länge $\Omega(n \log n)$, und wenn die Eingabe so permutiert ist, wie es den Testergebnissen längs dieses Pfades entspricht, muss Algorithmus A alle diese Vergleiche ausführen. □

Wenn die zu sortierenden Objekte aber Zahlen sind, mit denen man "rechnen" darf, gilt die untere Schranke von Theorem 2 nicht. Zum Beispiel geht der Algorithmus Radixsort folgendermaßen vor, um n k-stellige Zahlen zu sortieren: Zunächst werden leere 10 Listen $L_0, L_1, \ldots, L_9$ angelegt, je eine für jede Ziffer. Nun werden abwechselnd k Verteilungs- und Sammelphasen ausgeführt. Man beginnt mit der letzten der k Stellen, durchläuft die Eingabereihenfolge und hängt die Zahl a_i an diejenige Liste an, die zur letzten Ziffer von a_i gehört. Anschließend werden die Zahlen aus den Listen wieder eingesammelt, in der Reihenfolge $L_0, L_1, \ldots, L_9$ und unter Beibehalt der Reihenfolgen innerhalb der Listen. Mit der neuen Zahlenfolge verfährt man ebenso, verwendet aber nun die vorletzte Stelle, und so fort. Wenn man schließlich die Zahlen nach der ersten Stelle verteilt und wieder eingesammelt hat, sind sie sortiert (!).

Offensichtlich führt Radixsort nur $O(k\,n)$ viele Schritte aus. Wenn die Stellenzahl k als konstant betrachtet wird (was bei Standard-Zahlentypen meist der Fall ist), so ergibt sich die Laufzeit $O(n)$.

2.2 Closest Pair von n Punkten

Wir betrachten die folgende geometrische Aufgabenstellung. Für einer Menge S von n Punkten $p_1, p_2, \ldots, p_n$ in der Ebene, soll das Punktepaar mit kleinstem Euklidischen Abstand $d(\cdot, \cdot)$ berechnet werden.

Wir suchen also den Wert

$$d_S = \min_{1 \leq i,j \leq n, i \neq j} d(p_i, p_j).$$

Der Brute-Force Ansatz für dieses Problem vergleicht systematisch alle $O(n^2)$ vielen Distanzen und benötigt eine Laufzeit von $\Theta(n^2)$.

Für einen Divide-and-Conquer Algorithmus sortieren wir die Punkte zunächst nach X-Koordinaten und teilen die Punktmenge rekursiv in zwei gleichgroße Mengen anhand der X-Koordinaten auf.

1. Divide	Teile das Problem in Teilmengen S_r und S_l gleicher Größe anhand der X-Koordinaten auf
2. Conquer	Bestimme rekursiv d_{S_l} und d_{S_r}
3. Merge	Berechne d_S durch die Verwendung von d_{S_l} und d_{S_r}

Wir wollen im Merge-Schritt wiederum möglichst wenig Aufwand für die Bestimmung von d_S verwenden. Sei $d := \min\{d_{S_l}, d_{S_r}\}$. Wir brauchen für den Merge nur Punkte betrachten die von S_l zu Punkten aus S_r einen Abstand $< d$ haben. Nur die Punkte innerhalb des Streifens der Breite $2d$ symmetrisch entlang der *Splitvertikalen* können einen solchen Abstand $< d$ zueinander haben; siehe Abbildung 2.2. Im Merge-Schritt wird folgendes Verfahren angewendet:

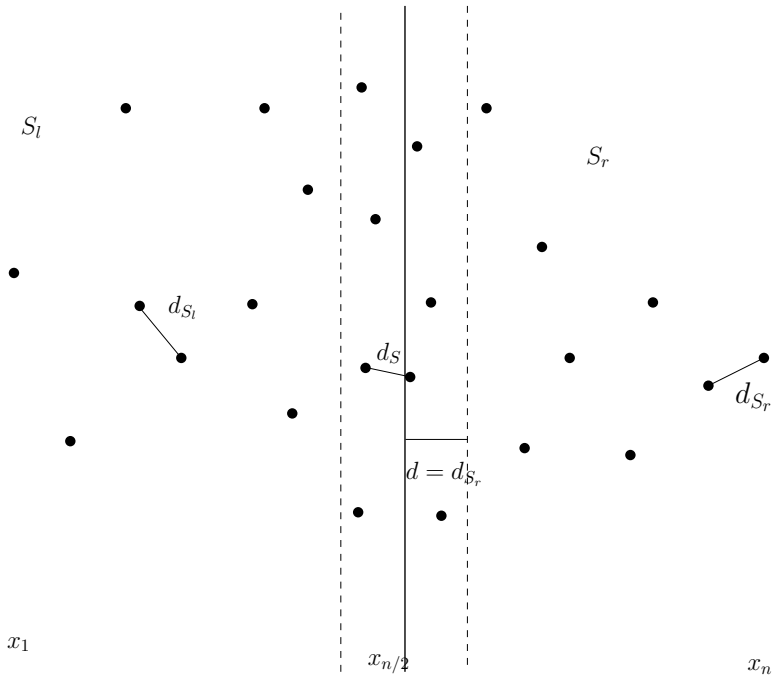


Abbildung 2.2: Für den Merge-Schritt müssen nur die Punkte der beiden Teilstreifen der Breite d entlang der Splitvertikalen betrachtet werden.

- Bestimme die Punkte in den zugehörigen rechten und linken Teilstreifen der Breite d und sortiere diese jeweils nach Y-Koordinaten
- Gehe die Punkte im linken Teilstreifen nach fallenden Y-Koordinaten durch
- Für jeden solchen Punkt p betrachte alle Punkte aus dem rechten Teilstreifen deren Y-Koordinaten im Bereich $[p_y - d, p_y + d]$ liegen
- Der zugehörige Punktbereich auf der rechten Seite wandert sukzessive nach *unten*

- Behauptung: Zu jedem Zeitpunkt sind nicht mehr als 10 Punkte im zugehörigen Bereich

Beweis. (Behauptung) Nehmen wir an, dass p und q ein dichtestes Punktepaar mit Abstand $< d$ sind. Dann können wir oBdA annehmen, dass p im linken und q im rechten Teilstreifen liegt. Der Punkt q kann nicht außerhalb des Rechtecks $R(p)$ der Breite d und Höhe $2d$ im rechten Teilstreifen liegen; siehe Abbildung 2.3. Außerdem haben alle Punkte in $R(p)$ einen Abstand von mindestens d zueinander. Dann sind die Kreisumgebungen $U_{\frac{d}{2}}(q)$ in $R(p)$ disjunkt. Für jeden Punkt q in $R(p)$ ist mindestens ein Viertel dieser Kreisfäche in $R(p)$ enthalten. $R(p)$ kann dann höchstens

$$\frac{2d^2}{\frac{\pi}{4}\left(\frac{d}{2}\right)^2} = \frac{32}{\pi} < 11$$

viele Punkte enthalten. □

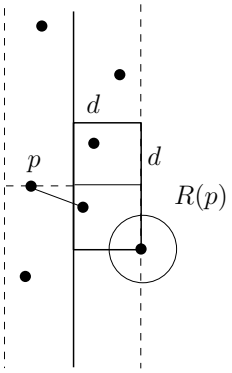


Abbildung 2.3: Für jeden Punkt p des linken Teilstreifens müssen nur konstant viele Punkte im rechten Teilstreifen getestet werden. Die zu testenden Punkte haben Y-Koordinaten, die im Bereich $[p_y - d, p_y + d]$ liegen.

Zur Analyse der Laufzeit betrachten wir nun die Einzelschritte des beschriebenen Algorithmus.

- In einem ersten Schritt werden die Punkte nach X-Koordinaten sortiert. Das kostet $O(n \log n)$ Zeit.
- Der Divide-Schritt kostet dann stets lineare Zeit.
- Für den Merge Schritt ergibt sich folgender Aufwand:
  1. Es müssen die Punkte mit X-Koordinaten aus den Teilstreifen ermittelt werden, das kostet $O(n)$ Zeit.
  2. Danach werden die Punkte in den jeweiligen Bereichen nach Y-Koordinate sortiert. Das kostet $O(n \log n)$ Zeit.
  3. In einem linearen Durchlauf wird für jeden Punkt p zu einer konstanten Anzahl (< 10) von Punkten q der Abstand ermittelt und der aktuell kleinste gespeichert. Insgesamt liegt der Aufwand in $O(n)$ für diesen Schritt.

Insgesamt erstellen wir daraus die folgende Rekursionsgleichung.

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ 2T\left(\frac{n}{2}\right) + c \cdot n \log n & \text{falls } n > 1 \end{cases} \tag{2.4}$$

Wie wir solche Rekursionsgleichungen im Allgemeinen lösen ist Gegenstand des nächsten Kapitels. Eine Übungsaufgabe besteht dann darin, die obige Gleichung abzuschätzen. Welche Vermutung haben Sie?

Kapitel 3

Lösen von Rekursionsgleichungen

Für die Analyse von Algorithmen, die das Prinzip der Rekursion verwenden, kann häufig eine Rekursionsgleichung aufgestellt werden.

Zunächst erwähnen wir einige Konventionen. Obwohl unsere Kostenfunktionen auf den natürlichen Zahlen definiert sind ignorieren wir diesen Sachverhalt. Deshalb können wir bei der Rekursionsformel für den Mergesort auch sogleich die Formel (2.1) verwenden und verzichten vollständig auf die Annahme $n = 2^k$.

Weiterhin liegen die Kosten für $T(n)$ für kleine n im Allgemeinen in $\Theta(1)$ und somit lassen wir diesen Teil der Rekursionsgleichung manchmal einfach weg. Die Bedingungen für kleine n werden auch Randbedingungen genannt. Die Randbedingungen spielen beim Induktionsbeweis für die Lösung einer Rekursionsgleichung eine Rolle. Insbesondere der Induktionsanfang ist davon betroffen.

Beispiele für Rekursionsgleichungen sind somit:

$T(n) = T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + c$ Binäre Suche

$T(n) = 2T\left(\frac{n}{2}\right) + cn$ Mergesort

$T(n) = 7\,T\left(\frac{n}{2}\right) + cn$ Strassen's Matrixmultiplikation

Die Frage lautet, wie wir diese Rekursionsgleichungen im Allgemeinen lösen können?

3.1 Substitutionsmethode

Die Substitutionsmethode besteht im Prinzip aus zwei Teilen.

1. Zunächst wird eine Lösung der Rekursionsgleichung *erraten*. Beispielsweise durch Erfahrung, durch Einsetzen einiger Werte oder durch die Analyse eines Rekursionsbaumes.

2. Dann wird die Lösung per *Induktion* verifiziert. Dabei können die verwendeten Konstanten präzisiert werden.

Beispiel:

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{falls } n > 1 \end{cases} \tag{3.1}$$

Wir vermuten, dass $T(n) \in O(n \log n)$ gilt und wollen $T(n) \leq c \cdot n \log n$ für geeignetes $c > 0$ per Induktion zeigen. Zunächst ist $T(1) = 1 > c \cdot 1 \log 1 = 0$. Deshalb muss hier $n_0 > 1$ gewählt werden. $T(2) = 4 \leq c \cdot 2 \log 2$ und $T(3) = 5 \leq c \cdot 3 \log 3$ für $c > 1$. Jetzt überprüfen wir die Aussage für $n_0 = 2$ per Induktion mit der *Substitutionsmethode*.

Induktionsanfang: $T(2) = 4 \leq c \cdot 2 \log 2$.

Induktionsschritt: Die Aussage gilt für alle Indizes von n_0 bis $n-1$. Zeige die Aussage auch für n.

$$\begin{aligned} T(n) &= 2T\left(\tfrac{n}{2}\right) + n \\ &\leq 2c \cdot \tfrac{n}{2} \log \tfrac{n}{2} + n && \text{(Induktionsannahme)} \\ &= c \cdot n \log n - c \cdot n \log 2 + n && \text{(Rechenregeln log)} \\ &\leq c \cdot n \log n + (1-c)n \\ &\leq c \cdot n \log n \text{ (für } c > 1) \end{aligned}$$

Als nächstes stellen wir zwei Methoden vor, wie die zu verifizierende Lösung erraten werden kann.

3.1.1 Raten durch Iteration

Wir betrachten die Rekursionsgleichung

$$T(n) = 3T\left(\frac{n}{4}\right) + n. \tag{3.2}$$

Für das Raten einer Lösung setzen wir die Gleichung sukzessive ein.

$$\begin{aligned} T(n) &= 3\,T\left(\tfrac{n}{4}\right) + n \\ &= 3\,\left(3\,T\left(\tfrac{n}{16}\right) + \tfrac{n}{4}\right) + n = 9T\left(\tfrac{n}{16}\right) + \left(\tfrac{3}{4}\right)^1 \cdot n + n && \text{(Einsetzen)} \\ &= 9\,\left(3\,T\left(\tfrac{n}{64}\right) + \tfrac{n}{16}\right) + 3\tfrac{n}{4} + n && \text{(Einsetzen)} \\ &= 27\,T\left(\tfrac{n}{64}\right) + \left(\tfrac{3}{4}\right)^2 \cdot n + \left(\tfrac{3}{4}\right)^1 \cdot n + \left(\tfrac{3}{4}\right)^0 \cdot n && \text{(Umformen)} \end{aligned}$$

Wenn wir $T(1) = c$ und $n = 4^k$, annehmen gelangen wir zur Vermutung

$$\begin{aligned} T(n) &= n \sum_{i=0}^{k-1} \left(\frac{3}{4}\right)^i + c \cdot 3^k \\ &= n \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{4}\right)^i + c \cdot n^{\log_4 3}. \end{aligned}$$

Hierbei wurde $3^{\log_4 n} = n^{\log_4 3}$ verwendet. Daraus wollen wir eine genaue Vermutung ableiten und diese danach beweisen.

$$
\begin{aligned}
T(n) &= n \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{4}\right)^i + c \cdot n^{\log_4 3} \\
&< n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + c \cdot n^{\log_4 3} && \text{(nach oben Abschätzen)} \\
&= n \cdot C + c \cdot n^{\log_4 3} && \text{(Potenzreihe } \textstyle\sum_{i=1}^{\infty} X^i \text{ Konvergenzradius } |X| \leq 1) \\
&< C' \cdot n && (\log_4 3 \leq 1)
\end{aligned}
$$

Also vermuten wir $T(n) \in O(n)$ und verifizieren die Aussage $T(n) \leq C' \cdot n$ durch Induktion mit Substitutionsmethode:

Induktionsanfang: $T(1) = c < C' \cdot 1$ gilt für $C' > c$.

Induktionsschritt: Die Aussage gilt für alle Indizes von n_0 bis $n-1$. Zeige die Aussage auch für n.

$$
\begin{aligned}
T(n) &= 3T\left(\tfrac{n}{4}\right) + n \\
&\leq 3C' \cdot \tfrac{n}{4} + n && \text{(Induktionsannahme)} \\
&= C' \cdot \tfrac{3}{4} \cdot n + n && \text{(Umformung)} \\
&= \left(1 + C' \cdot \tfrac{3}{4}\right) n && \text{(Umformung)} \\
&\leq C' \cdot n && \text{(für } C' \geq 4).
\end{aligned}
$$

3.1.2 Raten durch Rekursionsbaum

Die Idee bei der Verwendung eines Rekursionsbaumes ist, das obige iterative Einsetzen durch einen Baum darzustellen. Dabei werden die nicht-rekursiven Kosten und das Rekursionsargument notiert. Wir gehen wie folgt vor:

1. Jeder Knoten der Baumes stellt die Kosten des Teilproblems dar. Die Wurzel stellt die Kosten $T(n)$ dar und die Blätter die Kosten $T(1)$.
2. Wir summieren die Kosten auf jeder *Ebene* des Baumes.
3. Die Gesamtkosten ergeben sich durch die Summe der Kosten aller Ebenen.

Als Beispiel betrachten wir wiederum die Rekursionsgleichung (3.2) und nehmen $T(1) = c$ an. Damit der Baum *passend aufgeht* wählen wir $n = 4^k$.

Die Analyse über den Rekursionsbaum erfolgt durch

$$
T(n) = \underbrace{\sum_{i=0}^{\log_4 n-1}}_{\text{alle Ebenen}} \quad \underbrace{\left(\frac{3}{4}\right)^i \cdot n}_{\text{Kosten pro Ebene}} \quad + \quad \underbrace{c \cdot n^{\log_4 3}}_{\text{Kosten der Blätter}} \tag{3.3}
$$

Beachte: In der Praxis werden gelegentlich die Ratemethoden (insbesondere das iterative Einsetzen) bereits als *formaler Beweis* für eine Laufzeitanalyse verwendet. Auf die Induktion wird also tatsächlich gelegentlich verzichtet. Formal gesehen kann man das insbesondere unter

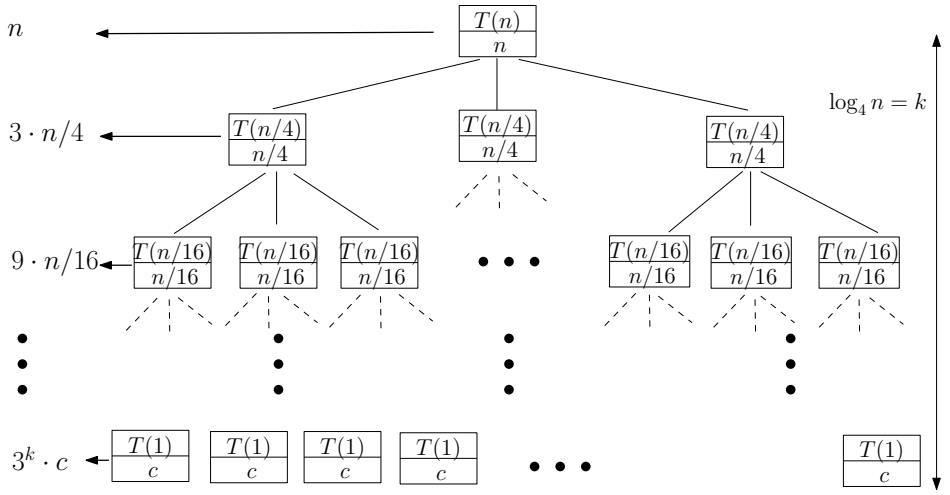


Abbildung 3.1: Der Rekursionsbaum zur Gleichung (3.2) hat $\log_4 n$ Ebenen. In jeder Ebene fallen konkrete Kosten (für das Aufteilen und das Kombinieren) an.

der Annahme, dass n eine bestimmte Potenz ist (zum Beispiel $n = 2^k$), gelten lassen. Das haben wir bespielsweise auch bei der Analyse des Mergesorts so gehandhabt. Die Aussage gilt dann aber streng genommen zunächst nur für exakt solche Potenzen.

3.2 Typische Probleme

Bei der Lösung von Rekursionsgleichungen können u.a. typischerweise die folgenden Probleme auftreten:

1. Der Induktionsbeweis ist aufgrund der Konstanten fehlerhaft.
2. Die Lösungsform ist zu ungenau.
3. Die Lösungsform erfordert eine Variablentransformation.

Wir geben hier Beispiele für die obigen Probleme an.

Korrekte Konstanten: Für

$$T(n) \quad = \quad 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

könnten wir

$$\begin{aligned} T(n) &\leq 2\left(c\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq c \cdot n + n \\ &= O(n) \qquad \text{Falsch!!} \end{aligned}$$

folgern, da c eine Konstante ist. Der Fehler liegt darin, dass wir nicht exakt $T(n) \leq cn$ bewiesen haben. Wir müssen die Konstante genau angeben.

Lösungsform ist zu ungenau: Für

$$T(n) \quad = \quad \begin{cases} 1 & \text{falls } n = 1 \\ 2T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 1 & \text{falls } n > 1 \end{cases}$$

vermuten wir $T(n) \leq cn$. Die Induktionsannahme ist für $c > 1$ erfüllt. Leider führt

$$\begin{aligned} T(n) &\leq c\left\lfloor \frac{n}{2} \right\rfloor + c\left\lceil \frac{n}{2} \right\rceil + 1 \\ &= cn + 1 > cn \end{aligned}$$

im Induktionsschluss durch Substitution nicht zum Ziel. Wir müssen hier $T(n) \leq cn - 1$ wählen und dann für die Induktionsannahme $c > 2$.

Lösung durch Variablentransformation:

Für

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T\left(\lfloor\sqrt{n}\rfloor\right) + \log n & \text{falls } n > 1 \end{cases}$$

können wir eine geeignete Umformung vornehmen. Durch $m = \log n$ gelangen wir vereinfacht zu
$T(2^m) = T\left(2^{m/2}\right) + m$ und erhalten durch Umbenennung die Gleichung $S(m) = 2S(m/2) + m$. Da, wie bereits gezeigt $S(m) = O(m \log m)$ gilt erhalten wir

$$T(n) = T(2^m) = S(m) \in O(m \log m) \in O(\log n \log\log n).$$

3.3 Das Mastertheorem

Wir wollen nun ein allgemeines Theorem für die Lösung solcher Rekursionsgleichungen erstellen. In der Regel sieht das Problem folgendermaßen aus:

- Das Problem teilt sich in $a \geq 1$ Teilprobleme auf.
- Jedes der Teilprobleme hat eine Größe $\frac{n}{b}$ für $b > 1$.
- Die Kosten für das Aufteilen und das Kombinieren der Teillösungen sind durch $f(n)$ gegeben.

Die Kostenanalyse wird dann durch die Gleichung

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{3.4}$$

beschrieben.

Theorem 3 *Seien $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ eine nichtnegative Funktion. Dann gilt für die Rekursionsgleichung (3.4):*

1. *Wenn $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$ für $\epsilon > 0$, dann gilt $T(n) \in \Theta\left(n^{\log_b a}\right)$.*
2. *Wenn $f(n) \in \Theta\left(n^{\log_b a}\right)$, dann gilt $T(n) \in \Theta\left(n^{\log_b a} \log n\right)$.*
3. *Wenn $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ für $\epsilon > 0$ und $a\ f(n/b) \leq c \cdot f(n)$ für $c < 1$ und für hinreichend große n, dann gilt $T(n) \in \Theta\left(f(n)\right)$.*

Die Lesart des obigen Theorems ist, dass wir hier einen Vergleich zwischen $f(n)$ und $n^{\log_b a}$ durchführen. Falls $f(n)$ asymptotisch kleiner ist als $n^{\log_b a}$ (Fall 1), wird $T(n)$ durch $n^{\log_b a}$ dominiert, falls $f(n)$ asymptotisch größer ist als $n^{\log_b a}$, wird $T(n)$ durch $f(n)$ dominiert (Fall 3). Wenn die beiden Funktionen die gleiche asymptotischen Größenordnung besitzen, gilt $T(n) = \Theta(f(n) \log n)$. Kleiner beziehungsweise größer ist hier als *polynomial* kleiner oder größer gedacht, also ein Unterschied in n^ϵ für ein $\epsilon > 0$.

Beispiel 1: Für $T(n) = 9\ T(n/3) + n$ gilt $a = 9$ und $b = 3$ und $f(n) = n$, dann ist $n^{\log_b a} = n^{\log_3 9} \in \Theta(n^2)$. Fall 1 tritt ein und $T(n) \in \Theta(n^2)$.

Beispiel 2: Für $T(n) = T(2n/3) + 1$ gilt $a = 1$ und $b = \frac{3}{2}$ und $f(n) = 1$ und es ist $n^{\log_b a} = n^{\log_{3/2} 1} = n^0$ und somit $f(n) \in \Theta(n^{\log_b a})$. Fall 2 ist anwendbar und es gilt $T(n) \in \Theta(\log n)$.

Beispiel 3: Für $T(n) = 3\ T(n/4) + n \log n$ gilt $a = 3$ und $b = 4$ und $f(n) = n \log n$ und es ist $n^{\log_b a} = n^{\log_4 3} = O(n^{0.8})$. Da nun $f(n) \in \Omega\left(n^{\log_4 3+\epsilon}\right)$ gilt und die *Regularitätsbedingung* $a\ f(n/b) \leq cf(n)$ für $c = \frac{3}{4}$ erfüllt ist, gilt $T(n) \in \Theta(n \log n)$. Es gilt $a\ f(n/b) = 3(n/4)\log(n/4) \leq 3/4n \log n = 3/4 f(n)$.

Beispiel 4: Für $T(n) = 2\ T(n/2) + n \log n$ gilt $a = 2$ und $b = 2$ und $f(n) = n \log n$ und $n^{\log_b a} = n^{\log_2 2} = n$. Nun ist aber $f(n)$ nicht polynomiell größer als $n^{\log_b a}$ da $\frac{f(n)}{n} = \log n$. Für jedes $\epsilon > 0$ ist n^ϵ asymptotisch größer als $\log n$. $f(n)$ ist zwar größer als $n^{\log_b a}$ aber nicht polynomiell größer. Das Theorem ist nicht anwendbar, eine Lücke zwischen Fall 2 und Fall 3. Das gleiche kann passieren, wenn $f(n)$ zwar kleiner ist als $n^{\log_b a}$ aber nicht polynomiell kleiner, das ist eine Lücke zwischen Fall 1 und Fall 2.

Wir beweisen das Mastertheorem in Teilen für Werte die durch Potenzen von b gegeben sind. Der allgemeine Beweis befindet sich zum Beispiel in [1]. Zunächst schätzen wir $T(n)$ für alle drei Fälle durch einen Rekursionsbaum ab. Danach werden für die Fälle die jeweiligen Eigenschaften der Funktion f benutzt.

Lemma 4 *Für Konstanten $a \geq 1$ und $b > 1$ und eine nichtnegative Funktion $f(n)$ sei*

$$T(n) \quad = \quad \begin{cases} \Theta(1) & \text{falls } n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & \text{falls } n = b^i \end{cases} \tag{3.5}$$

über die Potenzen von b definiert. Dann gilt

$$T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)\,. \tag{3.6}$$

Beweis. Wir schätzen $T(n)$ durch die Rekursionsbaummethode exakt für $n = b^i$ ab. Der Baum ist *vollständig* und hat eine Höhe von $i = \log_b n$. Jeder Knoten in Ebene j beschreibt ein Teilproblem mit Zusatzkosten $f(n/b^j)$. Bis auf die Blätter besitzt jeder Knoten a Kinder in der nächsten Ebene. Für die Darstellung des Baumes nehmen wir an, dass a eine ganze Zahl ist, logisch betrachtet ist das aber nicht notwendig. Das bedeutet, dass in Ebene j insgesamt a^j Knoten liegen. Die Blätter auf Ebene $\log_b n$ haben Kosten $\Theta(1)$. Insgesamt gibt es $a^{\log_b n} = n^{\log_b a}$ Blätter, siehe Abbildung 3.2. □

Die Gesamtkosten belaufen sich dann für die inneren Knoten auf

$$\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

und für die Blätter auf $c \cdot n^{\log_b a}$. Jetzt schätzen wir die Summe aus dem Term (3.6) gemäß des Mastertheorems ab.

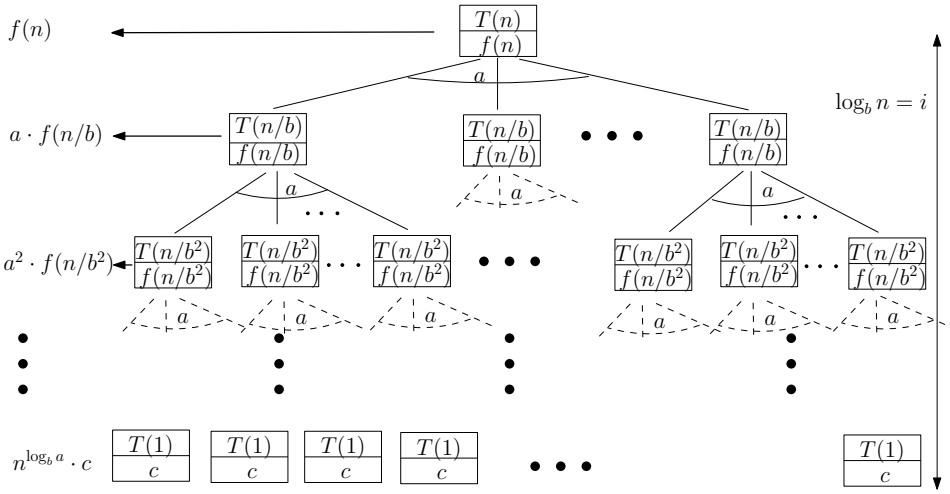


Abbildung 3.2: Der Rekursionsbaum zur Gleichung (3.5) hat $i = \log_b n$ Ebenen. In jeder Ebene fallen konkrete Kosten $a^j f(n/b^j)$ an.

Lemma 5 *Seien $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ eine nichtnegative Funktion. Dann gilt für die über Potenzen von b definierte Funktion*

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)\,. \tag{3.7}$$

1. *Wenn $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$ für $\epsilon > 0$, dann gilt $g(n) \in O\left(n^{\log_b a}\right)$.*
2. *Wenn $f(n) \in \Theta\left(n^{\log_b a}\right)$, dann gilt $g(n) \in \Theta\left(n^{\log_b a} \log n\right)$.*
3. *Wenn $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ für $\epsilon > 0$ und $a\ f(n/b) \leq cf(n)$ für $c < 1$ und für hinreichend große n, dann gilt $g(n) \in \Theta\left(f(n)\right)$.*

Beweis. Wir beweisen hier den ersten Fall, die anderen Fälle werden ähnlich behandelt, siehe auch [1]. Wir haben $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$ und somit $f(n/b^j) \in O\left((n/b^j)^{\log_b a - \epsilon}\right)$. Einsetzen in (3.7) ergibt

$$g(n) \in O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right)\,. \tag{3.8}$$

Wir vereinfachen den Term innerhalb der O-Notation und wenden die geometrische Reihe an.

$$\begin{aligned}
\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon} &= n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j \\
&= n^{\log_b a - \epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right) \\
&= n^{\log_b a - \epsilon} \left(\frac{n^\epsilon - 1}{b^\epsilon - 1}\right)
\end{aligned}$$

Da b und ϵ feste Konstanten sind kann der Term $\left(\frac{n^\epsilon - 1}{b^\epsilon - 1}\right)$ durch $O(n^\epsilon)$ abgeschätzt werden und es gilt: $g(n) \in O\left(n^{\log_b a}\right)$.

Die Fälle 2 und 3 sind noch etwas einfacher. □

Abschließend beweisen wir beispielhaft Nummer 1. in Theorem 3.

Beweis. (Beweis Theorem 3 Fall 1.) Aus Lemma 5 und Lemma 4 folgern wir

$$T(n) \in \Theta\left(n^{\log_b a}\right) + O\left(n^{\log_b a}\right) \in \Theta\left(n^{\log_b a}\right)\,.$$

□

Wie bereits erwähnt können die vollständigen Beweise sehr gut in [1] nachgelesen werden.

Kapitel 4

Dynamische Programmierung

Das Prinzip der dynamischen Programmierung kann dann angewendet werden, wenn eine rekursive Problemzerlegung an vielen Stellen die gleiche Teillösung verlangt. In diesem Fall ist es ratsam, die Lösungen dieser Teilprobleme zu speichern und gegebenenfalls darauf zurückzugreifen.

Falls die Teillösungen immer wieder neu berechnet werden, kann der Rechenaufwand exponentiell steigen. Grundsätzlich besteht die dynamische Programmierung aus

- einem rekursiven Aufruf für Teilprobleme und aus
- einer Tabellierung von Zwischenergebnissen die
- wiederverwendet werden sollen.

Wir betrachten zunächst ein einfaches klassisches Beispiel, das diese wesentlichen Bestandteile bereits charakterisiert. Generell werden durch die dynamischen Programmierung meistens Optimierungsprobleme gelöst. In solche Fällen wird dann versucht, zu jedem Teilproblem eine optimale Lösung zu finden.

Dynamische Programmierung ist dann sinnvoll, wenn das sogenannte *Optimalitätsprinzip von Bellman* vorliegt:

Eine optimale Lösung hat stets eine Zerlegung in optimale Teilprobleme.

Ein solches Beispiel wird in Abschnitt 4.2 vorgestellt.

4.1 Fibonacci Zahlen

Wir betrachten die Fibonacci-Zahlen, die rekursiv wie folgt definiert werden.

$$\begin{aligned} \text{fib}(0) &= 1 && (1)\\ \text{fib}(1) &= 1 && (2)\\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) && (3) \end{aligned}$$

Wollen wir nun beispielsweise fib(5) berechnen, so ergibt sich folgender Rekursionsbaum für die Aufrufe von fib; siehe Abbildung 4.1(I). Dieser Baum hat für fib(n) eine Größe von $\Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$, wie eine Übungsaufgabe zeigt.

Offensichtlich ist es ratsam, einige der Ergebnisse zu speichern, die rekursiven Abhängigkeiten können einfacher dargestellt werden, wie in Abbildung 4.1(II). Wir wollen zum Beispiel fib(2)

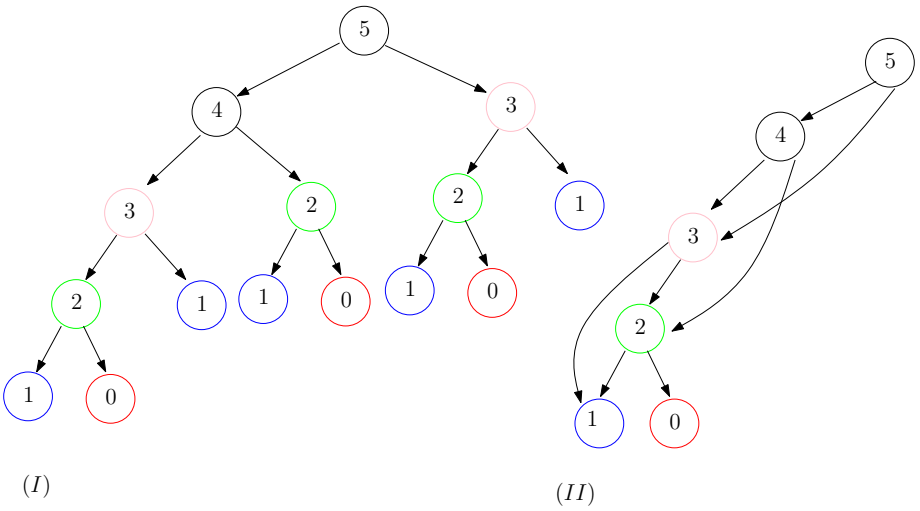


Abbildung 4.1: (I) Der vollständige Rekursionsbaum für fib(5) enthält einige Redundanzen. (II) Die rekursiven Abhängigkeiten können einfach dargestellt werden.

nicht dreimal wieder aus den Vorgängern neu berechnen, deshalb speichern wir die Werte ab. Es ist hier so, dass sich Teilprobleme überlappen und diese effizient gespeichert werden sollen. Der Abhängigkeitsgraph in Abbildung 4.1(II) hat nur eine Größe von $\Theta(n)$. Das Prinzip der Speicherung von Zwischenergebnissen wird auch *Memoisation* genannt.

Wenn wir nun die Abhängigkeiten nutzen wollen und von oben nach unten (Top-Down) die benötigten Werte fib(i) bestimmen wollen, könnten wir algorithmisch wie in Algorithmus 4 vorgehen, wobei FibMemo die memoisierende Prozedur 5 ist.

Algorithm 4 FibTopDown(n)

1: $F[1] := 1$; $F[0] := 1$;
2: **for** $i = 2$ to n **do**
3: $\quad F[i] := 0$;
4: **end for**
5: **RETURN** FibMemo(n, F)

Hierbei ist FibMemo die folgende memoisierende Prozedur.

Procedure 5 FibMemo(n,F)

1: **if** $F[n] > 0$ **then**
2: $\quad$ **RETURN** $F[n]$
3: **end if**
4: $F[n] :=$ FibMemo$(n-1, F)$ + FibMemo$(n-2, F)$;
5: **RETURN** $F[n]$

In Algorithmus 4 wird das Feld F zunächst mit Nullen initialisiert und beginnen dann den Aufruf der rekursiven Prozedur 5. Falls der Wert von $F[n]$ bekannt ist, geben wir diesen hier zurück, sonst wird er rekursiv gemäß der Rekursionsformel berechnet.

Laufzeitabschätzung: Wir benötigen im Algorithmus 4 $O(n)$ Zeit für die Initialisierung. Danach wird FibMemo(n, F) aufgerufen. Jeder Aufruf von FibMemo(m, F) in Prozedur 5 generiert höchstens einmal die Aufrufe FibMemo$(m-1, F)$ und FibMemo$(m-2, F)$. Für jedes

$l < n$ wird somit FibMemo(l, F) höchstens zweimal aufgerufen. Ohne die rekursiven Aufrufe wird nur $O(1)$ Zeit in Prozedur 5 benötigt. Insgesamt gilt für die Laufzeit $T(n) \in O(n)$.

Wir machen nun die folgende Beobachtung. Obwohl im obigen Algorithmus $F[n]$ in einer Top-Down Vorgehensweise errechnet wird, werden die eigentlichen Werte von unten nach oben berechnet. Erst wenn die Rekursion bei $F[1]$ und $F[0]$ angekommen ist, werden neue Felder belegt und die Memoisation setzt ein. Wenn wir beispielsweis $F[5]$ berechnen, rufen wir sukzessive FibMemo$(5, F)$, FibMemo$(4, F)$, FibMemo$(3, F)$, FibMemo$(2, F)$ und FibMemo$(1, F)$ auf bis hier zum ersten Mal ein Wert $F[1]$ zuückgegeben wird. Dann wird wieder FibMemo$(0, F)$ aufgerufen und $F[2]$ belegt. Dann wird FibMemo$(1, F)$ nochmal aufgerufen und $F[3]$ belegt. Danach rufen wir FibMemo$(2, F)$ auf und belegen $F[4]$. Dann FibMemo$(3, F)$ und $F[5]$ wird belegt und ausgegeben.

Deshalb ist es offensichtlich sinnvoller gleich die Werte von unten nach oben (Bottom-Up) zu berechnen. Da für die Berechnung von $F[n]$ nur zwei Werte aus $F[0], \ldots, F[n-1]$ benötigt werden, sparen wir außerdem Speicherplatz. Die obige Memoisation und die Top-Down vorgehensweise ist hier bezüglich des Speicherplatzes nicht effizient.

Insgesamt erhalten wir folgende effiziente dynamische Programmierungslösung.

Algorithm 6 FibDynProg(n)

1: $F_{\text{letzt}} := 1$; $F_{\text{vorletzt}} := 1$;
2: **for** $i = 2$ to n **do**
3: $\quad F := F_{\text{vorletzt}} + F_{\text{letzt}}$;
4: $\quad F_{\text{vorletzt}} := F_{\text{letzt}}$; $F_{\text{letzt}} := F$;
5: **end for**
6: **RETURN** F

- Es müssen insgesamt nur $\Theta(n)$ viele Berechnungen durchgeführt werden.
- Die Bottom-Up Berechnung stellt sicher, dass die benötigten Werte stets vorliegen.
- Der Speicherplatzbedarf liegt in $O(1)$.

Beachte: Manchmal kann eine Top-Down vorgehensweise auch sinnvoller sein, wenn zum Beispiel gar nicht alle Werte verwendet werden, die wir Bottom-Up berechnen würden. Die Prinzipien sind also nicht *dogmatisch* zu interpretieren.

Grundprinzip der dynamischen Programmierung:

- Formuliere das Problem insgesamt rekursiv. Stelle fest, dass sich Teillösungen überlappen und rekursiv voneinander abhängen.
- Stelle eine Rekursionsgleichnung (Top-Down) auf.
- Löse das Problem (Bottom-Up) durch das Ausfüllen von Tabellen mit Teillösungen (Memoisation).
- Ziel: Die Zahl der Tabelleneinträge soll klein gehalten werden.
- Anwendung: Bei einer direkten Ausführung der Rekursion ist mit vielen Mehrfachberechnungen zu rechnen.

Die Korrektheit der dynamischen Programmierung ergibt sich in der Regel dadurch, dass die rekursive Problembeschreibung korrekt durchgeführt wurde. Die Laufzeitanalyse ist in

der Regel nicht schwer, da die Algorithmen typischerweise aus geschachtelten FOR-Schleifen bestehen. Das eigentliche Problem der dynamischen Programmierung ist es, eine geeignete rekursive Formulierung des Problems zu finden. Wir werden uns deshalb ein signifikantes Optimierungsbeispiel ansehen in dem auch das *Optimalitätsprinzip von Bellman* zum tragen kommt.

4.2 Matrixmultiplikation

Wir betrachten das Multiplizieren von reellwertigen Matrizen. Seien $A \in \mathbb{R}^{i \times j}$ und $B \in \mathbb{R}^{j \times k}$ Matrizen mit i Zeilen und j Spalten bzw. j Zeilen und k Spalten, dann beschreibt $A \cdot B$ die Multiplikation dieser beiden Matrizen. Insgesamt sind zur Berechnung von $A \cdot B = C \in \mathbb{R}^{i \times k}$ $i \cdot k \cdot j$ Floating-Point Berechnungen notwendig. Genauer: Die Matrixeinträge von C werden durch

$$c_{st} = \sum_{l=1}^{j} a_{sl} \cdot b_{lt}$$

für alle $s = 1, \ldots i$ und alle $t = 1, \ldots, k$ berechnet, also $i \cdot j \cdot k$ Multiplikationen und Additionen.

