Skriptum zur Vorlesung über Datenstrukturen

Hans Ulrich Simon und Christoph Ries
Fakultät für Mathematik
Ruhr-Universität Bochum

12. August 2019

Aufbau des Skriptums.

Dieses Skriptum besteht aus

Teil I: Algorithmen und *Teil II: Grundlagen*.

Es ist so konzipiert, dass beide Teile „parallel“ gelesen werden können. Bei einer Vorlesung mit 4 Wochenstunden, die sich auf zwei Sitzungen pro Woche verteilen, könnte eine Sitzung sich Teil I und die andere Teil II widmen.
In Teil I werden Algorithmen auf einer höheren und problemorientierten Abstraktionsebene entworfen. Man nutzt zum Beispiel aus, dass bestimmte Operationen (wie zum Beispiel INSERT, DELETE und FIND) auf Mengen mit maximal n Elementen jeweils in Zeit $O(\log n)$ ausgeführt werden können, ohne sich um die Implementierung der entsprechenden Datenstruktur[1] zu kümmern. Oder man nutzt aus, dass man in einer Liste in konstanter Zeit von einem Eintrag zum nächsten übergehen kann, ohne sich darum zu sorgen, wie eine Liste in einem realen Rechner dargestellt ist. Kurz: in Teil I bedienen wir uns der Module, die ein Designer von Datenstrukturen uns zur Verfügung stellen kann. Wir bohren diese Module nicht auf und sind daher befreit von der Komplexität, die in ihnen eingekapselt ist.
In Teil II nehmen wir die Perspektive des Designers ein, der diese vielen schönen Module und Werkzeuge entwerfen muss, welche in Teil I Verwendung finden. Zudem sollte kontrollierbar sein, welche Rechenzeit dabei auf einem realen Rechner beansprucht wird. Wir führen zu diesem Zweck die „Random Access Machine (RAM)” als mathematisches Modell für einen realen Rechner ein. Danach skizzieren wir, wie sich komplexere Konzepte (wie sie etwa aus höheren Programmiersprachen bekannt sind) sowie einfache und zusammengesetzte Datenstrukturen auf einer RAM darstellen lassen.
Während wir in Teil I aus der Vogelperspektive auf die Rechenprobleme schauen und von nickligen, maschinennahen Details abstrahieren dürfen, müssen wir uns in Teil II, ausgehend von dem Modell der RAM, vielen Detailfragen stellen und die einzelnen Module nach und nach zusammenschrauben.

Inhalt von Teil I.

Teil I dieses Skriptums ist stark angelehnt an die Kapitel 4–6 des Buches „Algorithm Design“ von Kleinberg und Tardos [3].

- Kapitel 4 dieses Buches widmet sich den sogenannten „greedy algo-

[1] Das wäre hier das „Wörterbuch“ bzw. der „dictionary“.

rithms“. Darunter (Zitat aus [5]) „versteht man Algorithmen, die eine Lösung eines Optimierungsproblems iterativ bestimmen und dabei zu jedem Zeitpunkt die derzeit bestmögliche Alternative wählen“. Wir werden sehen, dass der Schlüssel zu einem guten „greedy algorithm“ in einer geeigneten Präzisierung von „derzeit bestmöglich“ liegt (und natürlich ist der „greedy approach“ nicht für jedes Optimierungsproblem geeignet). Im Skriptum begegnen wir dem „greedy approach“ in den Kapiteln 1–5 und 9–11.

- Kapitel 5 des Buches von Kleinberg und Tardos behandelt die Methode „Divide and Conquer“: Aufteilen eines Problems in kleinere Teilprobleme vom selben Typ, (rekursives) Lösen der Teilprobleme und Kombination der Teillösungen zu einer Gesamtlösung. Im Skriptum begegnen wir dem „Divide and Conquer“-Ansatz in den Kapiteln 6 und 7.

- Kapitel 6 des Buches von Kleinberg und Tardos folgt dem Ansatz der „dynamischen Programmierung“. Es handelt sich um das „bottom-up“ Gegenstück zu „Divide and Conquer“: man beginnt bei den kleinsten Teilproblemen, tabelliert das Ergebnis und schreitet zu komplexeren Teilproblemen fort bis schließlich das Gesamtproblem gelöst ist. Durch die Tabellierung der Lösungen für die Teilprobleme vermeidet man, dass dasselbe Teilproblem mehrmals gelöst wird. Im Skriptum begegnen wir dem Ansatz mit „dynamischem Programmieren“ in Kapitel 8.

Kapitel 12 besteht lediglich aus einem Literaturhinweis zur „Berechnung starker Komponenten von Digraphen“ mit Hilfe einer auf Güting zurückgehenden Methode [2]. Es handelt sich um eine Anwendung von „Depth First Search“ — eine Grundlagentechnik, die in Kapitel 23 von Teil II des Skriptums besprochen wird. In Kapitel 13 werden Schlüsselvergleichsverfahren zum Sortieren besprochen. Wir lehnen uns dabei an die Ausführungen der entsprechenden Kapitel in [2] und [1] an. „Sortieren durch Fachverteilung“, angelehnt an Abschnitt 3.2 in [1] ist Gegenstand von Kapitel 14.

Inhalt von Teil II.

In Kapitel 15 wird die Laufzeit eines Algorithmus als Funktion in der Eingabelänge eingeführt und es wird begründet, warum wir uns hauptsächlich für die asymptotische Größenordnung dieser Funktion interessieren. Kapitel 16 führt die „Random Access Maschine (RAM)“ als das Referenzmodell für die

Zwecke der Vorlesung ein. Kapitel 17 und 18 demonstrieren, wie Konzepte aus höheren Programmiersprachen mit Hilfe der RAM simuliert werden können. Hieraus ergeben sich Faustregeln zur Abschätzung der Laufzeit auf einer (im Vergleich zur RAM) höheren (und bequemeren) Ebene der Abstraktion. Kapitel 19 führt Schleifeninvarianten als Werkzeug für Korrektheitsbeweise ein. Binäre Suchbäume (mit und ohne Rebalancierung) sind Gegenstand der Kapitel 20 und 21 (wobei in Kapitel 21 lediglich auf Abschnitt 4.2.4 in [2] verwiesen wird). Kapitel 23 behandelt „Depth-First-Search“ und „Breadth-First-Search“ als Basistechniken zur Graphexploration. Hashing ist Gegenstand des abschließenden Kapitels 24 (wo aber lediglich ein Literaturhinweis auf entsprechende Abschnitte in [2] und [4] gegeben wird).

Die im Skriptum verwendete Literatur. Wie aus unserer obigen Inhaltsangabe deutlich geworden ist, machen wir exzessiven Gebrauch von Standardwerken wie [3, 2, 1, 4]. Die Hauptunterschiede zu der verwendeten Literatur sind wie folgt:

- In Teil I werden (im Unterschied zu den entsprechenden Abschnitten in [3]) Implementierungsdetails erarbeitet mit dem Ziel, zu einer möglichst kleinen Zeitschranke (meist von der Form $O(n \log n)$) zu gelangen.
- Im Unterschied zu den meisten Standardwerken verwenden wir keinen Pseudocode, welcher möglichst ähnlich zu einer realen höheren Programmiersprache (und damit beinahe *maschinell* kompilier- oder interpretierbar) ist. Stattdessen wählen wir einen Sprachstil, der *menschlichen* Hörerinnen und Hörern das Verständnis erleichtern soll (und hoffen, dass uns dies gelungen ist).

Abschnitte, in denen die Orginalliteratur den gleichen Prinzipien folgt, haben wir nicht überarbeitet, sondern nur mit einem Literaturhinweis versehen.

Bochum, der 12. August 2019

Hans Ulrich Simon Christoph Ries

Inhaltsverzeichnis

Teil I

Algorithmen

Kapitel 1

Intervall–Scheduling

Wir imaginieren uns folgendes Szenario. Eine zentrale Ressource (wie zum Beispiel ein Superrechner) kann von mehreren Nutzern in Anspruch genommen werden. Zu diesem Zweck kann jeder potenzielle Nutzer eine Anfrage stellen, ob ihm die Ressource für ein bestimmtes Zeitintervall zugeteilt werden kann. Anfragen, deren Zeitintervalle sich überlappen, können nicht simultan befriedigt werden. Unter dieser Nebenbedingung soll eine maximale Anzahl von Anfragen positiv beschieden werden. Dies führt zu folgendem Problem mit dem Namen „Intervall–Scheduling“.

Eingabe: eine Kollektion $R = (R_j)_{j=1,\dots,n}$ von Intervallen der Form $R_j = [a_j, b_j)$ mit $0 \leq a_j < b_j$. R_j repräsentiert den Wunsch, die zentrale Ressource in der Zeitspanne von a_j (= Anfangszeit) bis b_j (= Endzeit) zu nutzen.[1]

Aufgabe: Auffinden einer möglichst großen Menge $I \subseteq [n]$ mit $R_i \cap R_j = \emptyset$ für alle $i \neq j \in I$. I repräsentiert eine möglichst große Teilkollektion sich paarweise nicht überlappender Intervalle.

Es stellen sich einige Fragen:

- Welche Intervalle der Kollektion wollen wir in I aufnehmen?
- Welche Parameter müssen wir protokollieren, um die Idee umzusetzen?

[1] Zum Zeitpunkt b_j steht sie bereits dem nächsten Nutzer zur Verfügung. Deswegen sind die Intervalle R_j halboffen, d.h., die Anfangszeit gehört noch zum Intervall, die Endzeit aber nicht.

- Wie arbeitet der resultierende Algorithmus?
- Wie beweisen wir seine Korrektheit und Optimalität?
- Wie gelangen wir vom Algorithmus zu einer möglichst zeiteffizienten Implementierung?

Am Anfang eines algorithmischen Verfahrens steht meist eine zündende

Idee: Wähle als nächstes Intervall stets dasjenige mit der kleinstmöglichen Endzeit aus.[2]

Um die Idee umzusetzen, müssen wir folgende Parameter protokollieren:

- die Menge $I \subseteq [n]$, welche die bereits ausgewählten Intervalle repräsentiert (anfangs $I = \emptyset$)
- die Menge $J \subseteq [n]$, welche die noch zu inspizierenden Intervalle repräsentiert (anfangs $J = [n]$)
- die Endzeit t des zuletzt ausgewählten Intervalles (anfangs $t = 0$)

Hier ist nun der resultierende Algorithmus, den wir im Folgenden mit A_1 bezeichnen:

1. $I \leftarrow \emptyset$; $J \leftarrow [n]$; $t \leftarrow 0$.
2. Solange $J \neq \emptyset$:
   (a) Wähle $j \in J$ mit minimalem Wert von b_j.
   (b) Falls $a_j \geq t$: $I \leftarrow I \cup \{j\}$; $t \leftarrow b_j$.
   (c) $J \leftarrow J \setminus \{j\}$.
3. Gib I aus.

Beachte, dass I, J, t dynamische Parameter sind, deren Werte sich zur Laufzeit von A_1 ständig ändern.
Wir illustrieren die Vorgehensweise von A_1 an einem Beispiel bzw. in Abbildung 1.1.

[2] In der Vorlesung werden zum Aufwärmen noch ein paar weitere Ideen verfolgt.

Beispiel 1.0.1 *Im oberen Teil von Abbildung 1.1 sind* 9 *Intervalle gegeben, Diese sind so durchnummeriert, dass ihre Endzeiten eine aufsteigende Folge bilden. Im Folgenden wird der Algorithmus A_1 auf diese Eingabedaten angewendet. Dabei sind in I aufgenommene Intervalle als Linien in Fettdruck dargestellt. Gestrichelte Linien repräsentieren Intervalle, die sich mit dem zuletzt in I aufgenommenen Intervall überlappen und daher für die Aufnahme in I disqualifizieren.*

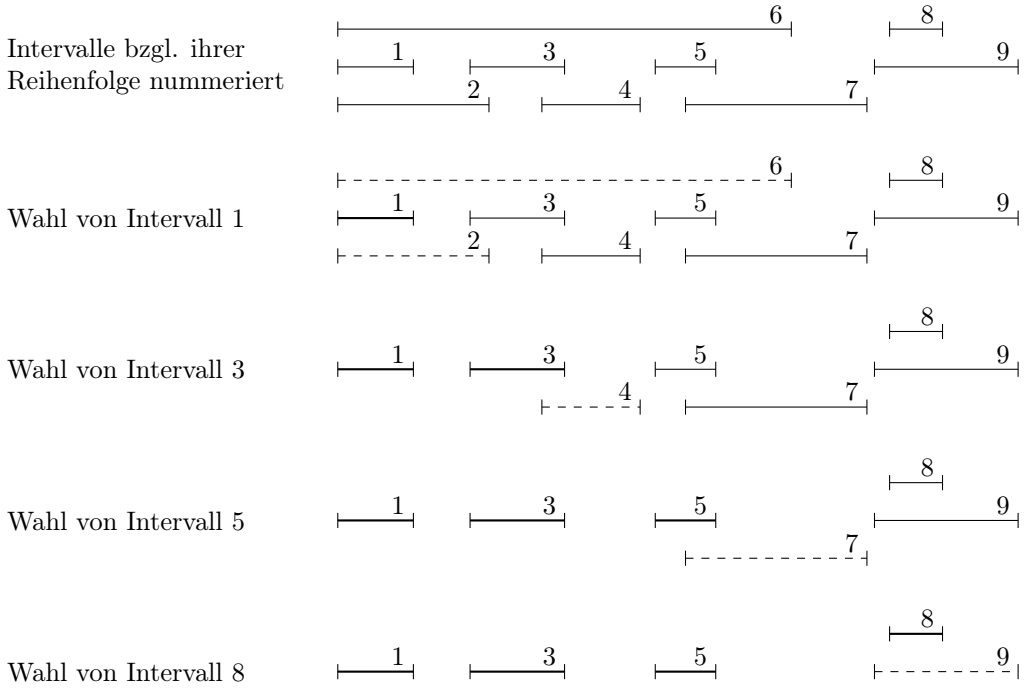


Abbildung 1.1: Beispiellauf des Algorithmus A_1.

Schritt 2(b) bewirkt, dass ein Intervall nur dann in I aufgenommen wird, wenn es sich mit den in I schon befindlichen Intervallen nicht überlappt. Daher ist die Ausgabe I korrekterweise eine überlappungsfreie Kollektion von Intervallen. Aber findet A_1 auch stets eine größte solche Kollektion? Die Antwort lautet „ja“:

Satz 1.0.2 *Der Algorithmus A_1 liefert zu einer gegebenen Kollektion von Intervallen eine größtmögliche Teilkollektion sich paarweise nicht überlappender Intervalle. D.h., A_1 liefert stets eine optimale Lösung zum Intervall–Scheduling Problem.*

Beweis Die zentrale Beobachtung (genannt „Schleifeninvariante“) zum Beweis des Satzes ist wie folgt:

Schleifeninvariante: Vor und nach jeder Iteration der Hauptschleife ist t der früheste Zeitpunkt, zu dem es $|I|$ überlappungsfreie Intervalle mit einer Endzeit von maximal t gibt.

In der Vorlesung wird der Beweis vervollständigt wie folgt:

- Wir zeigen, dass sich die Optimalität von A_1 unmittelbar aus der Schleifeninvariante ableiten lässt.

- Wir verifizieren die Schleifeninvariante mit Hilfe von vollständiger Induktion. Der Induktionsanfang erfolgt für den Zeitpunkt $t = 0$ (Zeitpunkt vor der ersten Iteration der Hauptschleife). Danach widmen wir uns dem Induktionsschritt und müssen nachweisen, dass die Schleifeninvariante *nach* einer Iteration der Hauptschleife gültig ist, sofern sie es *vor* dieser Iteration war.

Detailliertere Ausführungen hierzu gibt es mündlich in der Vorlesung. •

Korrektheitsbeweise mit Hilfe von Schleifeninvarianten bilden ein Thema, dass über die Dauer der Vorlesung „Datenstrukturen“ hinweg unser treuer Begleiter bleiben wird.

Bei der Implementierung spielt es eine Rolle, wie die vom Algorithmus A_1 manipulierten Objekte abgespeichert sind. Für die Ein- und Ausgabe ist Folgendes zweckmäßig:

- Die Intervallkollektion R ist durch zwei 1-dimensionale Arrays A und B gegeben: $A[j] = a_j$ ist die Anfangs- und $B[j] = b_j$ ist die Endzeit des j-ten Intervalles.

- Die von A_1 ausgewählte Teilkollektion $I \subseteq [n]$ wird als Liste ausgegeben (also die Liste $(i_1, \ldots, i_k)$ falls $I = \{i_1, \ldots, i_k\}$).

Wir merken kurz Folgendes an:

- 1-dimensionale Arrays sind, in der Sprache der Mathematik, Tupel. k-dimensionale Arrays sind eine naheliegende Verallgemeinerung.