Falls wir nun mehrere Matrizen multiplizieren wollen, also $E = A_1 \cdot A_2 \cdot \ldots \cdot A_n$ dann müssen auch hier die entsprechenden Matrizen gemäß der Größen *zusammenpassen*. Die Anzahl der Spalten von A_i muss der Anzahl der Zeilen von A_{i+1} entsprechen für $i = 1, \ldots, n-1$.

Die Matrixmultiplikation ist *assoziativ*, für drei Matrizen $A \in \mathbb{R}^{i \times j}$, $A \in \mathbb{R}^{j \times k}$ und $C \in \mathbb{R}^{k \times l}$ gilt also $(A \cdot B) \cdot C = A \cdot (B \cdot C) \in \mathbb{R}^{i \times l}$. Deshalb lassen sich auch allgemeine (zusammenpassende Ketten) beliebig Klammern. Diese Klammerung hat Einfluss auf die Anzahl der Floating-Point Operationen.

Beispiel: Für $A \in \mathbb{R}^{10 \times 50}$, $A \in \mathbb{R}^{50 \times 10}$ und $C \in \mathbb{R}^{10 \times 50}$, benötigt $(A \cdot B) \cdot C$ zuerst $10 \times 50 \times 10$ Operationen für $(A \cdot B)$ und danach $10 \times 10 \times 50$ Operationen für die Multiplikation des Ergebnisses mit C also insgesamt 10000 Operationen. Dahingegen benötigt $A \cdot (B \cdot C)$ zuerst $50 \times 10 \times 50$ Operationen für $(B \cdot C)$ und danach $10 \times 50 \times 50$ Operationen für die Multiplikation des Ergebnisses mit A also insgesamt 50000 Operationen. Die Klammerung $(A \cdot B) \cdot C$ ist also deutlich günstiger.

Allgemeine Problembeschreibung: Bestimme für die Berechnung von $E = A_1 \cdot A_2 \cdot \ldots \cdot A_n$ mit Dimensionen $d_0 \times d_1, d_1 \times d_2, \ldots, d_{n-1} \times d_n$ die optimale Klammerung für die minimale Anzahl an Operationen.

Dieses Problem wollen wir durch dynamische Programmierung lösen. Deshalb muss zunächst die Lösung rekursiv durch Teillösungen angegeben werden. Falls $M(n)$ alle Möglichkeiten beschreibt, wie man n Matrizen klammert, dann ist $M(n) = \sum_{i=1}^{n-1} M(k) \cdot M(n-k)$ mit $M(1) = 1$. Diese Rekursionsgleichung liegt in $\Omega(2^n)$. Es ist also nicht zweckmäßig, alle Möglichkeiten auszuprobieren.

Übungsaufgabe: Verifizieren Sie die Lösung $M(n) \in \Omega(2^n)$.

Wir wollen nun optimale Teillösungen wiederverwenden. Sei nun $K(l, k)$ der minimale Aufwand bzw. die optimale Klammerung für $A_l \cdot A_{l+1} \cdot \cdots \cdot A_k$. Zum Beispiel zerlegt sich zu Beginn ein erster Multiplikationsschritt für die Stelle s, d.h. der minimale Aufwand für $(A_1 \cdot A_2 \cdot \cdots \cdot A_s) \cdot (A_{s+1} \cdot A_{s+2} \cdot \cdots \cdot A_n)$, rekursiv in die Kosten

$$K(1, s) + K(s, n) + d_0 \cdot d_s \cdot d_n \tag{4.1}$$

Das Ziel ist nun, das optimale s zu finden, und das wollen wir rekursiv beschreiben.

- Offensichtlich ist $K(l, l) = 0$ für $l = 1, \ldots, n$

- Die Dimension eine Teilkette $A_l \cdot A_{l+1} \cdot \cdots \cdot A_k$ ist $d_{l-1} \times d_k$.
- Die Kosten für ein Teilen bei s sind $K(l,s) + d_{l-1} \cdot d_s \cdot d_k + K(s+1,k)$

Da wir $K(1,n)$ minimieren wollen erhalten wir insgesamt:

$$K(l,k) = \begin{cases} 0 & \text{falls } l = k \\ \min_{l \le s \le k-1} K(l,s) + d_{l-1} \cdot d_s \cdot d_k + K(s+1,k) & \text{falls } l < k \end{cases} . \tag{4.2}$$

Wir stellen fest, dass dynamische Programmierung geeignet ist:

- Wiederverwendung: Hier werden viele Teilprobleme mehrfach benutzt, zum Beispiel $K(1,2)$ wird verwendet von $K(1,3), K(1,4), \ldots, K(1,n)$.
- Beschränkte Anzahl: Für alle $1 \le l < k \le n$ gibt es genau ein Teilproblem, also insgesamt $\Theta(n^2)$ viele.
- Bearbeitungsreihenfolge: Falls alle Werte $K(l,s)$ und $K(s+1,k)$ bekannt sind, kann $K(l,k)$ in Zeit $O(k-l) \in O(n)$ berechnet werden.

Die geschickte Auswertung läßt sich sehr gut durch eine Matrix beschreiben. Wir verwenden ein Array $K[1..n, 1..n]$.

Zuerst werden die Diagonalen eingetragen, dann berechnen wir sukzessive die Nebendiagonalen von rechts unten nach links oben, zu jedem Zeitpunkt sind für $K(l,k)$ alle Werte $K(l,s)$ und alle Werte $K(s+1,k)$ für $l \le s$ und $s+1 \le k$ berechnet worden; siehe Abbildung 4.2.

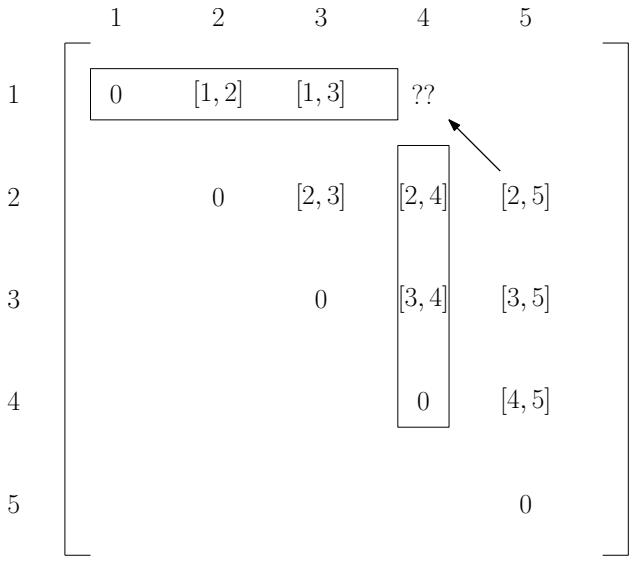


Abbildung 4.2: Die Nebendiagonalen können von unten nach oben berechnet werden. Nachdem zwei Nebendiagonalen bereits berechnet wurden beginnen wir die Berechnung der dritten Nebendiagonalen. Die benötigten Werte liegen bereits vor.

Korrektheit: Am Ende der Auswertung wird der Wert $[1,n]$ die minimale Anzahl an Operationen enthalten, wir haben den Wert rekursiv genauso festgelegt. Wir wollen uns natürlich auch die optimale Klammerung merken. Dazu müssen wir uns nur für jeden berechneten Matrixeintrag den jeweils optimalen Index $s[l,k]$ merken. Falls beispielsweise der optimale Index für die Berechnung von $[1,5]$ bei $s = 2$ liegt, ist die Klammerung $(A_1A_2)(A_3A_4A_5)$ rekursiv

die Beste und wir speichern den Wert $s[1,5] = 2$. Die optimale Klammerung von (A_1A_2) zur Berechnung von $[1,2]$ ist trivialerweise S= 1 also $s[1,2] = 1$. Rekursiv haben wir auch die optimale Klammerung aus $[3,5]$ gespeichert, diese kann $s = 3$ oder $s = 4$ sein, somit ist $s[3,5] \in \{3,4\}$. Insgesamt erhalten wir so die optimale Klammerung.

Laufzeitanalyse und Speicherplatz: Insgesamt wird Speicherplatz von $O(n^2)$ für beide Matrizen (Eintraäge $K[l,k]$ und $s[l,k]$) verwendet. Für die Berechnung eines Eintrages $K[l,k]$ und der Klammerung $s[l,k]$ wird ein Aufwand von $O(k-l) \in O(n)$ benötigt. Da wir $O(n^2)$ viele Aufrufe haben, ergibt sich ein Gesamtaufwand von $O(n^3)$.

Übungsaufgabe: Formulieren Sie einen Pseudocode, der den obigen Algorithmus beschreibt.

4.3 Rucksackproblem

Wir betrachten das Problem des optimalen Befüllens eines Rucksackes mit Gesamtgewichtskapazität G. Dazu sei eine Menge von n Gegenständen a_i mit Gewicht g_i und Wert w_i gegeben. Wir wollen den Rucksack optimal füllen.

Aufgabenstellung: Bestimme eine Teilmenge $\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\}$ der Elemente aus $\{a_1, \ldots, a_n\}$, so dass $\sum_{j=1}^{k} g_{i_j} \leq G$ gilt und $\sum_{j=1}^{k} w_{i_j}$ maximal ist. Bestimme also den Wert:

$$w_{\max} = \max_{\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\} \subseteq \{a_1, \ldots, a_n\}} \left\{ \sum_{j=1}^{k} w_{i_j} \,\middle|\, \sum_{j=1}^{k} g_{i_j} \leq G \right\}.$$

Wenn wir dieses Problem rekursiv lösen wollen, können wir naiv einen Rekursionsbaum angeben, der in jeder Ebene i die Entscheidung für den i-ten Gegenstand berücksichtigt; siehe Abbildung 4.3. Dadurch werden alle möglichen Fälle beschrieben. Das ist aber nicht effizient, da der Baum eine exponentielle Größe.

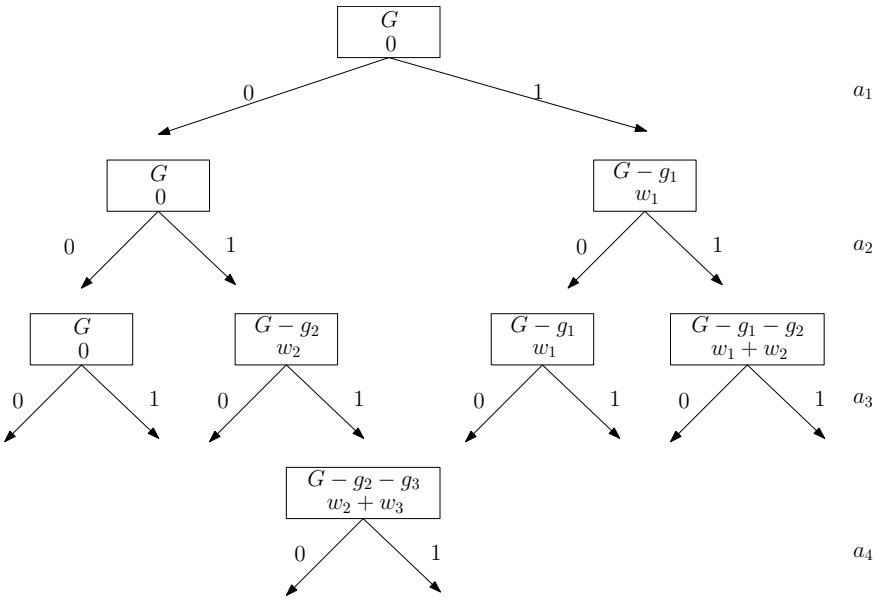


Abbildung 4.3: In Ebene i wird die Entscheidung für den i-ten Gegenstand getroffen. Der Baum hat exponentielle Größe.

Wir betrachten deshalb die folgende Vereinfachung und wollen Teilergebnisse verwenden. Sei dazu $W(i,j)$ der optimale Wert falls nur die ersten i Gegenstände betrachten werden und der

Rucksack die Kapazität j hat. Am Ende soll $w_{\max} = W(n, G)$ berechnet werden.

Wir können $W(i, j)$ rekursiv wie folgt angeben. Die Frage stellt sich, ob bei Lösungen mit den ersten $i-1$ Gegenständen der i-te Gegenstand hinzugenommen werden soll oder nicht. Falls nicht, bleibt der Wert bei $W(i-1, j)$ stehen. Falls ja, kommt der Wert w_i hinzu. Allerdings müssen wir zur Berechnung von $W(i, j)$ dann rekursiv auf $W(i-1, j-g_i)$ zurückgreifen, $W(i-1, j-g_i)+w_i$ ist dann die Lösung. Der maximale Wert aus diesen beiden Fällen ergibt den Wert $W(i, j)$.

Falls j negativ wird, ist das Ergebnis ungültig. Das drücken wir durch $W(i, j) = -\infty$ aus. Falls i null wird, setzen wir den Wert $W(i, j)$ auf null, da kein Gegenstand verwendet wird.

Wir kommen zu folgender Rekursionsgleichung, die wir danach Bottom-Up lösen wollen.

$$W(i,j) = \begin{cases} 0 & \text{falls } i = 0 \\ -\infty & \text{falls } j < 0 \\ \max\{W(i-1,j), W(i-1,j-g_i)+w_i\} & \text{sonst} \end{cases} \tag{4.3}$$

Nun wollen wir wiederum die Werte $W(i, j)$ in einem Array $W[0..n, 0..G]$ berechnen und initialisieren $W[0, j]$ und $W[i, 0]$ mit 0. Aus (4.3) ergibt sich offensichtlich, dass wir die Werte am besten Zeilenweise von links nach rechts und mit steigender Zeilenzahl berechnen. Zur Berechnung von $W(i, j)$ muss $W(i-1, j)$ und $W(i-1, j-g_i)$ vorliegen.

Beispiel: Für $n = 5$ notieren $a_i = (g_i, w_i)$ mit $a_1 = (6, 4)$ $a_2 = (1, 2)$, $a_3 = (2, 3)$, $a_4 = (5, 8)$ und $a_5 = (9, 10)$ mit $G = 12$.

Wir beschreiben einen Teilausschnitt der zugehörigen Matrix während der Berechnung. Zur Berechnung von $W(2, 7)$ wird $W(1, 7) = 4$ (ohne a_2) und $W(1, 7-g_2)+w_2 = W(1, 6)+2 = 4+2 = 6$ (mit a_2) verglichen. $W(1, 7)$ ist also gleich 6. Zeilenweise wird die Matrix von links nach rechts gefüllt.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	4	4	4	4	4	4	4
2	0	2	2	2	2	2	4	$W(2,7)$?	x	x	x	x	x
3	0	x	x	x	x	x	x	x	x	x	x	x	x
4	0	x	x	x	x	x	x	x	x	x	x	x	x
5	0	x	x	x	x	x	x	x	x	x	x	x	x

Auch für diesen Fall wollen wir uns am Ende die optimale Lösung generieren können. Dazu merken wir uns in einer zweiten Matrix $L[0..n, 0..G]$, woraus $W(i, j)$ entstanden ist. Zum Beispiel merken wir uns für $W(1, 7)$ das a_2 benutzt wird und der Plan aus $L(1, 6)$. Diese Information wird in $L(2, 7)$ gespeichert, zum Beispiel durch $L[2, 7] := (a_2, [1, 6])$. In $L(1, 6)$ steht dann wiederum, dass a_1 benutzt wird.

Korrektheit: Offensichtlich läßt sich nach Berechnung von $W(n, G)$ und $L(n, G)$ der Lösungsplan rekursiv angeben und das optimale Packen ist berechnet worden. Die Lösung ist rekursiv genauso formuliert worden.

Wir geben hier nochmal explizit die Formulierung im Pseudocode in Algorithmus 7 an und analysieren dann die Korrektheit und die Laufzeit.

Laufzeit und Speicherplatz: In Algorithmus 7 werden nach der Initialisierung der Werte $W[i, 0]$ und $W[0, j]$ zwei FOR-Schleifen geschachtelt, wobei die Äußere von 1 bis n läuft und die Innere von 1 bis G. Im Inneren ist der Aufwand $\Theta(1)$. Also ist der Aufwand insgesamt in $O(n \cdot G)$ und für das Array benötigen wir $O(n \cdot G)$ Speicherplatz.

Algorithm 7 RucksackDynProg($a_i = (g_i, w_i), i = 1, \ldots n$, Gesamtgew.: G)

```
1: for i = 0 to G do
2:    W[0, i] := 0;
3: end for
4: for j = 0 to n do
5:    W[j, 0] := 0;
6: end for
7: for i = 1 to n do
8:    for j = 1 to G do
9:       if (j − g_i) < 0 then
10:         W[i, j] := W[i − 1, j];    // a_i passt gar nicht rein
11:      else
12:         W[i, j] := max{W[i − 1, j], W[i − 1, j − g_i] + w_i};
13:      end if
14:   end for
15: end for
16: RETURN W[n, G]
```

Übungsaufgabe: Füllen Sie die Matrizen $W[0..5, 0..12]$ und $L[0..5, 0..12]$ für das obige Beispiel. Formulieren Sie die algorithmische Lösung im Pseudocode auch für die Belegung der Matrix L zur Rekonstruktion der optimalen Lösung.

Bemerkungen: Der obige Algorithmus ist für ganzzahlige Werte entworfen worden und die Laufzeit hängt polynomiell von der Größe der Rucksacks ab. Für nicht-ganzzahlige Werte ist das Problem deutlich schwerer zu lösen. Es ist nach heutiger Erkenntnis nicht zu erwarten, dass ein polynomieller Algorithmus für das allgemeine Rucksackproblem mit nicht-ganzzahligen Werten existiert.

Kapitel 5

Greedy Algorithmen

Neben den bereits behandelten klassischen Entwurfsmethoden der Algorithmik gibt es eine weitere Methode, die sich auf die intuitive Idee stütz, dass lokal optimale Berechnungsschritte auch global optimal sind. Es geht also auch in diesem Kapitel um Optimierungsprobleme. Der lokal optimale Berechnungsschritt soll dabei möglichst nicht wieder rückgängig gemacht werden. Außerdem werden keine Alternativen ausprobiert. In jedem Schritt wird der nächste lokal optimale Schritt berechnet und tatsächlich ausgeführt. In diesem Sinne nennt man diese Vorgehensweise auch *iterativ*. Die Kosten für einen *Iterationsschritt* sollen klein gehalten werden. Das ist meistens möglich, da wir zunächst gar nicht auf die Gesamtlösung achten. Zur Bestimmung des nächsten Schrittes wird allerdings häufig eine Liste von Kandidaten für den nächsten Schritt vorgehalten. Diese Liste wird nach dem Schritt aktualisiert. Greedy-Verfahren sind dennoch in der Regel einfach zu beschreiben und benötigen wenig Rechenaufwand.

Dieses *Greedy*-Prinzip wird leider nicht immer die optimale Lösung erzielen, drei Szenarien sind denkbar:

1. Wir erhalten die optimale Lösung.
2. Wir erhalten nicht die optimale Lösung, aber wir können zumindest beweisen, dass die Lösung im Vergleich zum Optimum nicht beliebig schlecht ist.
3. Die Lösung ist im Vergleich zum Optimum beliebig schlecht.

Zusammengefasst ergibt sich folgendes Grundprinzip der Greedy-Vorgehensweise:

- Lokal optimale Schritte werden iterativ ausgeführt.
- Eine Kandidatenliste wird dafür bereithalten und ggf. aktualisiert.
- Es werden keine Alternativen ausprobiert oder Zwischergebnisse tabellarisch abgespeichert.
- Die Kosten für den nächsten Schritt sind nicht sehr hoch.
- Die Lösung kann je nach Aufgabenstellung optimal, beliebig schlecht oder approximativ sein.

5.1 Rucksackproblem

Das bereits bekannte Rucksackproblem aus dem letzten Abschnitt läßt sich leicht in Greedy-Manier formulieren. Lokal optimal erscheint dabei, dass wir den Gegenstand mit dem jewei-

ligen besten Preis/Gewichtverhältnis als erstes Einfügen.

Beispiel: Für $n = 5$ notieren $a_i = (g_i, w_i)$ mit $a_1 = (6, 4)$ $a_2 = (1, 2)$, $a_3 = (2, 3)$, $a_4 = (5, 8)$ und $a_5 = (9, 10)$ mit $G = 12$. Die optimale Lösung ist hier offensichtlich (a_5, a_2, a_3) mit einem Wert von 15.

Zur Erinnerung, die Gegenstände haben ein Gewicht von g_i und einen Wert von w_i. Deshalb bestimmen wir zunächst eine Kandidatenliste gemäß des Verhältnisses $v_i = \frac{w_i}{g_i}$. Wir haben $v_1 = \frac{4}{6} = \frac{2}{3}$, $v_2 = \frac{2}{1} = 2$, $v_3 = \frac{3}{2} = 1.5$, $v_4 = \frac{8}{5}$ und $v_5 = \frac{10}{9}$. Das ergibt folgende Sortierung $v_2 > v_4 > v_3 > v_5 > v_1$. Ein Greedy Algorithmus wählt sukzessive den aktuell besten Kandidaten und revidiert diese Entscheidung nicht. Für den Algorithmus 8 nehmen wir zur einfachen Beschreibung an, dass die Gegenstände nach dem Preis/Gewichtsverhältnis sortiert vorliegen, also $v_1 \geq v_2 \geq \ldots \geq v_n$.

Algorithm 8 RucksackGreedy($v_i = \frac{w_i}{g_i}$ nach Größe sortiert, Gewicht: G)

```
1: for i = 1 to n do
2:    if g_i ≤ G then
3:       λ_i := 1; G := G − g_i;
4:    else
5:       λ_i := 0;
6:    end if
7: end for
8: RETURN Σ_{i_1}^{n} λ_i w_i
```

Mit einer entsprechenden Umbenennung bedeutet das in unserem Beispiel, der Algorithmus sukzessive a_2 und a_4 und a_3 auswählt und dann die Kapazität $12 - 1 - 5 - 2 = 4$ verbleibt. Nun kann a_5 und a_1 nicht mehr verwendet werden. Insgesamt ergibt sich ein Gesamtwert von $2 + 8 + 3 = 13$. Der Greedy-Algorithmus ist also nicht optimal.

Schlimmer noch, wir können ein ganz einfaches Beispiel angeben, indem der Greedy-Algorithmus beliebig schlecht wird.

Worst-Case Beispiel: Sei $a_1 = (1, 1)$ und $a_2 = (G, G - 1)$. Dann ist $v_1 = 1 > \frac{G-1}{G} = v_1$. Der Greedy Algorithmus legt $\lambda_1 = 1$ und $\lambda_2 = 0$ fest und das Ergebnis ist 1. Optimal wäre $\lambda_1 = 0$ und $\lambda_2 = 1$ mit Ergebnis $G - 1$. Da G beliebig groß werden kann, wird das Verhältnis zwischen der Greedy-Lösung und der optimalen Lösung $\frac{G}{1}$ beliebig groß.

Wir haben jetzt ein Beispiel gesehen, in dem eine Greedy-Vorgehensweise nicht empfehlenswert ist, da wir keine Garantie für die Güte abgeben können. Wir möchten eine obere Schranke für das Verhältnis von *Lösung_mit_Greedy(P)* und *Optimale_Lösung(P)* bei einem Optimierungssproblem für alle Probleme $P \in \Pi$ für eine Problemklasse Π angeben. Das Verhältnis soll durch eine Konstante $C \geq 1$ angegeben werden, die besagt, dass die Greedy-Lösung höchstens C mal schlechter ist als die optimale Lösung. Deshalb betrachten wir verschiedene Quotienten wie folgt:

Maximierungsproblem: Gilt

$$\forall P \in \Pi \quad \frac{\mathit{Optimale\_Lösung(P)}}{\mathit{Lösung\_mit\_Greedy(P)}} \leq C,$$

für eine Konstante C, so sagen wir, dass die Greedy-Lösung eine C-Approximation liefert.

Minimierungssproblem: Gilt

$$\forall P \in \Pi \quad \frac{\mathit{Lösung\_mit\_Greedy(P)}}{\mathit{Optimale\_Lösung(P)}} \leq C,$$

für eine Konstante C, so sagen wir, dass die Greedy-Lösung eine C-Approximation liefert.

5.2 Das optimale Bepacken von Behältern

Das Problem des optimalen Bepackens von mehreren Behältern gleicher Gewichtskapazität G mit verschiedenen Paketen (Binpacking) gestaltet sich wie folgt. Die Pakete p haben keinen Preis sondern lediglich verschiedene Gewichte g. Insgesamt haben wir zunächst n Behälter b_j für $j = 1, \ldots, n$ mit Kapazität G und n Gegenstände p_i mit Gewichten g_i für $i = 1, \ldots, n$. Wir können $g_i \leq G$ annehmen, sonst brauchen wir einen Gegenstand gar nicht zu betrachten.

Ziel der Aufgabenstellung ist es, die Anzahl der benötigten Behälter zu minimieren. Im Prinzip bestimmen wir eine Zuordnung $Z : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$ die jedem p_i einen Behälter b_j über $Z(i) = j$ zuweist. Wir suchen das minimale k, für das es eine gültige Zuordnung Z gibt. Gültig heißt, dass die Summe aller g_i mit $Z(i) = j$ nicht größer als G ist.

Aufgabenstellung: Minimiere k, so dass eine Zuordnung $Z : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$ existiert mit

$$\sum_{Z(i)=j} g_i \leq G \text{ für alle } i \in \{1, \ldots, n\}\,. \tag{5.1}$$

Für eine Greedystrategie benutzen wir hier keine besondere Kandidatenliste, wir nehmen die Pakete so wie sie gegeben sind in der Reihenfolge $a_1, a_2, \ldots, a_n$. Die Annahme ist wirklich sinnvoll, da wir uns vorstellen können, dass die Pakete so vom Band kommen oder so geliefert werden und zugeordnet werden müssen.

Wir berücksichtigen nicht, welche Pakete in der Zukunft auftreten. Die einzige Entscheidung die wir treffen, ist also, in welchen Behälter b_j legen wir das Paket a_i nachdem bereits die Pakete $a_1, a_2, \ldots, a_{i-1}$ einsortiert wurden. Der maximale Index j soll klein gehalten werden. Naheliegend sind die folgenden beiden Vorgehensweisen. Angenommen, wir haben bereits $a_1, a_2, \ldots, a_{i-1}$ Pakete zugeordnet. Platziere nun a_i.

First-Fit: Unter allen Behältern wähle den ersten aus, in den a_i noch hineinpasst. Genauer: Finde kleinstes j, so dass $g_i + \sum_{1 \leq l \leq (i-1), Z(l)=j} g_l \leq G$ gilt.

Best-Fit: Packe a_i in die aktuell vollste Kiste, in die es noch hinein passt. Genauer: Finde ein j, so dass $g_i + \sum_{1 \leq l \leq (i-1), Z(l)=j} g_l \leq G$ gilt und dabei $\sum_{1 \leq l \leq (i-1), Z(l)=j} g_l$ maximal ist.

Beispiel: Betrachten wir Behälter der Kapazität $G = 10$ und 7 Gegenstände der Größen $g_1 = 2$, $g_2 = 5$, $g_3 = 4$, $g_4 = 7$, $g_5 = 1$, $g_6 = 3$ und $g_7 = 8$ dann belegt First-Fit die Behälter wie in Abbildung 5.1 und benötigt 4 Behälter. Optimal wäre es gewesen, drei Behälter jeweils mit (g_1, g_7), (g_2, g_3, g_5) und (g_4, g_6) zu belegen. Beide Algorithmen lassen sich mit einer Laufzeit von $O(n^2)$ realisieren. Beispielsweise wird bei First-Fit für alle Behälter j der aktuelle Belegungswert gespeichert und wir überprüfen sukzessive von $j = 1, \ldots, n$ in welchen Behälter der Gegenstand a_i noch hineinpasst. Dieses Verfahren wird wiederum sukzessive für $i = 1, \ldots, n$ für die Gegenstände a_i durchgeführt. Insgesamte werden zwei FOR-Schleifen geschachtelt, die von $j = 1, \ldots, n$ respektive von $i = 1, \ldots, n$ laufen. Ähnlich kann bei Best-Fit vorgegangen werden.

Wir wollen nun zeigen, dass beide Algorithmen eine 2-Approximation der optimalen Anzahl der benötigten Behälter liefern. Wir gehen auch hier der Einfachheit halber davon aus, dass die Gegenstände ganzzahliges Gewicht haben.

Lemma 6 *Für das Binpacking Problem liefern die Algorithmen First-Fit und Best-Fit eine 2-Approximation für die optimale (minimale) Anzahl an benötigten Behältern.*

Beweis. Die Beweisidee stütz sich auf folgende Beobachtung. Nach Anwendung des jeweiligen Algorithmus gilt: Für je zwei *benutzte* Behälter gilt, dass das Gesamtgewicht der enthaltenen Gegenstände mindestens $G + 1$ ist.

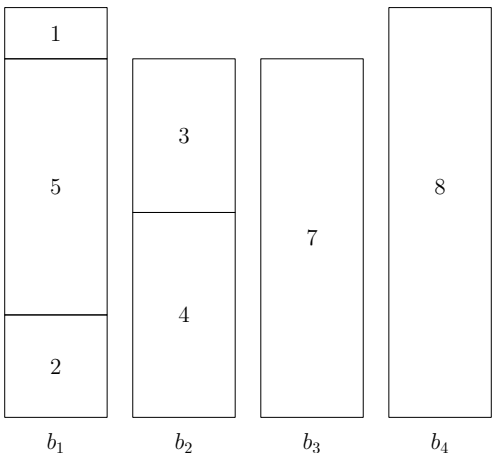


Abbildung 5.1: First-Fit für $g_1 = 2$, $g_2 = 5$, $g_3 = 4$, $g_4 = 7$, $g_5 = 1$, $g_6 = 3$ und $g_7 = 8$ benötigt einen Behälter zu viel.

Zum Beweis dieser Aussage führen wir einen Widerspruchsbeweis. Falls die obige Bedingung bei einer Belegung durch First-Fit bzw. Best-Fit nicht eintritt, ist die Belegung gar nicht durch den jeweiligen Algorithmen entstanden, siehe auch das Beispiel in 5.2.

Angenommen die obigen Bedingung stimmt nach der Ausführung einer der beiden Algorithmen nicht, dann gibt es zwei Behälter b_l und b_k mit $l < k$, so dass die Gegenstände von b_k vollständig in b_l gepasst hätten und umgekehrt.

First-Fit hätte einen Gegenstand a_i aus b_k in ein b_j mit $j < k$ einfügen müssen, egal zu welchem Zeitpunkt a_i aufgetreten ist. Zumindest b_l mit $l < k$ war ein Kandidat dafür. Die Belegung ist somit sicher nicht durch First-Fit entstanden. Ein Widerspruch.

Best-Fit kann ebenfalls nicht ausgeführt worden sein. Zwischen den Alternativen b_l und b_k benutzt Best-Fit stets den Behälter mit dem höchsten Füllstand. Falls zuerst b_l das erste Paket von beiden Behältern überhaupt erhält, kann der Algorithmus die weiteren Pakete unter dem Best-Fit Aspekt nur noch diesem Behälter (oder einem anderen Behälter $\neq b_k$) zuordnen. Der Behälter b_k erhält jedenfalls am Ende gar kein Paket. Ein Widerspruch zu der Annahme, dass überhaupt beide Behälter benutzt wurden.

Jetzt zeigen wir unter dieser Annahme, die 2-Approximation. Sei dazu g das kleinste Gewicht eines Behälters nach der Ausführung des jeweiligen Algorithmus und sei k_{opt} die optimale Anzahl an Behältern und k die Anzahl der jeweils verwendeten Behälter.

Fall 1, $g \geq \frac{G}{2}$**:** Die Gesamtbeladung, GB, ist $\geq k \times \frac{G}{2}$. Für k_{opt} gilt stets $k_{\text{opt}} \geq \frac{\text{GB}}{G}$. Daraus ergibt sich:

$$k_{\text{opt}} \geq \frac{k \times G/2}{G} = \frac{k}{2}\,.$$

Fall 2, $g < \frac{G}{2}$**,** $k = 1$**:** Dann gilt $k_{\text{opt}} = k$.

Fall 3, $g < \frac{G}{2}$**,** $k > 1$**:** Gemäß der obigen Aussage kann es nur einen einzigen Behälter mit Füllmenge $g < \frac{G}{2}$ geben. Dann sind $(k-1)$ Behälter mindestens bis $\frac{G}{2}$ gefüllt. Die Gesamtbeladung, GB, ist echt größer als $(k-1) \times \frac{G}{2}$. Für k_{opt} gilt dann $k_{\text{opt}} \geq \frac{\text{GB}}{G}$.

Daraus ergibt sich: $k_{\text{opt}} > \frac{(k-1)\times G/2}{G}$, woraus

$$k_{\text{opt}} \geq \frac{k}{2}$$

wegen der Ganzzahligkeit gefolgert werden kann.

□

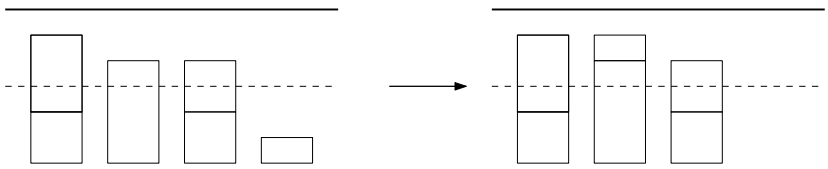


Abbildung 5.2: Nach Anwendung von First-Fit und Best-Fit haben je zwei Behälter ein Gesamtgewicht von $G + 1$.

Beide Algorithmen haben also eine beweisbare Güte. Sie sind in jedem Fall nicht schlechter als zweimal das Optimum.

Bemerkungen:

- Der obige Beweis läßt sich auch für nicht-ganzzahlige Gewichte führen. Die Hilfsaussage im Beweis von Lemma 6 lautet dann: *Für je zwei benutzte Behälter gilt, dass das Gesamtgewicht der enthaltenen Gegenstände mindestens $G+\epsilon$ für ein $\epsilon > 0$ ist.* Übungsaufgabe: Führen Sie den Beweis insgesamt für nicht-ganzzahlige Gewichte.
- Für beide Algorithmen läßt sich durch genauere Analyse ein Approximationsfaktor von $C \leq \frac{17}{10}$ beweisen.

Wir wollen abschließend auch noch eine untere Schranke für den Approximationsfaktor der beiden Algorithmen angeben. Das heißt, wir konstruieren eine Sequenz, die zu einem bestimmtem Faktor führt. Wir wollen dabei auch zeigen, dass das Ergebnis für Problemstellungen gilt, die beliebig viele Behälter benötigen.

Lemma 7 *Für das Binpacking Problem und die Algorithmen First-Fit und Best-Fit gibt es Beispiel-Sequenzen, die beliebig lang sein können und für die die Lösung aus den beiden Algorithmen niemals besser sein kann als $\frac{5}{3}$ mal die optimale Lösung.*

Beweis. Sei dazu $n = 18m$ und $G = 131$. Gegenstände mit Gewicht 20 plus Gewicht 45 plus Gewicht 66 füllen genau einen Behälter.

Insgesamt betrachten wir $18m$ Gegenstände, wobei in der Sequenz der Gegenstände

1. ... die ersten $6m$ ein Gewicht $g_i = 20$ für $i = 1, \ldots, 6m$,
2. ... die nächsten $6m$ ein Gewicht von $g_i = 45$ für $i = 6m + 1, \ldots, 12m$,
3. ... und die letzten $6m$ ein Gewicht von $g_i = 66$ für $i = 12m + 1, \ldots, 18m$

besitzen. Wir haben also drei Sequenzblöcke 1., 2. und 3. und exakt $6m$ Behälter reichen aus.

Wir zeigen, dass für diese Sequenz First-Fit und Best-Fit jeweils $10m$ Behälter befüllen. Beide Algorithmen füllen für den ersten Sequenzblock sukzessive die ersten m Behälter mit

Gewichten $6 \times 20 = 120$. Jeder Block hat eine Restkapazität von 11. Danach werden für den zweiten Block $3m$ Behälter mit jeweils 2×45 Gewicht und Restkapazität von 41. Danach werden $6m$ Behälter mit Gewicht 66 gefüllt und Restkapazität jeweils 65.

Insgesamt ergibt sich in beiden Fällen ein Quotient von $\frac{10m}{m} = 10$. □

Wenn die Sequenzen aus dem obigen Beweis in absteigend sortierter Reihenfolge vorgelegen hätten, dann hätten beide Algorithmen die optimale Belegung mit $6m$ Behältern erzielt.

5.3 Aktivitätenauswahl

Wir betrachten das Problem, dass eine Ressource für verschiedenen zeitlich beschränkte Aktivitäten benutzt werden soll. Die Aktivitäten können die Resource nicht zeitgleich benutzen. Es sollen aber soviel wie möglich Aktivitäten eingeplant werden. Beispielsweise soll die Belegung eines Konzertsaals für verschiedene Veranstaltungen geplant werden. Allgemein gibt es eine Menge von n Aktivitäten $a_i = (s_i, t_i)$ die jeweils durch einen Starttermin, s_i, und einen Endtermin, t_i, mit $s_i < t_i$ bestimmt sind.

Zwei Aktivitäten a_i und a_j sind kompatibel, wenn die Intervalle $[s_i, t_i)$ und $[s_j, t_j)$ nicht überlappen.

Aufgabenstellung Aktivitäten-Auswahl: Für eine Menge von n Aktivitäten $a_i = (s_i, t_i)$ für $i = 1, \ldots, n$ finde die maximale Menge gegenseitig kompatibler Aktivitäten. Mit maximal ist hier die Anzahl der einplanbaren Aktivitäten gemeint.

Zunächst ist klar, dass wir die Aktivitäten zwischen dem kleinsten Startzeitpunkt $S = \min\{s_i \mid i = 1, \ldots, n\}$ und dem größten Endzeit $T = \max\{t_i | i = 1, \ldots, n\}$ einplanen müssen. Eine naheliegende Greedy-Idee ist, die bislang nicht verwendete Zeit zu maximieren. Deshalb sortieren wir die Aktivitäten nach den Endzeitpunkten t_i. Nehmen wir an, dass wir die Folge a_i bereits so vorliegen haben, das heißt $t_1 \leq t_2 \leq \cdots \leq t_n$.

Greedy-Strategie: Wähle stets die Aktivität, die den frühesten Endzeitpunkt hat und noch legal eingeplant werden kann.

Beispiel:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
t_i	4	5	6	7	8	9	10	11	12	13	14

Hier sind zum Beispiel $\{a_3, a_9, a_{11}\}$ kompatibel, $\{a_1, a_4, a_8, a_{11}\}$ ist eine maximale kompatible Teilmenge, die durch die Greedy Strategie ausgewählt wird. Auch $\{a_2, a_4, a_9, a_{11}\}$ ist optimal bezüglich der Anzahl der einplanbaren Aktivitäten.

Algorithmisch beschreiben wir die Greedy-Strategie ganz einfach wie in Algorithmus 9.

Algorithm 9 AktivitätenGreedy($a_i = (s_i, t_i)$ nach t_i sortiert)

```
1: A_1 := {a_1}; last := 1;
2: for i = 2 to n do
3:    if s_i ≥ t_last then
4:       A_i := A_{i-1} ∪ {a_i}; last := i;
5:    else
6:       A_i := A_{i-1};
7:    end if
8: end for
9: RETURN A_n
```

Laufzeit: Es müssen lediglich vorab die Endzeitpunkte sortiert werden. Das kann in $O(n \log n)$ durchgeführt werden. Danach wird die Liste in $O(n)$ Zeit in der FOR-Schleife abgearbeitet.

Korrektheit: Der Algorithmus ist einfach und liefert eine Lösung, das ist ein Grundprinzip der Greedy-Algorithmen. Die Analyse, dass die Lösung optimal ist, kann manchmal sehr schwer sein. Im Folgenden versuchen wir die Optimalität formal zu beweisen.

Theorem 8 *Für das Aktivitäten-Auswahl-Problem berechnet der Greedy-Algorithmus 9 eine optimale Lösung.*

Beweis. Wir zeigen per Induktion, dass stets eine optimale Lösung A^* existiert, so dass für jedes i gilt, dass $A^* \cap \{a_1, a_2, \ldots, a_i\} = A_i$ ist. Sei $A^* = \{a_{i_1}, a_{i_1}, \ldots, a_{i_k}\}$ mit $i_1 < i_2 < \ldots < i_k$.

Induktionsanfang: Für $A_1 = \{a_1\}$ gilt, dass a_{i_1} nicht vor a_1 endet. Also ist auch $A^{*\prime} = \{a_1, a_{i_2}, \ldots, a_{i_k}\}$ eine optimale Lösung mit der gewünschten Eigenschaft.

Induktionsschritt: Nehmen wir an, die Aussage gilt bereits für $1, \ldots, i-1$.

Fall 1: Falls a_i nicht kompatibel zu A_{i-1} ist, ist a_i nicht in A_i enthalten. Da es eine optimale Lösung A^* gibt, die alle Elemente aus A_{i-1} enthält, ist diese Lösung ebenfalls nicht kompatibel zu a_i (enthält also a_i nicht) und die Aussage gilt.

Fall 2: Falls a_i kompatibel zu A_{i-1} ist und somit in A_i vorhanden ist, sei nun $B := A^* \setminus A_{i-1} = \{b_1, b_2, \ldots, b_j\}$, wobei die b_i's nach den Endzeitpunkten sortiert sind. A^* erfülle dabei die Bedingung für A_{i-1}, das heißt, A^* ist eine optimale Lösung, so dass $A^* \cap \{a_1, a_2, \ldots, a_{i-1}\} = A_{i-1}$ gilt. Nun gilt, dass b_1 nicht vor a_i enden kann und außerdem kompatibel zu A_{i-1} ist. Dann können wir b_1 durch a_i ersetzen. Es liegt eine Situation wie in Abbildung 5.3 vor. Somit ist auch $A^* = A_{i-1} \cup \{a_i\} \cup \{b_2, \ldots, b_j\}$ eine optimale Lösung mit der angegebenen Bedingung für A_i.