- Auf die j-te Komponente eines 1-dimensionalen Arrays B, notiert als $B[j]$, kann über die „Adresse“ j in konstanter Zeit zugegriffen werden. Bei einem k-dimensionalen Array B erfolgt entsprechend ein Zugriff auf die Komponente $B[j_1, \ldots, j_k]$ mit Adresse $(j_1, \ldots, j_k)$ in konstanter Zeit. Man spricht von „Random Access“, weil die Zugriffszeit nicht von der Lage der Komponente (Anfang, Ende, Mitte des Arrays) abhängt.

- Listen sind, in der Sprache der Mathematik, (endliche) Folgen.

- Listen werden i.A. sequentiell durchmustert. Man kann in konstanter Zeit auf das erste (oder letzte) Element der Liste zugreifen. Ebenso kann man, von einem gegebenen Element der Liste ausgehend, in konstanter Zeit auf das folgende (oder, im Falle von sogenannten doppeltverketteten Listen, auf das vorhergehende) Element zugreifen. Man benötigt daher i.A. Linearzeit, um auf ein Element aus einer Liste (wie zum Beispiel das mittlere Element) zuzugreifen.

- Eine Liste kann in konstanter Zeit um einen Eintrag (am Anfang oder am Ende der Liste) erweitert oder verkürzt werden.

- Komponenten eines Arrays oder einer Liste können selber wieder strukturierte Objeke (wie zum Beispiel Arrays oder Listen) sein.

Darüber hinausgehende Einzelheiten zu Arrays und Listen werden in Teil II des Skriptums besprochen.

Kommen wir schließlich zum Thema der Implementierung unseres Algorithmus A_1 für Intervall–Scheduling. A_1 manipuliert die Mengen I und J. Die entscheidende Frage ist, welche Operationen auf diesen Mengen unterstützt werden müssen:

Phase 1: In J müssen n Elemente (unter Beachtung ihrer Schlüsselwerte b_j) eingefügt werden: n-malige Anwendung einer Operation vom Typ INSERT.

Phase 2a: In J muss auf ein Element j mit minimalem Schlüsselwert b_j zugegriffen werden: 1-malige Anwendung einer Operation vom Typ MIN.

Phase 2b: In I muss ein Element eingefügt werden: 1-malige Anwendung von INSERT.

Phase 2c: Aus J muss das (vorher ausgewählte) Element mit minimalem Schlüsselwert entfernt werden: 1-malige Anwendung einer Operation vom Typ DELETE_MIN.

Da die Hauptschleife (= Phase 2) n-mal durchlaufen wird, können wir folgendes Résumée ziehen (wobei uns die Rechenzeit nur asymptotisch im Sinne der O-Notation interessiert):

- Für die Menge I muss ausschließlich die Operation INSERT unterstützt werden. Dies leistet die Datenstruktur „Liste“. Iteratives Einfügen von bis zu n Elementen in eine anfangs leere Liste kann in Zeit $O(n)$ geschehen.

- Für die Menge J müssen die Operationen INSERT, DELETE_MIN und MIN unterstützt werden. Dies leistet eine Datenstruktur namens „Heap“, die wir in Teil II des Skriptums detailliert kennenlernen werden. Bei einem Heap der Größe n kann jede der oben genannten Operationen in $\log n$ Schritten durchgeführt werden. n-malige Anwendung dieser Operationen kann also in Gesamtzeit $O(n \log n)$ geschehen.

Wir merken kurz an, dass die benötigte Laufzeit für Anweisungen wie $I \leftarrow \emptyset$ (= Initialisieren einer leeren Liste), $t \leftarrow b_j$ (= Zuweisung des Wertes $b_j = B[j]$ an die Variable t) oder die Abfrage „$J \neq \emptyset$?“ (= Abfrage, ob der Heap J leer ist) vernachlässigbar ist im Vergleich zu der Rechenzeit, welche für die Manipulationen der Menge J aufgewendet wird. Mit anderen Worten: die Laufzeit wird durch die Manipulationen der Menge J (vermöge der Operationen INSERT, MIN und DELETE_MIN) dominiert. Wir gelangen also zu folgendem Ergebnis:

Satz 1.0.3 *Bei geeigneter Implementierung hat der Algorithmus A_1 eine Laufzeit der Größenordnung $O(n \log n)$.*

Kapitel 2

Intervall–Färbungsproblem

Wie schon beim Intervall–Scheduling Problem betrachten wir auch hier ein Szenario mit Nutzern, die eine zentrale Ressource für ein bestimmtes Zeitintervall zugeteilt bekommen möchten. Diesmal verfügen wir jedoch über mehrere Duplikate der zentralen Ressource, sagen wir über identisch gebaute „Maschinen“ $M_1, M_2, M_3, \ldots$. Es können daher mehrere Nutzer gleichzeitig bedient werden. Natürlich gibt es weiterhin die Nebenbedingung, dass wir zwei Nutzern mit überlappenden Zeitintervallen nicht dieselbe Maschine M_i zuordnen können. Unter dieser Nebenbedingung stellen wir uns das Ziel, mit einer kleinstmöglichen Anzahl von Maschinen, jedem Nutzer eine Maschine für das von ihm gewünschte Zeitintervall zur Verfügung zu stellen. Dies führt zu folgendem Problem mit dem Namen „Intervall–Färbungsproblem“.

Eingabe: eine Kollektion $R = (R_j)_{j=1,\ldots,n}$ von Intervallen der Form $R_j = [a_j, b_j)$ mit $0 \leq a_j < b_j$. R_j repräsentiert den Wunsch, eine der Maschinen in der Zeitspanne von a_j (= Anfangszeit) bis b_j (= Endzeit) zu nutzen.

Aufgabe: Auffinden einer Abbildung $f : [n] \to [k]$ mit einem möglichst kleinen Wert von k, so dass die Bedingung

$$\forall 1 \leq i < j \leq n : R_i \cap R_j \neq \emptyset \Rightarrow f(i) \neq f(j) \tag{2.1}$$

erfüllt ist. Natürlich interpretieren wir hier $f(j)$ als die Nummer der Maschine, welche wir dem Nutzer j im Zeitintervall R_j zur Verfügung stellen.

Für diejenigen, die das Graphenfärbungsproblem kennen, stellen wir die folgende

Denksportaufgabe: Versuche, durch geeignete Wahl eines Graphen, das Intervall-Färbungsproblem als Spezialfall des Graphenfärbungsproblems darzustellen.

Wir definieren $d_R(t)$ als die Anzahl der Intervalle, welche t enthalten und setzen $d(R)$ auf den größtmöglichen dieser Werte, d.h.:

$$d_R(t) = |\{j \in [n] : t \in R_j\}| \text{ und } d(R) = \max_t d_R(t) \ .$$

Es ist offensichtlich, dass zum Zeitpunkt t mindestens $d_R(t)$ Maschinen benötigt werden. Somit werden für die Lösung des Intervall–Färbungsproblems mindestens $d(R)$ (und natürlich höchstens n) Maschinen benötigt. Der Algorithmus, den wir im Folgenden entwerfen, beruht auf folgender

Idee: Wähle als nächstes Intervall j stets dasjenige mit der kleinstmöglichen Anfangszeit a_j aus. Weise dann $f(j)$ die kleinste Nummer i zu mit der Eigenschaft: M_i ist zum Zeitpunkt a_j „frei“ (d.h., nicht aktuell in einem anderen Zeitintervall aktiv).

Die Umsetzung dieser Idee wird in folgendem Beispiel bzw. in Abbildung 2.1 illustriert.

Beispiel 2.0.1 *Im (a)-Teil der Abbildung ist eine Kollektion R bestehend aus* 10 *Intervallen gegeben. Diese sind so durchnummeriert, dass ihre Anfangszeiten eine aufsteigende Folge bilden. Offensichtlich gilt $d(R) = 3$. Im (b)-Teil der Abbildung werden diese Intervalle (gemäß der oben formulierten Idee)* 3 *Maschinen überlappungsfrei zugeordnet.*

Es bezeichne $U \subseteq [n]$ die Menge der Nummern der aktuell freien Maschinen. Um obige Idee umzusetzen, müssen wir die Menge U verwalten und stets auf dem aktuellen Stand halten. Anfangs sind alle Maschinen frei, d.h., $U = [n]$. Die kritischen Zeitpunkte, zu denen U sich ändert, sind die Anfangs- und Endzeiten der Intervalle R_j:

(a) Zum Zeitpunkt a_j wird die kleinste Nummer in U aus U entfernt (da die betreffende Maschine dem Nutzer j zur Verfügung gestellt wird).

(b) Zum Zeitpunkt b_j wird die im Zeitintervall R_j aktive Maschine $M_{f(j)}$ wieder frei und somit wird $f(j)$ in U eingefügt.

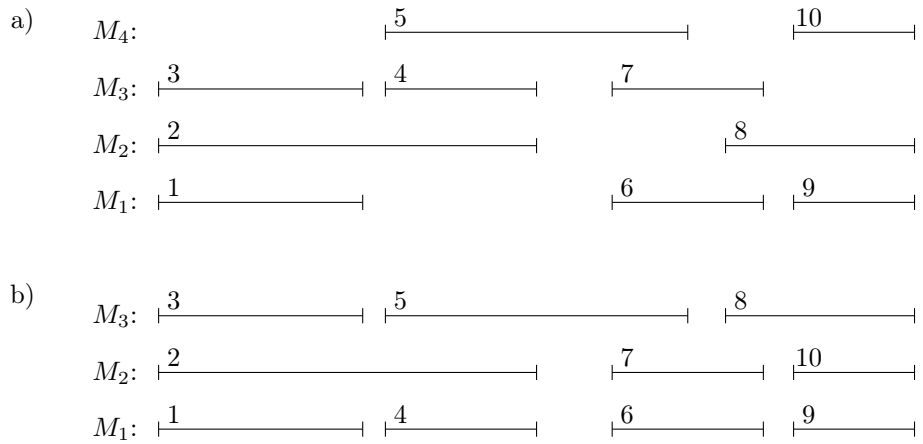


Abbildung 2.1: Eine 3-Färbung von 10 gegebenen Intervallen.

(c) Wenn t die Endzeit von R_j und zugleich die Anfangszeit von $R_{j'}$ ist (also $t = b_j = a_{j'}$), dann sollten wir die Aktualisierung vom Typ (b) für Index j durchführen bevor wir die Aktualisierung vom Typ (a) für den Index j' in Angriff nehmen, da die Maschine $M_{f(j)}$ zum Zeitpunkt b_j schon wieder für andere Maschinen genutzt werden kann.

Um die Menge U auf dem aktuellen Stand zu halten, müssen wir die Tripel $(j, a_j, 1)$ und $(j, b_j, 0)$ verwalten. Die zweite Komponente gibt den kritischen Zeitpunkt an. Das Indikatorbit in der dritten Komponente zeigt an, ob die zweite Komponente eine Anfangs- oder eine Endzeit ist. Ein Tripel $(j, t, \beta) \in \{(j, a_j, 1), (j, b_j, 0)\}$ ist zum Zeitpunkt t „erledigt“, falls $b_j < t$. Es bezeichne J die Menge der aktuell noch nicht erledigten Tripel. Wir verwenden für das Tripel (j, t, β) das Paar (t, β) als Schlüsselwert und verwenden die sogenannte lexikographische Ordnung.[1]
Wir fassen zusammen:

- Die Menge $U \subseteq [n]$ repräsentiert die Menge der aktuell freien Maschinen.
- die Menge J bezeichnet die Menge der aktuell noch nicht erledigten Tripel.

[1]Die β-Komponente wird beim Vergleich zweier Schlüsselwerte nur benötigt, falls die t-Komponenten übereinstimmen. In diesem Fall ist der Schlüsselwert $(t, 0)$ kleiner als $(t, 1)$. Die Wörter in einem Lexikon sind auf ähnliche Weise sortiert. Zum Beispiel finden wir das Wort „Aal“ vor dem Wort „Aas“, und „Aas“ vor „Abtei“.

Hier ist nun der resultierende Algorithmus, den wir im Folgenden mit A_2 bezeichnen:

1. $U \leftarrow [n]$; $J \leftarrow \{(j, a_j, 1) | j \in [n]\} \cup \{(j, b_j, 0) | j \in [n]\}$.
2. Solange $J \neq \emptyset$:
   (a) Wähle $(j, t, \beta) \in J$ mit minimalem Wert von (t, β).
   (b) Falls $\beta = 0$ (und somit $t = b_j$): $U \leftarrow U \cup \{f(j)\}$.
   Falls $\beta = 1$ (und somit $t = a_j$): $k \leftarrow \min U; f(j) \leftarrow k; U \leftarrow U \backslash \{k\}$.
   (c) $J \leftarrow J \setminus \{(j, t, \beta)\}$.
3. Gib f aus.

Die Aktualisierungen in der Hauptschleife (Phase 2) bewirken, dass U (als Menge der aktuell freien Maschinen) und J (als Menge der noch nicht erledigten Tripel) korrekt aktualisiert werden. Hieraus ergibt sich relativ leicht der

Satz 2.0.2 *A_2 konstruiert eine Abbildung f, welche die Nebenbedingung (2.1) erfüllt und dabei nur die Nummern von 1 bis $d(R)$ verwendet. Mit anderen Worten: A_2 ist ein optimaler Algorithmus für das Intervall–Färbungsproblem.*

Beweis Wir geben hier nur die Schleifeninvariante an:
Es bezeichne R' die Teilkollektion von R bestehend aus den aktuell (nach der letzten Iteration durch die Hauptschleife) bereits inspizierten Intervallen. Es sei (j, t, β) das in der letzten Iteration der Hauptschleife gewählte Tripel aus J. Dann sind die Maschinen mit den Nummern $d_{R'}(t) + 1, \ldots, n$ frei, d.h., $d_{R'}(t) + 1, \ldots, n \in U$.
Hieraus folgt direkt, dass die Abbildung f die Nummern $d(R) + 1, \ldots, n$ niemals einsetzt. In der Vorlesung werden wir die obige Schleifeninvariante mit Hilfe von vollständiger Induktion verifizieren. •

Es folgen ein paar Implementierungsdetails. Die Eingabedaten für A_2 sind (wie schon beim Algorithnus A_1 für Intervall–Scheduling) gegeben durch die Arrays $A[1 : n]$ und $B[1 : n]$ mit $A[j] = a_j$ und $B[j] = b_j$ für $j = 1, \ldots, n$. Die Ausgabe, also die Abbildung $f : [n] \to [n]$, kann auch als Array $F[1 : n]$ mit $F[j] = f(j)$ dargestellt werden.

Es ist leicht zu sehen, dass die Laufzeit dominiert wird durch die Operationen auf den Mengen U und J in den Phasen 1 und 2. Alle benötigten Mengenoperationen sind vom Typ INSERT, MIN oder DELETE_MIN. Wenn wir die Mengen U und J als Heap[2] (mit den jeweiligen Schlüsselwerten) implementieren, dann kann jede einzelne der genannten Operationen in $O(\log n)$ Schritten ausgeführt werden. In Phase 1 erfolgen bei der Initialisierung von U und J jeweils n Operationen vom Typ INSERT. Die Hauptschleife (Phase 2) wird insgesamt n-mal durchlaufen (einmal pro Tripel in J). In jeder Iteration werden nur konstant viele Mengenoperationen ausgeführt. Insgesamt werden also in beiden Phasen nur $O(n)$ Mengenoperationen ausgeführt. Wir gelangen daher zu folgendem Ergebnis:

Satz 2.0.3 *Bei geeigneter Implementierung hat der Algorithmus A_2 eine Laufzeit der Größenordnung $O(n \log n)$.*

Durch unsere Analyse hat sich herausgestellt, dass die von A_2 konstruierte Abbildung $f : [n] \to [n]$ die Nummern $d(R) + 1, \ldots, n$ gar nicht verwendet. Daher initialisiert A_2 in Phase 1 mit dem Kommando $U \leftarrow [n]$ einen unnötig großen Heap. Es wäre eine gute Übung, sich zu überlegen, dass eine raffiniertere Implementierung (allerdings mit derselben asymptotischen Laufzeitschranke) mit einem Heap U der Größe $d(R)$ auskommt.

Wir merken kurz an, dass es eine Alternative zu der geschilderten Implementierung gibt, welche ohne die Menge J auskommt und stattdessen eine Sortierung der Tripel (j, t, β) vornimmt, wobei wieder das Paar (t, β) als Schlüsselwert dient.[3] In der Hauptschleife (Phase 2) wird dann einfach die sortierte Liste abgearbeitet.

[2] die Datenstruktur, die wir bereits beim Algorithmus für Intervall–Scheduling eingesetzt hatten

[3] Randnotiz für Kenner der Szene: falls zum Sortieren der Algorithmus HEAPSORT verwendet wird, dann ist das alternative Verfahren zu unserem Algorithmus wieder sehr ähnlich.

Kapitel 3

Minimierung von Verspätungen

Wir haben die Aufgabe n (Rechen-)Jobs $J_1, \ldots, J_n$ mit Hilfe eines Prozessors zu erledigen. Wir identifizieren einen Job mit seiner Nummer und nennen daher J_i kurz den Job i. Jeder Job i hat eine (uns bekannte) Länge (= Ausführungsdauer) t_i und einen Schlusstermin d_i, zu dem er fertiggestellt sein sollte. Wenn wir ihn zur Zeit s_i auf dem Prozessor starten, ist er zur Zeit $f_i = s_i + t_i$ fertiggestellt. Wir bezeichnen dann $\ell_i = \max\{0, f_i - d_i\}$ als seine *Verspätung* (also Verspätung 0, falls der Job bis zu seinem Schlusstermin fertiggestellt ist). Da wir nur einen Prozessor zur Verfügung haben, müssen die Startzeiten so gewählt sein, dass für alle Wahlen von $1 \leq i < j \leq$ die Intervalle $[s_i, f_i)$ und $[s_j, f_j)$ disjunkt sind. Wir suchen eine Zuordnung von Jobs zu Startzeiten, welche unter dieser Nebenbedingung die größte auftretende Verspätung minimiert. Dies führt zu folgendem Scheduling–Problem:

Eingabe: Parameter $t_1, \ldots, t_n$ und $d_1, \ldots, d_n$ (dargestellt als Arrays)

Aufgabe: Auffinden einer Startzeitsequenz $s_1, \ldots, s_n \geq 0$ (dargestellt als Array), so dass mit der Setzung $f_i = s_i + t_i$ und $\ell_i = \max\{0, f_i - d_i\}$ die Bedingung

$$\forall 1 \leq i < j \leq n : [s_i, f_i) \cap [s_j, f_j) = \emptyset$$

erfüllt ist und, unter dieser Nebenbedingung, die maximale Verspätung $\max_{i \in [n]} \ell_i$ möglichst klein ist.

Es hat offensichtlich keinen Sinn, im Einsatzplan für den Prozessor „Lücken“ zu lassen, in denen er keine Arbeit hat, obwohl noch nicht alle Jobs erledigt sind. Daher ist das Kernproblem das Auffinden einer geeigneten Permutation

(= Umordnung) $\sigma = (\sigma(1), \ldots \sigma(n))$ von $(1, \ldots, n)$ mit der Festlegung, dass die Jobs in der durch σ gegebenen Reihenfolge lückenlos dem Prozessor zum Fraß vorgeworfen werden. Dies führt zu den Startzeiten

$$s_{\sigma(i)} = \sum_{j=1}^{i-1} t_{\sigma(j)} \ , \tag{3.1}$$

für $i = 1, \ldots, n$ d.h., der Job $\sigma(i)$ wird gestartet, sowie die Jobs $\sigma(1), \ldots, \sigma(i-1)$ fertiggestellt sind.[1] Es ist einfach zu sehen (Übung), dass die n Startzeiten gemäß (3.1) in Zeit $O(n)$ bestimmt werden können. Wir können uns im Folgenden auf die Frage konzentrieren, was die für unsere Zwecke beste Umordnung σ der Jobs $1, \ldots, n$ ist. (Eine natürliche Datenstruktur zur Angabe von σ wäre die Liste.)

Es wird sich zeigen, dass es optimal ist, eine Umordnung nach wachsenden Schlussterminen vorzunehmen.[2]

Idee: Sortiere die Jobs $1, \ldots, n$ aufsteigend bezüglich der Schlüsselwerte $d_1, \ldots, d_n$ (eine Strategie, die unter dem Namen „Earliest Deadline First“ bekannt ist).

Die sortierte Sequenz ist dann die gesuchte Umordnung.

Wie früher schon mal erwähnt, können n Objekte in Zeit $O(n \log n)$ sortiert werden. Der aus unseren Überlegungen resultierende Algorithmus, im Folgenden als A_3 bezeichnet, lässt sich leicht angeben:

1. Sortiere die Jobs nach wachsenden Schlüsselwerten. Es bezeichne $\sigma = (\sigma(1), \ldots, \sigma(n))$ die sortierte Sequenz.

2. Berechne aus σ die resultierende Startzeitsequenz gemäß (3.1).

Es ist klar, dass Phase 1 Rechenzeit $O(n \log n)$ und Phase 2 Rechenzeit $O(n)$ erfordert. Die Gesamtlaufzeit beträgt daher $O(n \log n)$. Es bleibt nur noch zu zeigen, dass A_3 ein optimaler Algorithmus ist, und dann ergibt sich der

[1] Im Falle $i = 1$ ist die Gleichung (3.1) so zu interpretieren, dass Job $\sigma(1)$ zum Zeitpunkt $s_{\sigma(1)} = 0$ gestartet wird. (Eine Summe mit 0 Summanden liefert den Wert 0.)

[2] In der Vorlesung betrachten wir zum Aufwärmen noch ein paar weitere (suboptimale) Umordnungskriterien.

Satz 3.0.1 *Algorithmus A_3 löst (in $O(n \log n)$ Rechenschritten) das in diesem Abschnitt vorgestellte Scheduling-Problem optimal, d.h. die von A_3 gewählten Startzeiten führen dazu, dass die größte auftretende Verspätung so klein wie möglich ist.*

Der Beweis erfolgt in der Vorlesung.

Kapitel 4

Minimierung von Cache-Fehlern

Reale Computer verfügen über eine Speicherhierarchie (die wir mündlich in der Vorlesung noch etwas detallierter besprechen). In diesem Abschnitt betrachten wir lediglich zwei aufeinander folgende Ebenen der Hierarchie:

Ebene 1: ein Cache-Speicher der Größe k mit einer sehr kurzen Zugriffszeit

Ebene 2: ein Speicher der Größe $N >> k$ mit einer vergleichsweise langen Zugriffszeit.

Der große Speicher ist in Blöcke (auch „Seiten“ genannt) unterteilt. Ein kleiner Teil der Blöcke kann im Cache eingelagert werden. Wenn ein Rechenprozess einen eingelagerten Block benötigt, so steht dieser mehr oder weniger sofort zur Verfügung. Ist der Block jedoch ausgelagert, so spricht man von einem „Seiten-“ oder „Cache-Fehler“ und muss einen zeitraubenden Transfer dieses Blockes in den Cache veranlassen. Wegen der begrenzten Kapazität des Cache muss im Gegenzug ein anderer Block aus dem Cache ausgewählt und vertrieben werden. Die Kunst besteht darin, die Auswahlentscheidungen so zu treffen, dass auf lange Sicht die Anzahl der Cache-Fehler minimiert wird. Zum Beispiel wäre es töricht, einen Block aus dem Cache zu vertreiben, der in naher Zukunft bereits wieder benötigt wird. In dem folgenden „Cache-Fehler-Minimierungsproblem“ identifizieren wir die Blöcke mit Zahlen. Weiterhin tun wir so, als könnten wir in die Zukunft schauen und wüssten, welche Blöcke in den nächsten m Phasen des laufenden Rechenprozesses benötigt werden. Gegen Ende des Abschnittes sprechen wir kurz eine Heuristik an,

die in der Praxis oft verwendet wird und die ohne die Kenntnis der Zukunft auskommt.

Eingabe: Eine natürliche Zahl N, eine Menge $K \subseteq [N]$ der Größe $k \leq N$ und eine Sequenz $\mathbf{a} = (a_1, \ldots, a_m) \in [N]^m$.

Aufgabe: Treffe die Auswahlentscheidungen in folgendem Szenario so, dass die Gesamtanzahl an Cache-Fehlern so klein wie möglich ist.

Szenario: Für $i = 1, \ldots, m$ mache Folgendes:

Runde i: Falls $a_i \notin K$ („Cache-Fehler“): Wähle ein $b \in K$ („Auswahlentscheidung“) und setze $K \leftarrow (K \setminus \{b\}) \cup \{a_i\}$, d.h. vertreibe b aus K und nimm a_i in K auf.

Zu einer Rundenzahl $i \in [m]$ und einem Element $b \in [N]$ sei $j_i(b)$ der kleinste Index $j \in \{i, \ldots, m\}$ mit $b = a_j$. Falls jedoch b in der Restfolge $a_i, \ldots, a_m$ gar nicht mehr vorkommt, so sei $j_i(b) = m + 1$. Dieser Index zeigt an, ab wann wir b wieder im Cache K haben sollten, um einen Cache-Fehler zu vermeiden (und im Falle $j_i(b) = m+1$ kann b gar keine Cache-Fehler mehr verursachen). Es ist eine naheliegende Idee, in erster Linie Elemente b aus dem Cache zu vertreiben, deren Indexwert $j_i(b)$ groß ist. Eine Präzisierung dieses Gedankens führt zu der

FF-Strategie: Falls $a_i \notin K$, dann vertreibe das Element b aus K, das den größten Index $j_i(b)$ hat.

„FF“ steht für „Farthest-in-Future“. Die FF-Strategie notieren wir im Folgenden als S_{FF}. Beachte, dass (abgesehen von Elementen b, die in $a_i, \ldots, a_m$ gar nicht mehr vorkommen), verschiedene Elemente aus $[N]$ verschiedene j_i-Indizes zugeordnet bekommen. Daher ist die FF-Strategie (im Wesentlichen) eindeutig.

Lemma 4.0.1 *Es sei $j \in \{0, 1, \ldots, m-1\}$. Es sei S eine Strategie, die bei Eingabe $N, K, \mathbf{a}$ in den ersten j Runden mit S_{FF} übereinstimmt. Dann kann S in eine Strategie S' umgewandelt werden, die 1. nicht mehr Cache-Fehler produziert als S und 2. in den ersten $j+1$ Runden mit S_{FF} übereinstimmt.*

Beweis Wenn in Runde $j+1$ kein Cache-Fehler vorliegt, stimmt S in dieser Runde zwangsläufig mit S_{FF} überein (da gar keine Auswahlentscheidung zu

treffen ist). Wir dürfen daher annehmen, dass in Runde $j+1$ ein Cache-Fehler vorliegt, dass S sich für die Vertreibung von a und S_{FF} sowie S' sich für die Vertreibung von $b \neq a$ aus dem Cache entscheiden. Wir dürfen weiterhin annehmen, dass $m \geq j+2$. Aus der Definition von S_{FF} folgt, dass b in der Restsequenz $a_{i+2}, \ldots, a_m$ nicht früher auftauchen kann als a. Im Folgenden wird S' bemüht sein, S zu simulieren, ohne dabei mehr Cache-Fehler als S zu produzieren. Die Simulation von S durch S' in den Runden $i = j+2, \ldots, m$ verläuft in mehreren Phasen. Wir beginnen bei Phase 1:

- Falls $a_i \notin \{a, b\}$ und S' (im Falle eines Cache-Fehlers) nicht ausgerechnet b aus dem Cache vertreibt, dann macht S' in Runde i dasselbe wie S.

- Falls $a_i \notin \{a, b\}$ einen Cache-Fehler produziert und S das Element b aus dem Cache vertreibt, dann vertreibt S' das Element a aus dem Cache. Danach stimmen die Caches von S und S' überein und es erfolgt ein direkter Übergang in die Phase 3 der Simulation.

- Falls $a_i = a$, dann liegt bei S (nicht aber bei S') ein Cache-Fehler vor. Es sei c das Element, das von S aus dem Cache vertrieben wird. Es erfolgt ein Übergang in die Phase 2 der Simulation.

Die Phase 2 ist dadurch gekennzeichnet, dass die Caches von S und S' bis auf zwei Elemente b, c übereinstimmen und dass S im Vergleich zu S' einen Cache-Fehler mehr auf dem Konto hat. Die Simulation in Phase 2 verläuft wie die in Phase 1 (mit c in der Rolle von a), bis auf die folgende Ausnahme:

- Falls $a_i = b$, dann liegt bei S' (nicht aber bei S) ein Cache-Fehler vor. Nun vertreibt S' das Element c aus dem Cache. Danach haben beide Strategien wieder die gleiche Anzahl an Cache-Fehlern und ihre Caches stimmen überein. Es erfolgt ein Übergang in die Phase 3.

Bei Eintritt in die Phase 3 gilt: S und S' haben aktuell die gleiche Anzahl an Cache-Fehlern und ihre Caches stimmen überein. In Phase 3 macht S' stets dasselbe wie S.
Auf diese Weise kann die Simulation bis zur Runde m aufrecht erhalten werden, und der Beweis des Lemma ist komplett. •

Lemma 4.0.1 besagt, dass wir eine beliebige Strategie S nach und nach in S_{FF} transformieren können, ohne dabei die Anzahl der Cache-Fehler zu

erhöhen. Da dies insbesondere auf eine optimale Strategie (mit einer minimalen Anzahl von Cache-Fehlern) zutrifft, ergibt sich unmittelbar die

Folgerung 4.0.2 *Die FF-Strategie ist optimal, d.h., sie produziert auf jeder Eingabeinstanz des Cache-Fehler-Minimierungsproblems die kleinstmögliche Anzahl von Cache-Fehlern.*

Der Rest dieses Abschnittes ist Implementierungsdetails gewidmet. Es sei M eine Menge der Maximalgröße n, deren Elemente mit eindeutigen Schlüsselwerten versehen sind. Wir werden später eine Datenstruktur D kennenlernen, die es erlaubt, jede der folgenden Operationen in Zeit $O(\log n)$ durchzuführen:[1]

MEMBER: MEMBER(a) liefert den Wahrheitswert TRUE genau dann, wenn M ein Element mit Schlüsselwert a enthält.

FIND: FIND(a) liefert uns das Element aus M mit dem Schlüsselwert a (sofern vorhanden).

INSERT: INSERT(x, a) fügt das Element x mit Schlüsselwert a in die Datenstruktur D ein (sofern nicht schon vorhanden).

DELETE: DELETE(a) entfernt das Element mit Schlüsselwert a aus D (sofern vorhanden).

MAX: MAX liefert das Element aus M mit dem maximalen Schlüsselwert.

CHOOSE: CHOOSE liefert irgendein Element aus M (sofern M nicht leer ist).

Die ersten vier Operationen heißen auch „Wörterbuch-“ oder „Dictionary-Operationen“. Die meisten Implementierungen für Wörterbücher (wie zum Beispiel diverse Arten balancierter Bäume) unterstützen neben diesen vier Operationen auch die Operationen MAX und CHOOSE.

Mit einer Eingabeinstanz $(N, K, \mathbf{a})$ mit $K = \{b_1, \ldots, b_k\}$ und $\mathbf{a} = (a_1, \ldots, a_m)$ und einer Rundenzahl i verbinden wir „Informationspakete“ der Form (β, j, b, L), wobei Folgendes gilt:

[1] Falls die Menge M Objekte nicht konstanter Größe enthält, wie zum Beispiel ein Array oder eine Liste, dann würde man in D aus Effizienzgründen nur die Basisadresse des Arrays oder des Listenkopfes eintragen.

1. $b \in K \cup \{a_i, \ldots, a_m\}$.
2. $j = j_i(b)$, d.h. j verrät uns das erste Vorkommen von b in der Folge $a_i, \ldots, a_m$. Es gilt $j = m + 1$, wenn b in dieser Folge nicht vorkommt.
3. $\beta \in \{0, 1\}$ ist ein Indikatorbit mit $\beta = 1$ genau dann, wenn $b \in K$.
4. L ist die geordnete Liste zur Menge $I(b) = \{j \in \{j_i(b)+1, \ldots, m\} | a_j = b\}$, d.h. L verrät uns die Vorkommen des Elementes b in der Folge $a_{j_i(b)+1}, \ldots, a_m$.

Wir betrachten (β, j, b) als den (lexikographischen) Schlüsselwert von (β, j, b, L). Dies hat folgenden Effekt:

1. Elemente im Cache (mit $\beta = 1$) sind grundsätzlich größer als Elemente außerhalb des Caches (mit $\beta = 0$).
2. Für Vergleiche von Elementen, die beide im Cache oder beide nicht im Cache sind, wird der Schlüsselwert j zu Rate gezogen.
3. Wenn beide Elemente im Cache sind aber nicht in der Folge $(a_i, \ldots, a_m)$ vorkommen, dann sind die ersten beiden Komponenten des Schlüsselwertes identisch: nämlich $\beta = 1$ und $j = m + 1$. In diesem (und nur diesem) Fall wird für den Schlüsselvergleich die Komponente b zu Rate gezogen.

Die Datenstruktur, welche alle Informationspakete der Form (β, j, b, L) unter dem Schlüsselwert (β, j, b) verwaltet und dabei die Wörterbuchoperationen inklusive MAX und CHOOSE unterstützt, notieren wir im Folgenden als D_i. Mit diesen Bezeichnungen gilt Folgendes:

Lemma 4.0.3 *Es sei $n = m + k$.*

1. *Die Datenstruktur D_1 lässt sich in Zeit $O(n \log n)$ aufbauen.*
2. *Mit Hilfe der Datenstruktur D_i und einem „Rundenzähler“ mit dem Wert i können wir in $O(\log n)$ Schritten*
   (a) *im Falle eines Cache-Fehlers die Auswahlentscheidung von S_{FF} in Runde i berechnen*
   (b) *und die Datenstruktur D_{i+1} generieren (sowie den Rundenzähler auf den Wert $i + 1$ hochsetzen).*

Beweis Es sei daran erinnert, dass wir mit $a_1, \ldots, a_m$ die (nicht notwendig verschiedenen) Komponenten der Folge **a** bezeichnen und mit $b_1, \ldots, b_k$ die paarweise verschiedenen Elemente, die sich anfangs im Cache K befinden. Zum Aufbau der Datenstruktur D_1 verwenden wir ein (anfangs leeres) Hilfswörterbuch D', welches bereits alle Informationspakete (β, j, b, L) enthält, diese aber einstweilen nur unter dem Schlüssel b verwaltet. Das Wörterbuch D' kann generiert werden wie folgt:

1. $D' \leftarrow \emptyset$.