Für $i = n$ erhalten wir somit durch A_n eine optimale Lösung, da es eine optimale Lösung A^* gibt, die genau aus den Elementen von A_n besteht. □

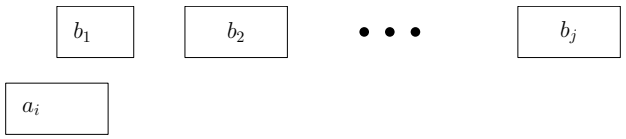


Abbildung 5.3: Nach Annahme gilt $A^* \cap \{a_1, a_2, \ldots, a_{i-1}\} = A_{i-1}$. Falls a_i kompatibel mit A_{i-1} ist, kann b_1 in $B := A^* \setminus A_{i-1} = \{b_1, b_2, \ldots, b_j\}$ durch a_i ersetzt werden.

Insegesamt haben wir drei Greedy Beispiele kennengelernt, in denen entweder eine optimale Lösung, eine gute Approximation oder eine beliebig schlechte Lösung erzielt wird.

Kapitel 6

Datenstrukturen

Nachdem wir einige algorithmische Standardtechniken betrachtet haben, wollen wir uns nun mit der geschickten Repräsentation von Daten im Rechner beschäftigen und stellen ein paar einfache Datenstrukturen vor.

Die einfache Datenstruktur des Arrays haben wir bereits kennengelernt, für ein Array $A[1..n]$ können wir die Werte $A[i]$ in $O(1)$ Zeit auslesen oder ändern. Arrays stehen in allen gängigen Programmiersprachen zur Verfügung, über die interne Realisierung müssen wir uns zunächst keine Gedanken machen. Wie bereits gesehen, lassen sich die Arrays auch in mehreren Dimensionen verwenden.

Im Abschnitt 2 haben wir gesehen, wie ein Array A mit n Werten sortiert werden kann. Den direkten Zugriff über $A[i]$ kann man zum Beispiel nutzen, um effizient festzustellen, ob ein Element x im Array $A[1..n]$ vorhanden ist. Falls wir die Einträge eines sortierten Arrays A sukzessive mit x vergleichen, liegt die Laufzeit dieser Vorgehensweise in $O(n)$.

Das Prinzip der *binären Suche* erlaubt eine effizientere Lösung. Dabei wird der Wert x jeweils mit dem Wert des Arrays an der Stelle $\left\lfloor \frac{p+q}{2} \right\rfloor$ verglichen und je nach Ergebnis wird in den Teilbereichen weitergesucht. Algorithmus 10 beschreibt dieses Verfahren.

Algorithm 10 BinarySearch(A,p,r,x)
Ist x ein Element des aufsteigend sortierten Arrays A zwischen Index p und r?

```
1:  if p < r then
2:     q := ⌊(p + r)/2⌋;
3:     if x < A[q] then
4:        BinarySearch(A, p, q, x);
5:     else if x > A[q] then
6:        BinarySearch(A, q + 1, r);
7:     else
8:        RETURN A[q] = x
9:     end if
10: else if p = r then
11:    if A[q] = x then
12:       RETURN A[q] = x
13:    else
14:       RETURN x ∉ A
15:    end if
16: end if
```

Laufzeitabschätzung: Dieser rekursive Algorithmus wird mit BinarySearch$(A, 1, n, x)$ aufgerufen und testet für $n = 1$ in konstanter Zeit, ob $A[1] = x$ ist. Sonst wird rekursiv ein Array der Größe $\lceil \frac{n}{2} \rceil$ betrachtet und konstanter Aufwand c für die konstante Anzahl an zusätzlichen Operationen verwendet.

Die zugehörige Rekursionsgleichnung lautet $T(n) = T\left(\lceil \frac{n}{2} \rceil\right) + c$ und $T(n)$ liegt beweisbar in $O(\log n)$.

Korrektheit: Falls das Array nur einen Eintrag hat, testen wir, ob dieser dem Element x entspricht. Ansonsten betrachten wir das mittlere Element $q := \lfloor (p + r)/2 \rfloor$ des aufsteigend sortierten Arrays $A[p..r]$, falls $x < A[q]$ suchen wir im linken Teil, sonst im rechten Teil des Arrays. Falls beides nicht zutrifft, muss $x = A[q]$ gelten und wir geben das Element aus.

6.1 Stacks

Die oben beschriebenen Arrays lassen sich auch für die Beschreibung komplizierterer Strukturen verwenden. Wir betrachten hier einen sogenannten *Stack* auch Stapel oder Keller genannt.

Der Stack ist intern realisiert durch ein unendliches Array $S[1..\infty]$ mit einem ausgewiesenen Kopfelement $S[\text{Top}]$ für die Integer-Variable Top. Dabei ist $S[1], S[2], \ldots, S[\text{Top}]$ der aktuelle Stack und $S[\text{Top}]$ das Kopfelement, der Stack ist leer, falls $\text{Top} = 0$ gilt.

Extern stehen dem Benutzer folgende Operationen zur Verfügung.

- Push(x), ein neues Kopfelement x wird auf den Stack gelegt. Intern wird $\text{Top} := \text{Top}+1$ und $S[\text{Top}] := x$ gesetzt
- Pop(x), gibt das aktuelle Top-Element in x zurück und löscht es aus dem Stack. Intern wird $x := S[\text{Top}]$ und $\text{Top} := \text{Top} - 1$ gesetzt. Falls Top bereits 0 ist, wird berichtet, dass der Stack leer ist, extern beispielsweise über den Wert Nil.
- Top(x) gibt das aktuelle Kopfelement in x zurück. Intern über $S[\text{Top}]$.

Wir wollen auch hier ein Beispiel für die effiziente Verwendung von Stacks angeben. Nehmen wir an, die Entwicklung verschiedener Aktien verläuft über einen bestimmten Zeitraum $I = [0, d]$ jeweils linear, das heißt beschrieben durch einen Funktion $f_i(x) = a_i x + b_i$ für $x \in I$. Nun möchte ein Analyst für jedes einzelne $x \in I$ schnell ermitteln, welche Aktie den größten Wert erzielt.

Naiv müsste man für den jeweiligen Wert x alle n Funktionswerte $f_i(x)$ miteinander vergleichen und könnte dann den maximalen Wert ermitteln.

Besser wäre es, gleich die sogenannte *obere Kontur* aller Funktionen explizit anzugeben. Zu jedem Zeitpunkt beschreibt eine der Funktionen f_i die momentan maximalen Werte. Wenn wir die Übergänge der jeweiligen *maximalen* Funktionen speichern könnten, können wir für die jeweiligen Teil-Intervalle schnell den optimalen Wert angeben.

Aufgabenstellung: Wir wollen die Funktion

$$f_{\max} : x \mapsto \max\{f_1(x), f_2(x), \ldots, f_3(x)\}$$

explizit durch Intervalle und zugeörige Teil-Funktionen beschreiben. $f_{\max}$ bezeichnen wir als die *obere Kontur* von $f_1, f_2, \ldots, f_n$ über dem Intervall $I = [0, d]$, siehe auch Abbildung 6.1.

Der Einfachheit halber nehmen wir an, dass die Werte b_i paarweise verschieden sind. Zunächst sortieren wir die Funktionen an der Stelle $x = 0$ nach den Y-Koordinaten b_i. Nehmen wir nun an, das o.B.d.A. $b_1 > b_2 > b_3 > \cdots > b_n$ gilt.

Wir verwenden einen Stack, und der Stack soll am Ende sukzessive die maximalen Funktionen und die Übergänge enthalten.

Zunächst ist f_1 maximal und wir legen $(f_1, (0, b_1))$ auf den Stack als Kopfelement. Das ist die momentane obere Kontur der betrachteten Funktionen. Nun nehmen wir f_2 und betrachten den aktuellen Stack. Falls die beiden Funktionen einen Schnittpunkt $s(f_1, f_2) = (x_{f_1,f_2}, y_{f_1,f_2})$ besitzen, wechselt ab diesem Punkt das Maximum von f_1 zu f_2. Wir legen somit f_2 mit $\text{Push}((f_2, s(f_1, f_2)))$ auf den Stack, der Stack repräsentiert die aktuelle obere Kontur. Das unterste Element f_1 startet bei $(0, b_1)$ und wird durch das nächste Element f_2 bei $s(f_1, f_2))$ abgewechselt.

Nun betrachten wir f_3. Falls f_3 keinen Schnittpunkt mit f_2 bildet, kann f_3 ausgelassen werden, die Funktion liegt vollständig unterhalb von f_2. Falls jedoch f_3 mit f_2 einen Schnittpunkt $s(f_2, f_3) = (x_{f_2,f_3}, y_{f_2,f_3})$ entscheidet die Lage des Schnittpunktes über die aktuelle Kontur der beteiligten Funktionen.

Fall 1.: Liegt der Punkt $s(f_2, f_3)$ mit seiner X-Koordinate x_{f_2,f_3} links vom X-Wert x_{f_1,f_2} des Startpunktes $s(f_1, f_2)$ des aktuellen Top-Elementes f_2, so kann f_2 offensichtlich insgesamt gar nicht an der oberen Kontur teilnehmen. In diesem Fall wird f_2 durch $\text{Pop}(x)$ vom Stack genommen. Der gleiche Test wird mit dem nächsten Top-Element (hier f_1) wiederholt.

Fall 2.: Der Punkt $s(f_2, f_3)$ liegt mit seiner X-Koordinate x_{f_2,f_3} rechts vom X-Wert x_{f_1,f_2} des Startpunktes $s(f_1, f_2)$ des aktuellen Top-Elementes f_2. Dann wird mit $\text{Push}((f_3, s(f_2, f_3)))$ der nächste Abschnitt der Kontur auf den Stack ge*pusht* und der aktuelle Stack beschreibt tatsächlich die obere Kontor der beteiligten Funktionen mit den jeweiligen Übergängen.

In Abbildung 6.1 sehen wir, wie dieses Prinzip im Allgemeinen funktioniert. Nachdem bereits sukzessive $(f_1, (0, b_1))$, $(f_2, s(f_1, f_2))$, $(f_3, s(f_2, f_3))$, $(f_4, s(f_3, f_4))$ auf den Stack ge*pusht* wurden, wird nun f_5 betrachtet. Wir betrachten jeweils die Lage des Schnittpunktes mit dem Top-Element. Da die Schnittpunkte $s(f_4, f_5)$ und $s(f_3, f_5)$ jeweils links von den Punkten $s(f_3, f_4)$ respektive $s(f_2, f_3)$ liegen, werden f_4 und f_3 nacheinander vom Stack ge*poppt*. Dann wird mit dem dann aktuellen Top-Element $(f_2, s(f_1, f_2))$ der Schnittpunkttest mit f_5 durchgeführt. Der Test der Schnittpunkte $s(f_2, f_5)$ und $s(f_1, f_2)$ ergibt, dass $s(f_2, f_5)$ rechts von $s(f_1, f_2)$ liegt und somit $(f_5, s(f_2, f_5))$ auf den Stack ge*pusht* werden muss, $(f_2, s(f_1, f_2))$ bleibt erhalten. Der Stack gibt die aktuelle Kontur wieder.

Korrektheit: Wir wissen, dass sich zwei Geraden nur einmal schneiden können. Induktiv nehmen wir an, dass der aktuelle Stack die obere Kontur für die ersten $f_1, f_2, \ldots, f_{k-1}$ Geraden wiedergibt. Für das erste Element f_1 ist das der Fall.

Das aktuelle Top-Element $(f_j, s(f_i, f_j))$ des Stacks ist das letzte Stück der Kontur und wird mit der nächsten Geraden f_k verglichen.

Falls der Schnittpunkt $s(f_j, f_k)$ links von $s(f_i, f_j)$ liegt, kann f_j nicht zur oberen Kontur gehören. Rechts von $s(f_j, f_k)$ dominiert f_k die Gerade f_j, links von $s(f_i, f_j)$ dominiert die Gerade f_i die Gerade f_j. Die Push-Operation ist also in diesem Fall korrekt und das nächste Top-Element des Stacks wird mit f_k verglichen.

Falls der Schnittpunkt $s(f_j, f_k)$ rechts von $s(f_i, f_j)$ liegt, wird korrekterweise $(f_k, s(f_j, f_k))$ ans Ende der Kontur und in den Stack eingefügt.

Zu jedem Zeitpunkt ist die Kontur korrekt, am Ende gibt der Stack die Kontur mit den jeweiligen Intervallen korrekt wieder.

Laufzeitabschätzung: Gemäß der Beschreibung betrachten wir sukzessive die Geraden $f_1, f_2, \ldots, f_n$ und vergleichen diese mit dem jeweiligen Top-Element. Jede Gerade kann nur einmal auf den Stack ge*pusht* werden wird aber auch nur einmal vom Stack ge*poppt*. Insgesamt haben wir somit nur $O(n)$ viele Push− und Pop−Operationen, das gleiche gilt dann für die Top-

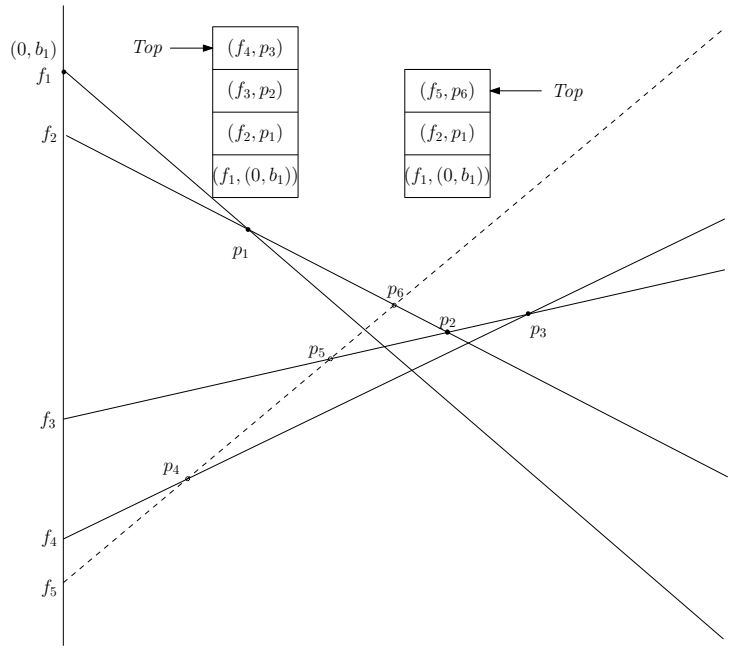


Abbildung 6.1: Vor dem Einfügen von f_5 beschreibt der Stack die obere Kontur durch die Funktionen f_1, f_2, f_3 und f_4 mit den jeweiligen Intervallgrenzen. Wenn f_5 betrachtet wird, müssen f_3 und f_4 den Stack verlassen.

Operationen. Für die Vergleiche muss das Top-Element betrachtet werden, das geht dann aber stets einer Push− oder Pop−Operation voraus. Insgesamt bestimmen wir die obere Kontur in $O(n)$ Zeit, für das anfängliche Sortieren war $O(n \log n)$ Zeit nötig.

6.2 Dynamische Listen

Eine weitere sehr nützliche Datenstruktur ist die verkettete Liste. Hierbei besitzt jedes Datenelement einen *Zeiger* auf das nachfolgende Element. Der Zeiger gibt an, in welcher Speicherzelle das nächste Listenobjekt zu finden ist.

Eine schematische Darstellung einer linearen Liste findet sich in Abbildung 6.2. Die Liste ist als Zeiger auf das Kopfelement gegeben, jedes weitere Element enhält einen Dateneintrag und einen Zeiger auf das nächste Element. Das letzte Element enthält den speziellen Zeiger Nil. Rechenintern kann man sich eine lineare Liste wie folgt vorstellen. Eine Liste von Zahlen

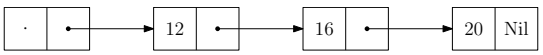


Abbildung 6.2: Die schematische Darstellung einer Liste.

$(16, 20, 33, 14, 5)$, die mit Adresse 7 angesprochen wird, kann beispielsweise so abgespeichert sein.

Adresse	1	2	3	4	5	6	7	8	9	10	11	...
Wert	20		33	16			·	5		14	12	...
Zeiger	3		10	1			4	Nil		8		...

Im Gegensatz zum Array mit fester Länge lassen sich verkettete Listen sehr effizient dynamisch

verändern. Es gibt dazu (programmiersprachenspezifische) Befehle zum *allokieren* von neuen Speicherelementen mit Datenobjekt und zugehörigem Zeiger. Außerdem können die Zeiger entsprechend umgelenkt werden.

Soll in die obige Liste nach dem Wert 33 ein neues Element 17 eingefügt werden, so wird zunächst (programmiersprachenspezifisch) eine freie Adresse $A = 6$ für ein neues Datenobjekt mit noch nicht spezifizertem Wert W_A und noch nicht spezifiziertem Zeiger Z_A erzeugt. Danach wird $W_A := 17$ eingetragen. Der Zeiger vom Element 33 wird auf den neuen Eintrag 6 gesetzt und der Zeiger Z_A verweist auf 10, also auf das nachfolgende Element 14. In Java läßt sich das beispielsweise durch Listen*objekte* mit der entsprechenden Ausgestaltung realisieren. Für ein neues Element wird ein neues Listenobjekt angelegt.

Der Vorteil der Listen liegt in der einfachen dynamischen Veränderung, für das Einfügen und Löschen von Listenelementen ist jeweils nur konstanter Aufwand notwendig. Will man bei Arrays den Zusammenhang erhalten, dann sind Kopiervorgänge mit linearer Laufzeit für das Einfügen und Löschen notwendig.

6.3 Bäume und Suchbäume

Eine natürliche Verallgemeinerung von Listen sind Bäume. Formal können Bäume wie folgt über die Knoten rekursiv definiert. Sei dazu die Menge $V = \{v_1, v_2, \ldots\}$ eine unendliche Menge von Knoten.

1. Jeder Knoten $v_i \in V$ ist ein Baum, v_i ist die Wurzel des Baumes.
2. Falls $T_1, T_2, \ldots, T_m$ mit $m \geq 1$ Bäume mit paarweise disjunkter Knotenmenge sind, dann ist auch $T = (v, T_1, T_2, \ldots, T_m)$ ein Baum mit Wurzel v und T_i ist der i-te *direkte Teilbaum* von T. Die Zahl m bezeichnet den *Grad* des Knoten v.

Grafisch werden Bäume dadurch dargestellt, dass für jeden Knoten v, ein Knoten angelegt wird und die Verbindungen zu den direkten Teilbäumen durch Kanten repräsentiert werden; siehe Abbildung 6.3. Genauer handelt es sich hierbei um eine *grafische Realisation* des Baumes. Im Folgenden werden wir die grafische Realisation und die obige formale Definition als identisch betrachten. Der i-te *direkte Teilbaum* soll von links nach rechts betrachtet an der i-ten Stelle per Kante angefügt werden.

Klassische Bezeichnungen: Für einen Baum T mit Wurzel v und direkten Teilbäumen $T_1, T_2, \ldots, T_m$ mit Wurzeln $w_1, w_2, \ldots, w_m$ heißt w_i der i-te *Sohn von* v und v der *Vater von* w_i. Kurzschreibweisen $v = \text{Wurzel}(T)$, $v = \text{Vater}(w_i)$, $w_i = \text{Sohn}(v, i)$. Ein Knoten v ist *Nachfolger* eines Knoten w innerhalb eines Baumes T falls es eine Folge von Knoten $(v_1, v_2, \ldots, v_n)$ mit $v_1 = v$ und $v_n = w$, so dass $v_{i+1} = \text{Vater}(v_i)$ für $i = 1, 2, \ldots, n-1$ gilt. Ein Knoten vom Grad 0 ist ein *Blatt*. Ein Knoten vom Grad > 0 ist ein *innerer Knoten*.

Die Tiefe eines Knoten $v \in T$ wird rekursiv wie folgt definiert.

$$\text{Tiefe}(v, T) = \begin{cases} 0 & \text{falls } v = \text{Wurzel}(T) \\ 1 + \text{Tiefe}(v, T_i) & \text{falls } T_i \text{ direkter Teilbaum} \\ & \text{von } T \text{ mit } v \in T_i \end{cases} \tag{6.1}$$

Die Höhe eines Baumes T wird durch

$$\text{Höhe}(T) = \max\{\text{Tiefe}(b, T) | b \text{ ist Blatt von } T\} \tag{6.2}$$

festgelegt

Die Abbildung 6.3 zeigt ein Beispiel eines Baumes der Höhe 4. Die Blätter v_{12} und v_{13} haben die maximale Tiefe 4.

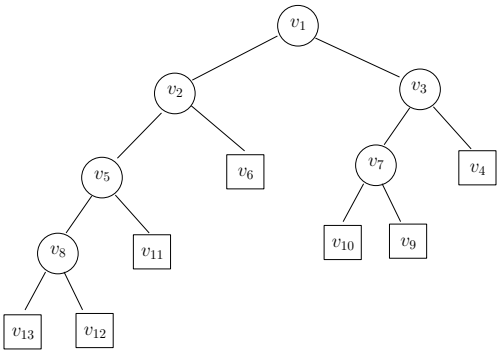


Abbildung 6.3: Die grafische Darstellung eines Baumes der Höhe 4.

Interne Realisation: Intern kann ein Baum analog zur verketteten Liste durch Zeiger realisiert werden. Ein Datenobjekt *Knoten* enthält dabei zum Beispiel einen Datensatz, einen Schlüssel (siehe unten) und eine Liste von Zeigern auf die Knoten der jeweiligen Söhne und ggf. einen Zeiger auf den Vaterknoten. Methoden für den Knoten erlauben es, auf den i-ten Sohn zuzugreifen, die Daten auszugeben oder die Zeiger zu verändern. Blätter werden als spezielle Knoten realisiert (zum Beispiel durch Nil-Zeiger, wenn keine Information in den Blättern gebraucht wird).

6.3.1 Binärbäume und Suchbäume

Bäume deren innere Knoten alle den Grad 2 haben werden Binärbäume genannt. Jeder innere Knoten hat dann einen eindeutigen linken und einen eindeutigen rechten direkten Teilbaum gemäß der eindeutigen grafischen Realisation. In den Knoten sollen außerdem Schlüssel abgespeichert werden. Über diese Schlüssel soll dann auf bestimmte Daten zugegriffen werden können. Je nachdem, ob sich die Schlüssel in den inneren Knoten oder nur in den Blättern befinden, gibt es die folgende allgemeine Unterteilung:

Art der Informationsspeicherung:

Suchbaum: Die Schlüssel befinden sich nur an den inneren Knoten, die Blätter sind leer (klassisch).

Blattsuchbaum: Die Schlüssel befinden sich nur in den Blättern, in den inneren Knoten gibt es nur *Wegweiser* zu den Knoten. Blattsuchbäume erlauben beispielsweise effiziente Bereichsanfragen.

Abbildung 6.4 zeigt einen Suchbaum (I) und einen Blattsuchbaum (II) für die Schlüssel $(1, 3, 5, 7, 12, 15)$.

Im Folgenden werden wir zunächst Suchbäume betrachten, in den Bättern gibt es also keine Informationen. Die inneren Knoten, die Blätter als Nachfolger haben verweisen intern auf Nil. Insgesamt ergeben sich nun Möglichkeiten, wie man bei einem Durchlauf (Besuch aller Knoten) durch den Baum die Ausgabe der Schlüssel oder Daten (oder die Besuchsreihenfolge der Knoten) einfach beschreiben kann.

Präorder: Besuche die Wurzel (gebe die Daten aus), besuche dann den linken Teilbaum rekursiv und danach den rechten Teilbaum rekursiv, kurz wLR.

Postorder: Besuche zuerst den linken Teilbaum rekursiv und danach den rechten Teilbaum rekursiv und danach die Wurzel (gebe die Daten aus), kurz LRw.

Lexikographische Ordnung (In-Order): Besuche zuerst den linken Teilbaum rekursiv und danach die Wurzel (gebe die Daten aus) und dann den rechten Teilbaum rekursiv, kurz LwR.

Wird beispielsweise der Baum in Abbildung 6.4(I) nach der lexikographischen Ordnung durchlaufen, erhalten wir die Schlüssel in der Reihenfolge $(1, 3, 5, 7, 12, 15)$. Die Präorder-Ausgabe ergibt: $(7, 5, 3, 1, 15, 12)$.

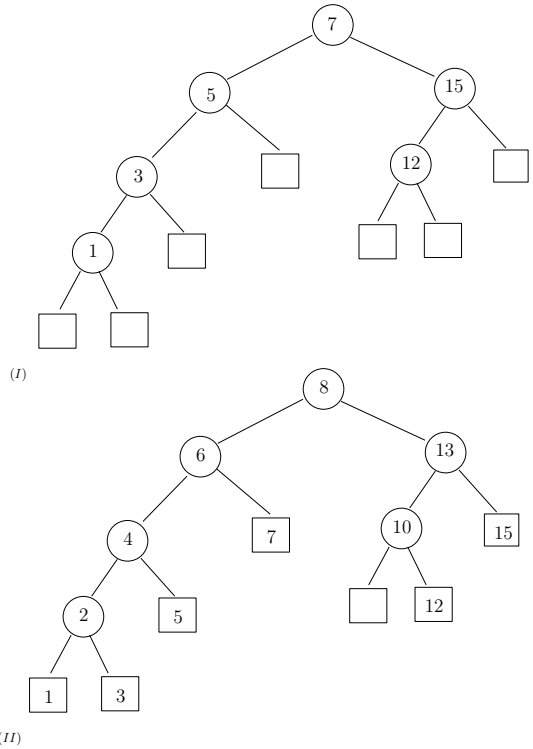


Abbildung 6.4: (I) Der Baum speichert Daten(Schlüssel) in den inneren Knoten, der Durchlauf nach *lexikographischer Ordnung* LwR gibt die Schlüssel sortiert in der Reihenfolge $(1, 3, 5, 7, 12, 15)$ aus. (II) Ein Blattsuchbaum für die gleichen Schlüssel. Innere Knoten enthalten Wegweiser zu den Blättern.

Für einen Knoten v eines Baumes bezeichne Schlüssel(v) den zugehörigen Schlüssel. Ein binärer Suchbaum hat die sogenannte *Suchbaumeigenschaft* falls für die Knoten und Schlüssel des Baumes gilt:

- Falls v ein Knoten des binären Suchbaumes ist und w ein Knoten im linken Teilbaum von v ist, dann gilt Schlüssel(w) $\leq$ Schlüssel(v).
- Falls v ein Knoten des binären Suchbaumes ist und w ein Knoten im rechten Teilbaum von v ist, dann gilt Schlüssel(w) $>$ Schlüssel(v).

In den Blattsuchbäumen erfüllen die Wegweiser zusammen mit den Schlüsseln der Blätter die Suchbaumeigenschaft. Suchbäume sollen es ermöglichen, möglichst effizient auf Daten über die Schlüssel zuzugreifen. Datenstrukturen sollen es insgesamt ermöglichen einfache Operationen effizient durchzuführen.

Ein klassisches Beispiel ist das **Wörterbuch-Problem** (Dictionary-Problem). Für eine Menge von Wörtern(Daten) wird dabei wie bereits oben erwähnt für jedes Wort ein Zahlenschlüssel verwendet, über den auf das Wort zugegriffen werden soll. Typische Operationen für dieses Problem sind:

Suche: Suche das Datenobjekt zu einem vorgegebenen Schlüssel x.

Einfügen: Füge ein neues Datenobjekt mit dem Schlüssel x ein.

Entfernen: Entferne ein Datenobjekt mit Schlüssel x.

Berichte Min/Max: Gebe das Datenobjekt mit kleinsten/größten Schlüssel aus.

Nächster Nachbar: Für einen Schlüssel m, finde ein Datenobjekt mit größten (kleinsten) Schlüssel $k \leq m$ ($k \geq m$).

Bereichsanfrage: Für ein Intervall $[l, r]$ finde alle Datenobjekte mit Schlüssel $k \in [l, r]$.

Im Folgenden beschreiben wir die Operationen jeweils für die Knoten eines Baumes mit zugehörigen Schlüssel. Dazu nehmen wir an, dass wir für einen Knoten v auf den linken Sohn bzw. rechten Sohn durch links(v) respektive rechts(v) zugreifen können. Blätter haben keine Nachfolger und können als Blatt identifiziert werden Die Operation Suchen kann nun wie folgt implementiert werden.

Algorithm 11 Search(T,k)
Suche im Baum T den Knoten v mit Schlüssel k oder gebe Blatt zurück, wenn kein solcher Knoten vorhanden ist

```
1: v := Wurzel(T);
2: while v ≠ Blatt and k ≠ Schlüssel(v) do
3:    if k < Schlüssel(v) then
4:       v := links(v);
5:    else
6:       v := rechts(v);
7:    end if
8: end while
9: RETURN v
```

Laufzeit: In jedem Schritt der Ausführung der WHILE-Schleife steigen wir definitiv eine Stufe tiefer in den Baum ab, Die Laufzeit der Suchoperation hängt somit von der Höhe des Baumes ab. Für $h = \text{Höhe}(T)$ liegt die Laufzeit in $O(h)$. Für einen Baum mit n Knoten kann die Laufzeit sogar in $\Omega(n)$ liegen, falls der Baum entsprechend entartet ist.

Korrektheit: Wenn der Baum die Suchbaumeigenschaft erfüllt, dann zweigen wir in der WHILE-Schleife jeweils in den richtigen Teilbaum ab. Wenn wir auf ein Blatt stoßen ist v = Blatt und die Schleife bricht ab. Falls der Schlüssel existiert gilt k = Schlüssel(v), die Schleife bricht ab und v wird zurückgegeben.

Bei den Operationen Einfügen und Entfernen soll natürlich die Suchbaumeigenschaft erhalten bleiben. Wir stellen hier das Einfügen vor. Zunächst wurde dabei bereits ein Knoten v mit Schlüssel(v) = k als Datenobjekt mit Verweisen links(v) = rechts(v) = Blatt angelegt.

Korrektheit und Laufzeit: Auch hier müssen wir wieder sukzessive tiefer in den Baum absteigen, um das Element an der richtigen Stelle einzufügen. Der neue Knoten v ersetzt ein Blatt des Baumes T. In der Variablen y merken wir uns den inneren Knoten an dem wir jeweils verzeigen, über marker merken wir uns außerdem die Verzweigungsrichtung. Wenn wir

Algorithm 12 Insert(T,v)
Füge v mit neuem Schlüssel(v) = k sortiert in nichtleeren Baum T ein

```
1: w := Wurzel(T);
2: y := Blatt;
3: while w ≠ Blatt do
4:    y := w
5:    if k < Schlüssel(w) then
6:       w := links(w); marker := links;
7:    else
8:       w := rechts(w); marker := rechts;
9:    end if
10: end while
11: if marker := links then
12:    links(y) := v;
13: else if marker := rechts then
14:    rechts(y) := v;
15: end if
```

am richtigen Blatt angekommen sind, können wir über y und die Verzweigungsrichtung den neuen Knoten einfügen. Die Laufzeit liegt wiederum in $O(h)$ wobei h = Höhe(T) gilt.

Das Löschen eines Knoten v eines binären Suchbaumes ist nicht ganz so einfach und hängt davon ab, welche Söhne der jeweilige Knoten besitzt. Der Knoten wird wiederum über seinen Schlüssel in $O(h)$ Zeit gefunden. Wir unterscheiden folgende Fälle.

Entfernen eines Knoten v:

- Falls der Knoten v nur Blätter als Söhne hat, entfernen wir ihn einfach. Zum Beispiel beim Entfernen des Knoten mit Schlüssel 1 in der Abbildung 6.4(I).
- Falls der Knoten v ein Blatt als Sohn hat und einen Teilbaum T_1 mit Wurzel w, ersetzt der Knoten w den Knoten v, der Knoten v wird ausgeschnitten. Zum Beispiel beim Entfernen des Knoten mit Schlüssel 5 in der Abbildung 6.4(I).
- Falls der Knoten v zwei echte Teilbäume T_1 und T_2 als Unterbäume hat, wählen wir den Knoten w mit dem Nachfolgerschlüssel des Schlüssels von v aus. Dieser hat entweder zwei Blätter aber mindestens ein Blatt als Sohn. Der Nachfolger w wird ausgeschnitten und durch seinen Nachfolger wie in den ersten beiden Fällen ersetzt. Der Knoten v wird nun allerdings durch w ersetzt. Zum Beispiel beim Entfernen des Knoten mit Schlüssel 7 in der Abbildung 6.4(I), kann der Knoten mit Schlüssel 12 den Knoten mit Schlüssel 7 ersetzen.

In jedem der beschriebenen Fälle haben wir im von der Struktur her einen Teilbaum durch seinen linken oder rechten Unterbaum ersetzt, der andere Teilbaum war dabei stets ein Blatt. Wir mussten dabei den Knoten des zugehörigen Schlüssels finden und ggf. den Knoten des Nachfolgeschlüssels. Die Operation kann in $O(h)$ ausgeführt werden.

Übungsaufgabe: Falls ein Knoten v mit Schlüssel k in einem Suchbaum zwei echte Unterbäume hat, dann hat der Knoten mit dem k nachfolgenden Schlüssel mindestens ein Blatt als Unterbaum.

Für Binärbäume lassen sich viele interessante Eigenschaften ableiten bzw. beweisen. Exemplarisch beweisen wir hier, dass für einen Binärbaum mit n Blättern die Höhe in $\Omega(\log n)$

liegt. Außerdem hängt die Anzahl der inneren Knoten fest von der Anzahl der Blätter ab und umgekehrt.

Lemma 9

1. *Ein Binärbaum mit n inneren Knoten hat $n+1$ Blätter.*
2. *Ein Binärbaum mit n Blättern hat $n-1$ innere Knoten.*
3. *In jedem Binärbaum gibt es maximal 2^i Knoten der Tiefe i.*
4. *Die Höhe eines Binärbaumes mit insgesamt n Knoten liegt in $\Omega(\log n)$.*

Beweis. 1.: Zunächst beweisen wir induktiv die Aussage über die Anzahl der Blätter bei n inneren Knoten-

Induktionsanfang: $n = 0$. Ein Baum mit keinem inneren Knoten hat genau ein Blatt.

Induktionsschritt: Sei w die Wurzel eines Binärbaumes mit insgesamt $n \geq 1$ inneren Knoten. Dann teilt sich der Baum bei w in zwei Teilbäume T_1 und T_2 auf, die beide zusammen $n_1 + n_2 = n - 1$ innere Knoten besitzen. Nach Induktionsannahme gilt: T_1 hat genau $n_1 + 1$ und T_2 hat genau $n_2 + 1$ Blätter, insgesamt hat T genau $n_1 + n_2 + 2 = n + 1$ Blätter.

2.: Der Beweis kann analog zum obigen Beweis geführt werden, eine leichte Übungsaufgabe. Man kann aber auch einfach $n := n - 1$ setzen.

3.: Dass es in jedem Binärbaum nur maximal 2^i Knoten der Tiefe i geben kann, läßt sich leicht induktiv zeigen bzw. folgt direkt aus der Definition der Tiefe und ist eine leichte Übungsaufgabe.

4.: Nun betrachten wir die Höhe eines Binärbaumes T_n mit n Knoten. Wir zeigen direkt, dass $\text{Höhe}(T_n) \geq \lceil \log(n+1) \rceil - 1$ für alle $n \geq 1$ gilt.

Die Höhe des Baumes ist die maximale Tiefe und deshalb wollen wir einen Binärbaum erstellen, dessen Knoten eine kleinstmögliche Tiefe haben. Nach obiger Aussage kann der Baum maximal $\sum_{i=1}^{l} 2^l = 2^{l+1} - 1$ Knoten der Tiefe $\leq l$ besitzen. Somit muss für n Knoten eine gewisse Mindesttiefe gelten, um alle Knoten unterzubringen.

Sei nun $k = \min_l \{l | 2^{l+1} - 1 \geq n\}$, offensichtlich ist k eine kleinstmögliche Höhe eines Baumes für n Knoten. Dann gilt offensichtlich $k \geq \lceil \log(n+1) \rceil - 1$ denn aus $2^{k+1} - 1 \geq n$ folgt $2^{k+1} \geq n + 1$ und wenn wir darauf den Logarithmus anwenden, dann folgt wegen der Ganzzahligkeit von $k+1$, dass $k + 1 \geq \lceil \log(n+1) \rceil$ gilt. Kein Baum mit geringerer Höhe kann alle n Knoten unterbringen und die Aussage gilt. □

Vorbereitend auf das nächste Kapitel wollen wir zeigen, dass wir in einem Suchbaum Knoten lokal umorganisieren können, ohne die Suchbaumeigenschaft zu verletzen. Die sogenannten Rotationen wollen wir dann im nächsten Kapitel für die Einhaltung von Höhenbalancen nutzen.

Linksrotation eines Knoten y mit Knoten x: Sei T ein Teilbaum mit folgender Eigenschaft. Die Wurzel x von T ist ein innerer Knoten mit linken Teilbaum A und rechten Teilbaum mit Wurzel y und seien B und C die Teilbäume (links und rechts) von y wie in der Abbildung 6.5. Dann können die Knoten y und x so nach *links* rotiert werden, dass y die Wurzel von T bildet mit rechten Teilbaum C. Der Knoten x wird linker Teilbaum von y mit linken und rechten Teilbäumen A und B.

Rechtsrotation eines Knoten x mit Knoten y: Symmetrisch zur Linksrotation, siehe Abbildung 6.5.

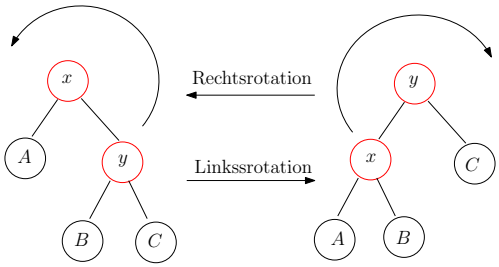


Abbildung 6.5: Bei einer Links- bzw. Rechtsrotation bleibt die Suchbaumeigenschaft erhalten.

Bei einer Linksrotation werden die Tiefen der Knoten in C um eins verringert, die Tiefen in B bleiben erhalten und die Tiefen in A werden um eins erhöht.

Bei einer Rechtsrotation werden die Tiefen der Knoten in A um eins verringert, die Tiefen in B bleiben erhalten und die Tiefen in C werden um eins erhöht.

Lemma 10 *Die Linksrotation erhält die Suchbaumeigenschaft des Teilbaumes T und des Gesamtbaumes. Der In-Order-Durchlauf gibt die gleiche Reihenfolge aus. Die Rotation kann mit konstanten Aufwand implementiert werden.*

Beweis. Dass die Rotation mit konstanten Aufwand implementiert werden kann, ergibt sich daraus, dass bei einer Implementierung mit Zeigern nur konstant viele Zeiger für die Rotation verändert werden müssen.

Die Schlüssel in A sind kleiner gleich Schlüssel$[x]$ und die Schlüssel in B und C sind größer gleich Schlüssel$[x]$. Es gilt Schlüssel$[x] \leq$ Schlüssel$[y]$. Schlüssel$[y]$ ist größer gleich den Schlüsseln in B aber kleiner gleich den Schlüsseln in C. Somit bleibt die Suchbaumeigenschaft insgesamt erhalten.

Der In-Order-Durchlauf beginnt jeweils mit einem In-Order-Durchlauf in A gibt dann x aus, führt einen In-Order-Durchlauf von B durch, gibt dann y aus und endet mit einem In-Order-Durchlauf von C. □

Naheliegend ist folgende Eigenschaft. Übungsaufgabe: Ein binärer Baum erfüllt genau dann die Suchbaumeigenschaft, wenn der In-Order-Durchlauf die Schlüssel in aufsteigend sortierter Reihenfolge ausgibt.

Im Allgemeinen werden auch Bäume mit einem Grad von ≤ 2 für jeden inneren Knoten als Binärbäume bezeichnet. In diesem Kontext wird ein Binärbaum mit Grad exakt 2 für jeden inneren Knoten als *vollständig* bezeichnet. Insbesondere weil für die Suchbäume manchmal auch die eigentlichen Blätter weggelassen werden, findet man in der Literatur auch allgemeine Binärbäume. Wir werden stets die (leeren) Blätter mit notieren und somit vollständige Binärbäume betrachten.

6.4 AVL-Bäume

Wie wir bereits im letzten Abschnitt gesehen haben, ist die Höhe des Baumes entscheident für elementare Operationen auf der Datenstruktur eines Binärbaumes. Das Lemma 9 zeigt uns allerdings bereits, dass eine bessere Laufzeit als $\Omega(\log n)$ für das Einfügen, Suchen und Entfernen nicht zu erwarten ist.

Ein binärer Suchbaum heißt AVL-Baum, falls für *jeden* Knoten v gilt, dass die Höhe der beiden Teilbäume von v sich nur um Eins unterscheiden.