2. Für $j = 1, 2, \ldots, m$ mache Folgendes:

   Fall 1: D' enthält keinen Eintrag mit dem Schlüsselwert a_j (MEMBER).
   Dann präpariere den Eintrag $(0, j, a_j, L_\varepsilon)$ und füge diesen in D' ein (INSERT).

   Fall 2: D' enthält einen Eintrag mit Schlüsselwert a_j (MEMBER).
   Es sei $(0, i, a_i, L)$ dieser Eintrag (FIND). Dann hänge den Eintrag j hinten an die Liste L an.

3. Für $j = 1, 2, \ldots, k$ mache Folgendes:

   Fall 1: D' enthält keinen Eintrag mit dem Schlüsselwert b_j (MEMBER).
   Dann präpariere den Eintrag $(1, m + 1, b_j, L_\varepsilon)$ und füge diesen in D' ein (INSERT).

   Fall 2: D' enthält einen Eintrag mit Schlüsselwert b_j (MEMBER).
   Es sei $(0, i, a_i, L)$ dieser Eintrag (FIND). Dann setze die erste Komponente von 0 auf 1 hoch.

Mit Hilfe von D' kann D_1 leicht erzeugt werden:

- Solange D' nicht leer ist mache Folgendes: Entnimm aus D' ein Informationspaket (β, j, b, L) (CHOOSE und DELETE) und füge es unter dem Schlüsselwert (β, j, b) in D_1 ein (INSERT).

Wir verfügen nun über die Datenstruktur $D = D_1$. Wir nehmen allgemein an, dass $D = D_i$ und i gegeben sind. Die Simulation von S_{FF} in Runde i und die Aktualisierung von D kann geschehen wie folgt:

Fall 1: D enthält einen Eintrag mit dem Schlüsselwert $(1, i, a_i)$ (MEMBER). Dann liegt kein Cache-Fehler vor und wir müssen lediglich $D = D_i$ auf $D = D_{i+1}$ aktualisieren:

1. Es sei $(1, i, a_i, L)$ der Eintrag mit Schlüsselwert $(1, i, a_i)$ in D (FIND).
2. Lösche diesen aus D (DELETE) und bearbeite das Quadrupel $(1, i, a_i, L)$ danach wie folgt:
   (a) Falls L leer ist, dann setze die Komponente i auf $m + 1$ hoch.
   (b) Falls L nicht leer ist, dann setze die Komponente i auf den Wert des ersten Eintrages in L und verkürze L entsprechend.
3. Füge das aktualisierte Quadrupel wieder in D ein (INSERT).

Fall 2: D enthält keinen Eintrag mit dem Schlüsselwert $(1, i, a_i)$ (MEMBER):
Dann liegt ein Cache-Fehler vor und D muss den Eintrag mit Schlüsselwert $(0, i, a_i)$ enthalten. Die Aktualisierung von $D = D_i$ auf $D = D_{i+1}$ geschieht wie zuvor im Fall 1 (nur dass diesmal das Quadrupel mit dem Schlüsselwert $(0, i, a_i)$ bearbeitet werden muss). Es muss aber (anders als im Fall 1) zusätzlich noch ein Element aus dem Cache vertrieben werden:

1. Es sei $(1, j, b, L)$ der Eintrag mit dem maximalen Schlüsselwert in D (MAX).
2. Lösche diesen aus D (DELETE).
3. Falls $j \neq m + 1$, dann setze die erste Komponente von $(1, j, b, L)$ auf 0 runter und füge den so aktualisierten Eintrag wieder in D ein.

•

Aus Lemma 4.0.3 ergibt sich unmittelbar durch scharfes Hinsehen:

Folgerung 4.0.4 *Die Strategie S_{FF} kann so implementiert werden, dass die Gesamtlaufzeit bei dem Szenario zu Eingabeinstanz $(N, K, \mathbf{a})$ mit $K \subseteq [N]$ und $\mathbf{a} \in [N]^m$ lediglich $O(n \log n)$ beträgt, wobei $n = m + |K|$.*

Der Beweis von Lemma 4.0.3 beschreibt eine Implementierung der FF-Strategie. Diese wird weiter unten in Beispiel 4.0.5 illustriert.

Die LRU-Strategie. In der Praxis kennt man die Folge **a** der zukünftig benötigten Speicherseiten nicht, so dass die FF-Strategie nicht ohne Weiteres anwendbar ist. Man verwendet dann stattdessen häufig die sogenannte LRU-Strategie, wobei „LRU“ für „Least-Recently-Used“ steht. Wenn ein Element aus dem Cache K vertrieben werden muss, dann entscheidet sich die LRU-Strategie für dasjenige, dessen letzte Nutzung am weitesten zurück in der Vergangenheit liegt. Dahinter steckt die Intuition, dass Speicherseiten, welche in der Vergangenheit nicht von Interesse waren, auch in der nahen Zukunft vermutlich eher nicht benötigt werden. Offenkundig ist LRU nichts anderes als eine spiegelsymmetrisch verkehrte Variante von FF: das Prinzip „Farthest-in-Future“ wird ersetzt durch das Prinzip „Farthest-in-Past“.

Beispiel 4.0.5 *Es sei* **a** *gegeben durch*

i	1	2	3	4	5	6	7	8	9	10
a_i	3	5	3	1	2	4	2	3	2	5

und für den Cache K *der Größe* 3 *gelte anfangs* $K = \{2, 4, 6\}$. *Beachte, dass* $m = 10$. *Die Datenstruktur* $D = D_1$ *enthält die folgenden Informationspakete der Form* (β, j, b, L)*:*

$$(0, 4, 1, ()), (1, 5, 2, (7, 9)), (0, 1, 3, (3, 8)), (1, 6, 4, ()), (0, 2, 5, (10)), (1, 11, 6, ())^*$$

Das Element mit maximalem Schlüsselwert $(1, 11, 6)$ *haben wir mit „*$*$*“ markiert. Wir spielen jetzt die* 10 *Runden durch, wobei das Element* 6 *gleich in Runde* 1 *aus* K *(dauerhaft) vertrieben wird:*

	$b = 1$	$b = 2$	$b = 3$	$b = 4$	$b = 5$
$a_1 = 3(F)$	$(0, 4, 1, ())$	$(1, 5, 2, (7, 9))$	$(\mathbf{1}, \mathbf{3}, \mathbf{3}, (\mathbf{8}))$	$(1, 6, 4, ())^*$	$(0, 2, 5, (10))$
$a_2 = 5(F)$	$(0, 4, 1, ())$	$(1, 5, 2, (7, 9))$	$(1, 3, 3, (8))$	$(0, 6, 4, ())$	$(\mathbf{1}, \mathbf{10}, \mathbf{5}, ())^*$
$a_3 = 3$	$(0, 4, 1, ())$	$(1, 5, 2, (7, 9))$	$(\mathbf{1}, \mathbf{8}, \mathbf{3}, ())$	$(0, 6, 4, ())$	$(1, 10, 5, ())^*$
$a_4 = 1(F)$	$(\mathbf{1}, \mathbf{11}, \mathbf{1}, ())^*$	$(1, 5, 2, (7, 9))$	$(1, 8, 3, ())$	$(0, 6, 4, ())$	$(0, 10, 5, ())$
$a_5 = 2$	$(1, 11, 1, ())^*$	$(\mathbf{1}, \mathbf{7}, \mathbf{2}, (\mathbf{9}))$	$(1, 8, 3, ())$	$(0, 6, 4, ())$	$(0, 10, 5, ())$
$a_6 = 4(F)$		$(1, 7, 2, (9))$	$(1, 8, 3, ())$	$(\mathbf{1}, \mathbf{11}, \mathbf{4}, ())^*$	$(0, 10, 5, ())$
$a_7 = 2$		$(\mathbf{1}, \mathbf{9}, \mathbf{2}, ())$	$(1, 8, 3, ())$	$(1, 11, 4, ())^*$	$(0, 10, 5, ())$
$a_8 = 3$		$(1, 9, 2, ())$	$(\mathbf{1}, \mathbf{11}, \mathbf{3}, ())$	$(1, 11, 4, ())^*$	$(0, 10, 5, ())$
$a_9 = 2$		$(\mathbf{1}, \mathbf{11}, \mathbf{2}, ())$	$(1, 11, 3, ())$	$(1, 11, 4, ())^*$	$(0, 10, 5, ())$
$a_{10} = 5(F)$		$(1, 11, 2, ())$	$(1, 11, 3, ())$		$(\mathbf{1}, \mathbf{11}, \mathbf{5}, ())^*$

Die Runden mit Cache-Fehler haben wir mit „(F)“ gekennzeichnet.

Kapitel 5

Huffman-Code und Datenkompression

Computer verarbeiten Bitstrings, also Zeichenfolgen aus Nullen und Einsen. Daher müssen die Zeichen aus größeren Alphabeten binär kodiert werden. Der ASCII-Code (mit 8 Bit = 1 Byte pro Zeichen) ist ein Beispiel dafür. Er ordnet jedem Zeichen ein Codewort derselben Bitlänge (nämlich 8) zu.

In Verbindung mit der Telegraphie (also lange vor der Erfindung des Computers) tauchte bereits die Frage auf, ob man durch Codewörter einer unterschiedlichen Bitlänge eine stärkere Textkompression erreichen kann. Die Grundidee ist, häufig vorkommende Zeichen kürzer zu kodieren als vergleichsweise selten vorkommende. Der Morse-Code[1] zum Beispiel operiert mit dem binären Alphabet $\{\cdot, -\}$. Der in Texten eher selten vorkommende Buchstabe „Y“ erhält das Codewort „$- \cdot - -$“. Der häufiger vorkommende Buchstabe „A“ wird mit „$\cdot -$“ kodiert, der Buchstabe „E“ mit „$\cdot$“ und der Buchstabe „T“ mit „$-$“.

Der Morse-Code ist (ohne weitere Maßnahmen) nicht eindeutig dekodierbar. Zum Beispiel erhalten die Wörter „ETET“, „AA“, „ETA“ und „AET“ alle dasselbe Codewort „$\cdot - \cdot -$“.[2] Wir werden uns im Folgenden auf Präfixcodes konzentrieren, bei denen eindeutige Dekodierbarkeit garantiert ist:

Definition 5.0.1 *Eine (binärer) Code für ein Alphabet A (formal gegeben durch eine Abbildung $\gamma : A \to \{0,1\}^+$) heißt* (binärer) Präfixcode*, wenn kein*

[1] Samuel Morse (1791–1872)

[2] Um dennoch eindeutige Dekodierbarkeit zu erreichen, werden beim Morsen Pausen hinter jedem Codewort eingelegt.

Codewort Präfix eines anderen Codewortes ist.

Beispiel 5.0.2 *Die folgenden Abbildungen $\gamma_0, \gamma_1, \gamma_2$ sind Präfixcodes für das Alphabet $A = \{a, b, c, d, e\}$.*

A	a	b	c	d	e
f	$0,32$	$0,25$	$0,20$	$0,18$	$0,05$
γ_0	1	011	010	001	000
γ_1	11	01	001	10	000
γ_2	11	10	01	001	000

(5.1)

Die Zeile für die Abbildung f kann im Moment noch ignoriert werden.

Ein A-markierter Binärbaum sei ein Binärbaum mit $|A|$ Blättern, welche mit den Zeichen aus A markiert sind. Präfixcodes für ein Alphabet A und A-markierte Binärbäume entsprechen sich in der offensichtlichen Weise. Der linke bzw. rechte Teilbaum der Wurzel ist zuständig für alle Zeichen, deren Codewort mit „0" bzw. „1" beginnt et cetera. In Abbildung 5.1 sind die zu $\gamma_0, \gamma_1, \gamma_2$ jeweils passenden Binärbäume zu sehen.

Es sei $f : A \to [0, 1]$ eine Abbildung mit $\sum_{a \in A} f(a) = 1$. Wir interpretieren $f(a)$ als die relative Häufigkeit, mit der das Zeichen a in Texten vorkommt. Zu einem Präfixcode $\gamma : A \to \{0, 1\}^+$ assoziieren wir die *mittlere Bitlänge* eines Codewortes, notiert als $ABL(\gamma)$:

$$ABL(\gamma) = \sum_{a \in A} f(a) \cdot |\gamma(a)| \ .$$

Auf den A-markierten Binärbaum T zum Code γ bezogen ist $ABL(\gamma)$ nichts anderes als die mittlere Tiefe eines Blattes in T.
Man rechnet leicht nach, dass für die Präfixcodes aus Tabelle (5.1) gilt:

$$ABL(\gamma_0) = 2,36 \ , \ ABL(\gamma_1) = 2,25 \ , \ ABL(\gamma_2) = 2,23 \ .$$

Dies wirft die folgende Frage auf:

- Wie finden wir einen optimalen Präfixcode, d.h. einen Präfixcode mit der kleinstmöglichen mittleren Bitlänge?

Schauen wir uns zum Beispiel den Shannon-Fano-Code[3] an. Er konstruiert einen A markierten Binärbaum nach der folgenden „Top-Down"-Methode:

[3] Claude Shannon (1916–2001) und Robert Fano (1917–2016)

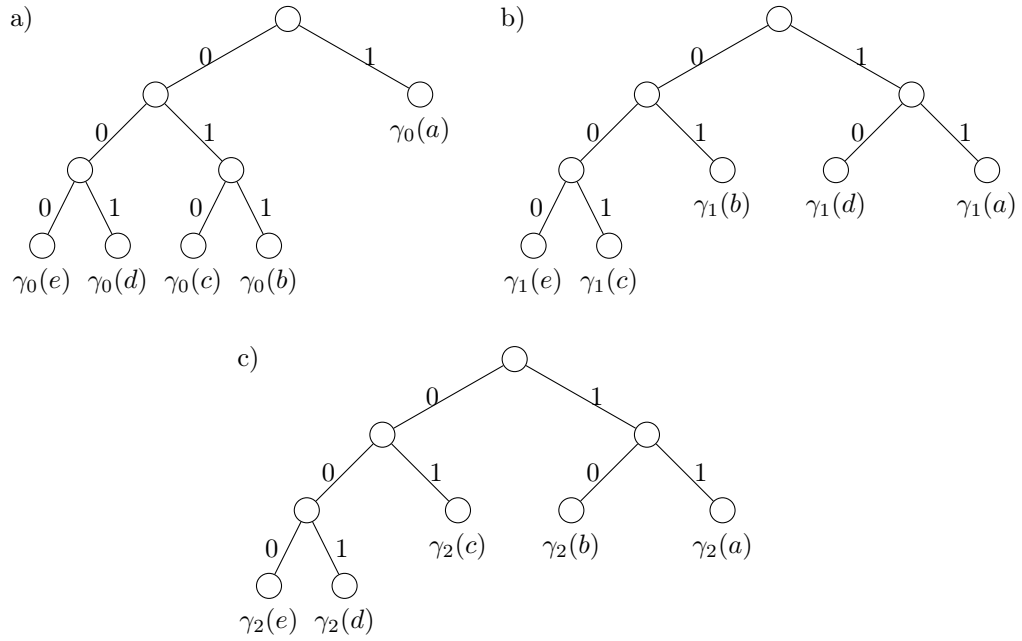


Abbildung 5.1: Die Binärbäume zu den Präfixcodes aus Tabelle (5.1)

1. Falls $|A| = 2$, dann kodiere ein Zeichen mit „0“ und das andere mit „1“, definiere T als den zugehörigen A-markierten Binärbaum (der Tiefe 1) und gehe direkt zu Schritt 5.

2. Zerlege das Alphabet A (mit $|A| \geq 3$) in zwei Teilalphabete A_0 und A_1, so dass $F_0 = \sum_{a \in A_0} f(a)$ und $F_1 = \sum_{a \in A_1} f(a)$ annähernd[4] gleich groß sind.

3. Wende das Verfahren rekursiv an und erhalte (als Ausgabe der rekursiven Aufrufe) für $i = 0, 1$ den A_i-markierten Binärbaum T_i.

4. Es sei T der A-markierte Binärbaum bestehend aus einer Wurzel mit linkem Unterbaum T_0 und rechtem Unterbaum T_1.

5. Gib T aus.

Wenn wir dieses Verfahren auf Beispiel 5.0.2 anwenden, dann erhalten wir (wie wir in der Vorlesung illustrieren werden) den Binärbaum zum Code γ_1 (der, wie wir bereits wissen, nicht optimal ist).

[4] so gut es eben geht

Shannon war sich der Tatsache bewusst, dass der von ihm und Fano entworfene Code i.A. nur suboptimal ist, sah aber keinen Weg, um effizient zu einer optimalen Lösung zu gelangen. Eine effiziente Methode zur Konstruktion eines optimalen Präfixcodes wurde schließlich von Huffman[5] gefunden. Sie basiert auf einer „Bottom-Up"-Methode, die sich rekursiv beschreiben lässt wie folgt:

Eingabe: Ein Alphabet $A = \{a_1, \ldots, a_n\}$ und eine (geordnete) Menge $F = \{f(1), \ldots, f(n)\}$ von Schlüsselwerten (= relativen Häufigkeiten)

Voraussetzung: $n \geq 2$, $f(1), \ldots, f(n) \geq 0$, $f(1) + \ldots + f(n) = 1$

Aufgabe: Konstruktion eines optimalen Präfixcodes (in Form eines A-markierten Binärbaums)

Methode: Huffman-Algorithmus (rekursive Variante)

1. Falls $n = 2$, dann kodiere a_1 mit „0" und a_2 mit „1", definiere T als den zugehörigen A-markierten Binärbaum und gehe direkt zu Schritt 5.
2. Es seien $a_k, a_\ell \in A$ die zwei Zeichen mit den kleinsten Schlüsselwerten (wobei $k > \ell$)[6]. Es sei A' ein neues Alphabet, das aus A hervorgeht, indem a_k, a_ℓ zu einem „Metazeichen" $Z(a_k, a_\ell)$ mit Schlüsselwert $f_k + f_\ell$ zusammengefasst werden. Es bezeichne F' die resultierende neue Menge von Schlüsselwerten.
3. Wende das Verfahren rekursiv auf A' und F' an und erhalte (als Ausgabe des rekursiven Aufrufes) einen Binärbaum T', dessen Blätter mit den Zeichen aus A' markiert sind.
4. Es sei z das Blatt von T', welches mit dem Metazeichen $Z(a_k, a_\ell)$ markiert ist. Kreiere zwei Kinder z_0 und z_1 von z, markiere z_0 mit a_k und z_1 mit a_ℓ (und lösche die Markierung von z). Es bezeichne T den resultierenden Binärbaum.
5. Gib T aus.

Wenn wir dieses Verfahren auf Beispiel 5.0.2 anwenden, dann erhalten wir (wie wir in der Vorlesung illustrieren werden) den Binärbaum zum Code γ_2 — ein optimaler Präfixcode wie uns folgendes Resultat lehrt:

[5] David A. Huffman (1925–1999)

[6] der Eindeutigkeit zuliebe

Satz 5.0.3 *Der Huffman Algorithmus konstruiert stets einen optimalen Präfixcode.*

Beweis Wir identifizieren einen Präfixcode mit dem entsprechenden A-markierten Binärbaum T. Zu einem Zeichen $a \in A$ bezeichne $v_T(a)$ das Blatt mit Markierung a in T. Der zentrale Baustein im Beweis ist die folgende

Behauptung: Es seien a_1 und a_2 zwei Zeichen aus A mit dem kleinsten und zweitkleinsten Schlüsselwert. Dann gibt es einen optimalen A-markierten Binärbaum $\tilde{T}$, bei dem a_1 und a_2 miteinander „verschmolzen“ werden, d.h., dass die Blätter $v_{\tilde{T}}(a_1)$ und $v_{\tilde{T}}(a_2)$ Geschwister sind (also einen gemeinsamen Vaterknoten in $\tilde{T}$ haben).

Beweis: Es sei v_1 ein Blatt maximaler Tiefe in T. Wir bezeichnen diese Tiefe mit d_{max}. Weiter sei v_2 der Geschwisterknoten von v_1. Dann hat v_2 die gleiche Tiefe d_{max} in T und muss somit ebenfalls ein Blatt sein. Die A-Markierung von v_1 bzw. v_2 in T bezeichnen wir mit b_1 bzw. b_2. Es sei d_1 bzw. d_2 die Tiefe von $v_T(a_1)$ bzw. $v_T(a_2)$ in T. Weiter sei $\tilde{T}$ der Baum, der aus T entsteht, wenn wir die Markierungen der Knoten v_1 und $v_T(a_1)$ sowie der Knoten v_2 und $v_T(a_2)$ miteinander vertauschen. Die Differenz aus der mittleren Blatttiefe in $\tilde{T}$ und der mittleren Blatttiefe in T ist dann gleich $d_{max}(f(a_1) + f(a_2)) + d_1 f(b_1) + d_2 f(b_2)$ minus $d_{max}(f(b_1) + f(b_2)) + d_1 f(a_1) + d_2 f(a_2)$. Dies ist, nach Umgruppierung von ein paar Termen, identisch zu

$$d_{max} \underbrace{(f(a_1) + f(a_2) - f(b_1) - f(b_2))}_{\leq 0} + f(b_1) \underbrace{(d_1 - d_{max})}_{\leq 0} + f(b_2) \underbrace{(d_2 - d_{max})}_{\leq 0}$$

und somit kleiner als oder gleich 0. $\tilde{T}$ muss daher ebenfalls ein optimaler A-markierter Binärbaum sein. $\tilde{T}$ hat die oben behauptete Eigenschaft.

Die obige Behauptung besagt gewissermaßen, dass die vom rekursiven Huffman-Algorithmus vorgenommene Verschmelzung von a_1 und a_2 zu einem Meta-Zeichen kein Schaden ist. Darauf aufbauend lässt sich nun die Optimalität des Huffman-Codes relativ leicht mit vollständiger Induktion über $n = |A|$ beweisen. Der Induktionsanfang mit $n = 2$ ist trivial. Sei nun $n \geq 3$. Es sei $\tilde{T}$ der optimale A-markierte Binärbaum gemäß obiger Behauptung. Es sei $\tilde{T}'$ der Binärbaum, der sich aus $\tilde{T}$ ergibt, wenn wir die Knoten $v_{\tilde{T}}(a_1)$ und $v_{\tilde{T}}(a_2)$ löschen und den gemeinsamen Vaterknoten (der nun zu

einem Blatt geworden ist) mit dem Metazeichen $Z(a_1, a_2)$ mit Schlüsselwert $f(a_1) + f(a_2)$ markieren. Es sei A' das modifizierte Alphabet mit dem Metazeichen $Z(a_1, a_2)$ anstelle von a_1 und a_2. Die mittlere Blatttiefe von $\tilde{T}$ ergibt sich aus der Summe von

- der mittleren Blatttiefe von $\tilde{T}'$
- und dem Kostenterm $f(a_1) + f(a_2)$,

da dieser Kostenterm in $\tilde{T}$, im Vergleich zu $\tilde{T}'$, mit einer um 1 vergrößerten Tiefe eingeht. Wir können also $\tilde{T}$ nicht verschlechtern, wenn wir $\tilde{T}'$ durch einen optimalen A'-markierten Binärbaum ersetzen. Da (nach Induktionsvoraussetzung) der Huffman-Code für A' optimal ist, ist dies aber exakt das, was der rekursive Huffman-Algorithmus macht. •

Um den Huffman-Algorithmus zu implementieren, benötigen wir eine Datenstruktur für A-markierte Binärbäume (mit $n = |A|$ Blättern und $n - 1$ inneren Knoten, also $2n - 1$ Knoten insgesamt). Diese können mit Hilfe von drei Arrays dargestellt werden: LS$[1 : 2n-1]$, RS$[1 : 2n-1]$ und LL$[1 : 2n-1]$. Hierbei steht „LS“ für „Left Son“, „RS“ für „Right Son“ und „LL“ für „Leaf Label“. Wir identifizieren dabei die $2n - 1$ Knoten von T mit den Adressen $1, \ldots, 2n - 1$ in den Arrays. LS, RS und LL haben dann die offensichtliche Bedeutung. Es folgt eine nicht-rekursive Variante des Huffman-Algorithmus, die mehr Implementierungsdetails sichtbar macht als die rekursive Variante zuvor.

Eingabe: zwei Arrays $A[1 : n]$ und $F[1 : n]$, wobei die Komponenten von A die Zeichen des zu kodierenden Alphabetes angeben und das Array F die zugehörigen Schlüsselwerte (= relative Häufigkeiten)

Voraussetzung: $n \geq 2$, $F[1], \ldots, F[n] \geq 0$ und $F[1] + \ldots + F[n] = 1$

Aufgabe: Konstruktion eines optimalen Präfixcodes in Form eines A-markierten Binärbaums T (der durch die Arrays LS$[1 : 2n-1]$, RS$[1 : 2n-1]$ und LL$[1 : 2n - 1]$ dargestellt wird)

Methode: Huffman-Algorithmus (nicht-rekursive Variante)

Datenstrukturen:

H: MIN_HEAP zur Verwaltung von (Meta-)Zeichen mit den Schlüsselwerten $F[i]$

N: Array der Größe n, dessen Komponente $N[i]$ — für den Fall, dass i ein Meta-Zeichen repräsentiert — die Adresse des zugehörigen Knotens in T angibt

LS,RS,LL: Arrays der Größe $2n-1$ (LS und RS initialisiert mit Nullen, LL initialisiert mit dem Leerzeichen „−“), welche am Ende den zu konstruierenden A-markierten Binärbaum T repräsentieren

Z: ein Baumknotenzähler (initialisiert mit $2n-1$), der immer die größte Adresse eines noch unbenutzten Baumknotens angeben soll

k,ℓ die beiden Nummern mit den kleinsten Schlüsselwerten in H.

l,r: Adressen für das linke und rechte Kind eines neu einzufügenden inneren Knotens.

Konzeption: Anfangs repräsentiert (LS,RS,LL) $2n-1$ unmarkierte und isolierte Knoten. Im weiteren Verlauf werden mehr und mehr dieser Knoten benutzt, um A-markierte Blätter oder innere Knoten des aufzubauenden Binärbaumes zu repräsentieren. Die „Inbetriebnahme“ der Knoten geschieht in der Reihenfolge $2n-1, , \ldots, 2, 1$, was mit Hilfe eines Zählers Z kontrolliert wird. Wird ein Knoten als Blatt mit Markierung $a_z = A[z]$ in Betrieb genommen, dann muss seine LL-Komponente auf a_z gesetzt. Im Gegenzug wird der Komponente $A[z]$ ein Leerzeichen „−“ zugewiesen. Wird ein Knoten z als innerer Knoten in Betrieb genommen, dann müssen seine LS- und RS-Komponenten entsprechend gesetzt werden. Zu jedem Zeitpunkt wird das Array-Tripel (LS,RS,LL) alle Veschmelzungen von (Meta-)Zeichen zu neuen Metazeichen reflektieren, die der Huffman-Algorithmus bereits getätigt hat.
Die Nummern von 1 bis n repräsentieren anfangs die Zeichen $a_1, \ldots, a_n$. Im Laufe des Algorithmus jedoch repräsentieren sie evtl. stattdessen auch Metazeichen. Genauer: wenn die (Meta-)Zeichen mit den Nummern k und $\ell < k$ zu einem neuen Metazeichen verschmolzen werden, dann wird ℓ zur Nummer des neuen Metazeichens (und die Nummer k wird arbeitslos). Wir weisen dann $N[\ell]$ die Nummer des Knotens zu, welcher das neue Metazeichen repräsentiert. Dieser Querverweis zwischen den Nummern von 1 bis n und den Knoten des aufzubauenden Binärbaums ist wichtig, um die LS- und RS-Komponenten korrekt zu belegen, wenn ein neuer innerer Knoten in Betrieb genommen wird.
Der MIN_HEAP H enthält diejenigen Nummern aus $[n]$, welche Zeichen oder Metazeichen repräsentieren, die nicht selber schon in Verschmel-

zungen aufgegangen sind. Das Array F wird so aktualisiert werden, dass es die entsprechenden Schlüsselwerte enthält.

die Vorgehensweise des Huffman-Algorithmus im Detail:

1. Füge die Nummern $1, \ldots, n$ mit den Schlüsselwerten $F[1], \ldots, F[n]$ in einen (anfangs leeren) MIN_HEAP H ein.
2. Initialisiere ein Array N$[1 : n]$ und die Arrays LS$[1 : 2n-1]$, RS$[1 : 2n-1]$ mit Nullen und das Array LL$[1 : 2n-1]$ mit „$-$“-Einträgen. Initialisiere einen Baumknotenzähler: $Z \leftarrow 2n-1$.
3. Durchlaufe folgende Schleife $(n-1)$-mal:
   (a) Finde die zwei Elemente k, ℓ (wobei $k > \ell$) in H mit den kleinsten Schlüsselwerten, entferne k und ℓ aus H und füge ℓ mit dem neuen Schlüsselwert $F[k] + F[\ell]$ wieder in H ein.
   (b) Falls $A[k] \neq -$:
   $l \leftarrow Z$; LL$[Z] \leftarrow A[k]$; $A[k] \leftarrow -$; $Z \leftarrow Z - 1$.
   Andernfalls: $l \leftarrow N[k]$.
   (c) Falls $A[\ell] \neq -$:
   $r \leftarrow Z$; LL$[Z] \leftarrow A[\ell]$; $A[\ell] \leftarrow -$; $Z \leftarrow Z - 1$.
   Andernfalls: $r \leftarrow N[\ell]$.
   (d) $N[\ell] \leftarrow Z$; LS$[Z] \leftarrow l$; RS$[Z] \leftarrow r$; $Z \leftarrow Z - 1$.

Beispiel 5.0.4 *Die Arrays A, F, N (mit (A, F) als Eingabe) sehen anfangs aus wie folgt:*

i	A	F	N
1	a	$0,32$	0
2	b	$0,25$	0
3	c	$0,20$	0
4	d	$0,18$	0
5	e	$0,05$	0

Der Heap H enthält anfangs die Paare $(i|F[i])$ für $i = 1, \ldots, 5$, wobei $F[i]$ als Schlüsselwert dient. Die Arrays LS,RS,LL sehen anfangs aus wie folgt:

i	LS	RS	LL
1	0	0	–
2	0	0	–
3	0	0	–
4	0	0	–
5	0	0	–
6	0	0	–
7	0	0	–
8	0	0	–
9	0	0	–

In der 1-ten Iteration der Hauptschleife werden die Indizes $k = 5$ und $\ell = 4$ zur Zeichenverschmelzung ausgewählt. Der Heap H enthält im Anschluss die Paare

$$(1|0,32), (2|0,25), (3|0,20), (4|0,23) \ .$$

Variable Z wird im Laufe der Iteration sukzessive von 9 auf 6 runtergezählt. Die Variablen l und r erhalten die Werte $l = 9$ und $r = 8$. Die diversen

Arrays werden aktualisiert wie folgt:

i	A	N
1	a	0
2	b	0
3	c	0
4	$-$	7
5	$-$	0

i	LS	RS	LL
1	0	0	$-$
2	0	0	$-$
3	0	0	$-$
4	0	0	$-$
5	0	0	$-$
6	0	0	$-$
7	9	8	$-$
8	0	0	d
9	0	0	e

In der 2-ten Iteration der Hauptschleife werden die Indizes $k = 4$ *und* $\ell = 3$ *zur Zeichenverschmelzung ausgewählt. Der Heap* H *enthält danach die Paare*

$$(1|0, 32), (2|0, 25), (3|0, 43) \ .$$

Im Laufe der Iteration wird Z *sukzessive von* 6 *auf* 4 *runtergezählt. Die Variablen* l *und* r *erhalten die Werte* $l = 7$ *und* $r = 6$*. Die Array-Aktualisierungen liefern:*

i	A	N
1	a	0
2	b	0
3	$-$	5
4	$-$	7
5	$-$	0

i	LS	RS	LL
1	0	0	$-$
2	0	0	$-$
3	0	0	$-$
4	0	0	$-$
5	7	6	$-$
6	0	0	c
7	9	8	$-$
8	0	0	d
9	0	0	e

Nach der 3-ten Iteration mit $k = 2$ *und* $\ell = 1$ *enthält* H *nur noch die Paare* $(1|0, 57)$ *und* $(3|0, 43)$*. Im Laufe der Iteration wird* Z *sukzessive von* 4 *auf* 1 *runtergezählt. Die Variablen* l *und* r *erhalten die Werte* $l = 4$ *und* $r = 3$*. Die*

Array-Aktualisierungen liefern:

i	A	N
1	–	2
2	–	0
3	–	5
4	–	7
5	–	0

i	LS	RS	LL
1	0	0	–
2	4	3	–
3	0	0	a
4	0	0	b
5	7	6	–
6	0	0	c
7	9	8	–
8	0	0	d
9	0	0	e

Nach der 4-ten und letzten Iteration enthält H nur noch das Paar $(1|1)$. Die Variable Z wird von 1 auf 0 runtergezählt. Die Variablen l und r erhalten die Werte $l = 5$ und $r = 2$. die Arrays sehen am Schluss aus wie folgt:

i	A	N
1	–	1
2	–	0
3	–	5
4	–	7
5	–	0

i	LS	RS	LL
1	5	2	–
2	4	3	–
3	0	0	a
4	0	0	b
5	7	6	–
6	0	0	c
7	9	8	–
8	0	0	d
9	0	0	e

Die Arrays (LS,RS,LL) repräsentieren jetzt den A-markierten Binärbaum zum optimalen Präfixcode.