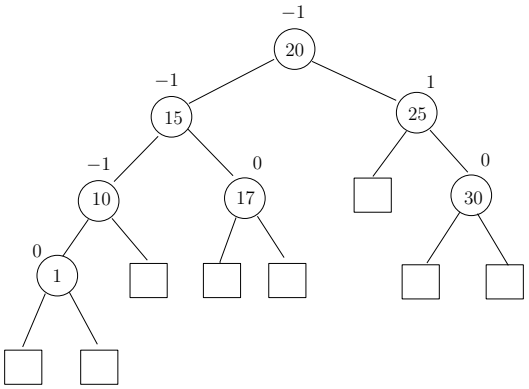


Abbildung 6.6: An jedem Knoten eines AVL-Baumes betragen die Höhenunterschieden der Teilbäume maximal 1.

Für jeden Knoten v gibt es einen sogenannten *Balancefaktor*.

$$\text{bal}(v) := \text{Höhe}(\text{rechts}(v)) - \text{Höhe}(\text{links}(v)) \tag{6.3}$$

In AVL-Bäumen (AVL nach Adelson, Velskij und Landis) gilt für alle Knoten $\text{bal}(v) \in \{-1, 0, 1\}$ ein Beispiel zeigt die Abbildung 6.6. Wir wollen nun die Höhe eines AVL-Baumes ermitteln. Dazu geben wir eine Höhe h vor und stellen fest, wieviele Knoten $n(h)$ ein AVL-Baum der Höhe h mindestens hat. Dann können wir umgekehrt feststellen welche maximale Höhe ein AVL-Baum für n Knoten aufweisen kann.

Ein AVL-Baum der Höhe h mit minimaler Knotenanzahl $n(h)$ heißt auch *minimaler AVL-Baum der Höhe h*. Wir zählen die Blätter $b(h)$ eines solchen Baumes und können nach Lemma 9 dann die Anzahl der Knoten ermitteln. Sei T_h ein minimaler AVL-Baum mit $b(h)$ Blättern. Betrachte Abbildung 6.7. Offensichtlich gilt:

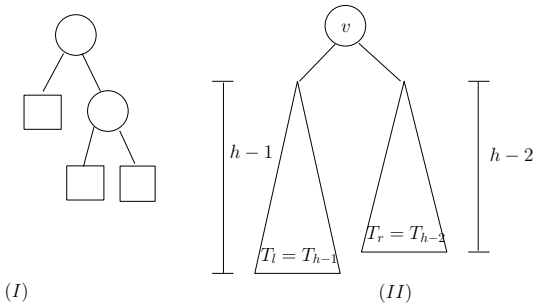


Abbildung 6.7: (I) Der minimale AVL-Baum der Höhe 2 hat $b(2) = 3$ Blätter. (II) Der minimale AVL-Baum der Höhe hat $h \geq 2$ hat $b(h) = b(h-1) + b(h-2)$ Blätter.

- Ein minimaler AVL-Baum der Höhe 0 hat ein Blatt, $b(0) = 1$.
- Ein minimaler AVL-Baum der Höhe 1 hat zwei Blätter, $b(1) = 2$.
- Ein minimaler AVL-Baum der Höhe 2 hat drei Blätter, $b(2) = 3$.
- Sei T_h eine minimaler AVL-Baum der Höhe h mit Wurzel v und seien T_l und T_r die Teilbäume von v. Dann hat zumindest einer der beiden Teilbäume eine Höhe von $h-1$ und der andere muss mindestens eine Höhe von $h-2$ haben. Da wir die Anzahl der

Bätter minimieren wollen, wählen wir o.B.d.A. $T_l = T_{h-1}$ und $T_r = T_{h-2}$ und für $h \geq 2$ gilt somit:

$$b(h) = b(h-1) + b(h-2) \tag{6.4}$$

Theorem 11 *Die Höhe eines AVL-Baumes mit n Blättern (und $n-1$ inneren Knoten) beträgt $O(\log n)$.*

Beweis. Sei $F_0 = 0$, $F_1 = 1$ und $F_2 = 1$ und $F_{h+2} = F_{h+1} + F_h$ für $h \geq 2$ dann ist offensichtlich $F_{h+2} = b(h)$ für $h \geq 0$. Die Folge F_h ist die bereits behandelte Fibonacci-Folge.

Wir zeigen zunächst, dass $F_{i+2} \geq 2F_i$ für $i \geq 1$ gilt. Das gilt für $i = 1$ und für $i \geq 2$ wenden wir direkt die Definition der Folge an, also $F_{i+2} = F_i + F_{i+1} = F_i + F_i + F_{i-1} \geq 2F_i$.

Mit $F_1 = F_2 = 1$ folgt nun $F_{h+2} \geq 2^{h/2}$ für $h \geq 1$. Das gilt zunächst für $h = 1$ und dann folgt induktiv, dass $F_h \geq 2^{(h-2)/2}$ für $h \geq 3$ bereits gilt. Dann ist $F_{h+2} \geq 2F_h \geq 2 \cdot 2^{(h-2)/2} = 2^{h/2}$.

Sei nun $b(h) = n$. Dann ist $n = F_{h+2} \geq 2^{h/2}$ und somit $h \leq 2 \log n$. □

Als nächstes wollen wir die Operationen Einfügen und Entfernen für AVL-Bäume betrachten. Diese Operationen sollen

- ... *strukturerhaltend* durchgeführt werden, die AVL-Eigenschaft soll erhalten bleiben.
- ... *kostengünstig* durchgeführt werden, der Durchlauf in $O(\log n)$ und eventuelles Ausbalancieren in konstanter Zeit durch Umhängen von Zeigern.

6.4.1 Einfügen eines neuen Knoten

Dazu betrachten wir das Beispiel aus Abbildung 6.6. Im Baum notieren wir den Wert bal(v) für jeden Knoten. Wir merken uns an jedem Knoten die Höhe des linken und rechten Teilbaumes, so können wir schnell den Wert bal(v) bestimmen bzw. neu berechnen. Wir wollen einen Knoten z mit Schlüssel 5 einfügen.

Zunächst fügen wir dabei abwärtslaufend wie in der Prozedur 12 den Knoten ein; siehe Abbildung 6.8. Danach werden aufwärtslaufend die bal(v)-Werte geändert bis zum ersten Mal eine Disbalance auftritt. Die neuen bal(v)-Werte werden durch die veränderten Höhen sukzessive berechnet. Am Knoten y mit Schlüssel 10 tritt zum ersten Mal eine Disbalance auf. Die Teilbäume des Knoten sollen umgeordnet werden. Eine einfache Rechtsrotation am Knoten x mit y führt hier leider nicht zum Erfolg, der neu eingefügte Knoten z (Baum B aus Abbildung 6.5) würde seine Höhe nicht verlieren und am neuen Knoten x wird dann lediglich der Wert bal(x) = +2 notiert.

Deshalb führen wir hier die folgende *Doppelrotation* aus:

1. Zunächst eine Linksrotation des Knoten z mit x.
2. Danach eine Rechtsrotation des Knoten z mit y.

Zunächst verliert dabei der Knoten x an Höhe, das wird aber durch die zweite Rotation wieder ausgeglichen. In Abbildung 6.9 sehen wir die Zwischenergebnisse der Rotationen. Der zugehörige Teilbaum ist danach ausbalanciert. Abschließend stellt sich die Frage ob gegebenenfalls noch die bal(v) Einträge bis Wurzel aktualisiert müssen. Wir haben den Teilbaum des Knoten y ausbalanciert aber er hat seine Höhe behalten, somit ändern sich die Werte nicht. Die Abbildung 6.10 zeigt das Ergebnis nach dem Einfügen.

Das Einfügen gestaltet sich insgesamt in den folgenden Schritten:

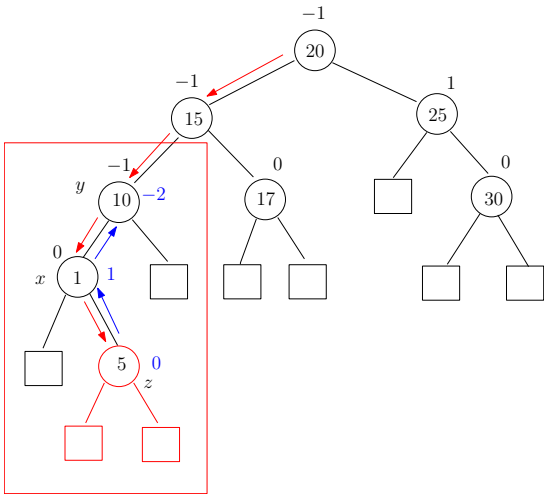


Abbildung 6.8: Das Einfügen eines Knoten z mit Schlüssel 5 in den AVL-Baum aus Abbildung 6.6 durch einen Durchlauf zum entsprechenden Blatt. Danach werden rückwärtslaufend die bal(v)-Werte geändert, bis zum ersten Mal eine Disbalance auftritt. Der Teilbaum in der eingezeichneten Box muss ausbalanciert werden.

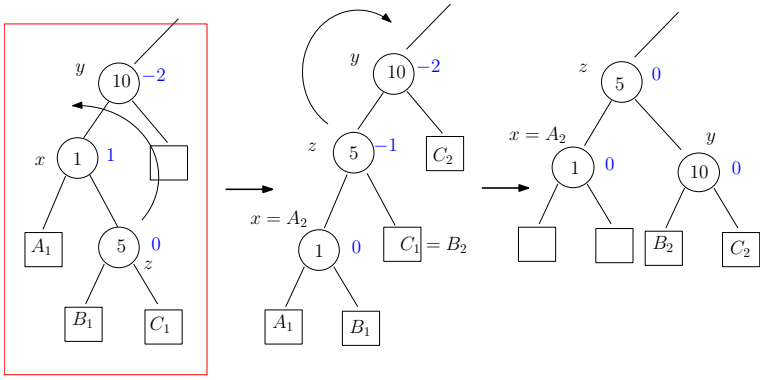


Abbildung 6.9: Zuerst wird eine Linksrotation am Knoten z mit x vorgenommen, danach eine Rechtsrotation am Knoten z mit y. Der Teilbaum ist danach balanciert.

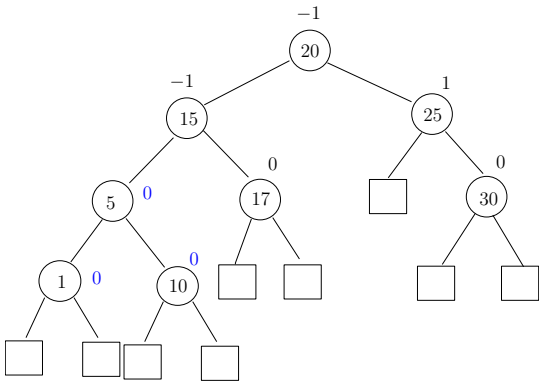


Abbildung 6.10: Der ausbalancierte Baum nach dem Einfügen des Knoten z des Schlüssels 5. Die bal(v)-Werte alle anderen Knoten bleiben gleich, da die Höhe des Teilbaumes erhalten bleibt.

1. Vorwärtsdurchlauf gemäß der Schlüssel und Eintrag des neues Knoten. Laufzeit: $O(\log n)$

2. Rückwärtsdurchlauf zum Anpassen der bal(v)-Werte bis die erste Disbalance an einem Knoten v auftritt. Laufzeit: $O(\log n)$

3. Ausbalancierung des zugehörigen Teilbaums durch Rotation. Laufzeit: $O(1)$, inklusive umsetzen der Zeiger

Kategorisierung der Rotationen:

Die Doppelrotation (erst links, dann rechts) im obigen Beispiel war notwendig, weil der Weg vom Knoten y der ersten Disbalance zum neu eingefügten Knoten z zuerst links und danach rechts abgebogen ist. Der neue Knoten liegt also vom Knoten y aus gesehen im rechten Unterbaum des linken Unterbaumes. Dieser rechte Unterbaum muss an Höhe verlieren.

Falls der besagte Weg zuerst rechts und danach links abbiegt, wird die Doppelrotation in der Reihenfolge rechts/links ausgeführt. Der neue Knoten liegt dann also vom Knoten y aus gesehen im linken Unterbaum des rechten Unterbaumes.

Manchmal reicht auch eine einfach Rechts- oder Linksrotation aus. Das ist der Fall, wenn der Weg vom Knoten y der ersten Disbalance zum neu eingefügten Knoten z nur rechts oder nur links abbiegt.

Falls wir beispielsweise in den neuen Baum aus Abbildung 6.10 einen weiteren Knoten z mit Schlüssel 32 einfügen, so erhalten wir wie in Abbildung 6.11 die erste Disbalance am Knoten y und führen eine einfache Linksrotation des Knoten x mit y aus. Das Ergebnis ist in Abbildung 6.12 zu sehen.

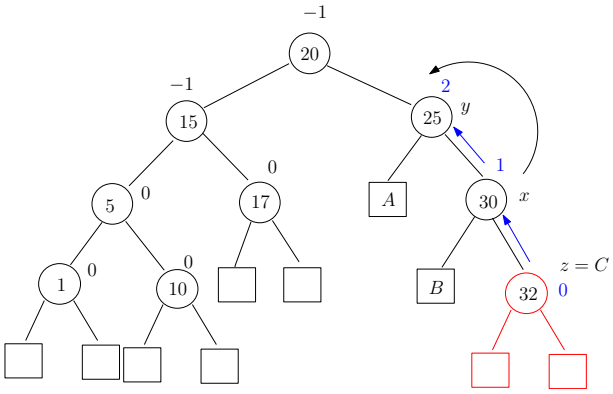


Abbildung 6.11: Nach dem Einfügen des Knoten z mit Schlüssel 32 reicht eine einfache Linksrotation.

Nach diesen Beispielen beschreiben wir die Doppelrotation (links/rechts) und die einfache Rotation (links) schematisch. Die Doppelrotation (rechts/links) und die einfache Rotation (rechts) verlaufen analog.

Doppelrotation (links/rechts): Nach dem Einfügen eines Knoten z und dem Ermitteln des untersten Disbalance-Knoten y läuft der Weg zum Knoten z in den rechten Unterbaum des linken Unterbaumes, siehe Abbildung 6.13. Zunächst wird eine Linksrotation des Knoten w mit x ausgeführt, siehe das Ergebnis in Abbildung 6.14. Danach wird eine Rechtsrotation des Knoten w mit y ausgeführt, siehe das Ergebnis in Abbildung 6.14. Der Baum ist ausgeglichen und das wäre auch der Fall, wenn der Knoten z in B eingefügt worden wäre, siehe z' in Abbildung 6.14.

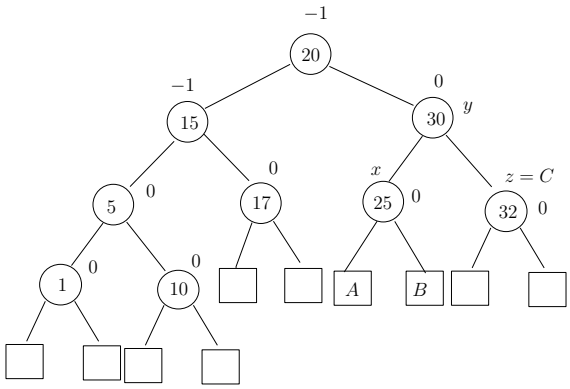


Abbildung 6.12: Die Höhe des ausbalancierten Baumes ändert sich nicht, alle anderen bal(v)-Werte bleiben gleich.

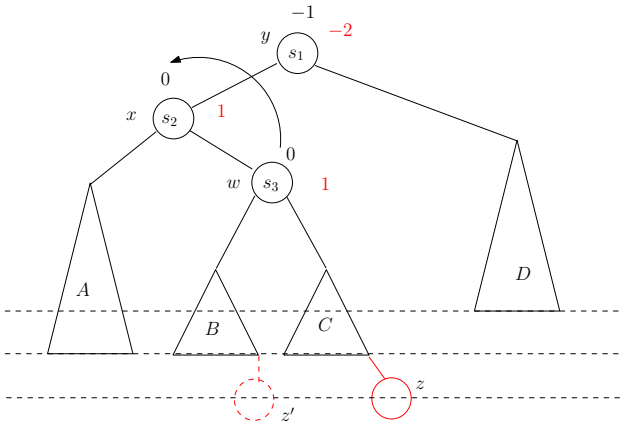


Abbildung 6.13: Die Situation vor einer (links/rechts) Doppelrotation. Der Knoten z könnte auch in B eingefügt worden sein.

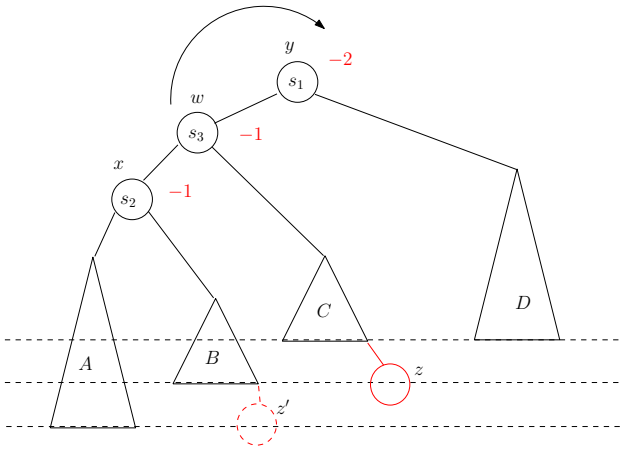


Abbildung 6.14: Die Situation vor einer (links/rechts) Doppelrotation. Der Knoten z könnte auch in B eingefügt worden sein.

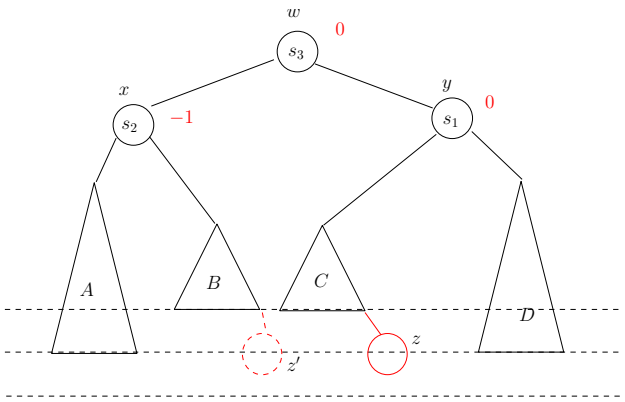


Abbildung 6.15: Nach der zweiten Rotation ist der Baum ausgeglichen. Falls der Knoten z in den Teilbaum B eingefügt worden wäre, wäre der Baum ebenfalls ausgeglichen. In jedem Fall ist die Höhe des Baumes wieder identisch zur Ausgangshöhe.

Einfache Rotation (links): Nach dem Einfügen eines Knoten z und dem Ermitteln des untersten Disbalance-Knoten y läuft der Weg zum Knoten z in den rechten Unterbaum des rechten Unterbaumes, siehe Abbildung 6.16.

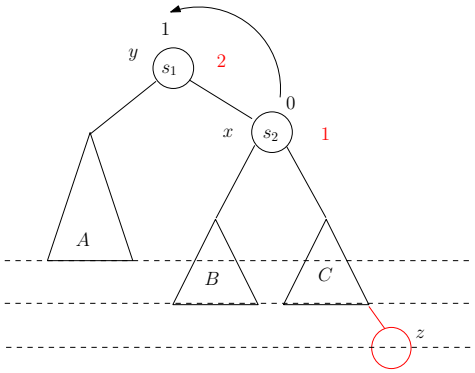


Abbildung 6.16: Die Situation vor einer einfachen Rotation (links). Der neue Knoten z liegt im rechten Unterbaum des rechten Unterbaumes von y.

Wir beschreiben nun in der Prozedur 13, wie die Zeiger bei einer Linksrotation verändert werden müssen.

Eine Rechtsrotation kann analog ausgeführt werden. Die Doppelrotation (links/rechts) die für einen Disbalance-Knoten y ausgeführt wird, kann dann durch folgende Prozedur beschrieben werden.

Übungsaufgabe: Beschreiben Sie die Schemata für eine Doppelrotation (rechts/links) und eine einfache Rotation (rechts) und die zugehörigen Zeigerprozeduren.

Korrektheit: Von der Logik her können Disbalancen am Knoten y durch das Einfügen eines Knoten z nur durch eine der vier Einfügefälle von z (links/rechts, rechts/links, links/links, rechts/rechts) entstehen. Wird der Knoten z beispielsweise direkt nur links unter y eingefügt dann kann rechts(y) nur aus einem Blatt oder einem Baum der Höhe 1 bestanden haben. In jedem Fall entsteht bei y gar keine Disbalance. Der andere Fall geht analog.

Außerdem wird durch das Beseitigen der Disbalance beim Knoten y stets der gesamte Baum ausgeglichen. Falls beispielsweise die Doppelrotation (links/rechts) durchgeführt wurde, muss

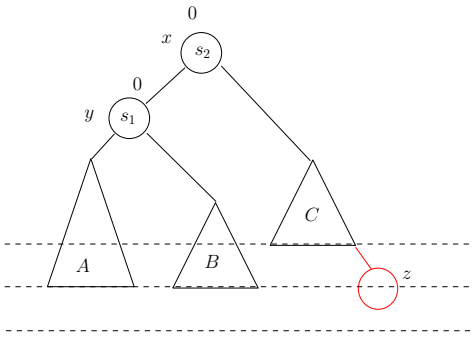


Abbildung 6.17: Die Situation nach einer einfachen Rotation (links). Der Teilbaum C hat an Höhe verloren, der Teilbaum A an Höhe gewonnen, der Baum ist ausgeglichen. Außerdem ist die Höhe des Baumes wieder identisch zur Ausgangshöhe.

Prozedur 13 Linksrotation(T,x)
Linksrotation des Knoten x mit $y = \text{Vater}(x)$

```
1:  y := Vater(x); w := Vater(y);
2:  if w ≠ Nil and y = LinkerSohn(w) then
3:      marker := links;
4:  else
5:      marker := rechts;
6:  end if
7:  rechts(y) := links(x);
8:  links(x) := y;
9:  if marker := links then
10:     links(w) := x;
11: else
12:     rechts(w) := x;
13: end if
```

Prozedur 14 DoppelrotationLR(T,y)
Doppelrotation (links/rechts) am Disbalance-Knoten y

```
1:  x := links(y);
2:  w := rechts(x);
3:  Linksrotation(T, w);
4:  Rechtsrotation(T, w);
```

der Knoten y vor dem Einfügen den Wert -1 gehabt haben. Nach der Rotation hat der Teilbaum mit neuer Wurzel w zwar den bal-Wert 0 aber die Höhe des Teilbaumes bleibt erhalten. Das heißt, alle anderen bal(v)-Werte bleiben erhalten. Der Baum ist insgesamt ausgeglichen. Die anderen Fälle gelten analog.

Insgesamt führen wir ein korrektes Einfügen eines Knoten z in Laufzeit $O(\log n)$ durch.

6.4.2 Entfernen eines Knoten

Beim Entfernen von Knoten kommen analoge Operationen zur Anwendung. Wie bereits in Abschnitt 6.3.1 bemerkt, wird stets ein Knoten v durch seinen linken oder rechten Teilbaum w ersetzt, der andere Teilbaum war dabei ein Blatt. Wenn der Baum nach dieser Operation am Knoten w unbalanciert ist, muss der Baum rebalanciert werden.

Wir geben zunächst ein Beispiel an und beschreiben dann die allgemeine Vorgehensweise. Wenn in Abbildung 6.12 der Knoten mit Schlüssel 15 und zwei echten Teilbäumen entfernt werden soll, so wird der Knoten zuerst durch den Knoten mit nächstgrößtem Schlüssel 17 ersetzt und der vormalige Knoten mit Schlüssel 17 entfernt. Dieser hatte mindestens ein Blatt, in unserem Fall sogar zwei Blätter. Der Teilbaum wurde entfernt und in diesem Fall durch ein Blatt ersetzt. Auf dem Weg zur Wurzel werden dann die bal(v)-Werte angepasst, bis zum ersten Mal eine Disbalance entdeckt wird. Nun müssen wiederum Rotationen ausgeführt werden, um die Disbalance zu beseitigen. Offensichtlich reicht hier eine einfache Rechtsrotation am Knoten mit Schlüssel 5 aus, siehe Abbildung 6.18. Es kann ebenfalls sein, dass

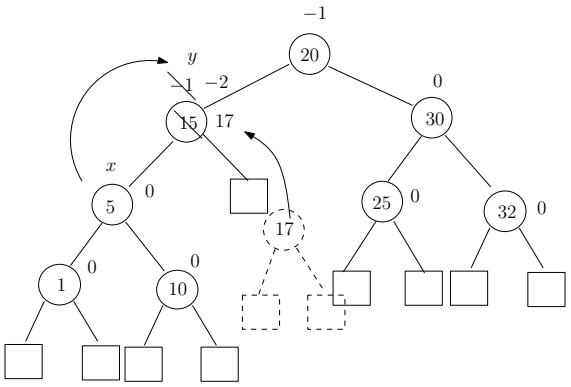


Abbildung 6.18: Die Situation vor einer einfachen Rotation nach dem Entfernen des Knoten mit Schlüssel 15. Der ehemalige Knoten des Schlüssels 17 wir entfernt.

Doppelrotationen notwendig sind. Wir beschreiben kurz das allgemeine Schema. Im Gegensatz zur Beschreibung des Einfügens eines Knoten beschreiben wir hier die einfache Rotation (rechts) und die Doppelrotation (links/rechts) die durch eine -2 Disbalance entstehen. Analoge Operationen einfache Rotation (links) und Doppelrotation (rechts/links) entstehen bei $+2$ Disbalancen.

Angenommen die Disbalance tritt am Knoten y zuerst auf. Falls die Disbalance im Knoten y durch einen -2 Wert gegeben ist, wurde ein rechter Teilbaum C von y angehoben. Für den linken Nachfolger x ergeben sich zwei Fälle. Falls der bisherige bal(x)-Wert 0 oder -1 war, reicht eine einfache Rechtsrotation am Knoten x mit Knoten y aus, siehe Abbildung 6.20. Der Teilbäume A von x wird um eins angehoben, der Teilbaum C um eins gesenkt. Der Teilbaum B respektive B' behält seine Höhe. Es kann auch vorkommen (Fall B'), dass die Höhe des Baumes insgesamt um Eins sinkt.

Falls allerdings der bal(x) Wert $+1$ war, hat der rechte Nachfolger w von x zwei Teilbäume B

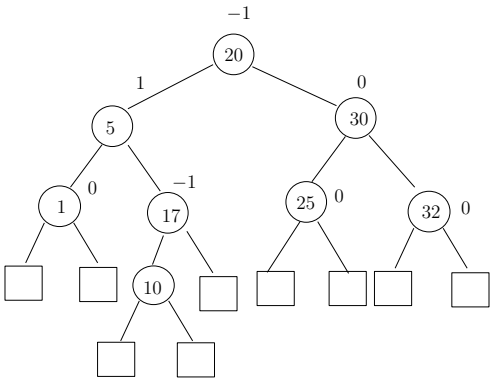


Abbildung 6.19: Nach der Rotation ist der Baum ausgeglichen.

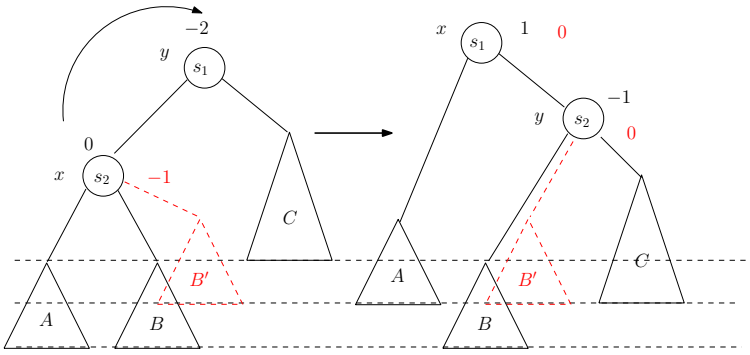


Abbildung 6.20: Die Situation vor einer einfachen Rotation. Der Disbalance-Knoten y ($\text{bal}(y) = -2$) hat einen linken Nachfolger x mit $\text{bal}(x) \in \{0, -1\}$. Dann reicht eine einfache Rechtsrotation des Knoten x mit y aus.

und C, von denen mindestens einer zusammen mit w eine höhere Tiefe als der linke Nachfolger A von x aufweist. Eine einfache Rechtsrotation von x mit y wird die Höhe des Baumes w erhalten, die Höhe von A aber absenken, somit entsteht eine $+2$ Disbalance.

Hier ist somit wiederum eine Doppelrotation notwendig. Zunächst eine Linksrotation des Knoten w mit x und danach eine Rechtsrotation des Knoten w mit y. Das vollständige Ergebnis ist in Abbildung 6.21 zu sehen. Beachte, dass C und B auch die Rollen tauschen können. Das heißt, dass entweder B oder C die Disbalance am Knoten y zusammen mit D verursacht. In jedem Fall wird der Baum rebalanciert, hat aber insgesamt in jedem Fall die Höhe um Eins verringert. Übungsaufgabe: Beschreiben Sie die Schemata für die Rotationen nach dem

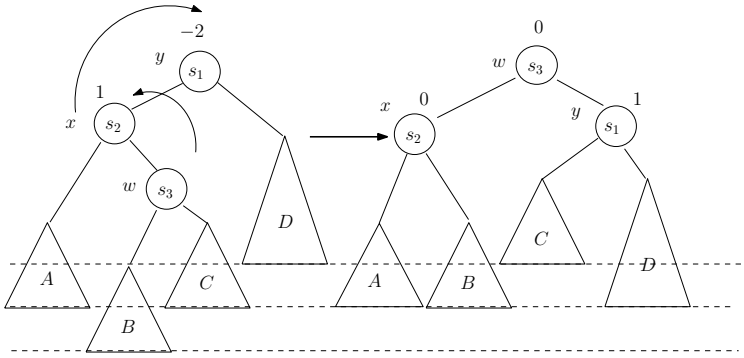


Abbildung 6.21: Die Situation vor einer Doppelrotation (links/rechts). Der Disbalance-Knoten y ($\text{bal}(y) = -2$) hat einen linken Nachfolger x mit $\text{bal}(x) = 1$. Dann ist eine Doppelrotation (zuerst Linksrotation Knoten w mit x und danach Rechtsrotation Knoten w mit y) notwendig.

Entfernen, falls sich die Disbalance durch einen ersten -2 Wert an einem Knoten y ergibt und somit ein rechter Teilbaum von y an Höhe verloren hat. Beschreiben Sie die zwei Unterfälle, dass der linke Nachfolger x von y den Wert $\text{bal}(x) = -1$ bzw. $\text{bal}(x) \in \{0, 1\}$ aufweist.

Korrektheit: Für den ersten Disbalance Knoten y gibt es nur die beiden Möglichkeiten $\text{bal}(y) \in \{2, -2\}$, wobei die Disbalance nur durch die jeweils beschriebenen Fälle aufgetreten sein konnten. Entweder wurde die Höhe eines rechten oder eines linken Teilbaumes von y reduziert. Für den jeweils anderen Nachfolger x von y ergeben sich dann die zwei möglichen Unterfälle. Insgesamt wird zur Rebalancierung von y entweder eine einfache Rotation (links oder rechts) oder eine Doppelrotation (links/rechts oder rechts/links) ausgeführt. Danach ist der Knoten y rebalanciert. Wir sehen aber, dass der Teilbaum von y (im Vergleich zur Situation vor dem Enfernen) insgesamt immer um 1 an Höhe verliert, deshalb kann es sein, dass die verringerte Höhe von y weitere $\text{bal}(v)$-Werte auf dem Weg zur Wurzel verändert.

Das ist nicht weiter tragisch. Wir wenden die Argumentation sukzessive wieder an. Für den nächsten Disbalance-Knoten auf dem Weg zur Wurzel führen wir die gleichen Operationen durch. Spätestens wenn wir an der Wurzel angekommen sind, ist der gesamte Baum ausgeglichen.

Laufzeit: Für das Entfernen wird zunächst ein Aufwand in $O(\log n)$ benötigt. Für die Rebalancierung laufen wir aufsteigend zu den nächsten zu rebalancierenden Knoten. Der Weg zur Wurzel hat eine Länge von $O(\log n)$. Jede einzelne Balancierungsoperation kostet $O(1)$ Zeit. Insgesamt wird nur $O(\log n)$ Zeit benötigt.

6.5 B-Bäume

Wenn die Datenmenge so groß wird, dass sie nicht mehr in den schnellen Hauptspeicher passt, muss sie in langsamere Speicher ausgelagert werden. Hier gilt das Modell der REAL RAM nur noch bedingt, denn die Laufzeit von Algorithmen wird jetzt von der *Anzahl der Zugriffe auf das langsame Speichermedium* dominiert.

Zum Beispiel muss bei externen Festplatten ein Schreib-/Lesekopf auf die richtige Spur bewegt werden; das dauert zwar nur einige Millisekunden, aber doch fast 10^3 mal so lang wie ein Hautpspeicherzugriff. Dafür liefert ein Festplattenzugriff nicht nur einen einzelnen Wert, sondern eine Datenseite (Datenblock) mit Volumen 512 Byte bis 1 KByte.

Diesem Umstand trägt das Modell der *B-Bäume* Rechnung.

Definition 12 Ein *B-Baum der Ordnung k* ist durch folgende Eigenschaften bestimmt:

1. Alle Blätter haben dieselbe Tiefe.
2. Alle Knoten außer der Wurzel haben mindestens k und höchstens $2k-1$ viele Söhne.
3. Die Wurzel hat mindestens 2 und höchstens $2k-1$ viele Söhne.

Wenn h die Höhe eines solchen B-Baums bezeichnet und n die Anzahl seiner Blätter, so gilt

$$2 \cdot k^{h-1} \;\leq\; n \;\leq\; (2k-1)^h;$$

die untere Schranke wird erreicht, wenn die Wurzel nur 2 und alle anderen internen Knoten nur k Söhne haben, die obere, wenn alle Knoten maximal viele, nämlich $2k-1$, Kinder besitzen. Daraus folgt

$$\log_{2k-1} n \;\leq\; h \;\leq\; 1 + \log_k \frac{n}{2} \tag{6.5}$$

Abbildung 6.22 zeigt, wie ein Knoten eines B-Suchbaums aufgebaut ist.

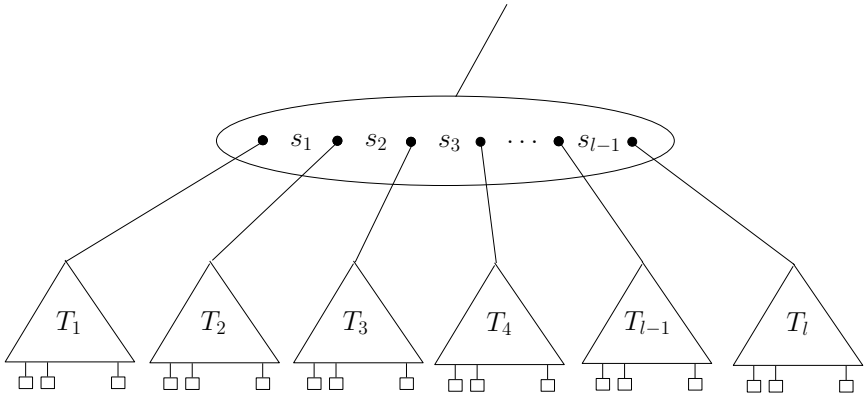


Abbildung 6.22: Im Knoten eines B-Baums wechseln Zeiger auf Teilbäume und Schlüssel einander ab. Alle Schlüssel in T_i sind kleiner als s_i, und s_i ist kleiner als die Schlüssel in T_{i+1}.

Schlüssel stehen nur in den inneren Knoten, zwischen den Zeigern auf die Teilbäume. Sie sind aufsteigend sortiert: Für alle i gilt

$$\text{Schlüssel von } T_i \;<\; s_i \;< \text{Schlüssel von } T_{i+1}.$$

Der Parameter k wird meist so gewählt, dass ein maximal voller Knoten mit $2k-1$ Zeigern und $2k-2$ Schlüsseln gerade auf eine Datenseite passt. Wegen der unteren Schranke k für die Anzahl der Söhne ist dann jede Datenseite (außer der Wurzel) zumindest halb voll.

Beim *Suchen* eines Schlüssels x startet man im Wurzelknoten und durchläuft von links nach rechts die dort gespeicherten Schlüssel $s_1, \ldots, s_{l-1}$, bis man zum letzten Schlüssel s_i mit $s_i \leq x$ gelangt. Gilt $s_i = x$, so ist man fertig. Andernfalls setzt man die Suche im Teilbaum T_{i+1} fort.

Um einen Schlüssel x *einzufügen*, wird x zunächst gesucht. Kommt x bereits vor, so wird eine Meldung generiert. Andernfalls endet die Suche erfolglos in einem Blatt b, das in seinem Vaterknoten v hinter einem Schlüssel $s_i < x$ angehängt ist. Wir fügen x als neuen Schlüssel hinter s_i in die Liste ein und ein neues Blatt b' hinter x. Wenn Knoten v jetzt immer noch $\leq 2k-1$ viele Söhne hat, ist man fertig. Wenn nicht, hat v jetzt genau $2k$ viele Söhne und wird in zwei gleich große Knoten v_1 und v_2 aufgeteilt; dabei wandert der k-te Schlüssel (nach Einfügen von x) in den Vater v' von v, siehe Abbildung 6.23. Möglicherweise ist jetzt auch Knoten v' überfüllt und muss geteilt werden. Das kann sich bis zur Wurzel w fortpflanzen; in dem Fall wird w in zwei Knoten aufgeteilt und eine neue Wurzel darübergesetzt, die genau diese zwei Söhne hat.

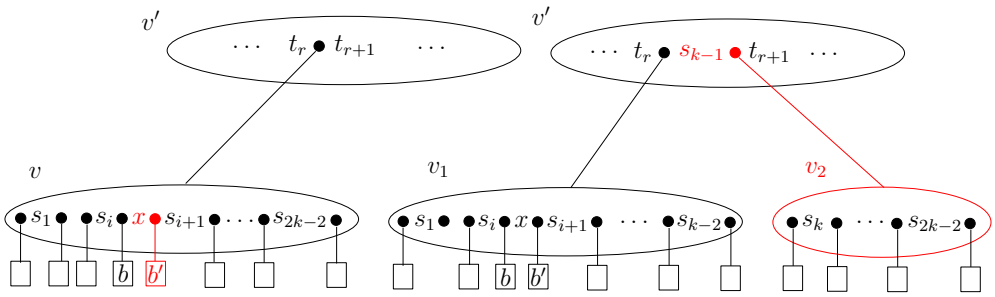


Abbildung 6.23: Knoten v wird geteilt, wenn er nach Einfügen von Schlüssel x und einem neuen Blatt b' nun $2k$ viele Söhne hat.

Auch beim *Entfernen* eines Schlüssels x wird dieser zunächst gesucht. Falls x nicht vorkommt, wird eine Meldung generiert. Andernfalls steht x in einem Knoten v hinter dem Zeiger auf einen Teilbaum T_j. Im ersten Fall sind T_j und alle anderen Söhne von v Blätter. Dann werden T_j und x einfach entfernt. Dabei kann es passieren, dass v nur noch $k-1$ Söhne bleiben; wir überlegen uns gleich, was dann zu tun ist. Im zweiten Fall hat v echte Teilbäume als Söhne. Dann sei s der größte Schlüssel in T_j, enthalten in einem Knoten w. Rechts von s steht in w nur noch der Zeiger auf ein Blatt b; siehe Abbildung 6.24. Schlüssel s ist der Vorgänger von x im Baum. Darum ersetzen wir im Knoten v den Schlüssel x durch s und entfernen dann im Knoten w den Schlüssel s und das Blatt b (so, als hätten wir von Anfang an s entfernen wollen, wie im ersten Fall). Auch hier kann es passieren, dass w nur $k-1$ Söhne behält.

Dann wird w mit einem Nachbarknoten u mit $l \geq k$ Söhnen vereinigt; das ist gerade die Umkehrung der Knotenteilung beim Einfügen. Falls $l > k$ war, können wir den Vereinigungsknoten gleich wieder in zwei Knoten mit jeweils $\geq k$ vielen Söhnen teilen und sind nach dieser Umverteilung von Schlüsseln fertig. Andernfalls hat der Vater w' von w und u durch die Vereinigung einen Sohn verloren und könnte jetzt selbst unterbelegt sein. Dann muss w' mit einem Nachbarknoten vereinigt werden, und so fort. Sollte die Wurzel nur noch einen Sohn haben, wird sie entfernt, und der Sohn wird zur neuen Wurzel.

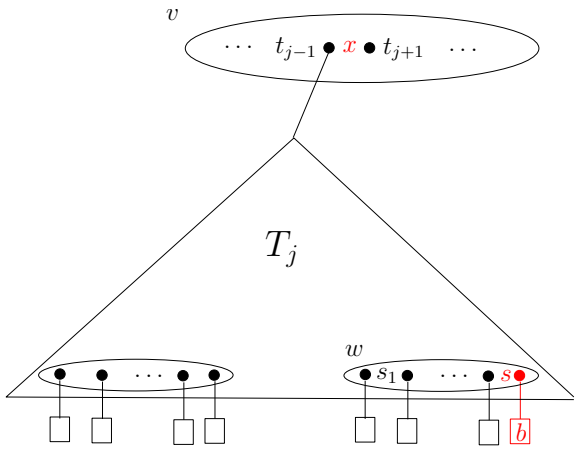


Abbildung 6.24: Der zu entfernende Schlüssel x wird durch seinen Vorgänger s ersetzt. Blatt b wird entfernt.