Die Laufzeit des Huffman-Algorithmus wird durch die Hauptschleife dominiert. In jeder der $n - 1$ Iterationen erfolgen konstant viele elementare Rechenschritte und konstant viele Operationen auf dem Heap. Da der Heap die Maximalgröße n hat, gelangen wir zu folgendem Ergebnis:

Satz 5.0.5 *Der Huffman-Algorithmus besitzt eine Implementierung, die den optimalen Präfixcode in $O(n \log n)$ Schritten bestimmt.*

Schlussbemerkung. Auf den ersten Blick scheint der Huffman-Algorithmus das Problem der Datenkompression optimal zu lösen. Das ist jedoch aus unterschiedlichen Gründen keineswegs der Fall. Wir führen zwei dieser Gründe an:

- Betrachte Schwarz-Weiß-Bilder gegeben durch $0,1$-wertige Pixelmatrizen. Häufig ist nur ein kleiner Bruchteil der Pixel schwarz. Diese Bilddaten liegen bereits in binärer Form (Schwarz-Weiß) vor, so dass der Huffman-Algorithmus nichts bewirken kann. Evtl. könnte aber bereits das Auflisten der Koordinaten der schwarzen Pixel eine erhebliche Datenkompression bewirken. In der Praxis kommen hier Methoden des „arithmetischen Kodierens“ zum Einsatz.

- Ein weiterer Nachteil des Huffman-Algorithmus ist seine Nicht-Adaptivität. Bei Text-, Bild- oder Ton-Material kann es zu signifikanten Veränderungen der Häufigkeitswerte kommen. Adaptive Kodierverfahren tragen solchen Umständen Rechnung und sind Gegenstand aktueller Forschung.

Kapitel 6

MergeSort und Zählen von Inversionen

Wir betrachten zunächst das Problem, zwei sortierte Listen $(a_i)_{i\in[m]}$ und $(b_j)_{j\in[n]}$ mit Einträgen

$$a_1 \leq \ldots \leq a_m \quad \text{und} \quad b_1 \leq \ldots \leq b_n$$

zu einer sortieren Gesamtliste $(c_k)_{k\in[m+n]}$ mit

$$c_1 \leq \ldots \leq c_{m+n}$$

zusammenzufügen. Dies kann mit folgender Methode, genannt MERGE (=Mischen), mit Hilfe dreier Zeiger i, j und k in $O(m+n)$ Schritten geschehen:

1. $i \leftarrow 1$; $j \leftarrow 1$; $k \leftarrow 1$.

2. Solange $i \leq m$ und $j \leq n$ mache Folgendes:

   (a) Falls $a_i \leq b_j$: $c_k \leftarrow a_i$; $i \leftarrow i+1$.
   Andernfalls: $c_k \leftarrow b_j$; $j \leftarrow j+1$.

   (b) $k \leftarrow k+1$.

3. Falls $i = m+1$:
   Solange $j \leq n$: $c_k \leftarrow b_j$; $j \leftarrow j+1$; $k \leftarrow k+1$.
   Andernfalls:
   Solange $i \leq m$: $c_k \leftarrow a_i$; $i \leftarrow i+1$; $k \leftarrow k+1$.

In Schritt 1 werden die drei Zeiger auf die jeweiligen Listenanfänge positioniert. In Schritt 2 wird wiederholt das kleinere der Elemente a_i und b_j in die c-Liste übertragen. Der Zeiger des nicht übertragenen Elementes bleibt stehen, die anderen zwei Zeiger rücken eine Position weiter. Dies geschieht solange, bis eine der a- und b-Listen vollständig in die c-Liste übertragen ist. In Schritt 3 wird der Rest der noch nicht vollständig übertragenen Liste in die c-Liste übertragen.

Die Laufzeit der Prozedur MERGE wird dominiert durch die beiden Schleifen in Schritt 2 und Schritt 3. Eine Iteration durch eine der Schleifen kostet nur Rechenzeit $O(1)$. Da bei jeder Iteration einer der beiden Zeiger i und j um eine Position weitergerückt wird, kann es nicht mehr als $n + m$ Iterationen geben. Wir fassen diese Diskussion zusammen:

Satz 6.0.1 *Die Prozedur MERGE mischt zwei sortierte Listen der Längen m und n in Rechenzeit $O(m+n)$ zu einer sortierten Gesamtliste zusammen.*

Die Prozedur MERGE ist der zentrale Baustein für das rekursive Sortierverfahren MergeSort, welches eine anfangs unsortierte Liste $L = (a_1, \ldots, a_n)$ nach folgendem Schema sortiert:

1. Falls $n = 1$, dann gib die Liste (a_1) aus und stoppe.
   Andernfalls (also wenn $n \geq 2$) mache weiter.

2. Zerlege L in zwei unsortierte Teillisten $L_1 = (a_1, \ldots, a_m)$ und $L_2 = (a_{m+1}, \ldots, a_n)$, wobei $m = \lfloor \frac{n+1}{2} \rfloor$.

3. Wende das Verfahren rekursiv auf die Listen L_1 und L_2 an und erhalte (nach dem Terminieren der rekursiven Aufrufe) diese in sortierter Form zurück.

4. Mische mit Hilfe der Prozedur MERGE die beiden sortierten Teillisten zu einer sortierten Gesamtliste zusammen und gib diese dann aus.

Es handelt sich um ein Divide&Conquer-Verfahren: ein Problem der Größe n wird in zwei Teilprobleme der Größe (etwa) $n/2$ zerlegt (Divide). Die Lösungen der beiden Teilprobleme werden zu einer Gesamtlösung kombiniert (Conquer). Die Divide-Phase beansprucht trivialerweise maximal Rechenzeit $O(n)$, die Conquer-Phase (Anwendung von MERGE) benötigt ebenfalls

nur Rechenzeit $O(n)$. Daher ist die von MergeSort beanspruchte Rechenzeit größenordnungsmäßig identisch zur Lösung der Rekursionsgleichung

$$T(1) = a \quad \text{und} \quad T(n) = 2 \cdot T\left(\frac{n}{2}\right) + cn \ .$$

Dabei sind a und c von n unabhängige Konstanten. Wie wir bereits wissen gilt $T(n) = O(n \log n)$. Es hat sich ergeben:

Satz 6.0.2 *Das Sortierverfahren MergeSort sortiert n anfangs unsortierte Daten in Rechenzeit $O(n \log n)$.*

Unsere obige Beschreibung von MergeSort blendet ein paar Implementierungsdetails aus, auf die wir nun kurz zu sprechen kommen. Nehmen wir an, dass die anfangs unsortierte Liste L durch ein Array $A[1:n]$ mit $A[i] = a_i$ gegeben ist. Die Prozedur MERGE kann nicht „in situ“ (also innerhalb eines Arrays) ablaufen: sie benötigt neben dem Array, in welchem sie die sortierten Teillisten vorfindet, mindestens ein weiteres Array für die Ausgabe. Andererseits reichen zwei Arrays, sagen wir $A[1:n]$ und $B[1:n]$, aus: in der Conquer-Phase werden Daten abwechselnd von A nach B und dann wieder von B nach A geschaufelt. (In der Divide-Phase wird das Array B noch nicht benötigt.)

Als kleine Fingerübung stellen wir uns die Aufgabe, die rekursive Prozedur so zu präzisieren, dass die Verwendung zweier Arrays (so wie oben beschrieben) dabei deutlich wird. Die Arrays A und B behandeln wir als globale Variable. Als Prozedurparameter übergeben wir die Array-Grenzen für die jeweiligen Teillisten und eine Boolesche Variable b: $b = 0$ (bzw. $b = 1$) zeigt an, dass der rekursive Aufruf sein Ergebnis im Array A (bzw. im Array B) abliefern soll.

Rekursive Prozedur: MergeSort(l, r, b)

Globale Variablen: Arrays $A[1:n]$ und $B[1:n]$

Parameter: l, r mit $1 \leq l \leq r \leq n$ und $b \in \{0, 1\}$

Lokale Variablen: Array-Index m sowie die für MERGE benötigten lokalen Variablen

Aufgabe: Sortiere die Daten im Teil-Array $A[l:r]$. Falls $b = 0$ (bzw. $b = 1$), so liefere das Ergebnis in $A[l:r]$ (bzw. in $B[l:r]$) ab.

Methode: Sortieren durch wiederholtes Mischen

1. Falls $l = r$ und $b = 0$: stoppe.
   Falls $l = r$ und $b = 1$: Setze $B[l] \leftarrow A[l]$ und stoppe.
   Falls $l \neq r$: mache weiter.
2. $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$; MergeSort$(l, m, \neg b)$; MergeSort$(m+1, r, \neg b)$.
3. Falls $b = 0$, dann mische (mit Hilfe der Prozedur MERGE) die in $B[l : m]$ und $B[m+1 : r]$ vorliegenden sortierten Teillisten zu einer sortierten Liste in $A[l : r]$ zusammen. Falls $b = 1$, dann mache das Gleiche mit vertauschten Rollen von A und B.

In Schritt 2 (bzw. 3) befinden wir uns in der Divide- (bzw. Conquer)-Phase. Schritt 1 wird (insgesamt n-mal) ausgeführt, wenn wir aus der Rekursion aussteigen. Im Hauptprogramm erfolgt der Aufruf MergeSort$(1, n, 0)$.

Beispiel 6.0.3 *Das Array* $A[1 : 8]$ *enthalte die Folge* $(2, 10, 7, 3, 1, 4, 6, 9)$. *In Abb. 6.1 ist der Rekursionsbaum zum Aufruf MergeSort*$(1, 8, 0)$ *zu sehen (wobei „MS“ für „MergeSort“ steht). Die* 8 *Aufrufe MergeSort*$(i, i, 1)$ *auf der Blattebene (mit* $i = 1, \ldots, 8$*) bewirken, dass die Folge* $(2|10|7|3|1|4|6|9)$ *von Array A in das Array B kopiert wird.*[1] *Auf der Ebene 2 gelangen die Daten in der Reihenfolge* $(2, 10|3, 7|1, 4|6, 9)$ *wieder in das Array A. Auf Ebene* 1 *werden sie in der Reihenfolge* $(2, 3, 7, 10|1, 4, 6, 9)$ *wieder nach B übertragen. Auf (Wurzel-)Ebene* 0 *schließlich gelangen sie sortiert, also in der Reihenfolge* $(1, 2, 3, 4, 6, 7, 9, 10)$*, in das Array A.*

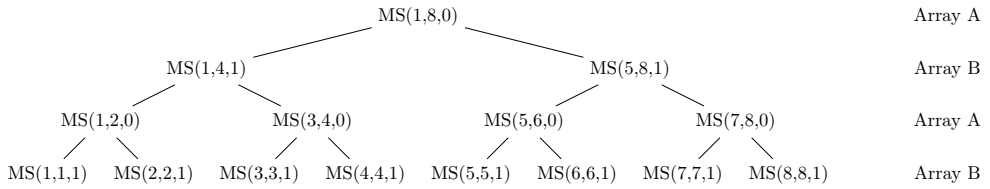


Abbildung 6.1: Der Rekursionsbaum zum Aufruf MergeSort$(1, 8, 0)$.

Wir betrachten abschließend ein Problem, das durch „Recommender Systems“ motiviert ist (worauf wir in der Vorlesung etwas näher eingehen).

[1] Mit dem Trennstrich "|" machen wir die Grenze zwischen zwei verschiedenen Aufrufen der Prozedur (auf derselben Ebene des Rekursionsbaums) deutlich.

Formal gesehen handelt es sich um das Problem, die Inversionszahl $z(L)$ einer gegebenen Liste $L = (a_1, \ldots, a_n)$ zu bestimmen wobei

$$z(L) = |\{(i,j) | 1 \leq i < j \leq n \text{ und } a_i > a_j\}| \ .$$

Zum Beispiel hat die aufsteigend sortierte Liste $(1, \ldots, n)$ die Inversionszahl 0. Die absteigend sortierte Liste $(n, \ldots, 1)$, das andere Extrem, hat die Inversionszahl $\binom{n}{2}$. „Feld-Wald-und-Wiesen-Listen“ liegen zwischen diesen Extremen, d.h., für jede Liste L der Länge n gilt $z(L) \in \{0, 1, \ldots, \binom{n}{2}\}$. Eine effiziente Berechnung von $z(L)$ basiert auf folgender Beobachtung:

1. Die Inversionszahl zu einer Liste der Länge $n = 1$ ist 0.

2. Für $n \geq 2$ betrachte eine Zerlegung von L in zwei nichtleere Teillisten $L_1 = (a_1, \ldots, a_m)$ und $L_2 = (a_{m+1}, \ldots, a_n)$. Dann gilt $z(L) = z(L_1) + z(L_2) + z(L_1, L_2)$ mit

$$z(L_1, L_2) = |\{(i,j) | 1 \leq i \leq m, m+1 \leq j \leq n \text{ und } a_i > a_j\}| \ .$$

Wir werden weiter unten aufzeigen, dass die Zahl $z(L_1, L_2)$ mit einer Prozedur namens „Extended_Merge“ (eine Erweiterung von MERGE) in Zeit $O(n)$ berechnet werden kann. Dies führt zu folgender Erweiterung von MergeSort, welche eine gegebene unsortierte Liste $L = (a_1, \ldots, a_n)$ sortiert und nebenbei die Inversionszahl $z(L)$ berechnet:

1. Falls $n = 1$, dann gib die Liste (a_1) und die Inversionszahl 0 aus und stoppe. Andernfalls (also wenn $n \geq 2$) mache weiter.

2. Zerlege $L = (a_1, \ldots, a_n)$ in zwei unsortierte Teillisten $L_1 = (a_1, \ldots, a_m)$ und $L_2 = (a_{m+1}, \ldots, a_n)$, wobei $m = \lfloor \frac{n+1}{2} \rfloor$.

3. Wende das Verfahren rekursiv auf die Listen L_1 und L_2 an und erhalte (nach dem Terminieren der rekursiven Aufrufe)

   - die sortierte Version $L_1' = (a_1', \ldots, a_m')$ von L_1 sowie die sortierte Version $L_2' = (a_{m+1}', \ldots, a_n')$ von L_2
   - und die zugehörigen Inversionszahlen $z(L_1)$ und $z(L_2)$.

4. Ermittle mit Hilfe der Prozedur Extended_Merge die sortierte Gesamtliste L' und die Zahl $z(L_1, L_2)$.

5. Gib L' und $z(L_1) + z(L_2) + z(L_1, L_2)$ als Ergebnis aus.

Es bleibt zu zeigen, wie die Zahl $z(L_1, L_2)$ effizient bestimmt werden kann. Die zentralen Beobachtungen sind wie folgt:

- Für die sortierten Versionen $L'_1 = (a'_1, \ldots, a'_m)$ und $L'_2 = (a'_{m+1}, \ldots, a'_n)$ von L_1 und L_2 gilt: $z(L'_1, L'_2) = z(L_1, L_2)$.
- Für eine feste Wahl von $j \in \{m+1, \ldots, n\}$ sei $i(j)$ der kleinste Index $i \in [m]$ mit $a'_i > a'_j$ (bzw. $i(j) = m+1$, falls ein solches i nicht existiert). Dann gilt
$$m - i(j) + 1 = |\{i \in [m] | a'_i > a'_j\}| \ ,$$
d.h., $m - i(j) + 1$ ist der Beitrag, den der Index j zur Zahl $z(L_1, L_2)$ leistet, und es gilt:
$$z(L_1, L_2) = \sum_{j=m+1}^{n} (m - i(j) + 1) \ .$$

Es folgt eine „High-Level“-Beschreibung der Prozedur Extended_Merge. Diese berechnet zu gegebenen sortierten Listen $L_1 = (a_1, \ldots, a_m)$ und $L_2 = (b_1, \ldots, b_n)$ mit $a_1 \leq \ldots \leq a_m$ und $b_1 \leq \ldots \leq b_n$

1. eine sortierte Gesamtliste $L = (c_1, \ldots, c_{m+n})$ mit $c_1 \leq \ldots \leq c_{m+n}$,
2. die Zahl $z(L_1, L_2)$.

Zu diesem Zweck verwendet sie neben den Zeigern i, j, k eine weitere lokale Variable z, um die aufgespürten „(L_1, L_2)-Inversionen“ mitzuzählen:

1. $i \leftarrow 1$; $j \leftarrow 1$; $k \leftarrow 1$; $z \leftarrow 0$.
2. Solange $i \leq m$ und $j \leq n$ mache Folgendes:
   (a) Falls $a_i \leq b_j$: $c_k \leftarrow a_i$; $i \leftarrow i + 1$.
   Andernfalls: $c_k \leftarrow b_j$; $j \leftarrow j + 1$; $z \leftarrow z + m - i + 1$.
   (b) $k \leftarrow k + 1$.
3. Falls $i = m + 1$:
   Solange $j \leq n$: $c_k \leftarrow b_j$; $j \leq j + 1$; $k \leftarrow k + 1$.
   Andernfalls:
   Solange $i \leq m$: $c_k \leftarrow a_i$; $i \leq i + 1$; $k \leftarrow k + 1$.