Bei den Operationen Suchen, Einfügen und Entfernen erfordert jeder Knotenbesuch einen Externzugriff auf eine Datenseite. Beim Einfügen läuft man erst längs des Suchpfads zu einem Blatt hinab; dann kann es nötig sein, zur Wurzel hinaufzulaufen und dabei jeden Knoten zu teilen. Mit der Abschätzung (6.5) für die Höhe eines B-Baums erhalten wir also folgendes Ergebnis:

Theorem 13 *In einem B-Baum der Ordnung k lassen sich Suchen, Einfügen und Entfernen mit maximal*

$$3 \log_k n + 3 \in O(\log_k n)$$

Externzugriffen ausführen, wobei n die Anzahl der Schlüssel bedeutet.

Für $k = 2$ sind B-Bäume auch als 2-3-Bäume bekannt und können als Alternative zu AVL-Bäumen als interne Datenstrukturen im Hauptspeicher verwendet werden.

6.6 Mittelwertbildung

Bevor wir weitere Datenstrukturen betrachten, wollen wir im Hinblick auf spätere Anwendungen verschiedene Arten der Mittelwertbildung diskutieren, die bei der Analyse vorkommen können, und jeweils typische Anwendungen untersuchen.

6.6.1 Amortisierte Kosten: Berechnung der konvexen Hülle

Manchmal muss eine Folge von n gleichartigen Aktionen ausgeführt werden, von denen einige hohe Kosten verursachen, andere aber nur geringe. Dann ist es vernünftig, die Gesamtkosten aller Aktionen zu betrachten und durch n zu teilen, um ein Maß für die mittleren Kosten pro Aktion zu bekommen. Dabei kann eine teure Aktion als "Investition in die Zukunft" angesehen werden, die sich später durch viele Aktionen mit niedrigen Kosten *"amortisiert"*.

Als Beispiel betrachten wir einen Algorithmus zur Konstruktion der *konvexen Hülle* von n Punkten in der Ebene, also der (bezüglich Inklusion) kleinsten Menge, die die gegebenen Punkte enthält und mit je zwei beliebigen Punkten a und b auch das Liniensegment $\overline{ab}$ von a nach b. Ihren Rand kann man sich als ein gespanntes Gummiband vorstellen, das die gegebenen Punkte umschließt; siehe Abbildung 6.25 (i).

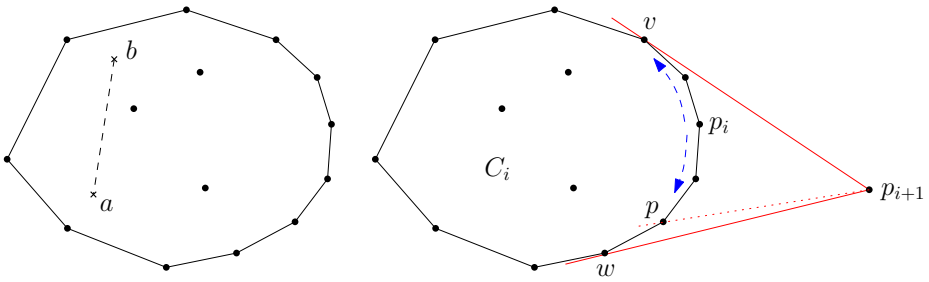


Abbildung 6.25: (i) Die konvexe Hülle von Punkten in der Ebene. (ii) Punkt p_{i+1} kommt hinzu.

Zu Beginn werden die n Punkte nach aufsteigenden X-Koordinaten sortiert; der Einfachheit halber wird angenommen, dass diese Koordinaten paarweise verschieden sind und dass außerdem keine drei Punkte auf einer Geraden liegen. Jetzt verfährt man inkrementell. Die konvexe Hülle von p_1, p_2, p_3 ist einfach das Dreieck mit diesen Eckpunkten. Angenommen, die konvexe Hülle C_i von $p_1, \ldots, p_i$ ist schon konstruiert, und p_{i+1} soll nun als weiterer Punkt hinzugenommen werden. Wegen der Sortierung liegt C_i links von der Senkrechten durch p_i, und p_{i+1} liegt rechts davon; siehe Abbildung 6.25 (ii). Deshalb kann p_{i+1} den Punkt p_i sehen. Wir starten bei p_i und durchlaufen den Rand von C_i nach oben und unten, bis die beiden Tangentenpunkte v und w gefunden sind. Unterwegs muss man bei jedem Eckpunkt p testen, ob die Halbgerade von p_{i+1} durch p die konvexe Hülle C_i berührt oder schneidet; das geht in konstanter Zeit, weil man nur die beiden Kanten von C_i betrachen muss, die sich in p treffen.

Wenn v und w gefunden sind, ergibt sich die neue konvexe Hülle C_{i+1}, indem man die Segmente $\overline{vp_{i+1}}$ und $\overline{wp_{i+1}}$ hinzufügt und das Randstück von C_i zwischen v und w, welches p_i enthält, entfernt.

Diese Aktion kann Kosten in $O(n)$ verursachen, wenn zwischen v und w viele Eckpunkte p liegen, die besucht werden müssen. Aber alle diese Eckpunkte werden anschließend entfernt und kommen nie wieder vor, weil sie jetzt innerhalb von C_{i+1} liegen. Also betragen die Gesamtkosten aller Einfügeoperationen $O(n)$, weil es ja nur n Punkte gibt, und eine einzelne Operation braucht im Mittel nur konstante Zeit. Damit haben wir folgendes Ergebnis:

Theorem 14 *Die konvexe Hülle von n Punkten in der Ebene lässt sich nach Sortieren der Punkte in Zeit $O(n)$ konstruieren, insgesamt also in Zeit $O(n \log n)$.*

Der Speicherplatzbedarf ist linear, wenn man die konvexen Hüllen als verkettete Listen verwaltet. Beide Schranken sind optimal. Der oben vorgestellte Algorithmus ist sowohl ein Beispiel für *Inkrementelle Konstruktion* als auch für *Sweep*, je nach Betrachtungsweise.

6.6.2 Randomisierter Input: Bucketsort

Bei manchen Algorithmen hängt die Laufzeit stark von der Eingabe ab. Dann kann es sinnvoll sein, die *mittlere Laufzeit* zu betrachten. Dazu muss man wissen, mit welcher Wahrscheinlichkeit jede mögliche Eingabe an die Reihe kommt.

Betrachten wir als Beispiel den Algorithmus *Bucketsort*, einen einfachen Verwandten vom Radixsort aus Kapitel 2. Um n reelle Zahlen x aus dem Intervall $[0, 1)$ zu sortieren, werden zunächst alle Eingabewerte in denselben Behälter (Bucket) gelegt, die im gleichen Intervall $[\frac{i}{n}, \frac{i+1}{n})$ liegen, für $i = 0, 1, 2, \ldots, n-1$. Nun werden die Zahlen eines jeden Behälters einzeln

sortiert, zum Beispiel mit einem quadratischen Verfahren wie Insertionsort. Schließlich werden die sortierten Behälterinhalte von links nach rechts ausgegeben.

Offensichtlich braucht Bucketsort Zeit $\Omega(n^2)$, wenn alle n Eingabewerte im selben Intervall $[\frac{i}{n}, \frac{i+1}{n})$ liegen, denn dann wird die gesamte Arbeit von der quadratischen Prozedur Insertionsort geleistet. Allgemein hat Bucketsort die Laufzeit

$$T(n) \;=\; d\,n \;+\; c\sum_{i=0}^{n-1} B_i^2,$$

wobei B_i die Anzahl der Eingabewerte im Intervall $[\frac{i}{n}, \frac{i+1}{n})$ bedeutet; dabei misst $d\,n$ den Aufwand für die Aufteilung in Buckets.

Die Summe der Quadrate der B_i entspricht genau der Anzahl aller Paare (x, y) von Eingabewerten, bei denen x und y im selben Intervall liegen. Das sind alle Paare (x, x), und außerdem gewisse Paare (x, y) mit $x \neq y$. Wenn man nun annehmen darf, dass die Eingabewerte gleichverteilt und unabhängig aus dem Intervall $[0, 1)$ gezogen werden, dann liegt zu gegebenem x der Wert y genau mit Wahrscheinlichkeit $\frac{1}{n}$ im gleichen Intervall wie x, denn die Intervalle haben alle die gleiche Länge $\frac{1}{n}$, und die Ziehung von y hängt nicht von der Ziehung von x ab. Für den Erwartungswert der Laufzeit ergibt sich demnach

$$\begin{aligned} E(T(n)) &= d\,n \;+\; c\Big(\sum_x 1 \;+\; \sum_x \sum_{y\neq x} \frac{1}{n}\Big) \\ &= d\,n \;+\; c\Big(n \;+\; n(n-1)\,\frac{1}{n}\Big) \\ &< (d+2c)\,n \end{aligned}$$

aus der Linearität des Erwartungswerts. Also gilt:

Theorem 15 *Bei gleichverteilten und unabhängig gezogenen Eingabewerten aus* $[0, 1)$ *hat Bucketsort die mittlere Laufzeit* $O(n)$.

Auch bei Bucketsort finden, wie schon bei Radixsort, nicht nur Schlüsselvergleiche statt, denn die Eingabewerte x müssen ja ihren Intervallen zugeordnet werden. Um die Bedingung $\frac{i}{n} \leq x < \frac{i+1}{n}$, oder äquivalent: $i \leq nx < i + 1$, testen zu können, wird die Rundungsfunktion $\lfloor nx \rfloor$ benötigt. Daher widerspricht Theorem 15 nicht der unteren Schranke in Theorem 2.

In manchen Fällen ist man nicht daran interessiert, die Laufzeit über alle möglichen Eingaben zu mitteln, sondern nur in der Umgebung besonders "schwieriger" Inputs; so kann man Aufschluss darüber gewinnen, ob es sich um singuläre Ausreißer handelt.

6.6.3 Randomisierter Algorithmus: Bestimmung des Maximums

In den meisten Fällen ist nicht bekannt, welcher Wahrscheinlichkeitsverteilung die Eingaben unterliegen; dann muss ein Algorithmus mit den konkreten Eingabewerten arbeiten, die er bekommt. Trotzdem kann er Zufall ins Spiel bringen, indem er würfelt, welche von mehreren möglichen Aktionen als nächste ausgeführt werden soll.

Wir beginnen mit einem ganz einfachen Beispiel. Um das Maximum von n Schlüsseln a_i zu bestimmen, verfährt man wie folgt: Eine Variable max wird zunächst auf a_1 gesetzt. Dann wird für $i = 2$ bis n getestet, ob $\max < a_i$ gilt, und gegebenfalls das bisherige Maximum durch $\max := a_i$ aktualisiert. Das erfordert $n-1$ Tests und schlimmstenfalls $n-1$ Aktualisierungen, wenn die Schlüssel aufsteigend sortiert sind.

Angenommen, ein Test kostet Zeit T, eine Aktualisierung aber Zeit $A >> T$, weil dabei Daten bewegt werden müssen. Dann würde man gern die Anzahl der Aktualisierungen begrenzen. Ein naheliegender Ansatz besteht darin, die gegebenen n Schlüssel zunächst in eine zufällige Reihenfolge zu bringen und dann obigen Algorithmus zu starten.

Wie wahrscheinlich ist es nun, dass die Aktualisierung max := a_i ausgeführt werden muss? Das geschieht genau dann, wenn a_i der größte Schlüssel in $A_i := \{a_1, a_2, \ldots, a_{i-1}, a_i\}$ ist. Die Wahrscheinlichkeit hierfür beträgt $\frac{1}{i}$, denn nach der anfänglichen Zufallspermutation ist a_i eine beliebige Zahl in der i-elementigen Menge A_i. Also hat der randomisierte Algorithmus die zu erwartenden Kosten

$$\begin{aligned} E(T(n)) &= (n-1)\cdot T \;+\; \sum_{i=2}^{n} A \cdot \frac{1}{i} && (6.6)\\ &< (n-1)\cdot T \;+\; A \cdot \int_1^n \frac{1}{x}\,dx && (6.7)\\ &< n\cdot T \;+\; \ln(n)\cdot A, && (6.8) \end{aligned}$$

was deutlich besser ist als der Worst Case $n \cdot (T + A)$. Die Abschätzung der Summe durch das Integral ergibt sich durch Vergleich der Flächen unter den Funktionen.

6.6.4 Randomisierter Algorithmus: Quicksort

Eine einfache und weit verbreitete Variante des Sortierens mit Schlüsselvergleichen funktioniert folgendermaßen: Gegeben ist ein Array A mit n Schlüsseln $a_1, \ldots, a_n$, die sortiert werden sollen. Eine dieser Zahlen wird als sogenanntes *Pivotelement* p ausgewählt. Dafür gibt es verschiedene Möglichkeiten; zum Beispiel könnte man einfach $p := a_n$ setzen. Jetzt werden alle $a_i < p$ in einer Folge L zusammengefasst, und alle $a_j > p$ in einer Folge R; das geht in Zeit $O(n)$. Nun wendet man Quicksort rekursiv auf L und R an, schreibt die sortierten Teilfolgen hintereinander, dazwischen p, und ist fertig. (In der Literatur wird beschrieben, wie man direkt im Array A die Folge L links von p anordnet und R rechts von p, ohne zusätzlichen Speicherplatz zu benutzen.)

Die Effizienz von Quicksort hängt von der Größe von L und R ab und damit von der Wahl des Pivotelements p. Im besten Fall sind L und R stets gleich groß. Dann ergibt sich für die Laufzeit die Rekursion $T(n) = 2T(\frac{n}{2}) + c\,n$, woraus $T(n) \in O(n \log n)$ folgt; vergleiche Kapitel 3. Tatsächlich könnte man in Zeit $O(n)$ den sogenannten *Median* von $a_1, \ldots, a_n$ bestimmen, also das Element, welches in der Mitte der sortierten Folge liegt, und damit $|L| \approx |R|$ garantieren. Der Median-Algorithmus ist aber selbst rekursiv, und die Konstante in der $O(n)$-Abschätzung recht hoch.

Der schlimmste Fall für Quicksort tritt ein, wenn die Eingabefolge bereits aufsteigend sortiert ist. Wir müssen dann $O(n)$ Zeit aufwenden, um festzustellen, dass die ersten $n-1$ Elemente kleiner als $p = a_n$ sind. Dann enthält L diese $n-1$ Elemente, und R ist leer. In diesem Fall braucht Quicksort quadratische Laufzeit.

Es liegt deshalb nahe, das Pivotelement p *zufällig* in $a_1, \ldots, a_n$ auszuwählen. Dann ist jede Position von p in der sortierten Folge gleich wahrscheinlich, und für die mittlere Laufzeit ergibt sich die Rekursion

$$\begin{aligned} E(T(n)) &= c\,(n+1) + \frac{1}{n}\sum_{i=1}^{n}\Big(E(T(i-1)) + E(T(n-i))\Big) && (6.9)\\ &= c\,(n+1) + \frac{2}{n}\sum_{i=0}^{n-1} E(T(i)), && (6.10) \end{aligned}$$

denn bei beiden Summanden läuft der Index von 0 bis $n-1$. Zur Abkürzung sei $e_i := E(T(i))$. In der Rekursion (6.10) können wir die Summation durch Differenzenbildung eliminieren und bekommen die Gleichungen

$$\begin{aligned} e_{n+1}(n+1) - e_n\, n &= c\Big((n+2)(n+1) - (n+1)n\Big) + 2\sum_{i=0}^{n} e_i - 2\sum_{i=0}^{n-1} e_i \\ &= 2\,c\,(n+1) + 2\,e_n \\ (n+1)\,e_{n+1} &= 2\,c\,(n+1) + (n+2)\,e_n \\ \frac{e_{n+1}}{n+2} &= \frac{2c}{n+2} + \frac{e_n}{n+1} \end{aligned}$$

Aus der letzten Zeile ergibt sich durch iteriertes Einsetzen

$$\begin{aligned} \frac{e_{n+1}}{n+2} &= 2\,c\,\Big(\frac{1}{n+2} + \frac{1}{n+1} + \ldots + \frac{1}{2}\Big) + \frac{e_0}{1} \\ &< 2\,c\,\ln(n+2) \end{aligned}$$

wie in (6.8), wobei $e_0 = 0$ ist. Damit haben wir folgendes Ergebnis:

Theorem 16 *Die mittlere Laufzeit von Quicksort, angesetzt auf eine beliebige Eingabe von n Schlüsseln, liegt in $O(n \log n)$.*

Quicksort und Mergesort sind beides Algorithmen vom Typ Divide-and-Conquer. Quicksort leistet die Hauptarbeit bei der Aufteilung in Teilmengen, Mergesort beim Zusammenfügen der Ergebnisse.

Im nächsten Kapitel werden wir Hash-Verfahren untersuchen, bei denen sowohl randomisierte Eingaben als auch amortisierte Kosten eine Rolle spielen.

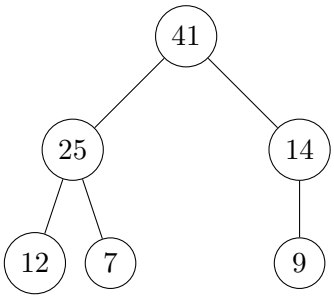
6.7 Heaps

In diesem Kapitel lernen wir eine Datenstruktur kennen, die sehr ähnlich zu den bisher behandelten Bäumen ist. Sie erbt die Vorzüge eines balancierten Baumes, muss jedoch nicht die volle Suchbaumeigenschaft erfüllen. Als Anwendungen der Datenstruktur betrachten wir zwei Beispiele. Zum einen den Sortieralgorithmus Heapsort, zum anderen die Prioritätenwarteschlange.

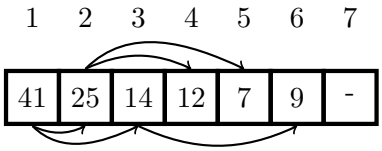
Ein Max-Heap ist ein fast vollständiger Binärbaum, welcher folgende Eigenschaften erfüllt:

- Der Baum ist, mit Ausnahme der untersten Ebene, vollständig besetzt. Die unterste Ebene ist von links nach rechts bis zu einem gewissen Punkt vollständig besetzt.
- Der Wurzelknoten enthält den größten Schlüsselwert. Alle anderen im Max-Heap gespeicherten Schlüssel sind nicht größer. Der linke und rechte Teilbaum der Wurzel ist jeweils ein Max-Heap.

Ganz ähnlich lässt sich auch ein Min-Heap definieren, in welchem der Elternknoten stets einen Wert kleiner oder gleich dem Schlüsselwert der Kinder enthält. In Abbildung 6.26a wird ein Beispiel für einen Max-Heap gegeben. Ein Heap kann sehr einfach in einem Array A abgelegt werden (s. Abbildung 6.26a). Dies hat unter anderem den Vorteil, dass für einen Wert in Zelle i leicht der Elternknoten bzw. das linke und rechte Kind bestimmt werden kann:



(a) Max-Heap



(b) Max-Heaps im Array

Abbildung 6.26: Beispiel eines Max-Heaps für die Schlüsselwerte 7,9,12,14,25,41 mit Einbettung im Array.

parent(i) : return $\lfloor i/2 \rfloor$;
links(i) : return $2i$;
rechts(i) : return $2i + 1$;

Im Folgenden werden wir uns nur mehr mit Max-Heaps beschäftigen. Die Bezeichnung Heap steht daher stets für einen Max-Heap.

Bevor wir uns damit befassen, wie ein Heap gebaut werden kann und wie Schlüssel eingefügt bzw. gelöscht werden können, betrachten wir den grundlegenden Algorithmus 15. Er stellt die Heap-Eigenschaft für einen Knoten i her, wenn sowohl der linke als auch der rechte Teilbaum von i bereits die Heap-Eigenschaft erfüllen.

Prozedur 15 Max-Heapify(A, i)

```
1: l := links(i), r := rechts(i);
2: if l ≤ A.heap-size und A[l] > A[i] then
3:    max := l;
4: else
5:    max := i;
6: end if
7: if r ≤ A.heap-size und A[r] > A[max] then
8:    max := r;
9: end if
10: if max ≠ i then
11:    vertausche A[i] und A[max];
12:    Max-Heapify(A, max);
13: end if
```

Heap-Eigenschaft herstellen: Ist im Knoten i die Heap-Eigenschaft nicht erfüllt, so vertauscht Algorithmus 15 den Schlüssel von i mit dem Schlüssel des Kindes, das den maximalen Schlüsselwert enthält. Aus diesem Grund ist die Heap-Eigenschaft nun mehr höchstens in dem Teilbaum nicht mehr erfällt, dessen Wurzel nun den Schlüssel von i enthält. Um dieses Problem zu lösen, ruft der Algorithmus sich rekursiv auf diesem Teilbaum auf. Der Schlüssel des Knotens i sinkt somit solange ab, bis die Heap-Eigenschaft im kompletten Baum erfüllt ist. Da der Binärbaum vollständig ist, hat er Höhe $O(\log n)$, wobei n die Anzahl der gespeicherten Knoten ist. Wie wir im Algorithmus sehen, so wird die Prozedur höchstens einmal rekursiv aufgerufen. Die Laufzeit von Max-Heapify hängt wegen der linearen Rekursion im Wesentlichen von der Höhe des Heaps ab, ist also $O(\log n)$.

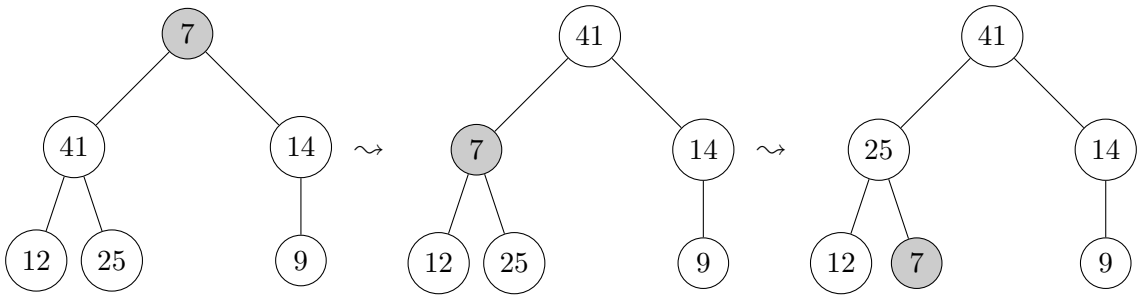


Abbildung 6.27: Beispiel für einen Aufruf von Max-Heapify$(A, 1)$. Die Heap-Eigenschaft ist im linken und rechten Teilbaum der Wurzel bereits erfüllt. Der Schlüssel der Wurzel wird mit dem größten Schlüssel der Kinder vertauscht und im Anschluss der Algorithmus rekursiv auf dem Teilbaum des linken Kindes aufgerufen, also Max-Heapify$(A, 2)$ aufgerufen.

Heaps bauen: Aus dem Algorithmus zum Herstellen der Heap-Eigenschaft können wir leicht einen Algorithmus zum Bau eines Heaps in einem Array A herleiten:

Prozedur 16 Build-Heap(A)

1: $A.\textit{heap-size} = A.\textit{length}$;
2: **for** $i = \lfloor A.\textit{length}/2 \rfloor$ to 1 **do**
3: Max-Heapify(A, i);
4: **end for**

Die Blätter erfüllen bereits die Heap-Eigenschaft. Daher können wir für alle Ebenen nacheinander, von der Blattebene bis zur Wurzelebene die Heap-Eigenschaft mit Aufrufen von Max-Heapify herstellen. Insgesamt liegt die Bauzeit somit in $O(n \log n)$. In den Übungen werden wir durch eine genauere Analyse zeigen, dass diese Bauzeit tatsächlich sogar linear ist.

6.7.1 Heapsort

Die Heap-Datenstruktur können wir auch verwenden, um Zahlenketten effizient zu sortieren. Dies veranschaulicht außerdem ein weiteres Entwurfsparadigma für Algorithmen, nämlich die Verwendung einer geeigneten Datenstruktur. Der resultierende Algorithmus Heapsort verbindet die Vorteile der bereits behandelten Algorithmen *Insertionsort* (Abschnitt 1.2) und *Mergesort* (Abschnitt 2.1). Wie auch Insertionsort benötigt er zusätzlich zur Eingabe nur konstant viel Speicherplatz. Solche Sortierverfahren werden auch als *in place* Verfahren bezeichnet. Darüber hinaus hat der Algorithmus eine worst-case Laufzeit von $O(n \log n)$, genau wie Mergesort. Er ist damit in puncto Laufzeit und Speicherplatz optimal.

Prozedur 17 Heapsort(A)

1: Baue Max-Heap für Array A;
2: **for** $i = A.\textit{length}$ to 2 **do**
3: vertausche $A[1]$ und $A[i]$;
4: $A.\textit{heap-size} = A.\textit{heap-size} - 1$;
5: Max-Heapify$(A, 1)$;
6: **end for**

Zunächst baut Algorithmus 17 über dem Array A einen Max-Heap auf. Aufgrund der Heap-

Eigenschaft, steht der größte Wert anschließend in der Wurzel des Heaps, also $A[1]$. Solange der Heap noch mindestens 2 Werte enthält, vertauschen wir den größten Wert in der Wurzel mit dem Wert, welcher in der untersten Ebene am weitesten rechts steht. Diesen Knoten können wir durch Reduktion der Größe des Heaps um 1 einfach herauslöschen. Der Rest des Arrays, welcher nicht vom Heap belegt ist, ist bereits sortiert. Im linken und rechten Teilbaum der Wurzel ist dann noch immer die Heap-Eigenschaft erfällt, nicht jedoch in der Wurzel. Um dies zu beheben, wird schließlich Algorithmus 15 auf dem Wurzelknoten aufgerufen.

6.7.2 Die Prioritätenwarteschlange

Als zweites Anwendungsbeispiel sehen wir im folgenden Abschnitt, wie wir Heaps zum Bau einer Prioritätenwarteschlange verwenden können. Eine Prioritätenwarteschlange ist eine Datenstruktur über einer Menge S von Elementen. Jedem Element x aus der Menge S ist zusätzlich ein Schlüsselwert (Priorität) zugeordnet. Wie auch bei den Heaps, unterscheiden wir zwei Typen, je nachdem, ob das Warten mit auf- oder absteigender Priorität erfolgt. Eine Max-Prioritätenwarteschlange unterstützt die folgenden Anfragen:

- MAX(S): gibt das Element mit der höchsten Priorität (größter Schlüssel) zurück;
- EXTRACT-MAX(S): entfernt das Element mit der höchsten Priorität (dem größten Schlüssel) und gibt es zurück;
- INSERT(S, x): fügt das Element x zur Warteschlange S hinzu;
- INCREASE-KEY(S, x, k): erhöht die Priorität des Elements x auf den Wert k (wobei $k >$ Schlüssel(x) gelten muss);

Diese Operationen erlauben es, Prioritätenwarteschlangen vielfältig einzusetzen. Beispielsweise bei der Abarbeitung von Rechenprozessen bzw. Rechenjobs auf mehreren CPUs oder Rechnern. Außerdem wird die Datenstruktur im Algorithmus von Dijkstra benötigt, wie wir später sehen werden.

Befassen wir uns nun damit, wie diese Operationen implementiert werden können. Sehr einfach ist es, das Element mir der höchsten Priorität in konstanter Zeit auszugeben:

Prozedur 18 Max(S)

```
1: return A[1];
```

Auch wie wir dieses Element aus dem Heap entfernen können, haben wir bereits in Sortieralgorithmus 17 gesehen. Wir ersetzen den Wurzelknoten durch den Knoten, der in der untersten Ebene am weitesten rechts steht. Anschließend lassen wir den Wurzelknoten absinken, bis die Heap-Eigenschaft erfüllt ist:

Prozedur 19 Extract-Max(A)

```
1: if A.heap-size < 1 then
2:    error "Heap ist leer"
3: end if
4: max = A[1];
5: A[1] = A[A.heap-size];
6: A.heap-size = A.heap-size − 1;
7: Max-Heapify(A, 1);
8: return max;
```

Die Laufzeit von Extract-Min hängt im Wesentlichen vom Aufruf von Max-Heapify ab, da zusätzlich nur konstante Zeit benötigt wird. Somit liegt die Laufzeit von Extract-Min in $O(\log n)$.

Erhöhen wir den Schlüssel eines Knotens, ist die Heap-Eigenschaft nicht mehr unbedingt erfüllt. In diesem Fall können wir jedoch nicht die Heapify-Prozedur benutzen, da diese einen Knoten im Heap absinken lässt. Wir benötigen hingegen eine Prozedur, die den entsprechenden Knoten so lange im Heap aufsteigen lässt, bis die Heap-Eigenschaft erfüllt ist:

Prozedur 20 Increase-Key(A, i, key)

```
1: if key < A[i] then
2:    error "Aktueller Schlüssel ist kleiner als key"
3: end if
4: A[i] = key;
5: while i > 1 und A[parent(i)] < A[i] do
6:    vertausche A[i] und A[parent(i)];
7:    i = parent(i);
8: end while
```

Der Beweis der Korrektheit der Prozedur ist eine gute Übung. Die Laufzeit hängt im Wesentlichen von der **while**-Schleife ab, in jedem Schleifendurchlauf sinkt jedoch der Abstand des Knotens i zur Wurzel des Heaps. Die Durchläufe der Schleife sind somit durch die Höhe des Heaps ($O(\log n)$) beschränkt.

Diese Prozedur, die einen Knoten im Heap aufsteigen lässt, können wir nun auch verwenden, um einen neuen Knoten einzufügen. Dazu legen wir zunächst einen Knoten mit dem Schlüsselwert $-\infty$ an. Anschließend erhöhen wir den Schlüssel dieses Knotens auf den gewünschten Wert und lassen ihn im Heap aufsteigen, bis die Heap-Eigenschaft erfüllt ist:

Prozedur 21 Insert(A, key)

```
1: A.heap-size = A.heap-size + 1;
2: A[A.heap-size] = ∞;
3: Increase-Key(A, A.heap-size, key);
```

Ein neuer Schlüssel kann also in $O(\log n)$ Zeit eingefügt werden. Insgesamt können wir also mit einem Heap jede der Operationen der Prioritätenwarteschlange in $O(\log n)$ Zeit ausführen.

6.8 Hashing

Wir haben gesehen, dass sich balancierte Bäume sehr gut dazu eignen, sortierte Schlüssel zu speichern. Für einen gegebenen Schlüssel findet sich so der zugehörige Datensatz in $O(\log n)$ Zeit. Das Einfügen und Entfernen eines neuen Objektes mit neuem Schlüssel hat die gleiche Zeitkomplexität.

Wie man sich leicht überlegt, gelten die gleichen Laufzeiten auch für die *mittlere* Laufzeit dieser Operationen. Wenn wir also eine Folge von n Operationen betrachten und $K(i)$ die Kosten der i-ten Operation beschreibt, so erhalten wir durch

$$\frac{1}{n}\sum_{i=1}^{n} O(i)$$

die mittleren Kosten der Operationen.

Für das Einfügen und Entfernen liegen die mittleren Kosten in jedem Fall in $\Theta(\log n)$, da bei jedem Einfügen und bei jedem Entfernen die Mindesttiefe $O(\log n)$ aus dem Beweis von Lemma 9 zu Durchlaufen ist und andererseits die Kosten nach den Betrachtungen des vorherigen Kapitels stets in $O(\log n)$ liegen.

Für Datenbanken mit sehr größen Datenmengen verzichtet man beim erwähnten **Wörterbuch-Problem** auf die sortierte Speicherung der Schlüssel. Stattdessen sollen die Operationen (Exakte) Suche, Einfügen und Entfernen im Mittel nur konstante Zeit verursachen.

In solchen Fälle kommt das sogenannte *Hashing* zur Anwendung. Gegeben ist also eine Menge von Objekten (Daten) die über einen eindeutigen Schlüssel (ganze Zahl) identifizierbar sind und es wird eine Struktur zur Speicherung gesucht, die mindestens die Operationen

- (Exakte) Suche eines Objektes mit Schlüssel x
- Einfügen eines Objektes mit Schlüssel x
- Entfernen eines Objektes mit Schlüssel x

effizient (im Mittel konstant) ausgeführt werden können. Dafür wird das Schlüsseluniversum partitioniert. Die Position des Datenelementes im Speicher ergibt sich durch eine Berechnung aus dem Schüssel.

Formal steht uns für die Verwaltung einer ungeordneten Menge S von Schlüsseln

1. ... ein Feld $T[0..m-1]$ für m Behälter $T[0], T[1], \ldots, T[m-1]$ die sogenannte *Hashtabelle* und
2. ... eine *Hash-Funktion* $h : U \to [0..m-1]$ die das Universum der Schlüssel auf das Feld T abbildet

zur Verfügung. Ziel ist es, für eine Schlüsselmenge $S \subseteq U$ für jedes Element $x \in S$ das zugehörige Datenobjekt in $T[h(x)]$ zu speichern. Im Folgenden lassen wir der Übersichtlichkeit halber die Datenobjekte weg und Speichern lediglich die Schlüssel in der Hashtabelle.

Beispiel: Sei $m = 5$ und $S = \{8, 22, 16, 29\}$ und $h(x) = x \bmod 5$. Die Hashtafel sieht dann wie folgt aus:

0	
1	16
2	22
3	8
4	29

Falls wir hier einen weiteren Schlüssel 21 verwenden gilt $h(21) = 1$ und wir sprechen dann von einer *Kollision*. Für eine Menge S von n Schlüsseln und für eine Menge von m Behältern $T[0], \ldots, T[m-1]$ mit $n \leq m$ heißt die Hashfunktion h *perfekt für* S falls es keine Kollisionen bei der Zuordnung gibt. Die Hashfunktion soll effizient ausgewertet werden können.

6.8.1 Kollisionsbehandlung mit verketteten Listen

Eine sehr einfache Lösung in der Kollisionsbehandlung ist die Verwendung verketteter Listen in den Einträgen $T[i]$ der Hashtabelle. Doppelte Einträge werden dann an den Anfang der Liste gesetzt und somit kann nach Auswertung der Hashfunktion in kostanter Zeit ein neuer Schlüssel eingefügt werden.

Beispiel: Für $m = 3$ betrachten wir die Schlüsselmenge $S = \{10, 22, 16, 29, 17, 3\}$ und die Hashfunktion $h(x) = x \bmod 3$. Die Abbildung beschreibt die Hashtabelle nach dem sukzessiven Einfügen der Schlüssel mit Kollisionsvermeidung mittels verketteter Listen.

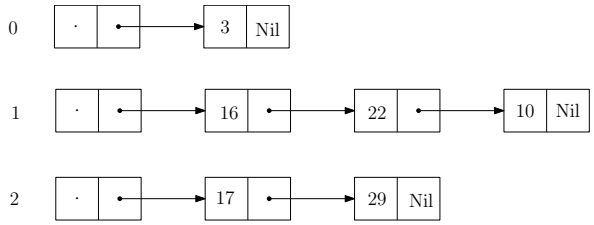


Abbildung 6.28: Die Kollisionsbehandlung durch die Verwendung von verketteten Listen $S = \{10, 22, 16, 29, 17, 3\}$ und $h(x) = x \bmod 3$.

Wir wollen nun die Verwendung von verketteten Listen für die Kollisionsbehandlung analysieren und dabei die mittleren Kosten berücksichtigen. Die Kosten einer Operation (Exakte) Suche und Entfernen ergeben sich jeweils durch das Berechnen des Hashwertes und den anschließenden linearen Suchdurchlauf durch die Liste des entsprechenden Eintrags.

Die Funktion $\delta_h(x, S)$ bezeichnet die Anzahl der Elemente $y \in S$ für die Funktion $h(x) = h(y)$ gilt. Dann kann eine Operation (Exakte) Suche und Entfernen für einen Schlüssel x in Zeit $O(1 + \delta_h(x, S))$ ausgeführt werden.

Wir betrachten dabei auch eine Folge von n möglichen zufälligen Operationen (Exakte) Suche, Einfügen und Entfernen und wollen die gesamte Laufzeit dafür nach oben Abschätzen.

Wir machen dafür die folgende Annahmen:

1. Die Hashfunktion $h : U \to [0..m]$ streut das Universum gleichmäßig über das Intervall $[0..m]$. Das ist zum Beispiel für $h(x) = x \bmod m$ gegeben.

2. Sämtliche Elemente des Universums sind mit gleicher Wahrscheinlichkeit Argument der nächsten Operation.

Falls nun x_k der Schlüssel ist, der in der k-ten Aktion verwendet wird, dann folgt nun, dass die Wahrscheinlichkeit, dass $h(x_k) = i$ ist für $i \in [0..m]$ durch $\frac{1}{m}$ gegeben ist, kurz $\text{prob}(h(x_k) = i) = \frac{1}{m}$.

Theorem 17 *Für die einzelnen Operationen (Exakte) Suche, Einfügen und Entfernen bezüglich einer Hashtabelle für die Schlüsselmenge S wird im worst-case $\Omega(|S|)$ Zeit benötigt.*

Unter den obigen Annahmen ist der Erwartungswert für die von einer Folge von n Operationen (Exakte) Suche, Einfügen und Entfernen verursachten Kosten kleiner gleich $\left(1+\frac{\alpha}{2}\right)n$, wobei $\alpha = \frac{n}{m}$ der maximale Belegungsfaktor der Hashtafel ist.

Beweis. Der erste Teil des Theorems ist leicht zu sehen. Im schlechtesten Fall gilt für alle Schlüssel $x \in S$, dass $h(x)$ auf den gleichen Wert i_0 abbildet, dann muss im schlimmsten Fall bei jeder Operation eine Liste der größe $|S|$ vollständig durchlaufen werden.

Für den zweiten Teil der Behauptung verwenden wir für die ersten k Operationen und für $i = 1, \ldots, m$ einen Zähler $l_k(i)$, so dass $l_k(i)$ nach k Operationen in jedem Fall einen Wert größer gleich der dann aktuellen Listenlänge der Liste von $T_k[i]$ hat. Falls nun eine weitere Operation die i-te Liste betrifft, wird $l_{k+1}(i) := l_k(i) + 1$ gesetzt und $l_k(i)$ ist dann stets eine obere Schranke der aktuellen Listenlänge nach k Operationen.

Anfangs sind alle Zähler auf 0 gesetzt. Immer wenn eine Operation die i-te Liste betrifft, erhöhen wir den Wert von $l_k(i)$ um Eins, unabhängig von der Art der Operation. Sei EK_{k+1} der Erwartungswert des Aufwandes der $(k+1)$-ten Operation bei insgesamt n Operationen. Nehmen wir an, dass x_{k+1} der Schlüssel der $(k+1)$-ten Operation ist, und es gelte $h(x_{k+1}) = i$. Sei nun $\mathrm{prob}(l_k(i) = j)$ die Wahrscheinlichkeit, dass der Zähler $l_k(i)$ nach der k-ten Operation den Wert j hat. Dann ist

$$\mathrm{EK}_{k+1} = 1 + \sum_{j=0}^{k} \mathrm{prob}(l_k(i) = j) \cdot j \,,$$

da wir den Schlüssel auswerten müssen und relativ zur Wahrscheinlichkeit der Größe von $l_k(i) = j$ die entsprechende Länge j durchlaufen müssen.

Für $j \geq 1$ gilt nun aber

$$\mathrm{prob}(l_k(i) = j) = \binom{k}{j} \left(\frac{1}{m}\right)^j \left(1 - \frac{1}{m}\right)^{k-j} .$$

Diese Aussage läßt sich leicht kombinatorisch erklären. Es gibt insgesamt $\binom{k}{j}$ j-elementige Sequenzen aus den ersten k-Operationen die zu beachten sind. Für jede dieser Sequenzen ist die Wahrscheinlichkeit, dass genau j-Operationen die i-te Liste betreffen $\left(\frac{1}{m}\right)^j \left(1 - \frac{1}{m}\right)^{k-j}$. Genauer, das Produkt der jeweiligen Wahrscheinlichkeit $\frac{1}{m}$, dass eine einzelne der j-Operation die i-te Liste betrifft mal dem Produkt der jeweiligen Wahrscheinlichkeit $\left(1 - \frac{1}{m}\right)$, dass eine einzelne der restlichen $k-j$-Operationen *nicht* die i-te Liste betrifft.

Zunächst berechnen wir nun EK_{k+1}:

$$\begin{aligned}
\mathrm{EK}_{k+1} &= 1 + \sum_{j=0}^{k} \binom{k}{j} \left(\frac{1}{m}\right)^j \left(1 - \frac{1}{m}\right)^{k-j} \cdot j \\
&= 1 + \frac{k}{m} \sum_{j=1}^{k} \binom{k-1}{j-1} \left(\frac{1}{m}\right)^{j-1} \left(1 - \frac{1}{m}\right)^{k-j} \\
&\qquad \left(\tfrac{k}{j} \text{ und } \tfrac{1}{m} \text{ Ausklammern}\right) \\
&= 1 + \frac{k}{m} \left(1 - \frac{1}{m}\right)^{-1} \sum_{j=1}^{k} \binom{k-1}{j-1} \left(\frac{1}{m}\right)^{j-1} \left(1 - \frac{1}{m}\right)^{k-(j-1)} \\
&\qquad \left(\left(1 - \tfrac{1}{m}\right)^{-1} \text{ Ausklammern}\right)
\end{aligned}$$

$$
\begin{aligned}
&= 1 + \frac{k}{m}\frac{m}{m-1}\left(\frac{1}{m} + \left(1 - \frac{1}{m}\right)\right)^{k-1} \\
&\qquad \text{(Binominalkoeffizienten)} \\
&= 1 + \frac{k}{m-1}
\end{aligned}
$$

Nun kann der Erwartungswert aller n Operation, $\mathrm{EK}(n)$, folgendermaßen abgeschätzt werden:

$$
\begin{aligned}
\mathrm{EK}(n) &= \sum_{k=1}^{n} \mathrm{EK}_k = \sum_{k=0}^{n-1} \mathrm{EK}_{k+1} \\
&= \sum_{k=0}^{n-1} \left(1 + \frac{k}{m-1}\right) \\
&= n + \frac{1}{m-1}\sum_{k=0}^{n-1} k = n + \frac{1}{m-1}\frac{n(n-1)}{2} \\
&= n \cdot \left(1 + \frac{n-1}{2(m-1)}\right) < \left(1 + \frac{n}{2m}\right) \cdot n
\end{aligned}
$$

Wobei wir für den letzten Schritt $n < m$ annehmen. Somit gilt für $\alpha = \frac{n}{m}$, dass $\left(1 + \frac{\alpha}{2}\right) n$ eine obere Schranke für den erwarteten Aufwand der n Operationen darstellt. □

Falls $\alpha = \frac{n}{m}$ konstant bleibt, sagt dass obige Theorem, dass im Mittel nur konstant viel Aufwand zu erwarten ist, also eine deutliche Verbesserung im Vergleich zur AVL-Baumlösung.

Abschließend erwähnen wir noch, wie sich die Bedingung, dass $\alpha = \frac{n}{m}$ konstant bleibt gewährleisten können, ohne an Performanz zu verlieren.

Wir passen das m sukzessive an. Dazu verwenden wir eine Folge von Hashtafeln $T_0, T_1, T_2, \ldots, T_i, \ldots$ der Größen $m, 2m, \ldots, 2^i m, \ldots$ mit Hashfunktionen $h_i : U \to [0..2^i m - 1]$. Die Tafeln werden so verwendet, dass der aktuelle Belegungsfaktor α stets strikt zwischen $\frac{1}{4}$ und 1 liegt.

Die Tabelle T_i muss dann umkopiert werden, wenn der Belegungsfaktor aus dem Intervall herausfällt:

α **wird zu 1:** Wir speichern alle $2^i m$ Elemente von T_i nach T_{i+1} in $\Theta(2^i m)$ Zeit. Der Belegungsfaktor von T_{i+1} ist dann $\frac{1}{2}$. Bis zur nächsten Umspeicherung müssen mindestens $\frac{1}{4}2^{i+1}m$ Operationen durchgeführt werden bevor die nächste Umspeicherung notwendig wird.

α **wird zu** $\frac{1}{4}$**:** Wir speichern alle $\frac{1}{4}2^i m$ Elemente von T_i nach T_{i-1} in $\Theta(\frac{1}{4}2^i m)$ Zeit. Der Belegungsfaktor von T_{i-1} ist dann $\frac{1}{2}$ und mindestens $\frac{1}{4}2^{i-1}m$ Operationen können durchgeführt werden bevor die nächste Umspeicherung notwendig wird.

In beiden Fällen sind die Kosten für die Umspeicherung maximal zweimal so groß, wie die Anzahl der Operationen. Die Kosten für das Umspeichern verteilen wir deshalb auf die einzelnen Operationen. Der Erwartungswert $O(1)$ für die mittleren Kosten bleibt dann erhalten. Der Aufwand amortisiert sich hier über die Zeit. Auch wenn gelegentlich viele Elemente umkopiert werden müssen, so ist der Aufwand im Mittel gering.

6.9 Union-Find Datenstruktur

Neben der Speicherung einer Menge von Objekten ist es manchmal nützlich, ein System von Mengen zu speichern. Die Union-Find Datenstruktur dient zur Verwaltung einer Menge von disjunkten Mengen $S_1, \ldots, S_k$. Ein Universum $U = \{x_1, x_2, \ldots, x_n\}$ von Elementen ist gegeben und wir haben anfangs n disjunkte Mengen $S_i = \{x_i\}$. Über ein Array $U[1..n]$ können wir stets auf die Elemente durch $U[i] = x_i$ zugreifen. Jede Menge besitzt einen eindeutigen Repräsentaten $s_i \in S_i$, Im Folgenden beschreiben wir die Mengen der Einfachheit halber über die Indizes des Arrays U, das heißt $S_i = \{i\}$ und auch $s_i = i$.

Die Datenstruktur soll die folgenden Operationen effizient unterstützen:

- UNION(x, y): Falls zwei Mengen S_x und S_y mit $x \in S_x$ und $y \in S_y$ existieren, so werden diese entfernt und durch die Menge $S_x \cup S_y$ ersetzt. Der neue Repräsentant von $S_x \cup S_y$ kann ein beliebiges Element dieser vereinigten Menge sein.
- FIND(x): Liefert den Repräsentanten der Menge S mit $x \in S$ zurück.

Beispiel: Im folgenden Beispiel soll $x : A$ bedeuten, dass wir eine Menge A mit Repräsentant x gespeichert haben. Welches Element nach einer UNION-Operation Repräsentant der neuen Menge wird, ist nicht eindeutig bestimmt. Es wurde jeweils ein beliebiges Element dafür ausgewählt.

Operation	Zustand der Datenstruktur
Initialisierung	$1 : \{1\}$, $2 : \{2\}$, $3 : \{3\}$, $4 : \{4\}$, $5 : \{5\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(1,2)	$2 : \{1, 2\}$, $3 : \{3\}$, $4 : \{4\}$, $5 : \{5\}$
UNION(3,4)	$2 : \{1, 2\}$, $3 : \{3, 4\}$, $5 : \{5\}$
FIND(1)	Ausgabe: 2, keine Zustandsänderung
UNION(1,5)	$2 : \{1, 2, 5\}$, $3 : \{3, 4\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(5,4)	$3 : \{1, 2, 3, 4, 5\}$

Intern kann eine solche Struktur effizient wie folgt realisiert werden. Eine einzelne Menge S_i wird durch eine verkettete Liste dargestellt, wobei aber jedes einzelne Listenelement auch einen Zeiger auf den Kopf S_i der Liste speichert. Der Kopf der Liste enthält den Namen S_i, den Repräsentaten i und die Größe der Liste. Zusätzlich verwenden wir ein Feld $U[1..n]$ dass für jedes Element j durch $U[j]$ einen Zeiger auf das Listenelement j in einer Liste S_i bereithält. Abbildung 6.29 illustriert die Situation für eine Menge $S_5 = \{1, 5, 4, 8\}$. Die Operation FIND(x) kann nun in $O(1)$ auf das Element x über $U[x]$ zugreifen und findet den Repräsentanten i von S_i durch einen Verweis.

Die Operation UNION(x, y) kann wie folgt realisiert werden. Sei $|S_x| \geq |S_y|$.

- Jeder Namenszeiger von S_y wird auf den Namen von S_x gesetzt.
- Die Verkettete Liste von S_x wird an S_y angehängt. Der Mengenanfangszeiger von S_x wird auf den Anfang von S_y gesetzt. Die Größen werden addiert und im Kopfelement S_x eingetragen.
- Das alte Kopfelement S_y wird entfernt.

Es ergibt sich ein Aufwand von $O(|S_y|)$.

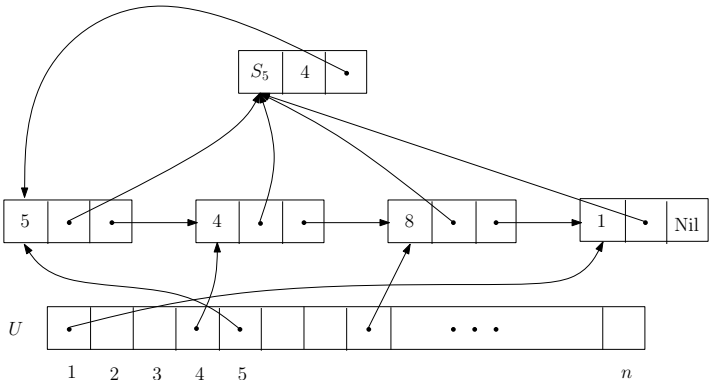


Abbildung 6.29: Die Zeigerstruktur einer Union-Find Realisierung der Menge $S_5 = \{1, 5, 4, 8\}$.

Theorem 18 *Eine Folge von m* FIND*-Operationen und $k < n$* UNION*-Operationen läßt sich in Zeit $O(m + n \log n)$ ausführen.*

Beweis. Die Laufzeit $O(m)$ für die FIND-Operationen ist klar. Die abschließende Frage lautet, wie oft ein Namenszeiger eines Elementes x insgesamt umgehängt wird. Das Element gehörte dann immer zur kleineren Menge. Nach dem ersten Umhängen gehört x zu einer Menge mit größer gleich 2 Elementen. Nach dem zweiten Umhängen gehört x zu einer Menge mit größer gleich 4 Elementen. Nach dem i-ten Umhängen gehört x also zu einer Menge mit größer gleich 2^i Elementen. Eine solche Menge S_j kann aber nur maximal n Elemente enthalten. Also kann x nur maximal $\log n$ mal ungehängt worden sein. Das gilt für jedes $x \in U$ und somit ergibt sich die Laufzeit. □

Gelegentlich wird in einigen Beschreibungen zur Union-Find Datenstruktur auch eine Operation MAKE-SET(x) angegeben, die eine Menge $S_x = \{x\}$ initialisiert. Wir haben hier diese Initialisierung vorweggenommen. An den Laufzeiten ändert sich nichts.

6.10 Datenstrukturen für Graphen

Im nächsten Kapitel wollen wir einige klassische Algorithmen betrachten, die auf einem Netzwerk von Knoten und Kanten angewendet werden und viele praktische Anwendungen haben.

Ein sogenannter *Graph* ist eine natürliche Erweiterung der Bäume. Formal beschreiben wir einen (ungerichteten) Graphen durch eine Menge $V = \{v_1, v_2, \ldots, v_n\}$ von Knoten und eine Menge $E \subseteq V \times V$ von Kanten, kurz $G = (V, E)$. Jeder formale Graph G kann grafisch (oder geometrisch) in der Ebene analog zu den Bäumen realisiert werden. Für die Knoten $v_i \in V$ legen wir disjunkte Punkte in der Ebene fest und für je zwei Knoten v_1, v_2 in V mit $(v_1, v_2) \in E$ *zeichnen* wir einen einfachen (ohne Selbstschnitte) Weg zwischen den zugehörigen Orten. Falls wir eine bestimmte geometrische Realisation meinen, sprechen wir auch von einem *geometrischen* Graphen.

Nicht jeder Graph läßt sich so geometrisch realisieren, dass die Kantenwege paarweise schnittfrei bleiben. Graphen, die eine schnittfreie geometrische Realisiation erlauben, heißen auch *planare* Graphen. Die Realisation selbst, also der geometrische Graph heißt dann auch *kreuzungsfrei*. Bei einem kreuzungsfreien geometrischen Graphen kann man in der Ebene eindeutige Flächen charakterisieren, die von Kanten eingeschlossen sind.

Wir erlauben auch Kanten der Form (v, v), der *Grad* eines Knoten v ist die Anzahl der Kanten, die diesen Knoten als Endpunkt haben. Für *ungerichtete* Graphen ist die Kante

(v_i, v_j) gleichbedeutend mit (v_j, v_i). Bei *gerichteten* Graphen wird durch (v_i, v_j) eine Kante beschrieben, die von v_i zu v_j verläuft. Im allgemeinen kann E auch eine Multimenge sein, dass heißt, es können mehrere Kanten (v_i, v_j) in E enthalten sein. In diesem Fall wird der Graph auch als *Multigraph* bezeichnet.

Für $G = (\{v_1, v_2, v_3, v_4, v_5\}, \{(v_1, v_2), (v_1, v_3), (v_1, v_1), (v_2, v_3), (v_4, v_5), (v_3, v_4), (v_1, v_5)\})$ zeigt Abbildung 6.30 eine kreuzungsfreie geometrische Realisation von G. Wir interessieren uns hier

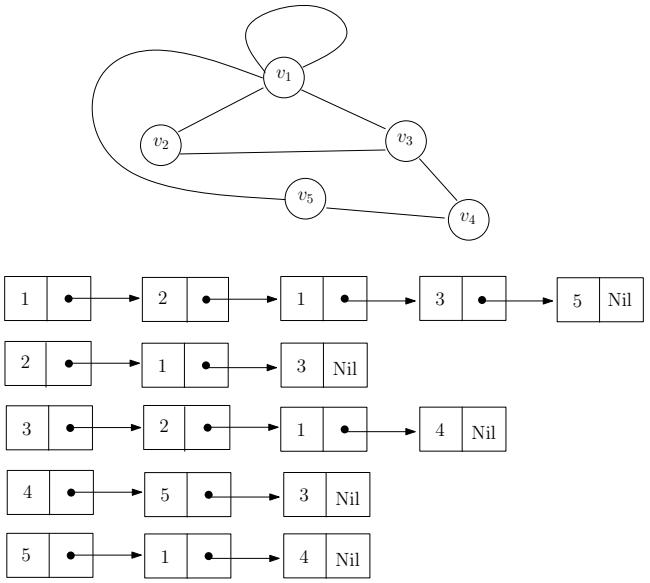


Abbildung 6.30: Eine geometrische Realisation des Graphen $G = (\{v_1, v_2, v_3, v_4, v_5\}, \{(v_1, v_2), (v_1, v_3), (v_1, v_1), (v_2, v_3), (v_4, v_5), (v_3, v_4), (v_1, v_5)\})$. Die Adjazenzliste speichert für jeden Knoten die adjazenten Knoten in einer Liste ab. Es gibt Crossreferenzen zu den jeweiligen Listen.

zunächst darum, wie Graphen im allgemeinen so abgespeichert werden, dass wir effizient auf die notwendigen Informationen zugreifen können. Ein *Weg* in einem Graphen ist eine Folge von Knoten, die über Kanten miteinander verbunden sind. Bei gerichteten Graphen muss die Kantenfolge entsprechend *gerichtet* sein.

Beispielsweise möchten wir nun wissen, ob tatsächlich alle Knoten von einem Startknoten aus sukzessive über Kanten erreichbar sind. Wie stellen also die Frage, ob die Knoten untereinander *wegzusammenghängend* sind oder kurz ob der Graph insgesamt *zusammenhängend* ist. Zusammenhangskomponenten (maximale wegszusammenhängende Teilmengen) des Graphen werden über die Knoten definiert. Außerdem kann es sinnvoll sein, alle Knoten und Kanten (bzw. weitere Daten dazu) des Graphen sukzessive auszugeben.

6.10.1 Adjazenzlisten

Eine klassische Speichermethode ist die sogenannte *Adjazenzliste*. Falls für einen Graphen $G = (V, E)$ eine Kante (v_i, v_j) existiert, sind v_i und v_j *Nachbarknoten* und werden als *adjazent* bezeichnet.

Für jeden Knoten v gibt es in der Adjazenliste eine verkettete Liste aller Knoten, die zu v adjazent sind, also eine Kante bilden, in Abbildung 6.30 ist ein Beispiel angegeben. Genauer handelt es sich um eine verkettete Liste von den zu v adjazenten Knoten.

Der Einfachheit halber können wir Knoten v_i auch durch Indizes $i = 1, \ldots, n$ repräsentieren.

Dann können wir von einem Knoten in einer Adjazenzliste leicht zu seiner Adjazenzliste gelangen und auch für jeden Knoten global eine Information abspeichern. Die Listenköpfe lassen sich dann beispielsweise durch ein Array $A[1..n]$ erreichen und die Information für die Knoten wird in $V[1..n]$ gespeichert. Das läßt sich aber genauso durch Zeiger und Namen v_i realisieren.

Offensichtlich benötigen wir für die Adjazenzliste einen Speicherplatz von $O(|V| + |E|)$. Das ist in dem Sinne optimal, dass unser Graph auch $|V| + |E|$ Objekte enthält. Wir stellen nun klassische Traversierungsmöglichkeiten für Graphen vor. Wir wollen alle Knoten des Graphen von einem Startknoten aus berichten.

Tiefensuche (DFS): Geht von einem Startknoten aus rekursiv in die *Tiefe* und besucht rekursiv den nächsten noch nicht besuchten Nachbarknoten. Falls kein solcher Knoten mehr existiert wird zum letzten Ausgangsknoten der rekursiven Suche zurückgegangen (Backtracking). Dort wird rekursive der nächste noch nicht besuchte Knoten in die Tiefe *exploriert*.

Breitensuche (BFS): Geht von einem Startknoten aus rekursiv in die *Breite* und besucht zunächst sukzessive durch Vorwärts- und Rückwärtsbewegung alle unbesuchten Nachbarknoten mit Abstand 1 zum Startknoten. Wenn alle Nachbarknoten besucht wurden, geht die Suche rekursiv beim ersten Nachbarknoten weiter.

Wir beschreiben das Verfahren DFS zunächst als Pseudocode und geben ein Beispiel an. Hierbei markieren wir die Knoten in der DFS Reihenfolge. Wir verwenden einen Stack S.

Prozedur 22 DFS(A, s)
Graph $G = (V, E)$ repräsentiert durch (kopierte) Adjazenzsliste A. $A[v]$ ist Zeiger auf Nachbarschaftsliste von v. Vom Startknoten s aus, werden alle Knoten markiert.

```
1:  Push(s);
2:  Markiere s
3:  Top(v);
4:  while v ≠ Nil do
5:    if A[v] ≠ Nil then
6:      v' := first(A[v]);
7:      Lösche v' aus A[v]
8:      if v' nicht markiert then
9:        Push(v');
10:       Markiere v'
11:     end if
12:   else
13:     Pop(v);
14:   end if
15:   Top(v);
16: end while
```

Wenn wir beispielsweise den Graphen aus Abbildung 6.30 betrachten, und den Knoten v_5 als Startknoten verwenden, dann können wir die Markierungen leicht in einem Array $M[1..5]$ festhalten. Die Abbbildung 6.31 zeigt einen Zwischenstand bei der Durchführung des Algorithmus für Startknoten v_5. Jeder Knoten wird nur einmal markiert und somit gibt es $|V|$ Push-Operationen. Da in jedem While-Durchlauf entweder ein Element aus einer nichtleeren Liste gestrichen wird, oder ein Element vom Stack gepoppt wird, liegt die Laufzeit des Algorithmus in $O(|V| + |E|)$.

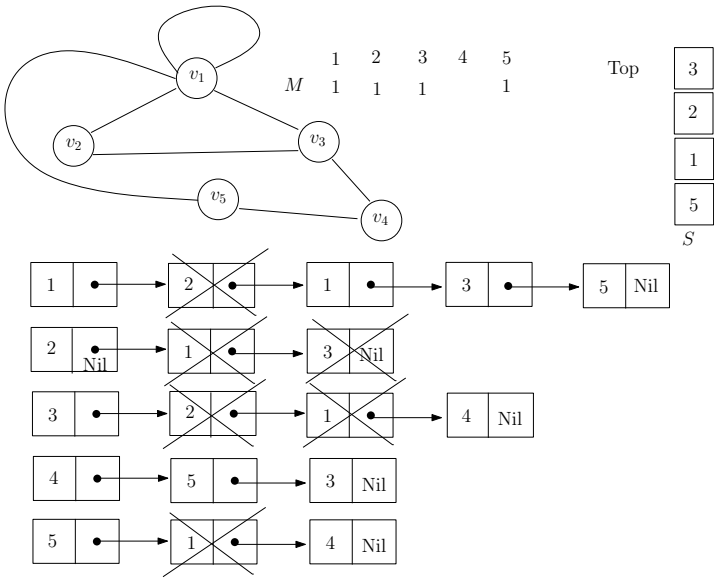


Abbildung 6.31: Ein Zwischenstand des DFS-Aufrufs für den Startknoten v_5. Im nächsten Durchlauf der While-Schleife wird das erste Element der Liste von v_3 betrachtet.

6.10.2 Nachbarschaftsmatrix

Eine weitere simple Methode, um einen Graphen effizient abzuspeichern, ist die Verwendung einer Nachbarschaftsmatrix. Wir identifizieren die n Knoten des Graphen wiederum durch die ganzzahligen Indizes und für $V = \{1, 2, \ldots, n\}$ verwenden wir eine $n \times n$ Matrix N mit Einträgen

$$N_{ij} = \begin{cases} 1 & \text{falls } (i,j) \in E \\ 0 & \text{sonst} \end{cases} \tag{6.11}$$

Die Matrix im Beispiel aus Abbildung 6.30 sieht dann wie folgt aus:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Der Vorteil der Speicherung in einer Nachbarschaftsmatrix liegt darin, dass wir in konstanter Zeit testen können, ob eine Kante zwischen zwei Knoten existiert. Allerdings benötigen wir $\Omega(|V|^2)$ viel Speicherplatz.

Literatur

Eine andere Darstellung der hier betrachteten Datenstrukturen findet sich beispielsweise in [2].

Kapitel 7

Elementare Graphalgorithmen

Dieses Kapitel entstammt in wesentlichen Teilen aus dem Vorlesungsskript von Prof. Dr. Heiko Röglin, der die Vorlesung im Wintersemester 2010/2011 an der Universiät Bonn gelesen hat und freundlicherweise den Quellcode zur Verfügung gestellt hat.

7.1 Minimale Spannbäume

Sei $G = (V, E)$ ein ungerichteter zusammenhängender Graph. Sei $w : E \to \mathbb{R}_{>0}$ eine *Gewichtung der Kanten*, d.h. eine Funktion, die jeder Kante $e \in E$ ein Gewicht $w(e)$ zuordnet. Ein Graph heißt *kreisfrei*, falls keine Folge von Kanten $(v_1, v_2)(v_2, v_3), \ldots, (v_{n-1}, v_n)$ mit $v_1 = v_n$ existiert.

Definition 19 *Eine Kantenmenge $T \subseteq E$ heißt* Spannbaum *von G, wenn der Graph $G = (V, T)$ zusammenhängend und kreisfrei ist. Wir erweitern die Funktion w auf Teilmengen von E, indem wir definieren*

$$w(T) = \sum_{e \in T} w(e),$$

und wir nennen $w(T)$ das Gewicht von T. *Ein Spannbaum $T \subseteq E$ heißt* minimaler Spannbaum *von G (kurz auch MST für* Minimum Spanning Tree*) wenn es keinen Spannbaum von G gibt, der ein kleineres Gewicht als T hat.*

Die Berechnung eines minimalen Spannbaumes ist ein Problem, das in vielen Anwendungskontexten auftritt, da der MST den Graphen bezüglich des Zusammenhanges gut approximiert und eine geringere Komplexität hat. Für unser Anwendungsbeispiel der optimalen Platzierung von Agenten aus Kapitel 1 und Abschnitt 1.1.1 läßt sich beispielsweise zeigen, dass die beste Lösung für den MST des Graphen stets eine 2-Approximation der optimalen Lösung für den gesamten Graphen ergibt.

Wir entwerfen in diesem Abschnitt einen Algorithmus, der effizient für einen gegebenen Graphen einen minimalen Spannbaum berechnet.

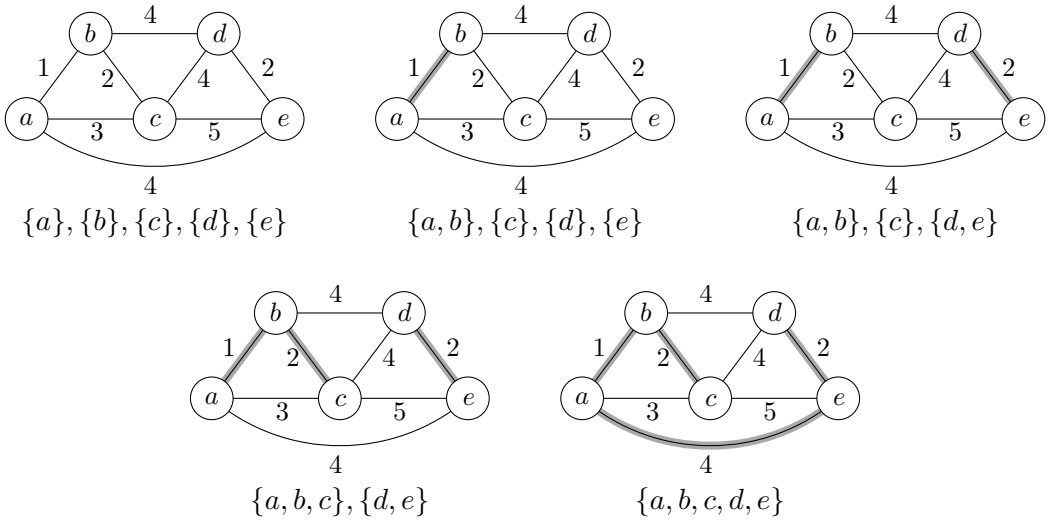
7.1.1 Algorithmus von Kruskal

Mit Hilfe einer Union-Find-Datenstruktur können wir den *Algorithmus von Kruskal* zur Berechnung eines minimalen Spannbaumes angeben. Er ist nach Joseph Kruskal benannt, der diesen Algorithmus 1956 veröffentlichte.

```
KRUSKAL(G, w)
1   Teste mittels DFS, ob G zusammenhängend ist. Falls nicht Abbruch.
2   for all v ∈ V { MAKE-SET(v) }
3   T = ∅
4   Sortiere die Kanten in E gemäß ihrem Gewicht. Danach gelte E = {e_1, ..., e_m} mit
    w(e_1) ≤ w(e_2) ≤ ... ≤ w(e_m). Außerdem sei e_i = (u_i, v_i).
5   for i = 1 to m {
6       if FIND(u_i) ≠ FIND(v_i) {
7           T = T ∪ {e_i}
8           UNION(u_i, v_i)
9       }
10  }
11  return T
```

Die Mengen, die in der Union-Find-Datenstruktur im Algorithmus verwendet werden, entsprechen den Zusammenhangskomponenten des Graphen $G = (V, T)$. Wir gehen die Kanten in der Reihenfolge ihres Gewichtes durch und fügen eine Kante zur Menge T hinzu, wenn sie nicht innerhalb einer bereits zusammenhängenden Menge verläuft.

Bevor wir die Korrektheit beweisen und die Laufzeit analysieren, demonstrieren wir das Verhalten an einem kleinen Beispiel.



Korrektheit

Die Korrektheit des Algorithmus folgt aus dem folgenden Lemma.

Lemma 20 *Sei $V_1, \ldots, V_k$ eine disjunkte Zerlegung von V. Sei $E_i \subseteq E$ die Menge der Kanten $(u, v) \in E$ mit $u, v \in V_i$ und sei $T_i \subseteq E_i$ ein Spannbaum des Graphen (V_i, E_i).*

Ferner sei $E' = E \setminus (E_1 \cup \ldots \cup E_k)$, also

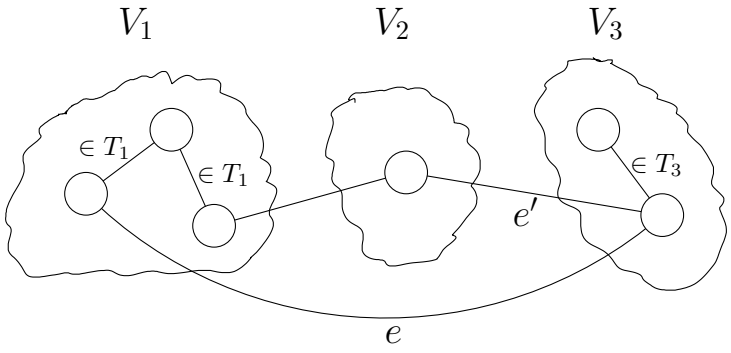
$$E' = \{(u, v) \in E \mid u \in V_i, v \in V_j, i \neq j\},$$

und sei e eine Kante mit dem kleinsten Gewicht aus E'.

Unter allen Spannbäumen von $G = (V, E)$, die die Kanten aus $T_1 \cup \ldots \cup T_k$ enthalten, gibt es einen mit kleinstem Gewicht, der zusätzlich auch die Kante e enthält.

Beweis. Sei T ein Spannbaum von $G = (V, E)$, der $T_1 \cup \ldots \cup T_k$ aber nicht e enthält. Wir konstruieren einen Spannbaum T', der $T_1 \cup \ldots \cup T_k$ und e enthält und für den gilt $w(T') \leq w(T)$. Daraus folgt direkt das Lemma, denn angenommen wir haben einen Spannbaum T mit kleinstmöglichem Gewicht, der e nicht enthält, so können wir einen anderen Spannbaum T' konstruieren, der e enthält und ebenfalls das kleinstmögliche Gewicht hat.

Sei $T'' = T \cup \{e\}$. Dann enthält der Graph (V, T'') einen Kreis, auf dem e liegt. Da e zwei verschiedene Komponenten V_i und V_j mit $i \neq j$ verbindet, muss auf diesem Kreis noch eine weitere Kante $e' \in E'$ liegen, die zwei verschiedene Komponenten verbindet. Diese Situation ist in dem folgenden Bild dargestellt:



Aufgrund der Voraussetzung, dass e die billigste Kante aus E' ist, muss $w(e) \leq w(e')$ gelten. Wir betrachten nun die Menge $T' = T'' \setminus \{e'\} = (T \cup \{e\}) \setminus \{e'\}$.

Zunächst können wir festhalten, dass (V, T') zusammenhängend ist. Der Graph (V, T) ist nach Voraussetzung zusammenhängend. Gibt es in (V, T) zwischen zwei Knoten $u, v \in V$ einen Weg, der Kante e' nicht benutzt, so gibt es diesen Weg auch in (V, T'). Gibt es in (V, T) einen Weg P zwischen u und v, der Kante $e' = (w_1, w_2)$ benutzt, so erhält man auch in (V, T') einen Weg zwischen u und v, indem man P nimmt und die Kante e' durch den Weg zwischen w_1 und w_2 über die Kante e ersetzt.

Ebenso können wir argumentieren, dass (V, T') kreisfrei ist: Angenommen es gibt in (V, T') einen Kreis C. Wenn C die Kante e nicht enthält, so gibt es den Kreis C auch in (V, T), im Widerspruch zur Definition von T. Enthält der Kreis die Kante $e = (u_1, u_2)$, so erhalten wir einen Kreis in (V, T), indem wir in C die Kante e durch den Weg zwischen u_1 und u_2 über e' ersetzen.

Somit können wir festhalten, dass (V, T') ein Spannbaum ist. Außerdem gilt

$$w(T') = w(T) + w(e) - w(e') \leq w(T).$$

Damit ist das Lemma bewiesen. □

Wir können nun das folgende Theorem beweisen.

Theorem 21 *Der Algorithmus von Kruskal berechnet einen minimalen Spannbaum.*

Beweis. Wir benutzen die folgende Invariante, die aus drei Teilen besteht:

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen am Ende des Schleifenrumpfes der for-Schleife stets den Zusammenhangskomponenten des Graphen (V, T), wobei T die in Zeilen 3 und 7 betrachtete Kantenmenge ist.

2. Die Zusammenhangskomponenten von (V, T) stimmen am Ende des Schleifenrumpfes der for-Schleife stets mit den Zusammenhangskomponenten von $(V, \{e_1, \ldots, e_i\})$ überein.

3. Die Kantenmenge $T \subseteq E$ kann am Ende des Schleifenrumpfes der for-Schleife stets zu einer Kantenmenge $T' \supseteq T$ erweitert werden, sodass T' ein minimaler Spannbaum von G ist. Insbesondere ist (V, T) azyklisch.

Ist diese Invariante korrekt, so ist auch der Algorithmus korrekt. Das sehen wir wie folgt: Am Ende des letzten Schleifendurchlaufes garantiert uns der zweite Teil der Invariante, dass die Zusammenhangskomponenten von (V, T) mit denen von $G = (V, E)$ übereinstimmen. Da G laut Voraussetzung zusammenhängend ist, ist also auch (V, T) zusammenhängend. Ist G keine korrekte Eingabe, da er nicht zusammenhängend ist, wird dies im ersten Schritt des Algorithmus erkannt.

Der dritte Teil der Invariante garantiert uns, dass T zu einer Kantenmenge $T' \supseteq T$ erweitert werden kann, sodass T' ein minimaler Spannbaum von G ist. Da (V, T) bereits zusammenhängend ist, würde die Hinzunahme jeder Kante einen Kreis schließen. Somit muss $T' = T$ gelten, woraus direkt folgt, dass $T = T'$ ein minimaler Spannbaum von G ist.

Nun müssen wir nur noch die Korrektheit der Invariante zeigen. Dazu überlegen wir uns zunächst, dass die Invariante nach dem ersten Schleifendurchlauf gilt. In diesem ersten Durchlauf fügen wir die billigste Kante e_1 in die bisher leere Menge T ein, da ihre beiden Endknoten u_1 und v_1 zu Beginn in verschiedenen einelementigen Mengen liegen. Nach dem Einfügen von e_1 vereinigen wir diese beiden einelementigen Mengen, so dass nach dem ersten Schleifendurchlauf die Knoten u_1 und v_1 in derselben und alle anderen Knoten in getrennten einelementigen Mengen liegen. Dies sind genau die Zusammenhangskomponenten von $(V, T) = (V, \{e_1\})$. Damit sind der erste und zweite Teil der Invariante gezeigt. Für den dritten Teil wenden wir Lemma 20 an. Wählen wir in diesem Lemma $V_1 = \{v_1\}, \ldots, V_m = \{v_m\}$, wobei wir $V = \{v_1, \ldots, v_m\}$ annehmen, so impliziert es direkt, dass wir die Menge $T = \{e_1\}$ zu einem minimalen Spannbaum von G erweitern können.

Schauen wir uns nun einen Schleifendurchlauf an und nehmen an, dass die Invariante am Ende des vorherigen Schleifendurchlaufs erfüllt war.

1. Falls wir Kante e_i nicht in T einfügen, so ändern sich weder die Mengen in der Union-Find-Datenstruktur noch die Zusammenhangskomponenten von (V, T). Teil eins der Invariante bleibt dann also erhalten. Fügen wir e_i in T ein, so fallen die Zusammenhangskomponenten von u_i und v_i in (V, T) zu einer gemeinsamen Komponente zusammen. Diese Veränderung bilden wir in der Union-Find-Datenstruktur korrekt ab. Also bleibt auch dann der erste Teil erhalten.

2. Bezeichne $V_1, \ldots, V_k$ die Zusammenhangskomponenten von (V, T) am Ende des vorherigen Schleifendurchlaufs und bezeichne $T_1, \ldots, T_k$ die Kanten innerhalb dieser Komponenten. Falls wir e_i nicht zu T hinzufügen, so verläuft es innerhalb einer der gerade definierten Komponenten $V_1, \ldots, V_k$. Das heißt die Komponenten von $(V, \{e_1, \ldots, e_{i-1}\})$ und $(V, \{e_1, \ldots, e_i\})$ sind identisch und die Invariante bleibt erhalten. Fügen wir e_i zu T hinzu, so verschmelzen in $(V, \{e_1, \ldots, e_i\})$ die Komponenten, die u_i und v_i enthalten. Diese Verschmelzung passiert aber genauso in (V, T), da wir e_i zu T hinzufügen. Also stimmen die Zusammenhangskomponenten von (V, T) wieder mit den Zusammenhangskomponenten von $(V, \{e_1, \ldots, e_i\})$ überein.

3. Aufgrund der Invariante können wir $T = T_1 \cup \ldots \cup T_k$ zu einem minimalen Spannbaum von $G = (V, E)$ erweitern. Fügen wir in dieser Iteration e_i zu T hinzu, so garantiert Lemma 20, dass wir auch $T \cup \{e_i\}$ zu einem minimalen Spannbaum erweitern können: Da alle Kanten $e_1, \ldots, e_{i-1}$ aufgrund des zweiten Teils der Invariante innerhalb der Komponenten $V_1, \ldots, V_k$ verlaufen, ist e_i die billigste Kante, die zwei verschiedene Komponenten miteinander verbindet. Somit sind die Voraussetzungen von Lemma 20 erfüllt.

□

Laufzeit

Nachdem wir die Korrektheit des Algorithmus von Kruskal bewiesen haben, analysieren wir nun noch die Laufzeit. Wir haben bereits im letzten Kapitel gesehen, wie effizient wir die Operationen der Union-Find-Datenstruktur implementieren können, siehe Theorem 18.

Nun können wir die Laufzeit des Algorithmus von Kruskal abschließend bestimmen.

Theorem 22 *Für zusammenhängende ungerichtete Graphen mit m Kanten benötigt der Algorithmus von Kruskal Laufzeit $O(m \log m) = O(|E| \cdot \log |V|)$.*

Beweis. Die Tiefensuche im ersten Schritt benötigt Zeit $O(|V| + m)$. Da der Graph laut Voraussetzung zusammenhängend ist, gilt $m \geq |V|-1$, also $O(|V|+m) = O(m)$. Das Sortieren der Kanten benötigt Zeit $O(m \log m)$. Die for-Schleife wird m mal durchlaufen und abgesehen von UNION und FIND werden in jedem Durchlauf nur konstant viele Operationen durchgeführt, so dass wir eine Laufzeit von $O(m)$ für alle Operationen in der Schleife abgesehen von UNION und FIND ansetzen können.

Wir führen in der Union-Find-Datenstruktur $2m$ FIND-Operationen und $m-1$ UNION-Operationen durch. Mit Theorem 18 ergibt sich also eine Laufzeit von $O(m \log m+2m) = O(m \log m)$. Wegen $|E| \leq |V|^2$ folgt $O(m \log m) = O(|E| \cdot \log |V|^2) = O(|E| \cdot \log |V|)$. Damit ist das Theorem bewiesen. □

Literatur

Der Algorithmus von Kruskal kann in Kapitel 23 von [1] nachgelesen werden. Kapitel 21 enthält zudem eine Beschreibung von Union-Find-Datenstrukturen.

7.2 Kürzeste Wege

Sei $G = (V, E)$ ein gerichteter Graph und sei $w : E \to \mathbb{R}$ eine Gewichtung der Kanten. Da wir uns in diesem Abschnitt mit kürzesten Wegen beschäftigen, werden wir $w(e)$ auch als die *Länge der Kante e* bezeichnen.

Definition 23 *Für zwei gegebene Knoten $s, t \in V$ ist $P = (v_0, v_1, \ldots, v_\ell)$ ein* Weg von s nach t*, wenn $v_0 = s$, $v_\ell = t$ und wenn $(v_i, v_{i+1}) \in E$ für alle $i \in \{0, 1, \ldots, \ell-1\}$ gilt. Wir definieren die* Länge von P *als $w(P) = \sum_{i=0}^{\ell-1} w(v_i, v_{i+1})$. Wir sagen, dass P ein* kürzester Weg von s nach t *ist, falls es keinen anderen Weg P' von s nach t mit $w(P') < w(P)$ gibt. Wir nennen die Länge $w(P)$ des kürzesten Weges P die* Entfernung von s nach t *und bezeichnen diese mit $\delta(s, t)$. Existiert kein Weg von s nach t, so gelte $\delta(s, t) = \infty$.*

Zur besseren Intuition können wir uns im Folgenden vorstellen, dass die Knoten des Graphen Kreuzungen entsprechen und dass für jede Straße von Kreuzung $u \in V$ zu Kreuzung $v \in V$ eine Kante $e = (u, v)$ in E enthalten ist, wobei $w(e) = w((u, v))$ die Länge der Straße angibt. Es gibt jedoch auch zahlreiche andere Anwendungskontexte, in denen kürzeste Wege gefunden werden müssen. Insbesondere auch solche, in denen negative Kantengewichte auftreten können.

Wir sind an der Lösung der folgenden zwei Probleme interessiert:

1. Im *Single-Source Shortest Path Problem (SSSP)* ist zusätzlich zu G und w ein Knoten $s \in V$ gegeben und wir möchten für jeden Knoten $v \in V$ einen kürzesten Weg von s nach v und die Entfernung von s nach v berechnen.

2. Im *All-Pairs Shortest Path Problem (APSP)* sind nur G und w gegeben und wir möchten für jedes Paar $u, v \in V$ einen kürzesten Weg von u nach v und die Entfernung von u nach v berechnen.

Für den Spezialfall, dass $w(e) = 1$ für alle Kanten $e \in E$ gilt, haben wir mit Breitensuche bereits einen Algorithmus kennengelernt, der das SSSP löst. Breitensuche lässt sich allerdings nicht direkt auf den Fall verallgemeinern, dass wir beliebige Kantengewichte haben.

Bevor wir Algorithmen für das SSSP und das APSP beschreiben, schauen wir uns zunächst einige strukturelle Eigenschaften von kürzesten Wegen an.

Optimale Substruktur

Für die Algorithmen, die wir vorstellen werden, ist es wichtig, dass ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$ aus kürzesten Wegen zwischen anderen Knoten zusammengesetzt ist.

Lemma 24 *Sei $P = (v_0, \ldots, v_\ell)$ ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$. Für jedes Paar i, j mit $0 \leq i < j \leq \ell$ ist $P_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ ein kürzester Weg von v_i nach v_j.*

Beweis. Wir zerlegen den Weg $P = P_{0\ell}$ in die drei Teilwege P_{0i}, P_{ij} und $P_{j\ell}$. Dies schreiben wir als

$$P = v_0 \overset{P_{0i}}{\rightsquigarrow} v_i \overset{P_{ij}}{\rightsquigarrow} v_j \overset{P_{j\ell}}{\rightsquigarrow} v_\ell.$$

Insbesondere gilt $w(P) = w(P_{0i}) + w(P_{ij}) + w(P_{j\ell})$.