Es ist nicht schwer, induktiv folgende Schleifeninvariante nachzuweisen. Wenn i aktuell den Wert i_0 und j aktuell den Wert j_0 hat, dann gilt:

- Die bisher aufgebaute c-Folge enthält die Elemente aus $M(i_0, j_0) := \{a_1, \ldots, a_{i_0-1}\} \cup \{b_1, \ldots, b_{j_0-1}\}$, und zwar aufsteigend sortiert.
- Keine der Zahlen aus $M(i_0, j_0)$ ist größer als $\min\{a_{i_0}, b_{j_0}\}$.
- Falls $a_{i_0} > b_{j_0}$, dann ist i_0 der kleinste Index $i \in [m]$ mit $a_i > b_{j_0}$, d.h., $i_0 = i(j_0)$.
- Der Zähler z hat den Wert $\sum_{j=1}^{j_0-1}(m - i(j) + 1)$.

Aus dieser Invariante kann man ablesen, dass die c-Folge am Ende eine sortierte Gesamtliste darstellt, und dass die Variable z am Ende den Wert $z(L_1, L_2)$ hat.

Beispiel 6.0.4 *Wir setzen Beispiel 6.0.3 fort und wollen* $(2, 10, 7, 3, 1, 4, 6, 9)$ *nicht nur sortieren sondern auch die Inversionszahl dieser Folge bestimmen. Wir konzentrieren uns auf die oberste Rekursionsebene. Der Aufruf MergeSort*$(1, 4, 1)$ *liefert neben der Liste* $L_1 = (2, 3, 7, 10)$ *die Inversionszahl* $z(L_1) = 3$. *Analog liefert der Aufruf MergeSort*$(5, 8, 1)$ *neben der Liste* $L_2 = (1, 4, 6, 9)$ *die Inversionszahl* $z(L_2) = 0$. *Der Aufruf von Extended_Merge liefet neben der sortierten Gesamtliste die Zahl*

$$z(L_1, L_2) = 9 = 4 + 2 + 2 + 1 \ .$$

Dabei sind $4, 2, 2, 1$ *die Beiträge, welche die Komponenten* $1, 4, 6, 9$ *von* L_2 *zu* $z(L_1, L_2)$ *leisten. Die Inversionszahl von* $L = (2, 10, 7, 3, 1, 4, 6, 9)$ *ergibt sich dann aus*

$$z(L) = z(L_1) + z(L_2) + z(L_1, L_2) = 3 + 0 + 9 = 12 \ .$$

Kapitel 7

Paare von Punkten mit minimaler Distanz

Es sei $P \subset \mathbb{R}^2$ eine Menge von n Punkten in der Euklidischen Ebene. Im Folgenden bezeichnen wir mit einem „Punktepaar“ immer ein Paar mit zwei verschiedenen Punkten aus P. Wir suchen ein Paar (p_0^*, p_1^*) von Punkten, deren Distanz zueinander so klein wie möglich ist, d.h.,

$$d(p_0^*, p_1^*) = \min\{d(p_0, p_1) | p_0, p_1 \in P, p_0 \neq p_1\} \ , \tag{7.1}$$

wobei $d(p_0, p_1)$ für Punkte $p_0 = (x_0, y_0)$ und $p_1 = (x_1, y_1)$ gegeben ist durch

$$d(p_0, p_1) = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} \ .$$

Im Folgenden bezeichnen wir ein solches Punktepaar als „optimal“.

Wir setzen in diesem Abschnitt (idealisierend) voraus, dass die Operationen $+, -, \cdot, \sqrt{}$ auf den reellen Zahlen jeweils in konstanter Zeit ausführbar sind.[1] Daher kann auch $d(p_0, p_1)$ in konstanter Zeit bestimmt werden.

Betrachten wir als Aufwärmübung den Spezialfall, bei dem alle n Punkte auf einer Geraden liegen. In diesem Fall genügt es, (a) die Punkte (entlang der Geraden) zu sortieren und (b) die sortierte Liste zu durchmustern, wobei die Distanzen von auf der Geraden benachbarten Punkten berechnet werden

[1] Auf einer RAM (oder einem realen Rechner) würde man statt dem exakten Ergebnis nur eine Approximation desselben in konstanter Zeit berechnen können. Die Kontrolle des Approximationsfehlers (obschon grundsätzlich möglich) würde uns aber zu weit von dem Kern des hier diskutierten Problems ablenken.

und der aktuelle Champion[2] mitprotokolliert wird. Das Sortieren in Phase (a) kostet Zeit $O(n \log n)$. Für Phase (b) wird nur Linearzeit benötigt.

Wir setzen uns das Ziel, das Problem in zwei Dimensionen ebenfalls in Zeit $O(n \log n)$ zu lösen. Da Sortieren von n Daten in Zeit $O(n \log n)$ möglich ist, können wir folgende Objekte vorab berechnen:

- die Liste P_x aller Punkte aus P aufsteigend sortiert nach ihrer x-Koordinate
- sowie die Liste P_y aller Punkte aus P aufsteigend sortiert nach ihrer y-Koordinate

Generell gilt: wenn wir eine geordnete Liste L von Elementen haben, sowie eine Element-Eigenschaft E, die in konstanter Zeit überprüfbar ist, dann können wir in Linearzeit die geordnete Teilliste L' aller Elemente von L mit Eigenschaft E berechnen. Dazu müssen wir lediglich die Liste L einmal durchforsten, bei jedem Element in L die Eigenschaft E testen und die Elemente aus L, welche den Test bestehen, in eine (anfangs leere) Liste L' aufnehmen.

Es ist daher naheliegend, unser Problem in zwei Teilprobleme der halben Größe aufzuteilen wie folgt (zur Illustration s. Abb. 7.1):

- Ermittle in P_x den Punkt $p' = (x', y')$ in der Mitte der Liste.
- Extrahiere aus P_x (bzw. aus P_y) die geordnete Teilliste Q_x (bzw. Q_y) aller Punkte $p = (x, y)$ mit $x \leq x'$.
- Extrahiere aus P_x (bzw. aus P_y) die geordnete Teilliste R_x (bzw. R_y) aller Punkte $p = (x, y)$ mit $x > x'$.

Damit repräsentiert (Q_x, Q_y) das Teilproblem in der „linken Hälfte“ und (R_x, R_y) das Teilproblem in der „rechten Hälfte“ des Gesamtproblems. Es sollte klar sein, dass die Aufteilung in diese zwei Teilprobleme nur Zeit $O(n)$ erfordert.

Die beiden Teilprobleme können rekursiv gelöst werden[3], sagen wir (q_0^*, q_1^*) ist das optimale Punktepaar in der linken Hälfte und (r_0^*, r_1^*) ist das optimale Punktepaar in der rechten Hälfte. Freilich muss keines dieser beiden Paare optimal für das Gesamtproblem sein: das optimale Paar (p_0^*, p_1^*) könnte einen

[2] Dies ist das Punktepaar, das bisher die kleinste Distanz aufweist.

[3] mit Ausstieg aus der Rekursion bei Listen mit nur 3 oder weniger Punkten

Punkt in der linken und den anderen in der rechten Hälfte haben (und dies sind Paare, die bei den Teilproblemen nicht in Betracht gezogen werden). Es ist auf Anhieb nicht zu sehen, wie wir effizient zu einem global optimalen Punktepaar gelangen. Dass dies dennoch möglich ist, zeigt das folgende (evtl. überraschende) Resultat:

Lemma 7.0.1 *Aus P_x, P_y, (q_0^*, q_1^*) und (r_0^*, r_1^*) können wir in Zeit $O(n)$ ein optimales Punktepaar (p_0^*, p_1^*) für das Gesamtproblem berechnen.*

Den Beweis liefern wir weiter unten nach.

Fassen wir zusammen: nach einem anfänglichen Sortiervorgang (in Zeit $O(n \log n)$) können wir das Gesamtproblem in Zeit $O(n)$ in zwei Teilprobleme der halben Größe zerlegen und die Teillösungen in Zeit $O(n)$ in eine Gesamtlösung transformieren. Daraus ergibt sich (vom anfänglichen Sortieren einmal abgesehen) für die Laufzeit eine Rekursionsgleichung vom Typ "MergeSort", deren Lösung $O(n \log n)$ ist. Wir erhalten daher die

Folgerung 7.0.2 *Zu einer gegebenen Menge P, bestehend aus n Punkten in der Ebene, können wir in Zeit $O(n \log n)$ ein Punktepaar (p_0^*, p_1^*) aufspüren, welches die Optimalitätsbedingung (7.1) erfüllt.*

Beweis[Lemma 7.0.1] Es sei

$$\delta = \min\{d(q_0^*, q_1^*), d(r_0^*, r_1^*)\} \ . \tag{7.2}$$

Es sei L eine durch p' (dem Punkt in der Mitte von P_x) gezogene vertikale Linie. Wir parzellieren den vertikalen Streifen S der Weite 2δ, mit L in der Mitte, in Quadrate der Seitenlänge $\delta/2$ (s. Abb. 7.1) und beobachten:

- Falls ein Paar (p_0^*, p_1^*) mit einer Distanz unterhalb von δ existiert, dann müssen beide Punkte im Streifen S liegen, und zwar einer links und einer rechts von L.

- Jede Parzelle mit Seitenlänge $\delta/2$ in S enthält maximal einen Punkt (weil jede Parzelle entweder vollständig links oder vollständig rechts von L liegt, der Durchmesser der Parzelle kleiner als δ ist, und es aber weder links noch rechts von L ein Punktepaar mit einer Distanz unterhalb von δ geben kann).

- Es sei S_y die geordnete Teilliste aller Punkte $p = (x, y)$ von P_y, die im Streifen S liegen (die also die Bedingung $x' - \delta \leq x \leq x' + \delta$ erfüllen). Wenn S_y ein Punktepaar (p_0^*, p_1^*) mit einer Distanz unterhalb von δ enthält, dann sind die Positionen dieser Punkte in der Liste S_y maximal um 11 voneinander entfernt (s. Abb. 7.1).

Um ein optimales Punktepaar (p_0^*, p_1^*) zu berechnen, können wir vorgehen wie folgt:

- Wir berechnen die Teilliste S_y von P_y.
- Wir durchforsten S_y, wobei wir
  - zu jedem Punkt mit Position i in S_y die Distanzen zu allen Punkten mit Position $i + 1, \ldots, i + 11$ in S_y bestimmen
  - und das bisher beste Paar immer mitprotokollieren.
- Es bezeichne (s_0^*, s_1^*) das beste Paar, das wir in S_y aufgespürt haben (der „Streifen-Champion“). Es sei dann (p_0^*, q_0^*) das Paar mit dem kleinsten Distanzwert aus der Kandidatenmenge $\{(s_0^*, s_1^*), (q_0^*, q_1^*), (r_0^*, r_1^*)\}$, bestehend aus dem „Streifen-Champion“ und den Champions für das linke und rechte Teilproblem.
- Wir geben (p_0^*, q_0^*) als optimales Punktepaar aus.

Offensichtlich benötigen diese Maßnahmen nur Zeit $O(n)$. •

Wie geben zum Abschluss den Gesamtalgorithmus an:

Eingabe: eine Menge $P \subset \mathbb{R}^2$ bestehend aus n Punkten in der Euklidischen Ebene

Aufgabe: Bestimme ein Punktepaar, das die Optimalitätsbedingung (7.1) erfüllt.

Methode: Vorausberechnung (Sortieren) und „Divide & Conquer“

1. Berechne die Liste P_x aller Punkte aus P aufsteigend sortiert nach ihrer x-Koordinate.
2. Berechne die Liste P_y aller Punkte aus P aufsteigend sortiert nach ihrer y-Koordinate.

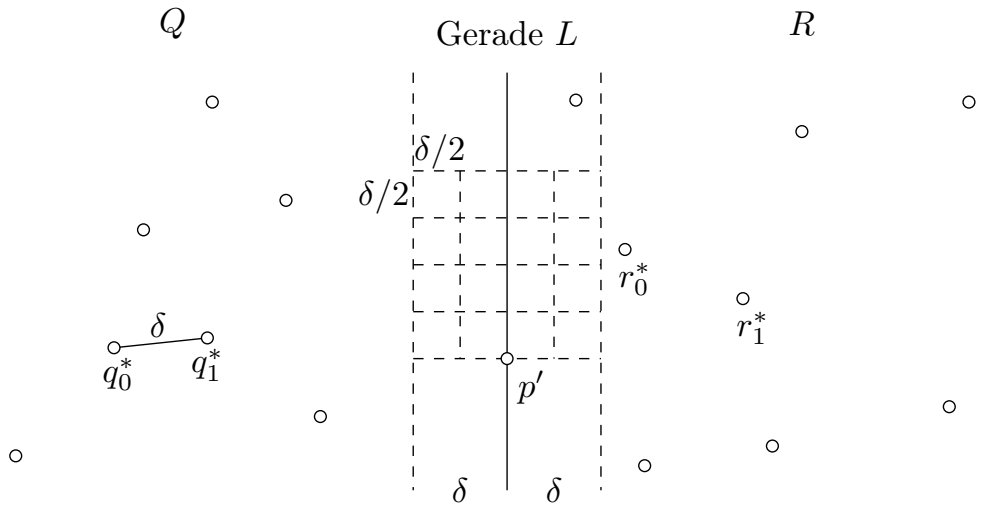


Abbildung 7.1: Gegebene Punktemenge, Grenzlinie L und der Streifen S.

3. Rufe die rekursive Prozedur Closest_Points auf mit den Parametern P_x und P_y.

Dabei ist Closest_Points die folgende Prozedur:

Parameter: P_x und P_y
zwei Listen derselben Menge von Punkten in der Ebene, eine aufsteigend sortiert nach x- die andere aufsteigend sortiert nach y-Koordinaten.

Vorgehensweise:

1. Falls es sich nur um 3 (oder noch weniger) Punkte handelt, dann bestimme ein optimales Punktepaar durch Berechnung der Distanzwerte für alle 3 (oder noch weniger) möglichen Punktepaare und stoppe. Andernfalls mache weiter.
2. Ermittle in P_x den Punkt $p' = (x', y')$ in der Mitte der Liste.
3. Extrahiere aus P_x (bzw. aus P_y) die geordnete Teilliste Q_x (bzw. Q_y) aller Punkte $p = (x, y)$ mit $x \leq x'$.
4. Extrahiere aus P_x (bzw. aus P_y) die geordnete Teilliste R_x (bzw. R_y) aller Punkte $p = (x, y)$ mit $x > x'$.
5. Rufe rekursiv die Prozedur Closest_Points auf mit den Parametern Q_x und Q_y und erhalte ein optimales Punktepaar (q_0^*, q_1^*) für das betreffende Teilproblem (linke Hälfte).

6. Rufe rekursiv die Prozedur Closest_Points auf mit den Parametern R_x und R_y und erhalte ein optimales Punktepaar (r_0^*, r_1^*) für das betreffende Teilproblem (rechte Hälfte).
7. Berechne δ gemäß (7.2) sowie die geordnete Teilliste S_y aller Punkte (x, y) aus P_y, welche die Bedingung $x' - \delta \leq x \leq x' + \delta$ erfüllen (Punkte im Streifen).
8. Durchforste S_y, wobei
   (a) zu jedem Punkt mit Position i in S_y die Distanzen zu allen Punkten mit Positionen $i+1, \ldots, i+11$ in S_y bestimmt werden
   (b) und das bisher beste Paar immer mitprotokolliert wird.
9. Es bezeichne (s_0^*, s_1^*) das beste Paar, das wir in S_y aufgespürt haben. Es sei dann (p_0^*, q_0^*) das Paar mit dem kleinsten Distanzwert aus der Kandidatenmenge $\{(s_0^*, s_1^*), (q_0^*, q_1^*), (r_0^*, r_1^*)\}$.
10. Gib (p_0^*, q_0^*) als optimales Punktepaar aus.

Kapitel 8

Intervall–Scheduling mit Gewichten

Wir betrachten eine Verallgemeinerung des Intervall–Scheduling Problems, welches wir früher bereits kennengelernt haben. In der uns bekannten Version gibt es n potenzielle Nutzer einer zentralen Ressource. Nutzer j spezifiziert ein Zeitintervall $R_j = [a_j, b_j)$, in dem er die Ressource zugeteilt bekommen möchte. Anfragen mit überlappenden Zeitintervallen können nicht simultan befriedigt werden. Ziel ist es, eine möglichst große Anzahl von Anfragen positiv zu bescheiden. In der im Folgenden zu diskutierenden allgemeineren Variante ist jedes Intervall R_j mit einem Gewicht $w_j \in \mathbb{N}$ versehen. Intuitiv könnte w_j die Wichtigkeit des Jobs reflektieren, welchen Nutzer j mit Hilfe der zentralen Ressource auszuführen gedenkt. Oder w_j ist der Geldbetrag, den Nutzer j für die Zuteilung der Ressource zu zahlen bereit ist. Dies führt uns zu folgendem Problem mit dem Namen „Intervall–Scheduling mit Gewichten“.