Nehmen wir nun an, dass P_{ij} kein kürzester Weg von v_i nach v_j ist. Dann gibt es einen Weg P'_{ij} von i nach j mit $w(P'_{ij}) < w(P_{ij})$. Dann erhalten wir ebenfalls einen anderen Weg P' von v_0 nach v_ℓ:

$$P' = v_0 \overset{P_{0i}}{\rightsquigarrow} v_i \overset{P'_{ij}}{\rightsquigarrow} v_j \overset{P_{j\ell}}{\rightsquigarrow} v_\ell.$$

Für diesen Weg gilt $w(P') = w(P) - w(P_{ij}) + w(P'_{ij}) < w(P)$ im Widerspruch zur Voraussetzung, dass P ein kürzester Weg von v_0 nach v_ℓ ist. Also kann es keinen kürzeren Weg als P_{ij} von v_i nach v_j geben. □

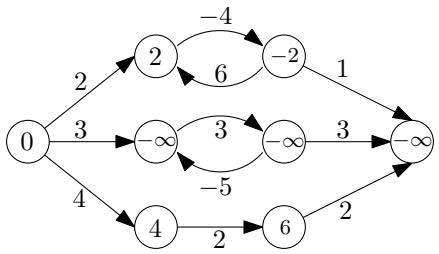
Daraus erhalten wir direkt das folgende Korollar.

Corollary 25 *Sei $P = (v_0, \ldots, v_\ell)$ ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$. Dann gilt*

$$\delta(v_0, v_\ell) = \delta(v_0, v_{\ell-1}) + w(v_{\ell-1}, v_\ell).$$

Negative Kantengewichte

Wir haben in der obigen Definition auch erlaubt, dass Kantengewichte negativ sein können. Dies kann in einigen Anwendungskontexten tatsächlich auftreten, es führt jedoch zu zusätzlichen Schwierigkeiten, insbesondere dann, wenn der Graph einen Kreis mit negativem Gesamtgewicht enthält. Dann ist nämlich für einige Paare von Knoten kein kürzester Weg mehr definiert, da man beliebig oft den Kreis mit negativem Gesamtgewicht entlang laufen könnte. Die folgende Abbildung zeigt einen Graphen mit einem negativen Kreis. Die Zahlen in den Knoten entsprechen dabei den Entfernungen der Knoten vom linken Knoten.



Der *Algorithmus von Dijkstra*, den wir für das SSSP kennenlernen werden, funktioniert nur für Graphen, die keine negativen Kantengewichte haben. Der *Floyd-Warshall-Algorithmus* für APSP funktioniert auch für negative Kantengewichte solange es keine negativen Kreise gibt. Gibt es einen negativen Kreis, so kann der Floyd-Warshall-Algorithmus zumindest benutzt werden, die Existenz eines solchen zu verifizieren.

7.2.1 Single-Source Shortest Path Problem

Es sei ein Graph $G = (V, E)$, eine Gewichtung $w : E \to \mathbb{R}_{\geq 0}$ und ein Startknoten $s \in V$ gegeben. In diesem Abschnitt gehen wir davon aus, dass alle Kantengewichte nicht-negativ sind.

Darstellung kürzester Wege

Zunächst halten wir fest, dass ein kürzester Weg in einem Graphen ohne negativen Kreis keinen Knoten mehrfach besuchen muss. Wäre $P = (v_0, \ldots, v_\ell)$ nämlich ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$ und gäbe es v_i und v_j mit $i < j$ und $v_i = v_j$, so wäre der Teilweg $P_{ij} = (v_i, \ldots, v_j)$ ein Kreis, der laut Voraussetzung keine negativen Gesamtkosten hat. Das heißt, er hat entweder Kosten 0 oder echt positive Kosten. In jedem Falle können wir einen neuen Weg P' von v_0 nach v_ℓ konstruieren, indem wir den Kreis überspringen: $P' = (v_0, \ldots, v_i, v_{j+1}, \ldots, v_\ell)$. Für diesen neuen Weg P' gilt $w(P') \leq w(P)$. Somit können wir davon ausgehen, dass die kürzesten Wege zwischen zwei Knoten, die wir berechnen werden, aus höchsten $n-1$ Kanten bestehen.

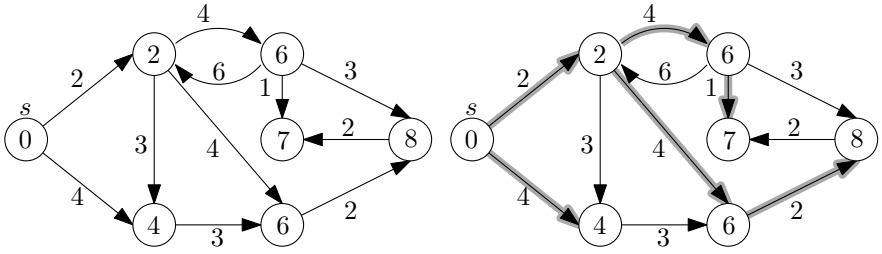
Beim SSSP berechnen wir n kürzeste Wege, nämlich einen vom Startknoten s zu jedem anderen Knoten. Beim APSP berechnen wir sogar $\binom{n}{2}$ kürzeste Wege. Da wir als obere Abschätzung für die Anzahl Kanten auf einem kürzesten Weg nur $n-1$ haben, müssten wir damit rechnen, dass wir Platz $\Theta(n^2)$ bzw. $\Theta(n^3)$ brauchen, um alleine die Ausgabe aufzuschreiben. Wir wollen uns nun überlegen, wie man die Ausgabe effizienter kodieren kann. Dabei machen wir uns die Eigenschaft der optimalen Substruktur aus Lemma 24 zu Nutze.

Wir betrachten in diesem Abschnitt nur das SSSP. Statt alle n Wege separat zu speichern, wird der Algorithmus von Dijkstra einen Kürzeste-Wege-Baum mit Wurzel s gemäß der folgenden Definition berechnen.

Definition 26 *Wir nennen (V', E') mit $V' \subseteq V$ und $E' \subseteq E$ einen* Kürzeste-Wege-Baum mit Wurzel s*, wenn die folgenden Eigenschaften erfüllt sind.*

1. *V' ist die Menge der Knoten, die von s aus in G erreichbar sind.*
2. *G' ist ein gewurzelter Baum mit Wurzel s. (Das bedeutet, dass G' azyklisch ist, selbst wenn wir die Richtung der Kanten ignorieren, und dass alle Kanten von s weg zeigen.)*
3. *Für alle $v \in V'$ ist der eindeutige Weg von s nach v in G' ein kürzester Weg von s nach v in G.*

Kürzeste-Wege-Bäume benötigen Platz $O(n)$, da wir für jeden Knoten nur seinen Vorgänger speichern müssen. Sie sind also eine platzsparende Möglichkeit, um alle kürzesten Wege von s zu den anderen Knoten des Graphen zu kodieren. Die folgende Abbildung zeigt rechts einen Kürzeste-Wege-Baum für den Graphen auf der linken Seite.



Relaxation von Kanten

Der Algorithmus von Dijkstra verwaltet für jeden Knoten $v \in V$ zwei Attribute: $\pi(v) \in V \cup \{\text{nil}\}$ und $d(v) \in \mathbb{R}$. Der Knoten $\pi(v)$ wird am Ende des Algorithmus der Vorgänger von v in einem Kürzeste-Wege-Baum mit Wurzel s sein (sofern $\pi(v) \neq \text{nil}$) und die Zahl $d(v) \in \mathbb{R}$ wird am Ende die Entfernung von s nach v angeben. Wir initialisieren $\pi(v)$ und $d(v)$ wie folgt:

```
    INITIALIZE-DIJKSTRA(G, s)
1   for each v ∈ V {
2       d(v) = ∞
3       π(v) = nil
4   }
5   d(s) = 0
```

Im Algorithmus wird $d(v) \geq \delta(s, v)$ zu jedem Zeitpunkt und für jeden Knoten $v \in V$ gelten. Das heißt, $d(v)$ ist stets eine obere Schranke für die Länge des kürzesten Weges von s nach v. Direkt nach der Initialisierung ist dies sicherlich der Fall. Die Idee ist nun, die oberen Schranken Schritt für Schritt zu verbessern, bis irgendwann $d(v) = \delta(s, v)$ für alle Knoten v gilt. Dazu benutzen wir die folgende Beobachtung.

Lemma 27 *Für $x, y, z \in V$ gilt stets*

$$\delta(x, y) \leq \delta(x, z) + w(z, y).$$

Beweis. Wir konstruieren einen Weg von x nach y, indem wir von x auf einem kürzesten Weg nach z laufen und dann der direkten Kanten von z zu y folgen. Dieser Pfad hat Länge $\delta(x, z) + w(z, y)$. Da er offensichtlich nicht kürzer sein kann als der kürzeste Weg von x nach y, muss $\delta(x, z) + w(z, y) \geq \delta(x, y)$ gelten. □

Aus dieser Beobachtung folgt direkt eine Möglichkeit, neue obere Schranken zu erhalten: Seien $u, v \in V$ und gelte $d(u) \geq \delta(s, u)$. Dann gilt

$$\delta(s, v) \leq d(u) + w(u, v).$$

Das nutzen wir, um die oberen Schranken mittels der folgenden Methode zu verbessern.

```
    RELAX(u, v)
```

```
1   if (d(v) > d(u) + w(u,v)) {
2       d(v) = d(u) + w(u,v)
3       π(v) = u
4   }
```

Aus den bisherigen Überlegungen folgt direkt das folgende Lemma.

Lemma 28 *Nach einem Aufruf von* Initialize-Dijkstra *gefolgt von einer beliebigen Sequenz von* Relax*-Aufrufen gilt $d(v) \geq \delta(s,v)$ für alle Knoten v. Das heißt, der Wert $d(v)$ ist zu jedem Zeitpunkt eine obere Schranke für die Entfernung $\delta(s,v)$ von s nach v.*

Algorithmus von Dijkstra

Der Algorithmus von Dijkstra wird eine Folge von Relax-Aufrufen durchführen und sicherstellen, dass am Ende tatsächlich $d(v) = \delta(s,v)$ für alle Knoten v gilt. In der folgenden Implementierung wird $S \subseteq V$ eine Menge von Knoten bezeichnen, sodass für jeden Knoten $v \in S$ bereits $d(v) = \delta(s,v)$ gilt. Es sei außerdem $Q = V \setminus S$. Wir gehen davon aus, dass der Graph G in Adjazenzlistendarstellung gegeben ist. Die Methode Extract-Min(Q) entfernt einen Knoten $u \in Q$ aus Q, der unter all diesen Knoten den kleinsten Wert $d(u)$ hat, und gibt diesen Knoten zurück.

```
    Dijkstra(G, w, s)
1   Initialize-Dijkstra(G, s)
2   S = ∅; Q = V;
3   while Q ≠ ∅ {
4       u = Extract-Min(Q)
5       S = S ∪ {u}
6       for each v ∈ Adj[u] {
7           Relax(u, v)
8       }
9   }
```

Ein Beispiel für die Ausführung des Algorithmus von Dijkstra findet sich in Abbildung 7.1.

Theorem 29 *Der Algorithmus von Dijkstra terminiert auf gerichteten Graphen $G = (V,E)$ mit nicht-negativen Kantengewichten $w : E \to \mathbb{R}_{\geq 0}$ und Startknoten $s \in V$ in einem Zustand, in dem $d(v) = \delta(s,v)$ für alle $v \in V$ gilt.*

Beweis. Wir beweisen das Theorem mit Hilfe der folgenden Invariante, bei der wir die Konvention $w((u,v)) = \infty$ für $(u,v) \notin E$ anwenden.

Am Ende jeder Ausführung der **while**-Schleife in Zeilen 4–8 gilt:

1. $\forall v \in S : d(v) = \delta(s,v)$,
2. $\forall v \in Q = V \setminus S : d(v) = \min\{\delta(s,x) + w(x,v) \mid x \in S\}$.

Gilt diese Invariante, so folgt aus ihr direkt das Theorem: In jedem Schritt der **while**-Schleife fügen wir einen weiteren Knoten aus Q zu der Menge S hinzu. Da zu jedem Zeitpunkt $S \cup Q = V$ gilt, muss nach n Durchläufen der **while**-Schleife $S = V$ gelten. Dann ist $Q = \emptyset$ und

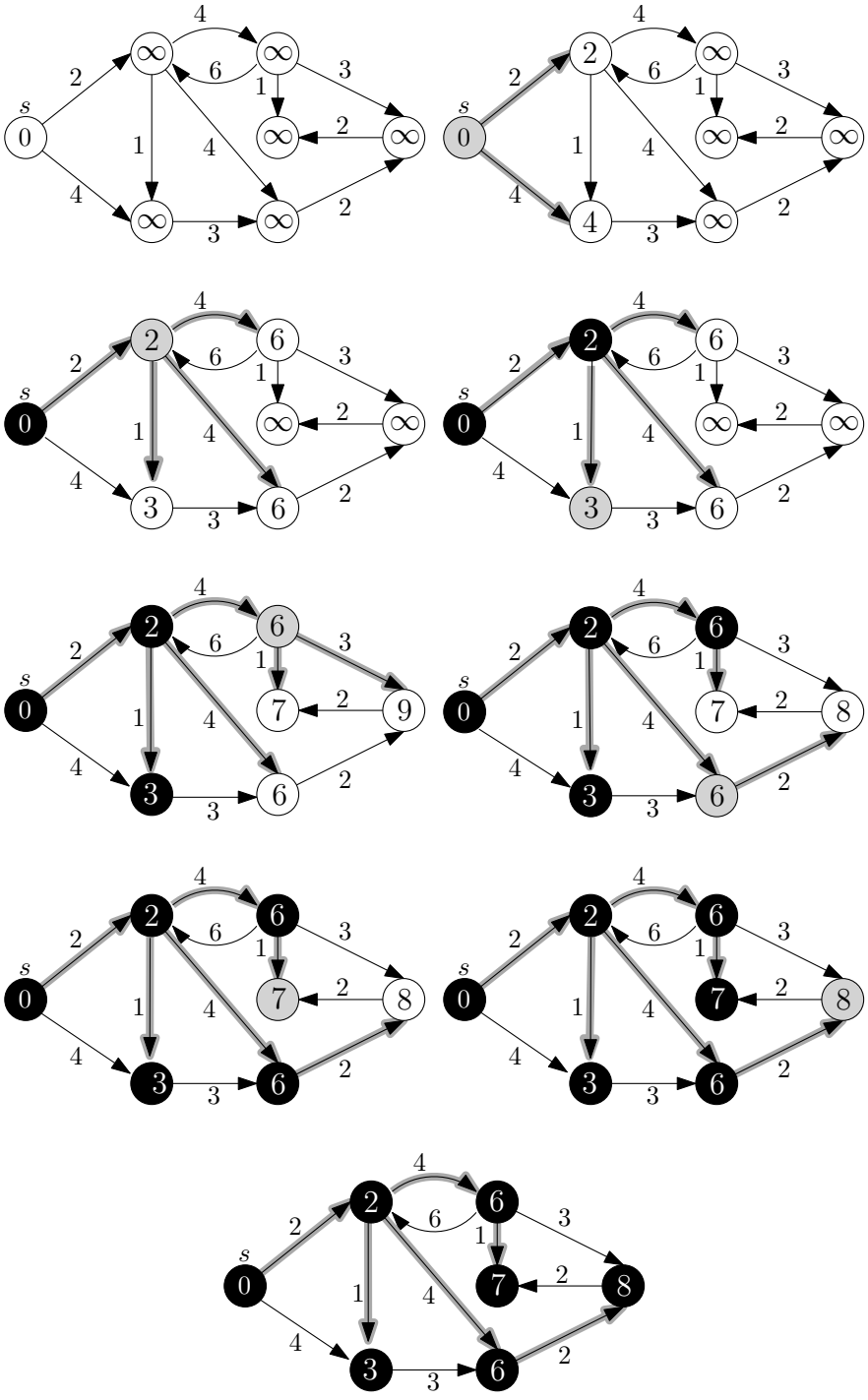


Abbildung 7.1: Beispiel für den Ablauf des Dijkstra-Algorithmus. Schwarze Knoten gehören zu S und der graue Knoten ist jeweils der Knoten u, der aus Q extrahiert wird. Eine graue Kanten (u, v) bedeutet, dass $\pi(v) = u$ gilt.

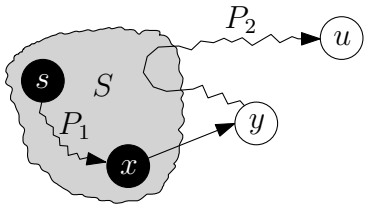
der Algorithmus terminiert. Die Invariante besagt dann, dass $d(v) = \delta(s, v)$ für alle Knoten $v \in S = V$ gilt.

Am Ende des ersten Schleifendurchlaufs gilt $S = \{s\}$ und $d(s) = \delta(s, s) = 0$. Somit ist der erste Teil der Invariante erfüllt. Den zweiten Teil der Invariante können wir in diesem Falle umschreiben zu: $\forall v \in V \setminus \{s\} : d(v) = \delta(s, s) + w(s, v) = w(s, v)$. Die Gültigkeit dieser Aussage folgt daraus, dass wir für jeden Knoten $v \in \text{Adj}[s]$ die Methode $\textsc{Relax}(s, v)$ aufgerufen haben, die laut ihrer Definition $d(v) = \min\{d(s) + w(s, v), \infty\} = w(s, v)$ gesetzt hat.

Wir müssen uns nun nur noch überlegen, dass die Invariante erhalten bleibt. Gehen wir also davon aus, wir befinden uns in einem Durchlauf der **while**-Schleife und am Ende des vorherigen Durchlaufs galt die Invariante. Während dieser Iteration fügen wir den Knoten $u \in Q$, der momentan unter allen Knoten aus Q den kleinsten Wert $d(u)$ hat, zur Menge S hinzu.

Wir zeigen zunächst den ersten Teil der Invariante, indem wir zeigen, dass für diesen Knoten $d(u) = \delta(s, u)$ gilt. Sei dazu P ein kürzester Weg von s nach u und sei y der erste Knoten auf dem Pfad P, der nicht zur Menge S gehört. Da u selbst noch nicht zu S gehört, muss es einen solchen Knoten y geben. Weiterhin muss y einen Vorgänger haben, da der erste Knoten auf dem Weg P der Knoten $s \in S$ ist. Wir nennen diesen Vorgänger x. Den Teilweg von P von s nach x nennen wir P_1 und den Teilweg von y nach u nennen wir P_2. Diese Teilwege können auch leer sein.

Die Notation ist in der folgenden Abbildung veranschaulicht.



Wegen Lemma 24 muss P_1 ein kürzester Weg von s nach x sein. Somit gilt

$$\delta(s, u) = w(P_1) + w(x, y) + w(P_2) = \delta(s, x) + w(x, y) + w(P_2) \geq d(y) + w(P_2) \geq d(y),$$

wobei wir bei der vorletzten Ungleichung den zweiten Teil der Invariante für $y \in Q$ ausgenutzt haben. Wegen Lemma 28 gilt $d(u) \geq \delta(s, u)$ und somit folgt insgesamt $d(u) \geq d(y)$. Da u ein Knoten aus Q mit kleinstem d-Wert ist, muss auch gelten $d(u) \leq d(y)$. Alles in allem erhalten wir die Ungleichungskette

$$d(u) \geq \delta(s, u) \geq d(y) \geq d(u),$$

deren einzige Lösung $d(u) = \delta(s, u)$ ist. Dies ist genau die Aussage, die wir für den ersten Teil der Invariante zeigen mussten.

Der zweite Teil der Invariante folgt nun einfach daraus, dass vorher $d(v) = \min\{\delta(s, x) + w(x, v) \mid x \in S\}$ für all $v \in Q$ galt und dass nun für jeden Knoten aus $v \in \text{Adj}[u]$ die Operation $\textsc{Relax}(u, v)$ aufgerufen wird, die dazu führt, dass gilt:

$$\begin{aligned} d(v) &= \min\{\min\{\delta(s, x) + w(x, v) \mid x \in S\}, \delta(s, u) + w(u, v)\} \\ &= \min\{\delta(s, x) + w(x, v) \mid x \in S \cup \{u\}\}. \end{aligned}$$

□

Nun fehlt nur noch der Beweis, dass die Vorgänger-Relation, die in π berechnet wird, wirklich einen Kürzesten-Wege-Baum mit Wurzel s ergibt. Dies folgt unmittelbar aus dem folgenden Lemma, welches wir in der Vorlesung jedoch nicht beweisen werden.

Lemma 30 *Es gelte nach einem Aufruf von* INITIALIZE-DIJKSTRA *gefolgt von einer Sequenz von* RELAX*-Aufrufen $d(v) = \delta(s, v)$ für jeden Knoten v. Sei*

$$V_\pi = \{v \in V \mid \pi(v) \neq nil\} \cup \{s\} \quad \textit{und} \quad E_\pi = \{(\pi(v), v) \in E \mid v \in V_\pi \setminus \{s\}\}.$$

Dann ist der Graph (V_π, E_π) ein Kürzeste-Wege-Baum mit Wurzel s.

Implementierung und Laufzeit Wir werden nun die Laufzeit des Algorithmus von Dijkstra analysieren. Dazu müssen wir uns zunächst überlegen, in welcher Datenstruktur wir die Menge Q verwalten. Diese Datenstruktur muss eine Menge verwalten können und sollte die folgenden drei Operationen möglichst effizient unterstützen:

- INSERT(Q, x, d): Füge ein neues Element x mit Schlüssel $d \in \mathbb{R}$ in die Menge Q ein.
- EXTRACT-MIN(Q): Entferne aus Q ein Element mit dem kleinsten Schlüssel und gib dieses Element zurück.
- CHANGE-KEY(Q, x, d_1, d_2): Ändere den Schlüssel des Objektes $x \in Q$ von d_1 auf d_2.

Eine solche Datenstruktur heißt *Min-Priority Queue* und sie lässt sich leicht durch z. B. eine verkettete Liste realisieren, wenn wir auf Effizienz keinen Wert legen. Sei n die Anzahl der Einträge der Priority Queue, dann würde eine Realisierung als verkettete Liste Zeit $\Theta(n)$ für EXTRACT-MIN und DECREASE-KEY benötigen. Dies wollen wir nun verbessern.

Wir haben mit AVL-Bäumen schon eine Möglichkeit in der Vorlesung kennengelernt, um Priority Queues effizienter zu realisieren. Wir legen dafür einen AVL-Baum an und fügen bei INSERT(Q, x, d) einfach das Objekt x mit Schlüssel d in den AVL-Baum ein. Wir müssen auf den Fall aufpassen, dass wir zwei Objekte x und y mit dem gleichen Schlüssel einfügen wollen. Wir nehmen für diesen Fall an, dass es eine inhärente Ordnung auf den Objekten gibt und dass bei eigentlich gleichem Schlüssel d diese inhärente Ordnung entscheidet, welcher Schlüssel als kleiner betrachtet wird. In unserem Beispiel sind die Objekte in Q die Knoten aus V. Wir können davon ausgehen, dass diese mit $v_1, \ldots, v_n$ durchnummeriert sind und falls wir v_i und v_j mit gleichem Schlüssel d in Q einfügen, so wird v_i als kleiner als v_j betrachtet, wenn $i < j$ gilt. Auf diese Weise ist stets eine eindeutige Ordnung auf den Schlüsseln gegeben. Da die Höhe eines AVL-Baumes mit n Knoten durch $O(\log(n))$ beschränkt ist, können wir INSERT(Q, x, d) in Zeit $O(\log(n))$ ausführen.

EXTRACT-MIN(Q) können wir in einem AVL-Baum realisieren, indem wir ausgehend von der Wurzel immer dem Zeiger zum linken Kind folgen. Auf diese Weise finden wir das Objekt mit dem kleinsten Schlüssel. Auch dies geht wegen der beschränkten Höhe in Zeit $O(\log(n))$. Haben wir es einmal gefunden, so können wir es in Zeit $O(\log n)$ löschen und zurückgeben. Ebenso können wir DECREASE-KEY in Zeit $O(\log(n))$ realisieren, indem wir zunächst nach x suchen (dafür ist wichtig, dass wir seinen momentanen Schlüssel d_1 übergeben bekommen), es dann löschen und mit dem neuen Schlüssel d_2 wieder einfügen.

Theorem 31 *Die Laufzeit des Algorithmus von Dijkstra beträgt $O((n + m) \log n)$.*

Beweis. Die Initialisierung erfolgt in Zeit $O(n)$. Da die Adjazenzliste eines Knoten $v \in V$ dann das einzige Mal durchlaufen wird, wenn der Knoten v der Menge S hinzugefügt wird, wird für jede Kante $e = (u, v) \in E$ genau einmal die Operation RELAX(u, v) aufgerufen. Innerhalb des Aufrufs von RELAX(u, v) wird gegebenenfalls der Wert $d(v)$ reduziert. Dies entspricht einem Aufruf von CHANGE-KEY, welcher Laufzeit $O(\log n)$ benötigt. Alle Aufrufe der RELAX-Methode zusammen haben somit eine Laufzeit von $O(m \log n)$. Wir benötigen außerdem genau n Aufrufe von EXTRACT-MIN. Diese haben eine Gesamtlaufzeit von $O(n \log n)$. Damit folgt das Theorem. □

7.2.2 All-Pairs Shortest Path Problem

In diesem Abschnitt beschäftigen wir uns mit dem APSP. Dazu sei ein gerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to \mathbb{R}$ gegeben. Im Gegensatz zum Dijkstra-Algorithmus funktioniert der Floyd-Warshall-Algorithmus, den wir gleich beschreiben werden, auch, wenn es negative Kantengewichte gibt. Enthält der Graph einen negativen Kreis, so stellt der Algorithmus dies fest und bricht ab.

Für den Floyd-Warshall-Algorithmus gehen wir davon aus, dass $V = \{1, \ldots, n\}$ gilt, und wir untersuchen Wege, die nur bestimmte Knoten benutzen dürfen. Sei $P = (v_0, \ldots, v_\ell)$ ein kürzester Weg von v_0 nach v_ℓ. Wir haben bereits im SSSP argumentiert, dass es für jedes Paar von Knoten u und v in einem Graphen ohne negativen Kreis stets einen kürzesten Weg von u nach v gibt, der keinen Knoten mehrfach besucht. Wir nehmen an, dass dies auf P zutrifft, das heißt, die Knoten $v_0, \ldots, v_\ell$ seien paarweise verschieden. Wir nennen einen solchen Weg auch *einfachen Weg*. Dann nennen wir $v_1, \ldots, v_{\ell-1}$ die *Zwischenknoten von* P. Wir schränken nun die Menge der erlaubten Zwischenknoten ein.

Wir betrachten für jedes Paar $i, j \in V$ alle einfachen Wege von i nach j, die nur Zwischenknoten aus der Menge $\{1, \ldots, k\}$ für ein bestimmtes k benutzen dürfen. Sei P_{ij}^k der kürzeste solche Weg von i nach j und sei $\delta_{ij}^{(k)}$ seine Länge. Wir überlegen uns, wie wir den Weg P_{ij}^k und $\delta_{ij}^{(k)}$ bestimmen können, wenn wir für jedes Paar $i, j \in V$ bereits den Weg P_{ij}^{k-1} kennen, also den kürzesten Weg von i nach j, der als Zwischenknoten nur Knoten aus $\{1, \ldots, k-1\}$ benutzen darf. Hierzu unterscheiden wir zwei Fälle:

- Enthält der Weg P_{ij}^k den Knoten k nicht als Zwischenknoten, so enthält er nur die Knoten aus $\{1, \ldots, k-1\}$ als Zwischenknoten und somit ist P_{ij}^k auch dann ein kürzester Weg, wenn wir nur diese Knoten als Zwischenknoten erlauben. Es gilt dann also $\delta_{ij}^{(k)} = \delta_{ij}^{(k-1)}$ und $P_{ij}^k = P_{ij}^{k-1}$.

- Enthält der Weg P_{ij}^k den Knoten k als Zwischenknoten, so zerlegen wir ihn wie folgt in zwei Teile:
$$P_{ij}^k = i \overset{P}{\rightsquigarrow} k \overset{P'}{\rightsquigarrow} j.$$
Da P_{ij}^k ein einfacher Weg ist, kommt k nur einmal vor und ist somit nicht in den Teilwegen P und P' enthalten. Diese müssen laut Lemma 24 kürzeste Wege von i nach k bzw. von k nach j sein, die nur Knoten aus $\{1, \ldots, k-1\}$ als Zwischenknoten benutzen. Somit gilt in diesem Fall
$$\delta_{ij}^{(k)} = \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}.$$
und
$$P_{ij}^k = i \overset{P_{ik}^{k-1}}{\rightsquigarrow} k \overset{P_{kj}^{k-1}}{\rightsquigarrow} j.$$

Aus dieser Beobachtung können wir direkt rekursive Formeln zur Berechnung der Entfernungen $\delta^{(k)}(i, j)$ und der Wege P_{ij}^k herleiten. Wir betrachten zunächst nur die Entfernungen. Für alle $i, j \in V$ gilt
$$\delta_{ij}^{(k)} = \begin{cases} w(i,j) & \text{für } k = 0 \\ \min\{\delta_{ij}^{(k-1)}, \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}\} & \text{für } k > 0. \end{cases}$$

Für $k = n$ erlauben wir alle Knoten $\{1, \ldots, n\}$ als Zwischenknoten. Demzufolge besteht für $n = k$ keine Einschränkung mehr und es gilt $\delta(i, j) = \delta_{ij}^{(n)}$ für alle $i, j \in V$.

Wir weisen an dieser Stelle nochmals explizit darauf hin, dass wir für diese Rekursionsformel essentiell ausgenutzt haben, dass es im Graphen G keinen Kreis mit negativem Gesamtgewicht

gibt. Gibt es nämlich einen solchen Kreis, der von einem Knoten $i \in V$ erreichbar ist und von dem aus der Knoten $j \in V$ erreichbar ist, so ist der kürzeste Weg von i nach j nicht mehr einfach. Er würde stattdessen den Kreis unendlich oft durchlaufen und demzufolge die Knoten auf dem Kreis mehrfach besuchen.

Floyd-Warshall-Algorithmus

Der Floyd-Warshall-Algorithmus nutzt die rekursive Darstellung der $\delta_{ij}^{(k)}$, die wir soeben hergeleitet haben, um das APSP zu lösen. Zur Vereinfachung der Notation gehen wir davon aus, dass $V = \{1, \ldots, n\}$ gilt und dass der Graph als $(n \times n)$-Adjazenzmatrix $W = (w_{ij})$ gegeben ist. Die Adjazenzmatrix W enthält bei gewichteten Graphen an der Stelle w_{ij} das Gewicht der Kante (i, j). Existiert diese Kante nicht, so gilt $w_{ij} = \infty$. Des Weiteren gilt $w_{ii} = 0$ für alle $i \in V$.

```
    FLOYD-WARSHALL(W)
1   D^(0) = W
2   for k = 1 to n {
3       Erzeuge (n × n)-Nullmatrix D^(k) = (δ_ij^(k))
4       for i = 1 to n {
5           for j = 1 to n {
6               δ_ij^(k) = min{δ_ij^(k-1), δ_ik^(k-1) + δ_kj^(k-1)}
7           }
8       }
9   }
10  return D^(n)
```

Theorem 32 *Der Floyd-Warshall-Algorithmus löst das APSP für Graphen ohne Kreise mit negativem Gesamtgewicht, die als Adjazenzmatrix dargestellt sind und n Knoten enthalten, in Zeit* $\Theta(n^3)$.

Beweis. Die Korrektheit des Algorithmus folgt direkt aus der rekursiven Formel für $\delta_{ij}^{(k)}$, die wir oben hergeleitet haben, da der Algorithmus exakt diese rekursive Formel berechnet. Die Laufzeit von $\Theta(n^3)$ ist leicht ersichtlich, da wir drei ineinander geschachtelte **for**-Schleifen haben, die jeweils über n verschiedene Werte zählen, und innerhalb der innersten Schleifen nur Operationen ausgeführt werden, die konstante Zeit benötigen. Das Erstellen der Nullmatrizen benötigt insgesamt ebenfalls Zeit $\Theta(n^3)$. □

Wenn wir zusätzlich zu den Entfernungen auch die kürzesten Wege ermitteln wollen, so können wir dies auch wieder mittels einer rekursiv definierten Vorgängerfunktion π machen. Sei $\pi_{ij}^{(k)}$ der Vorgänger von j auf dem Pfad P_{ij}^k, das heißt auf dem kürzesten Pfad von i nach j, der als Zwischenknoten nur Knoten aus $\{1, \ldots, k\}$ benutzen darf. Da für $k = 0$ überhaupt keine Zwischenknoten erlaubt sind, gilt für alle Knoten $i, j \in V$

$$\pi_{ij}^{(0)} = \begin{cases} \text{nil} & \text{falls } i = j \text{ oder } w_{ij} = \infty, \\ \text{i} & \text{falls } i \neq j \text{ und } w_{ij} < \infty. \end{cases}$$

Das heißt, $\pi_{ij}^{(0)}$ ist genau dann gleich i, wenn es eine direkte Kante von i nach j gibt. Für $k > 0$ orientieren wir uns an der rekursiven Formel für $\delta_{ij}^{(k)}$, die wir oben hergeleitet haben.

Ist $\delta_{ij}^{(k)} = \delta_{ij}^{(k-1)}$, so ist auch $P_{ij}^k = P_{ij}^{k-1}$ und somit ist der Vorgänger von j auf dem Pfad P_{ij}^k derselbe wie der auf dem Pfad P_{ij}^{k-1}. Ist $\delta_{ij}^{(k)} < \delta_{ij}^{(k-1)}$, so gilt

$$P_{ij}^k = i \overset{P_{ik}^{k-1}}{\rightsquigarrow} k \overset{P_{kj}^{k-1}}{\rightsquigarrow} j,$$

und damit ist insbesondere der Vorgänger von j auf dem Pfad P_{ij}^k derselbe wie auf dem Pfad P_{kj}^{k-1}. Zusammenfassend halten wir für $k > 0$ fest

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{falls } \delta_{ij}^{(k-1)} \leq \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{falls } \delta_{ij}^{(k-1)} > \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}. \end{cases}$$

Die $\pi_{ij}^{(k)}$ können zusammen mit den $\delta_{ij}^{(k)}$ im Floyd-Warshall-Algorithmus berechnet werden, ohne dass sich die Laufzeit signifikant ändert.

Negative Kreise Wir wollen nun noch kurz diskutieren, wie man den Floyd-Warshall-Algorithmus benutzen kann, um die Existenz eines Kreises mit negativem Gesamtgewicht zu detektieren. Dazu müssen wir uns nur überlegen, was die Einträge $\delta_{ii}^{(k)}$ auf der Diagonalen der Matrix $D^{(k)}$ bedeuten. Für $k = 0$ werden diese Einträge alle mit 0 initialisiert. Gibt es nun für $k > 0$ einen Kreis mit negativem Gesamtgewicht, der den Knoten i enthält und sonst nur Knoten aus der Menge $\{1, \ldots, k\}$, so gilt $\delta_{ii}^{(k)} < 0$ und somit auch $\delta_{ii}^{(n)} < 0$, da es einen Pfad negativer Länge von i nach i gibt. Also genügt es, am Ende des Algorithmus, die Einträge auf der Hauptdiagonalen zu betrachten. Ist mindestens einer von diesen negativ, so wissen wir, dass ein Kreis mit negativem Gesamtgewicht existiert. In diesem Fall, gibt die vom Algorithmus berechnete Matrix $D^{(n)}$ nicht die korrekten Abstände an.

Vergleich mit Algorithmus von Dijkstra Wir haben weiter oben bereits angesprochen, dass das APSP auch dadurch gelöst werden kann, dass wir einen Algorithmus für das SSSP für jeden möglichen Startknoten einmal aufrufen. Hat der Algorithmus für das SSSP eine Laufzeit von $O(f)$, so ergibt sich zur Lösung des APSP eine Laufzeit von $O(nf)$. Für den Dijkstra-Algorithmus ergibt sich also eine Laufzeit von $O(nm \log n)$.

Für Graphen mit nicht-negativen Kantengewichten stellt sich somit die Frage, ob es effizienter ist, den Floyd-Warshall-Algorithmus mit Laufzeit $O(n^3)$ oder den Dijkstra-Algorithmus mit Lösung $O(nm \log n)$ zur Lösung des APSP zu nutzen. Diese Frage lässt sich nicht allgemein beantworten. Für dichte Graphen mit $m = \Theta(n^2)$ vielen Kanten ist zum Beispiel der Floyd-Warshall-Algorithmus schneller. Für dünne Graphen mit $m = O(n)$ Kanten ist hingegen die n-malige Ausführung des Dijkstra-Algorithmus schneller.

Literatur

Das SSSP und APSP werden in den Kapiteln 24 und 25 von [1] beschrieben. Außerdem sind sie in den Kapiteln 4.3.1 und 4.3.3 von [2] erklärt.

7.3 Flussprobleme

In diesem Abschnitt lernen wir mit Flussproblemen eine weitere wichtige Klasse von algorithmischen Graphproblemen kennen, die in vielen Bereichen Anwendung findet. Ein *Flussnetzwerk* ist ein gerichteter Graph $G = (V, E)$ mit zwei ausgezeichneten Knoten $s, t \in V$ und einer

Kapazitätsfunktion $c : V \times V \to \mathbb{N}_0$, die jeder Kante $(u, v) \in E$ eine ganzzahlige nicht-negative Kapazität $c(u, v)$ zuweist. Wir definieren $c(u, v) = 0$ für alle $(u, v) \notin E$. Außerdem nennen wir den Knoten s die *Quelle* und den Knoten t die *Senke*. Wir definieren $n = |V|$ und $m = |E|$. Außerdem nehmen wir an, dass jeder Knoten von der Quelle s zu erreichen ist. Das bedeutet insbesondere, dass $m \geq n - 1$ gilt.

Definition 33 *Ein* Fluss *in einem Flussnetzwerk ist eine Funktion* $f : V \times V \to \mathbb{R}$*, die die folgenden beiden Eigenschaften erfüllt.*

1. Flusserhaltung*: Für alle Knoten* $u \in V \setminus \{s, t\}$ *gilt*
$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$
2. Kapazitätsbeschränkung*: Für alle Knoten* $u, v \in V$ *gilt*
$$0 \leq f(u, v) \leq c(u, v).$$

Wir definieren den Wert *eines Flusses* $f : V \times V \to \mathbb{R}$ *als*
$$|f| := \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$
und weisen explizit darauf hin, dass die Notation $|.|$ *hier keinen Betrag bezeichnet, sondern den Wert des Flusses gemäß obiger Definition.*

Das Flussproblem, das wir uns in diesem Abschnitt anschauen werden, lässt sich nun wie folgt formulieren:

> Gegeben sei ein Flussnetzwerk G. Berechne einen maximalen Fluss auf G, d. h. einen Fluss f mit größtmöglichem Wert $|f|$.

Anschaulich ist die Frage also, wie viele Einheiten Fluss man in dem gegebenen Netzwerk maximal von der Quelle zur Senke transportieren kann.

Als erste einfache Beobachtung halten wir fest, dass der Fluss, der in der Quelle entsteht, komplett in der Senke ankommt und nicht zwischendurch versickert.

Lemma 34 *Sei* f *ein Fluss in einem Flussnetzwerk* G*. Dann gilt*
$$|f| = \sum_{v \in V} f(v, t) - \sum_{v \in V} f(t, v).$$

Beweis. Als erstes beobachten wir, dass die folgenden beiden Summen gleich sind:
$$\sum_{u \in V} \sum_{v \in V} f(v, u) = \sum_{u \in V} \sum_{v \in V} f(u, v).$$
Dies folgt einfach daraus, dass wir sowohl links als auch rechts über jedes Knotenpaar genau einmal summieren. Nun nutzen wir die Flusserhaltung für alle Knoten $u \in V \setminus \{s, t\}$ und erhalten aus obiger Gleichung:
$$\begin{array}{ll} \sum_{u \in \{s,t\}} \sum_{v \in V} f(v, u) & = \sum_{u \in \{s,t\}} \sum_{v \in V} f(u, v) \\ \iff \sum_{v \in V} f(v, s) + \sum_{v \in V} f(v, t) & = \sum_{v \in V} f(s, v) + \sum_{v \in V} f(t, v) \\ \iff \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) & = \sum_{v \in V} f(v, t) - \sum_{v \in V} f(t, v), \end{array}$$
woraus mit der Definition von $|f|$ direkt das Lemma folgt. □

Anwendungsbeispiele

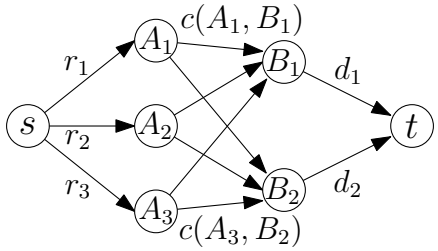
Bevor wir Algorithmen zur Lösung des Flussproblems vorstellen, präsentieren wir ein Anwendungsbeispiel, das wir aus Kapitel 4.5 von [2] übernommen haben. Nehmen wir an, in den Seehäfen $A_1, \ldots, A_p$ liegen Bananen zur Verschiffung bereit. Es gibt Zielhäfen $B_1, \ldots, B_q$, zu denen die Bananen transportiert werden sollen. Für $i = 1, \ldots, p$ bezeichne r_i, wie viele Tonnen Bananen im Hafen A_i bereit stehen, und für $j = 1, \ldots, q$ bezeichne d_j wie viele Tonnen Bananen am Zielfhafen B_j angefordert wurden. Für $i = 1, \ldots, p$ und $j = 1, \ldots, q$ gibt es zwischen den Häfen A_i und B_j eine Schifffahrtslinie, die maximal $c(A_i, B_j)$ Tonnen Bananen transportieren kann. Es kann auch sein, dass es zwischen zwei Häfen A_i und B_j keine Schifffahrtslinie gibt; in diesem Falle definieren wir $c(A_i, B_j)$ als 0. Die folgenden Fragen stellen sich dem Handelsunternehmen:

1. Ist es möglich, alle Anforderungen zu befriedigen?
2. Falls nein, wie viele Tonnen Bananen können maximal zu den Zielhäfen transportiert werden?
3. Wie sollen die Bananen verschifft werden?

Alle diese Fragen können als Flussproblem modelliert werden. Dazu konstruieren wir einen Graphen $G = (V, E)$ mit den Knoten $V = \{s, t, A_1, \ldots, A_p, B_1, \ldots, B_q\}$. Es gibt drei Typen von Kanten:

- Für $i = 1, \ldots, p$ gibt es eine Kante (s, A_i) mit Kapazität r_i.
- Für $j = 1, \ldots, q$ gibt es eine Kante (B_j, t) mit Kapazität q_j.
- Für jedes $i = 1, \ldots, p$ und $j = 1, \ldots, q$ gibt es eine Kante (A_i, B_j) mit Kapazität $c(A_i, B_j)$.

Die folgende Abbildung zeigt ein Beispiel für $p = 3$ und $q = 2$.



Sei $f : E \to \mathbb{R}$ ein maximaler Fluss im Graphen G. Man kann sich anschaulich schnell davon überzeugen, dass man mit Hilfe dieses Flusses alle drei Fragen, die oben gestellt wurden, beantworten kann:

1. Wir können alle Anforderungen befriedigen, wenn der Wert $|f|$ des Flusses gleich $\sum_{j=1}^{q} d_j$ ist.
2. Wir können maximal $|f|$ Tonnen Bananen zu den Zielhäfen transportieren.
3. Der Fluss $f(e)$ auf Kanten $e = (A_i, B_j)$ gibt an, wie viele Tonnen Bananen entlang der Schifffahrtslinie von A_i nach B_j transportiert werden sollen.

Wir verzichten an dieser Stelle auf einen formalen Beweis, dass dies die korrekten Antworten auf obige Fragen sind, möchten aber den Leser dazu anregen, selbst darüber nachzudenken.

Flussprobleme können auch dazu genutzt werden, andere Probleme zu modellieren, die auf den ersten Blick ganz anders geartet sind. Ein Beispiel (siehe Übungen) ist es, zu entscheiden, ob eine Fußballmannschaft bei einer gegebenen Tabellensituation und einem gegebenen restlichen Spielplan gemäß der alten Zwei-Punkte-Regel noch Meister werden kann.

7.3.1 Algorithmus von Ford und Fulkerson

Der folgende Algorithmus von Ford und Fulkerson berechnet für ein gegebenes Flussnetzwerk $G = (V, E)$ mit Kapazitäten $c : V \times V \to \mathbb{N}_0$ einen maximalen Fluss. Wir gehen in diesem Kapitel davon aus, dass das Flussnetzwerk dergestalt ist, dass für kein Paar von Knoten $u, v \in V$ die beiden Kanten (u, v) und (v, u) in E enthalten sind. Dies ist keine wesentliche Einschränkung, da jedes Netzwerk, das diese Eigenschaft verletzt, einfach in ein äquivalentes umgebaut werden kann, das keine entgegengesetzten Kanten enthält. Es ist eine gute Übung für den Leser, sich zu überlegen, wie das funktioniert.

```
    FORD-FULKERSON(G, c, s ∈ V, t ∈ V)
1   Setze f(e) = 0 für alle e ∈ E.   // f ist gültiger Fluss mit Wert 0
2   while ∃ flussvergrößernder Weg P {
3       Erhöhe den Fluss f entlang P.
4   }
5   return f;
```

Wir müssen noch beschreiben, was ein *flussvergrößernder Weg* ist und was es bedeutet, den Fluss entlang eines solchen Weges zu erhöhen. Dazu führen wir den Begriff des *Restnetzwerkes* ein.

Definition 35 *Sei $G = (V, E)$ ein Flussnetzwerk mit Kapazitäten $c : V \times V \to \mathbb{N}_0$ und sei f ein Fluss in G. Das dazugehörige* Restnetzwerk *$G_f = (V, E_f)$ ist auf der gleichen Menge von Knoten V definiert wie das Netzwerk G. Wir definieren eine Funktion $rest_f : V \times V \to \mathbb{R}$ mit*

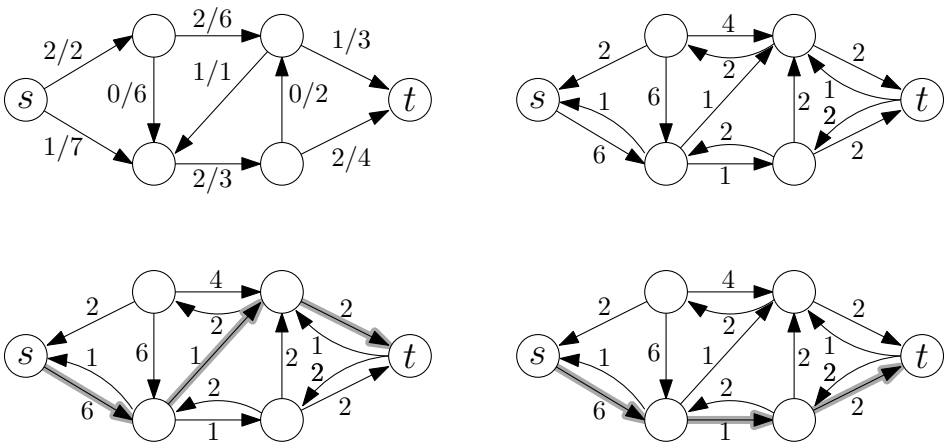
$$rest_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \textit{falls } (u, v) \in E, \\ f(v, u) & \textit{falls } (v, u) \in E, \\ 0 & \textit{sonst.} \end{cases}$$

Die Kantenmenge E_f ist definiert als

$$E_f = \{(u, v) \in V \times V \mid rest_f(u, v) > 0\}.$$

Ein flussvergrößernder Weg *ist ein einfacher Weg von s nach t im Restnetzwerk G_f.*

Die folgende Abbildung zeigt oben links ein Beispiel für ein Flussnetzwerk mit einem Fluss f. Dabei bedeutet die Beschriftung a/b an einer Kante e, dass $f(e) = a$ und $c(e) = b$ gilt. Rechts oben ist das zugehörige Restnetzwerk dargestellt, und in der unteren Reihe sind zwei flussvergrößernde Wege zu sehen.



Nun müssen wir noch klären, was es bedeutet, den Fluss entlang eines Weges zu erhöhen.

Definition 36 *Sei G ein Flussnetzwerk mit Kapazitäten $c : V \times V \to \mathbb{N}$, sei $f : V \times V \to \mathbb{R}$ ein Fluss und sei P ein Weg im Restnetzwerk G_f von s nach t. Wir bezeichnen mit $f \uparrow P : V \times V \to \mathbb{R}$ den Fluss, der entsteht, wenn wir f entlang P erhöhen. Dieser Fluss ist definiert durch*

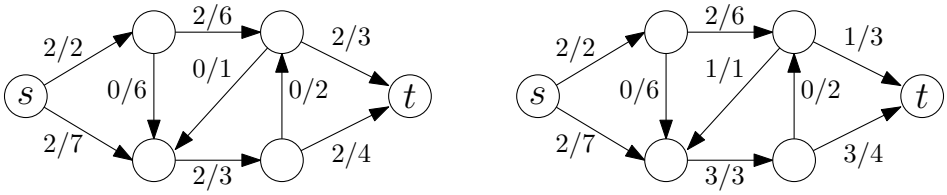
$$(f \uparrow P)(u,v) = \begin{cases} f(u,v) + \delta & \text{falls } (u,v) \in E \text{ und } (u,v) \in P, \\ f(u,v) - \delta & \text{falls } (u,v) \in E \text{ und } (v,u) \in P, \\ f(u,v) & \text{sonst,} \end{cases}$$

wobei

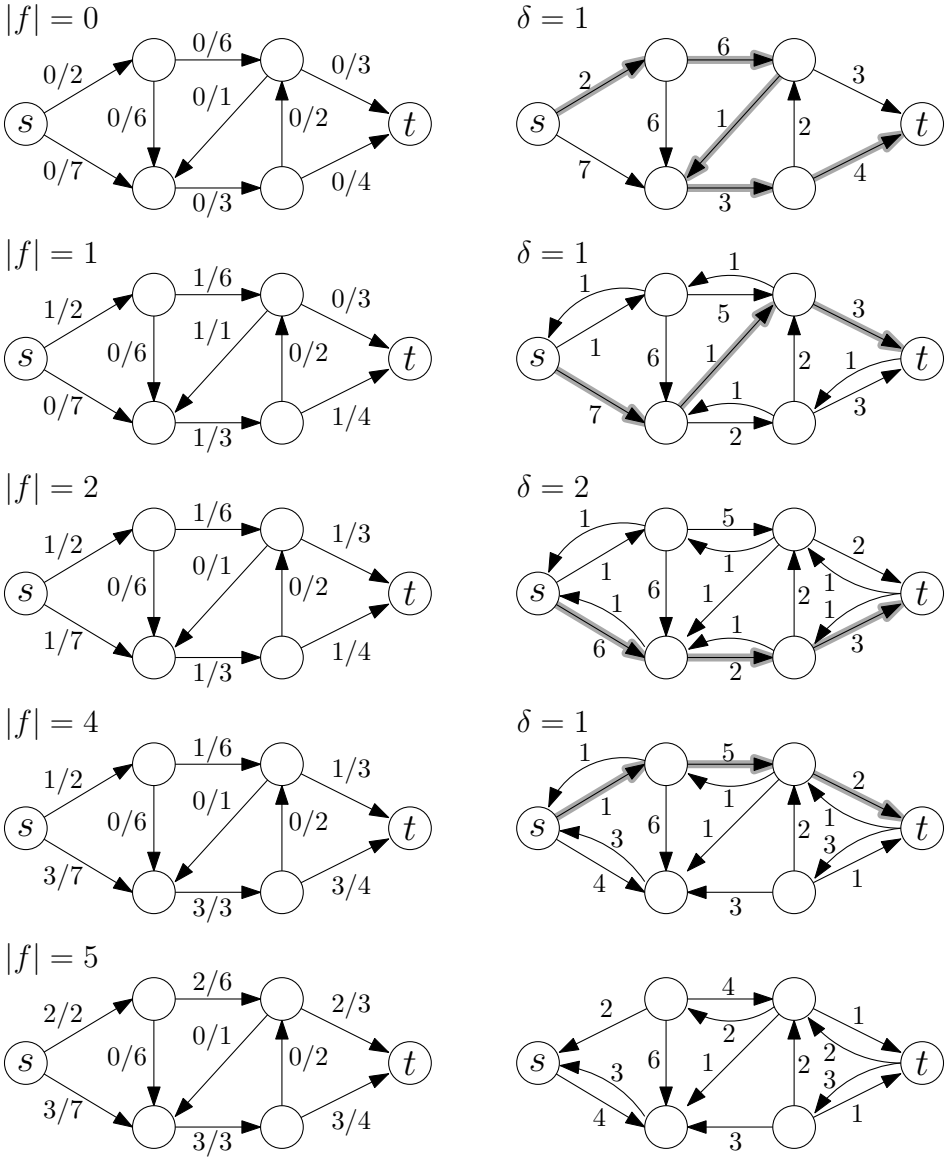
$$\delta = \min_{e \in P}(rest_f(e)).$$

Aus der Definition von E_f folgt, dass $\delta > 0$ gilt.

In der folgenden Abbildung sind die Flüsse dargestellt, die entstehen, wenn wir den Fluss in obiger Abbildung entlang der beiden oben dargestellten Wege verbessern. In beiden Beispielen ist $\delta = 1$.



Die folgende Abbildung zeigt ein Beispiel für die Ausführung des Algorithmus von Ford und Fulkerson.



Korrektheit

Um die Korrektheit des Algorithmus von Ford und Fulkerson zu zeigen, müssen wir zunächst zeigen, dass $f \uparrow P$ wieder ein Fluss ist.

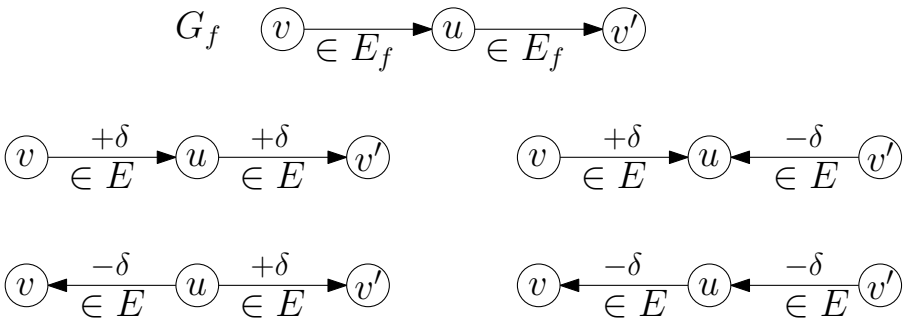
Lemma 37 *Sei f ein Fluss in einem Netzwerk G und sei P ein Weg von s nach t im Restnetzwerk G_f. Die Funktion $f \uparrow P : V \times V \to \mathbb{R}$ ist wieder ein Fluss. Für diesen Fluss gilt*

$$|f \uparrow P| = |f| + \delta.$$

Beweis. Wir müssen zeigen, dass die beiden Eigenschaften aus Definition 33 erfüllt sind.

- *Flusserhaltung*: Sei $u \in V \setminus \{s, t\}$. Falls u auf dem Weg P nicht vorkommt, so gilt $f(u, v) = (f \uparrow P)(u, v)$ und $f(v, u) = (f \uparrow P)(v, u)$ für alle Knoten $v \in V$ und somit folgt die Flusserhaltung für u in $f \uparrow P$ aus der Flusserhaltung für u im Fluss f.

Kommt u auf dem Weg P vor, so gibt es genau eine Kante $e \in P$ der Form (v, u) für ein $v \in V$ und es gibt genau eine Kante e' der Form (u, v') für ein $v' \in V$. Die Eindeutigkeit folgt daraus, dass P per Definition einfach ist und somit den Knoten u nur einmal besucht. Die Kanten e und e' sind die einzigen zu u inzidenten Kanten, auf denen sich der Fluss ändert. Wir brauchen also nur diese beiden Kanten näher zu betrachten. Es gilt entweder $(v, u) \in E$ oder $(u, v) \in E$. Ebenso gilt entweder $(v', u) \in E$ oder $(u, v') \in E$. Wir schauen uns alle vier Fälle einzeln an und überprüfen, dass die Flusserhaltung erhalten bleibt. In der folgenden Abbildung ist oben der relevante Ausschnitt des Restnetzwerkes dargestellt und darunter die vier Fälle. In jedem der Fälle ist leicht zu sehen, dass die Flusserhaltung an u erhalten bleibt.



- *Kapazitätsbeschränkung*: Seien $u, v \in V$ beliebig so gewählt, dass $e = (u, v) \in E_f$ eine Kante auf dem Weg P in G_f ist. Wir zeigen, dass $0 \leq (f \uparrow P)(e) \leq c(e)$ gilt. Dafür unterscheiden wir zwei Fälle.
  - Ist $e = (u, v) \in E$, so erhöhen wir den Fluss auf e um δ. Auf Grund der Definition von δ und wegen $f(e) \geq 0$ und $\delta \geq 0$ gilt
    $$0 \leq f(e) + \delta \leq f(e) + \text{rest}_f(e) = f(e) + (c(e) - f(e)) = c(e).$$
  - Ist $e' = (v, u) \in E$, so verringern wir den Fluss auf e' um δ. Auf Grund der Definition von δ und wegen $f(e') \leq c(e')$ und $\delta \geq 0$ gilt
    $$c(e') \geq f(e') - \delta \geq f(e') - \text{rest}_f(e) \geq f(e') - f(e') = 0.$$

□

Aus Lemma 37 folgt direkt, dass die Funktion f, die der Algorithmus von Ford und Fulkerson verwaltet, nach jeder Iteration ein Fluss ist. Wir zeigen nun, dass der Algorithmus, wenn er terminiert, einen maximalen Fluss berechnet hat. Dazu führen wir zunächst die folgende Definition ein.

Definition 38 *Sei $G = (V, E)$ ein Flussnetzwerk mit Kapazitäten $c : V \times V \to \mathbb{N}$. Sei $s \in V$ die Quelle und $t \in V$ die Senke. Ein* Schnitt *von G ist eine Partitionierung der Knotenmenge V in zwei Teile $S \subseteq V$ und $T \subseteq V$ mit $s \in S$, $t \in T$ und $T = V \setminus S$. Wir nennen*

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

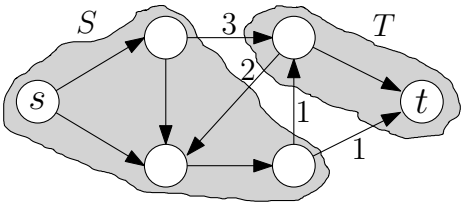
die Kapazität *des Schnittes (S, T). Ein Schnitt (S, T) heißt* minimal*, wenn es keinen Schnitt (S', T') mit $c(S', T') < c(S, T)$ gibt.*

Für einen Fluss f und einen Schnitt (S, T) sei

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

der Fluss über den Schnitt.

Die folgende Abbildung zeigt einen Schnitt mit Kapazität 5.



Anschaulich ist leicht einzusehen, dass die Kapazität $c(S,T)$ eine obere Schranke dafür ist, wie viel Fluss von der Quelle zur Senke geschickt werden kann, und dass für einen Fluss f der Fluss über den Schnitt (S,T) genau $|f|$ sein muss. Formal zeigen wir das in folgendem Lemma.

Lemma 39 *Sei (S,T) ein Schnitt eines Flussnetzwerkes G und sei f ein Fluss in G. Dann gilt*

$$|f| = f(S,T) \leq c(S,T).$$

Beweis. Ähnlich zu dem Beweis von Lemma 34 summieren wir zunächst über alle Kanten, die innerhalb der Menge S verlaufen. Wir erhalten die folgende Gleichung:

$$\begin{aligned} \sum_{v\in S} f(s,v) + \sum_{u\in S\setminus\{s\}} \sum_{v\in S} f(u,v) &= \sum_{v\in S} f(v,s) + \sum_{u\in S\setminus\{s\}} \sum_{v\in S} f(v,u) \\ &\Longleftrightarrow \\ \sum_{v\in S} f(s,v) - \sum_{v\in S} f(v,s) &= \sum_{u\in S\setminus\{s\}} \sum_{v\in S} f(v,u) - \sum_{u\in S\setminus\{s\}} \sum_{v\in S} f(u,v). \end{aligned} \tag{7.1}$$

Für alle Knoten $u \in S \setminus \{s\}$ gilt die Flusserhaltung. Dafür müssen wir aber auch die Kanten betrachten, die von u in die Menge T führen, und die Kanten, die von der Menge T zu u führen. Wir erhalten

$$\sum_{v\in S} f(u,v) + \sum_{v\in T} f(u,v) = \sum_{v\in S} f(v,u) + \sum_{v\in T} f(v,u).$$

Damit können wir Geichung 7.1 schreiben als

$$\sum_{v\in S} f(s,v) - \sum_{v\in S} f(v,s) = \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(u,v) - \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(v,u). \tag{7.2}$$

Für die Quelle s gilt entsprechend

$$\begin{aligned} \sum_{v\in S} f(s,v) + \sum_{v\in T} f(s,v) &= |f| + \sum_{v\in S} f(v,s) + \sum_{v\in T} f(v,s) \\ &\Longleftrightarrow \\ \sum_{v\in S} f(s,v) - \sum_{v\in S} f(v,s) &= |f| + \sum_{v\in T} f(v,s) - \sum_{v\in T} f(s,v). \end{aligned} \tag{7.3}$$

Wir setzen Gleichung 7.3 in Gleichung 7.2 ein und erhalten

$$\begin{aligned} |f| + \sum_{v\in T} f(v,s) - \sum_{v\in T} f(s,v) &= \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(u,v) - \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(v,u) \\ &\Longleftrightarrow \\ |f| &= \sum_{u\in S} \sum_{v\in T} f(u,v) - \sum_{u\in S} \sum_{v\in T} f(v,u) = f(S,T). \end{aligned}$$

Die Ungleichung folgt einfach aus obiger Gleichung, da $f(u,v) \leq c(u,v)$ und $f(v,u) \geq 0$ für alle $u, v \in V$:

$$f(S,T) \leq \sum_{u \in S} \sum_{v \in T} f(u,v) \leq \sum_{u \in S} \sum_{v \in T} c(u,v) \leq c(S,T).$$

□

Mit Hilfe von Lemma 39 zeigen wir das folgende Theorem, aus dem direkt folgt, dass der Algorithmus von Ford und Fulkerson einen maximalen Fluss gefunden hat, wenn er terminiert.

Theorem 40 *Sei f ein Fluss in einem Netzwerk G. Dann sind die folgenden drei Aussagen äquivalent.*

a) f ist ein maximaler Fluss.

b) Das Restnetzwerk G_f enthält keinen flussvergrößernden Weg.

c) Es gibt einen Schnitt (S,T) mit $|f| = c(S,T)$.

Dieses Theorem besagt insbesondere, dass es einen Schnitt (S,T) geben muss, dessen Kapazität so groß ist wie der Wert $|f|$ des maximalen Flusses f. Zusammen mit Lemma 39, das besagt, dass es keinen Schnitt (S',T') mit kleinerer Kapazität als $|f|$ geben kann, folgt die wichtige Aussage, dass die Kapazität des minimalen Schnittes gleich dem Wert des maximalen Flusses ist. Die Korrektheit des Algorithmus von Ford und Fulkerson bei Terminierung folgt einfach daraus, dass der Algorithmus erst dann terminiert, wenn es keinen flussvergrößernden Weg mehr gibt. Nach dem Theorem ist das aber genau dann der Fall, wenn der Fluss maximal ist.

Beweis.[Beweis von Theorem 40] Wir werden zeigen, dass b) aus a) folgt, dass c) aus b) folgt und dass a) aus c) folgt. Diese drei Implikationen bilden einen Ringschluss und damit ist gezeigt, dass die drei Aussagen äquivalent sind.

- **a) ⇒ b)** Wir führen einen Widerspruchsbeweis und nehmen an, dass es einen flussvergrößernden Weg P in G_f gibt. Dann können wir den Fluss f entlang P erhöhen und erhalten den Fluss $f \uparrow P$, welcher nach Lemma 37 einen um $\delta > 0$ höheren Wert als f hat. Damit kann f kein maximaler Fluss sein.

- **c) ⇒ a)** Sei f ein Fluss mit $|f| = c(S,T)$ für einen Schnitt (S,T) und sei f' ein maximaler Fluss. Dann gilt $|f| \leq |f'|$. Wenn wir Lemma 39 für den Fluss f' und den Schnitt (S,T) anwenden, dann erhalten wir $|f'| \leq c(S,T)$. Zusammen ergibt das

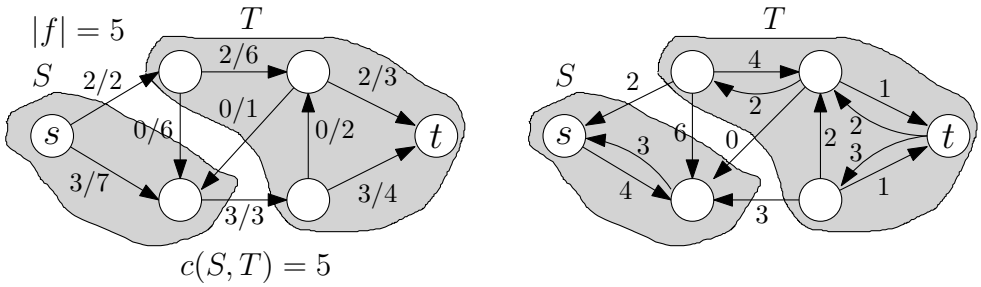
  $$c(S,T) = |f| \leq |f'| \leq c(S,T),$$

  woraus $c(S,T) = |f| = |f'|$ folgt. Damit ist auch f ein maximaler Fluss.

- **b) ⇒ c)** Laut Voraussetzung gibt es keinen flussvergrößernden Weg, das heißt, es gibt keinen Weg von s nach t im Restnetzwerk G_f. Wir teilen die Knoten V des Restnetzwerkes in zwei Klassen S und T ein:

  $$S = \{v \in V \mid \text{ es gibt Weg in } G_f \text{ von } s \text{ nach } v\} \quad \text{und} \quad T = V \setminus S.$$

  Da der Knoten t in G_f nicht von s aus erreichbar ist, gilt $s \in S$ und $t \in T$. Damit ist (S,T) ein Schnitt des Graphen. Die Situation ist in folgender Abbildung dargestellt.



Wir argumentieren nun, dass $|f| = c(S,T)$ für diesen Schnitt (S,T) gilt.

- Sei $(u,v) \in E$ eine Kante mit $u \in S$ und $v \in T$. Da u in G_f von s aus erreichbar ist, v aber nicht, gilt $(u,v) \notin E_f$. Mit der Definition des Restnetzwerkes folgt $\text{rest}_f(u,v) = 0$, also $f(u,v) = c(u,v)$.
- Sei $(v,u) \in E$ eine Kante mit $u \in S$ und $v \in T$. Da u in G_f von s aus erreichbar ist, v aber nicht, gilt $(u,v) \notin E_f$. Mit der Definition des Restnetzwerkes folgt $\text{rest}_f(u,v) = 0$, also $f(v,u) = 0$.

Somit gilt mit Lemma 39

$$\begin{aligned} c(S,T) = \sum_{u\in S}\sum_{v\in T} c(u,v) &= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \\ &= f(S,T) = |f| \leq c(S,T). \end{aligned}$$

Damit folgt $|f| = f(S,T) = c(S,T)$.

□

Laufzeit

Wir haben den Korrektheitsbeweis noch nicht abgeschlossen. Wir haben lediglich gezeigt, dass der Algorithmus *partiell korrekt* ist. Das bedeutet, wir haben gezeigt, dass der Algorithmus das richtige Ergebnis liefert, falls er terminiert. Wir haben aber noch nicht gezeigt, dass er wirklich auf jeder Eingabe terminiert. Das erledigen wir nun implizit bei der Laufzeitabschätzung.

Theorem 41 *Für ganzzahlige Kapazitäten $c : V \times V \to \mathbb{N}_0$ ist die Anzahl an Iterationen der* **while***-Schleife im Algorithmus von Ford und Fulkerson durch $C = \sum_{e\in E} c(e)$ nach oben beschränkt. Die Laufzeit des Algorithmus ist $O(mC)$.*

Beweis. Man kann sehr einfach (z. B. mit Hilfe einer Invariante) zeigen, dass der aktuelle Fluss f, den der Algorithmus verwaltet stets ganzzahlig ist, dass also stets $f : V \times V \to \mathbb{N}_0$ gilt. Dies folgt einfach daraus, dass für einen ganzzahligen Fluss f und einen flussvergrößernden Weg P stets $\delta \in \mathbb{N}$ gilt. Demzufolge steigt der Wert des Flusses mit jeder Iteration der **while**-Schleife um mindestens eins an. Da der Wert eines jeden Flusses durch $\sum_{e\in E} c(e)$ beschränkt ist, garantiert dies die Terminierung nach höchstens C vielen Schritten.

Für einen gegebenen Fluss f kann das Restnetzwerk G_f in Zeit $O(m)$ berechnet werden. Ein Weg P von s nach t in G_f kann mittels Tiefensuche in Zeit $O(m)$ gefunden werden. Ist ein solcher Weg P gefunden, so kann in Zeit $O(m)$ der neue Fluss $f \uparrow P$ berechnet werden. Insgesamt dauert also jede Iteration Zeit $O(m)$, woraus das Theorem folgt. □

Aus diesem Theorem und seinem Beweis folgt direkt, dass es bei ganzzahligen Kapazitäten auch stets einen ganzzahligen maximalen Fluss gibt. Dies ist eine sehr wichtige Eigenschaft für viele Anwendungen und deswegen halten wir sie explizit als Korollar fest.

Corollary 42 *Sind alle Kapazitäten ganzzahlig, so gibt es stets einen ganzzahligen maximalen Fluss.*

7.3.2 Algorithmus von Edmonds und Karp

Die Laufzeit $O(mC)$, die wir in Theorem 41 bewiesen haben, ist nicht besonders gut, weil sie mit den Kapazitäten der Kanten wächst. Dies steht im Gegensatz zu den Laufzeiten von zum Beispiel den Algorithmen von Kruskal und Dijkstra. Für diese Algorithmen haben wir Laufzeitschranken bewiesen, die nur von der Größe des Graphen abhängen, nicht jedoch von den Gewichten oder Längen der Kanten. Eine Laufzeit wie $O(mC)$, die von den Kapazitäten abhängt, heißt *pseudopolynomielle Laufzeit*. Wir werden diesen Begriff im nächsten Semester weiter vertiefen. Hier sei nur angemerkt, dass man, wenn möglich, Laufzeiten anstrebt, die nur von der Größe des Graphen, nicht jedoch von den Kapazitäten abhängen, da diese sehr groß werden können. Wir werden nun den *Algorithmus von Edmonds und Karp* kennenlernen. Dabei handelt es sich um eine leichte Modifikation des Algorithmus von Ford und Fulkerson, deren Laufzeit nur von der Größe des Graphen abhängt.

Wir haben in der Beschreibung des Algorithmus von Ford und Fulkerson offen gelassen, wie die flussvergrößernden Wege gewählt werden, wenn es mehrere zur Auswahl gibt. Unsere Analyse von Korrektheit und Laufzeit war unabhängig von dieser Wahl. Wir zeigen nun, dass für eine geeignete Wahl die Laufzeit reduziert werden kann.

```
    EDMONDS-KARP(G, c, s ∈ V, t ∈ V)
1   Setze f(e) = 0 für alle e ∈ E.   // f ist gültiger Fluss mit Wert 0
2   while ∃ flussvergrößernder Weg P {
3       Wähle einen flussvergrößernden Weg P mit so wenig Kanten wie möglich.
4       Erhöhe den Fluss f entlang P.
5   }
6   return f;
```

Der einzige Unterschied zum Algorithmus von Ford und Fulkerson ist also, dass immer ein kürzester flussvergrößernder Weg gewählt wird, wobei die Länge eines Weges als die Anzahl seiner Kanten definiert ist. Wir werden in diesem Zusammenhang auch von der *Distanz von s zu $v \in V$ im Restnetzwerk G_f* sprechen. Damit meinen wir, wie viele Kanten der kürzeste Weg (d. h. der Weg mit den wenigsten Kanten) von s nach v in G_f hat.

Theorem 43 *Die Laufzeit des Algorithmus von Edmonds und Karp ist $O(m^2n) = O(n^5)$.*

Beweis. Eine Iteration der **while**-Schleife kann immer noch in Zeit $O(m)$ durchgeführt werden. Einen kürzesten flussvergrößernden Weg findet man nämlich zum Beispiel, indem man von s startend eine Breitensuche im Restnetzwerk durchführt. Diese benötigt Zeit $O(m)$. Zum Beweis des Theorems genügt es also zu zeigen, dass der Algorithmus nach $O(mn)$ Iterationen terminiert. Dazu zeigen wir zunächst die folgende strukturelle Eigenschaft über die Distanzen im Restnetzwerk.

Lemma 44 *Sei $x \in V$ beliebig. Die Distanz von s zu x in G_f wird im Laufe des Algorithmus nicht kleiner.*

Beweis. Wir betrachten eine Iteration der **while**-Schleife, in der der Fluss f entlang des Weges P zu dem Fluss $f \uparrow P$ erhöht wird. Die Kantenmenge $E_{f\uparrow P}$ unterscheidet sich von der Kantenmenge E_f wie folgt:

- Für jede Kante $(u, v) \in P \subseteq E_f$ verringert sich $\text{rest}_f(u, v)$ um δ, das heißt $\text{rest}_{f\uparrow P}(u, v) = \text{rest}_f(u, v) - \delta$. Eine Kante mit $\text{rest}_f(u, v) = \delta$ heißt *Flaschenhalskante* und sie ist in $E_{f\uparrow P}$ nicht mehr enthalten.

- Für jede Kante $(u, v) \in P \subseteq E_f$ erhöht sich die Restkapazität $\text{rest}_f(v, u)$ der entgegengesetzten Kante um δ, das heißt $\text{rest}_{f\uparrow P}(v, u) = \text{rest}_f(v, u) + \delta$. War $\text{rest}_f(v, u) = 0$, so war $(v, u) \notin E_f$, nun gilt aber $(v, u) \in E_{f\uparrow P}$.

Wir spalten den Übergang von E_f zu $E_{f\uparrow P}$ in mehrere Schritte auf. Zunächst fügen wir alle neuen Kanten nach und nach ein, die gemäß dem zweiten Aufzählungspunkt entstehen. Anschließend entfernen wir die Kanten nach und nach gemäß dem ersten Aufzählungspunkt, um die Kantenmenge $E_{f\uparrow P}$ zu erhalten. Nach jedem Einfügen und Löschen einer Kante untersuchen wir, wie sich die Distanz von s zu x geändert hat.

Als erstes betrachten wir den Fall, dass wir eine Kante (v, u) einfügen. Gemäß dem zweiten Aufzählungspunkt bedeutet das, dass die Kante (u, v) auf dem Weg P liegen muss. In dem Algorithmus wurde der Weg P so gewählt, dass er ein kürzester Weg von s nach t ist. Wegen der optimalen Substruktur, die wir uns für kürzeste Wege überlegt haben, bedeutet das insbesondere, dass die Teilwege von P, die von s nach u und v führen, kürzeste Wege von s nach u bzw. v sind. Die Länge des Teilweges nach u bezeichnen wir mit ℓ. Dann ist die Länge des Teilweges nach v genau $\ell + 1$. Somit verbindet die neue Kante einen Knoten mit Distanz $\ell + 1$ von s mit einem Knoten mit Distanz ℓ von s. Dadurch kann sich jedoch die Distanz von s zu keinem Knoten des Graphen verringern. Um die Distanz zu verringern, müsste die neue Kante einen Knoten mit Distanz k von s für ein $k \geq 0$ mit einem Knoten mit Distanz $k + i$ von s für ein $i \geq 2$ verbinden. Damit folgt, dass selbst nach dem Einfügen aller neuen Kanten der Knoten x immer noch die gleiche Distanz von s hat.

Es ist leicht einzusehen, dass das Löschen von Kanten keine Distanzen verringern kann. Deshalb hat der Knoten x nach dem Löschen mindestens die gleiche Distanz von s wie in G_f. □

Mit Hilfe von Lemma 44 können wir nun das Theorem beweisen. Bei jeder Iteration der **while**-Schleife gibt es mindestens eine Flaschenhalskante (u, v). Diese wird aus dem Restnetzwerk entfernt. Sei ℓ die Distanz von s zu u, bevor die Kante entfernt wird. Die Distanz zu v ist dann $\ell + 1$. Es ist möglich, dass diese Kante zu einem späteren Zeitpunkt wieder in das Restnetzwerk eingefügt wird, aber nur dann, wenn die Kante (v, u) in einer späteren Iteration auf dem gewählten flussvergrößernden P' liegt. Da sich die Distanz von s zu v nicht verringern kann, muss zu diesem Zeitpunkt die Distanz von s nach v immer noch mindestens $\ell + 1$ sein. Da P' ein kürzester Weg ist, können wir wegen der optimalen Substruktur wieder folgern, dass die Distanz von s zu u zu diesem Zeitpunkt mindestens $\ell + 2$ sein muss.

Zusammenfassend können wir also festhalten, dass das Entfernen und Wiedereinfügen einer Kante (u, v) im Restnetzwerk die Distanz von s zu u um mindestens 2 erhöht. Da die maximale Distanz $n - 1$ ist, kann eine Kante demnach nicht öfter als $n/2$ mal entfernt werden. In jeder Iteration wird mindestens eine Kante entfernt und es gibt $2m$ potentielle Kanten im Restnetzwerk. Damit ist die Anzahl an Iterationen der **while**-Schleife nach oben beschränkt durch $\frac{n}{2} \cdot 2m = nm$. □

Literatur

Flussprobleme werden in Kapitel 26 von [1] beschrieben. Außerdem sind sie in Kapitel 4.5 von [2] erklärt.

7.3.3 Der Heiratssatz

Flussmaximierung hat zahlreiche überraschende Anwendungen in anderen Gebieten. Als Beispiel betrachten wir den Satz von Hall, der auch als *Heiratssatz* bekannt ist. Gegeben sind eine Menge V_1 von Damen und eine Menge V_2 von Herren. Jede Dame $a \in V_1$ hat eine nicht leere Menge $F(a) \subseteq V_2$ von Freunden in V_2.

Die Mengen $F(a)$ brauchen nicht disjunkt zu sein. Deshalb stellt sich die Frage, ob jede Dame a einen ihrer Freunde in $F(a)$ heiraten kann, ohne daß es zu Polygamie kommt; gesucht wird also eine *injektive* Abbildung

$$h : V_1 \longrightarrow V_2$$

mit $h(a) \in F(a)$ für alle $a \in V_1$.

Eine (scheinbar) schwächere Forderung ist durch die sogenannte *Party-Bedingung* gegeben: Wenn eine Teilmenge von Damen alle ihre Freunde einlädt, gibt es genügend viele Tanzpartner (die aber keine Freunde zu sein brauchen). Das heißt, für jede Teilmenge $W \subseteq V_1$ gilt

$$|W| \;\leq\; |\bigcup_{a \in W} F(a)|$$

Überraschenderweise gilt nun folgendes

Theorem 45 *Eine Heirat h existiert genau dann, wenn die Party-Bedingung erfüllt ist.*

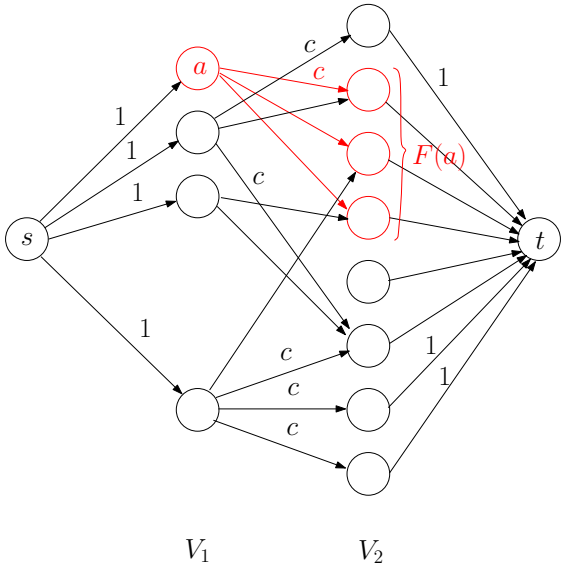


Abbildung 7.2: Die Kapazität c hat den Wert $|V_1| + 1$.

.

Beweis. Wenn eine Heirat h existiert, kann jede Dame a auf jeder Party mit ihrem Ehemann $h(a) \in F(a)$ tanzen; deshalb ist die Party-Bedingung erfüllt. Umgekehrt nehmen wir nun an, daß die Party-Bedingung erfüllt ist, und betrachten das in Abbildung 7.2 gezeigte Flussnetzwerk. Links stehen die Damen, rechts die Herren, und von jeder Dame gehen Kanten mit Kapazität $c := |V_1| + 1$ zu ihren Freunden aus. Quelle s ist mit allen Damen verbunden, und alle Herren sind mit der Senke verbunden; diese Kanten haben alle die Kapazität 1.

Behauptung: Die von s ausgehenden Kanten bilden einen minimalen Schnitt mit Kapazität $|V_1|$.

Zum Beweis nehmen wir an, es gäbe einen Schnitt mit Kapazität $< |V_1|$. Kanten von V_1 nach V_2 können darin nicht vorkommen, denn deren Kapazität c ist zu groß. Also kann der Schnitt nur Kanten von s zu einer Teilmenge V_1^0 von V_1 enthalten sowie Kanten von einer Teilmenge $V_2^0 \subseteq V_2$ nach t, und es gilt $|V_1^0| + |V_2^0| < |V_1|$.

Sei nun $a \in V_1 \setminus V_1^0$ und $w \in F(a)$. Dann muß w zu V_2^0 gehören, denn sonst gäbe es ja den Pfad $s \to a \to w \to t$ im Widerspruch zur Eigenschaft eines Schnitts. Das bedeutet

$$\bigcup_{a \in V_1 \setminus V_1^0} F(a) \;\subseteq\; V_2^0,$$

also

$$|\bigcup_{a \in V_1 \setminus V_1^0} F(a)| \;\leq\; |V_2^0| \;<\; |V_1| - |V_1^0| \;=\; |V_1 \setminus V_1^0|$$

im Widerspruch zur Party-Bedingung. Also ist die Behauptung richtig, und aus Theorem 40 folgt, daß es einen Fluss mit Wert $|V_1|$ geben muß. Dieser Fluss muß über $|V_1|$ *verschiedene* Knoten von V_2 zur Senke laufen und markiert deshalb eine Heirat. □

Literaturverzeichnis

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. MIT Press, 3. Auflage, 2009.

[2] Norbert Blum. Algorithmen und Datenstrukturen: Eine anwendungsorientierte Einführung. Oldenbourg Verlag, 2004.