Eingabe: eine Kollektion $R = (R_j)_{j=1,\ldots,n}$ von Intervallen der Form $R_j = [a_j, b_j)$ mit $0 \leq a_j < b_j$ sowie eine entsprechende Kollektion $(w_j)_{j=1,\ldots,n}$ von natürlich-zahligen Gewichten.

Aufgabe: Auffinden einer Menge $I \subseteq [n]$, welche unter der Nebenbedingung

$$\forall i \neq j \in I : R_i \cap R_j = \emptyset \tag{8.1}$$

ein maximales Gesamtgewicht

$$w(I) = \sum_{i \in I} w_i$$

aufweist.

Offensichtlich kollabiert das allgemeinere Problem zu dem uns von früher bekannten Problem, wenn wir alle Gewichtsparameter auf den Wert 1 setzen, d.h., „Intervall–Scheduling mit Einheitsgewichten“ ist dasselbe wie „Intervall–Scheduling“.

Wie wir wissen, lässt sich Intervall–Scheduling optimal lösen mit Hilfe folgender „gierigen Regel“ zur Auswahl von in I aufzunehmenden Intervallen:
Wähle als nächstes Intervall stets dasjenige mit der kleinstmöglichen Endzeit aus.[1]
Dass dies bei der Variante mit Gewichten i.A. nicht optimal ist, zeigt folgendes Beispiel:

$$R_1 = [1,3), R_2 = [2,5), R_3 = [4,6), w_1 = 1, w_2 = 3, w_3 = 1 \ .$$

Die gierige Auswahlregel nimmt zunächst 1 in I auf (was 2 zur Aufnahme in I disqualifiziert) und dann 3, d.h., sie führt zur Lösung $I = \{1,3\}$ mit Gesamtgewicht 2. Optimal wäre die Lösung $I^* = \{2\}$ mit Gesamtgewicht 3.

Wir nehmen im Folgenden an, dass die Intervalle $R_j = [a_j, b_j)$ aufsteigend nach dem Schlüssel b_j sortiert sind. Zudem nummerieren wir die Intervalle so um, dass anschließend die Bedingung $b_1 \leq b_2 \leq \ldots \leq b_n$ gilt. Sortieren plus Umnummerieren beansprucht Rechenzeit $O(n \log n)$.

Die folgende Abbildung $j \mapsto p(j)$ wird bei der Lösung unseres Problems eine zentrale Rolle spielen:
Es sei $p(j)$ der größte Index i mit $b_i \leq a_j$ (bzw. $p(j) = 0$, falls kein solcher Index i existiert).
Diese Definition impliziert, dass $p(j)$ kleiner als j ist und dass die Intervalle mit den Indizes $p(j)+1, \ldots, j-1$ sich mit R_j überlappen. Die Zahl $p(j)$ ist zudem gleich der Anzahl der Intervalle, die vor (oder genau zum) Zeitpunkt a_j bereits beendet sind.

Beispiel 8.0.1 *Für*

$$R_1 = [1,4), R_2 = [2,6), R_3 = [5,7), R_4 = [3,10), R_5 = [8,11), R_6 = [9,12)$$

gilt

$$p(1) = 0, p(2) = 0, p(3) = 1, p(4) = 0, p(5) = 3, p(6) = 3 \ .$$

[1] natürlich unter Beachtung der Nebenbedingung (8.1)

Wir werden später nachweisen, dass sich die $p(j)$-Werte in Zeit $O(n \log n)$ berechnen lassen (bzw. sogar in Linearzeit, wenn vorher die Daten geeignet sortiert wurden). S. Lemma 8.0.4 weiter unten.

Es sei I^* eine optimale Lösung unseres Problems. Obwohl wir momentan keine Ahnung haben, wie I^* aussieht, lassen sich folgende Beobachtungen machen:

1. Falls $n \in I^*$, dann folgt $p(n)+1, \ldots, n-1 \notin I^*$ und I^* besteht aus $\{n\}$ vereinigt mit der optimalen Lösung für das Teilproblem zur Kollektion $R_1, \ldots, R_{p(n)}$.

2. Falls $n \notin I^*$, dann ist I^* identisch zu der optimalen Lösung für das Teilproblem zur Kollektion $R_1, \ldots, R_{n-1}$.

Im Folgenden bezeichne $\mathrm{OPT}(j)$ das Gesamtgewicht der optimalen Lösung für das Teilproblem zur Kollektion $R_1, \ldots, R_j$, wobei wir $\mathrm{OPT}(0) = 0$ setzen. Offensichtlich gilt folgende Rekursion:

1. $\mathrm{OPT}(0) = 0$.

2. $\mathrm{OPT}(n) = \max\{w_n + \mathrm{OPT}(p(n)) \ , \ \mathrm{OPT}(n-1)\}$.

Der erste (bzw. zweite) Term in dem Max-Ausdruck steht für die Festlegung n in I aufzunehmen (bzw. nicht aufzunehmen).

Die obige Rekursion könnte uns auf die Idee bringen, das Problem rekursiv zu lösen. Dies führt leider zu einem Algorithmus mit exponentieller Laufzeit (im worstcase):

Beispiel 8.0.2 *Für $j = 1, \ldots, n$ sei $R_j = [2j-2, 2j+1)$. Dann führt der rekursive Aufruf mit einem Parameter $j \geq 2$ zu zwei rekursiven Aufrufen: einer mit Parameter $j-2$ und einer mit Parameter $j-1$. Die Anzahl der Knoten in dem aus Aufruf OPT(n) resultierenden Rekursionsbaum beträgt daher $F(n)$: das n-te Glied der Fibonacci-Folge. Wie wir wissen hat die Folge $(F_n)_{n\geq 0}$ eine exponentielle Wachstumsrate.*

Das Problem mit der rekursiven Prozedur ist, dass der Aufruf für ein und denselben Parameter j vielfach getätigt wird.[2] Eine Technik, dies zu vermeiden, besteht in Rekursion plus „Memorisation“:

[2]Bei MergeSort und anderen erfolgreichen „Divide & Conquer“-Verfahren tritt dieses Problem *nicht* auf, weil diese Verfahren ein Problem in *disjunkte* Teilprobleme zerlegen.

Beim ersten rekursiven Aufruf mit Parameter j wird das Ergebnis tabelliert. Vor jedem Aufruf wird kontrolliert, ob bereits eine Tabellierung vorliegt und, falls dem so ist, erfolgt ein „Table-Lookup“ anstelle des rekursiven Aufrufes.

Es ist jedoch konzeptionell einfacher, den „Top-Down“-Ansatz der Rekursion durch einen „bottom-up“ Ansatz zu ersetzen:
Man beginnt mit Teilproblemen einer trivialen Größe und kämpft sich zu immer größeren Teilproblemen durch. Jedes Ergebnis wird tabelliert. Beim Betrachten eines Teilproblemes kann man daher stets auf die Lösungen noch kleinerer Teilprobleme zurückgreifen.
Diesen Ansatz, bekannt unter dem Namen „dynamische Programierung“, werden wir nun auf unser Problem der Berechnung der OPT(j)-Werte anwenden. Dabei setzen wir voraus, dass die $p(j)$-Werte bereits berechnet wurden und in einem Array $p[1:n]$ zur Verfügung stehen. Die Tabelle (= Array) $OPT[0:n]$ ergibt sich dann in Linearzeit wie folgt:

1. OPT[0] $\leftarrow 0$.

2. Für $j = 1, \ldots, n$: OPT[j] $\leftarrow \max\{w_j + \text{OPT}[p[j]]\ ,\ \text{OPT}[j-1]\}$.

Der Wert OPT[n] ist dann identisch zum Gesamtgewicht $w(I^*)$ einer optimalen Lösung I^*.

Freilich sind wir nicht nur an dem Gesamtgewicht OPT[n] $= w(I^*)$ einer optimalen Lösung interessiert, sondern auch an der optimalen Lösung selbst. Dazu müssen wir lediglich in obigem Rechenschema mit Hilfe eines Booleschen Arrays $B[1:n]$ protokollieren, welcher der beiden Vergleichsterme mit dem Maximum übereinstimmt. Genau dann, wenn dies der erste von beiden Termen ist, setzen wir $B[j]$ auf TRUE. Danach durchlaufen wir die Indizes $j \in [n]$ in absteigender Reihenfolge, beginnend bei $j = n$. Im Falle $B[n] =$ TRUE müssen wir n in I^* aufnehmen und mit dem Index $j = p(n)$ (in der gleichen Weise) weitermachen. Im Falle $B[n] =$ FALSE nehmen wir n nicht in I^* auf und machen mit dem Index $n-1$ (inder gleichen Weise) weiter. Ein entsprechend erweitertes Rechenschema liest sich wie folgt:

1. OPT[0] $\leftarrow 0$.

2. Für $j = 1, \ldots, n$:
   Falls $w_j + \text{OPT}[p[j]] \geq \text{OPT}[j-1]$:
   OPT[j] $\leftarrow w_j + \text{OPT}[p[j]]$; $B[j] \leftarrow$ TRUE.
   Andernfalls:
   OPT[j] $\leftarrow$ OPT[$j-1$]; $B[j] \leftarrow$ FALSE.

3. $I^* \leftarrow \emptyset$; $j \leftarrow n$.

4. Solange $j \neq 0$ mache Folgendes:
   Falls $B[j]$: $I^* \leftarrow I^* \cup \{j\}$; $j \leftarrow p(j)$.
   Andernfalls: $j \leftarrow j - 1$.

I^* wird dabei als Liste verwaltet. Offensichtlich beansprucht diese Konstruktion von I^* lediglich Linearzeit (also Zeit $O(n)$).

Fassen wir das Hauptergebnis dieses Abschnittes zusammen:

Satz 8.0.3 *Das Problem „Intervall–Scheduling mit Gewichten“ ist in Rechenzeit $O(n \log n)$ optimal lösbar. Genauer: wenn die Hilfswerte $p(1), \ldots, p(n)$ bereits vorliegen und die Intervalle bereits nach aufsteigenden Endzeiten sortiert sind, so genügt Linearzeit.*

Bleibt uns noch die Pflicht darzulegen, wie die Hilfswerte $p(j)$ in Zeit $O(n \log n)$ berechnet werden können:

Lemma 8.0.4 *Betrachte die Tripelmenge*

$$T = \{(1, a_1, 1), \ldots, (n, a_n, 1)\} \cup \{(1, b_1, 0), \ldots, (n, b_n, 0)\} \ ,$$

wobei wir $(a_j, 1)$ bzw. $(b_j, 0)$ als Schlüsselwert des Tripels $(j, a_j, 1)$ bzw. $(j, b_j, 0)$ betrachten. Es sei L eine nach diesen Schlüsselwerten aufsteigend sortierte Liste aller Tripel aus T. Dann gilt: aus L lassen sich die Werte $p(1), \ldots, p(n)$ in Linearzeit berechnen.

Beweis Das Rechenschema ist wie folgt:

1. $p \leftarrow 0$.

2. Solange L nicht vollständig abgearbeitet ist, mache Folgendes:

   (a) Entnimm L das nächste Tripel, sagen wir (j, c_j, β).

   (b) Falls $\beta = 1$ (und somit $c_j = a_j$): $p(j) \leftarrow p$.
   Andernfalls (also falls $\beta = 0$ und somit $c_j = b_j$): $p \leftarrow p + 1$.

Die Variable p zählt, wieviele Intervalle aktuell schon beendet sind. Dem Index j wird ein p-Wert zugewiesen, wenn wir in der Liste L den Zeitpunkt a_j erreicht haben. Somit wird dem Index j über $p(j) \leftarrow p$ stets der richtige Wert zugewiesen. •

Da das Sortieren der Tripel aus T nur Zeit $O(n \log n)$ beansprucht, ergibt sich die

Folgerung 8.0.5 *Die Hilfswerte $p(1), \ldots, p(n)$ sind aus den Eingabedaten des Problems „Intervall–Scheduling mit Gewichten“ in Zeit $O(n \log n)$ berechenbar.*

Kapitel 9

Minimale Spannbäume

Ein Graph $G = (V, E)$ heißt *zusammenhängend*, wenn es für zwei beliebige Knoten $u, v \in V$ in G stets einen Pfad von u nach v gibt. Ein *Baum* ist ein kreisloser zusammenhängender Graph. Ein nicht zusammenhängender Graph zerfällt (auf die offensichtliche Weise) in maximale zusammenhängende Komponenten.

Ein *Spannbaum* für einen zusammenhängenden Graphen $G = (V, E)$ ist ein Baum mit der Knotenmenge V und einer Kantenmenge $E' \subseteq E$. Wenn $n = |V|$ die Anzahl der Knoten bezeichnet, dann hat jeder zusammenhängende Teilgraph von G mindestens $n-1$ und jeder Spannbaum für G genau $n-1$ Kanten (wie man sich leicht überlegen kann), d.h. ein Spannbaum für G ist ein kleinster zusammenhängender Teilgraph von G, der die volle Knotenmenge V umfasst.

Ein Wald ist eine knotendisjunkte Kollektion von Bäumen. Ein *Spannwald* für einen Graphen $G = (V, E)$ ist ein Wald mit der Knotenmenge V und einer Kantenmenge $E' \subseteq E$.

Beim Aufbau eines möglichst kostengünstigen zusammenhängenden Netzwerkes zwischen n gegebenen Knotenpunkten hat man es mit dem folgenden sogenannten MST-Problem[1] zu tun:

Eingabe: Ein zusammenhängender Graph $G = (V, E)$ mit Kantenkosten $w(e)$ für jede Kante $e \in E$.

Aufgabe: Berechne einen *minimalen Spannbaum* für G, d.h., berechne eine Kantenmenge $E' \subseteq E$, sodass $T = (V, E')$ ein Spannbaum für G mit den kleinstmöglichen Gesamtkosten $w(T) = \sum_{e \in E'} w(e)$ ist.

[1] MST = Minimum Spanning Tree.

Dabei repräsentiert $w(e)$ mit $e = \{i, j\}$ die Kosten, eine Punkt–zu–Punkt Verbindung zwischen den Knoten i und j herzustellen. Wir diskutieren im Folgenden drei Algorithmen zur Lösung des MST-Problems:

Kruskal's Algorithmus: Initialisiere E' als leere Menge. Inspiziere danach die Kanten, eine nach der anderen, in der Reihenfolge aufsteigender Kantenkosten (also billigere Kanten zuerst). Für jede Kante $e = \{i, j\}$ teste, ob i und j zu verschiedenen (Zusammenhangs-)Komponenten von (V, E') gehören. Falls dem so ist, nimm e in E' auf. Nach Inspektion aller Kanten gib E' bzw. $T = (V, E')$ als Spannbaum von G aus. Zur Illustration s. Teil (a) der Abb. 9.1: die Nummerierung der Kanten zeigt die Reihenfolge an, in der Kruskal's Algorithmus sie in den minimalen Spannbaum einfügt.

Prim's Algorithmus: Wähle aus V einen Startknoten s aus. Initialisiere eine Knotenmenge S mit $\{s\}$ und E' als leere Menge. Solange $S \neq V$, mache Folgendes. Wähle aus $S \times (V \setminus S)$ eine billigste Kante $e = (i, j)$ aus. Nimm e in E' und j in S auf. Nachdem die obige Schleife wegen der Abbruchbedingung $S = V$ verlassen wurde, gib E' bzw. (V, E') als Spannbaum von G aus. Zur Illustration s. Teil (b) der Abb. 9.1: die Nummerierung der Kanten zeigt die Reihenfolge an, in der Prim's Algorithmus sie in den minimalen Spannbaum einfügt.

Reverse-Delete Algorithmus: Initialisiere E' mit E. Inspiziere danach die Kanten, eine nach der anderen, in der Reihenfolge absteigender Kantenkosten (also teurere Kanten zuerst). Für jede Kante $e = \{i, j\}$ teste, ob der Teilgraph (V, E') nach Wegnahme von e aus E' immer noch zusammenhängend ist. Falls dem so ist, entferne e aus E'. Nach Inspektion aller Kanten gib E' bzw. $T = (V, E')$ als Spannbaum von G aus.

Lemma 9.0.1 *Jeder der drei Algorithmen konstruiert einen Spannbaum für G.*

Beweis Beim Algorithmus von Kruskal gilt für $T = (V, E')$ zu jedem Zeitpunkt: T ist ein Spannwald für G und, falls T aus mehr als einem Baum besteht, so ist noch keine Kante aus E, welche zwei dieser Bäume verbindet, bisher inspiziert worden. Diese Invarianzbedingung lässt sich leicht mit

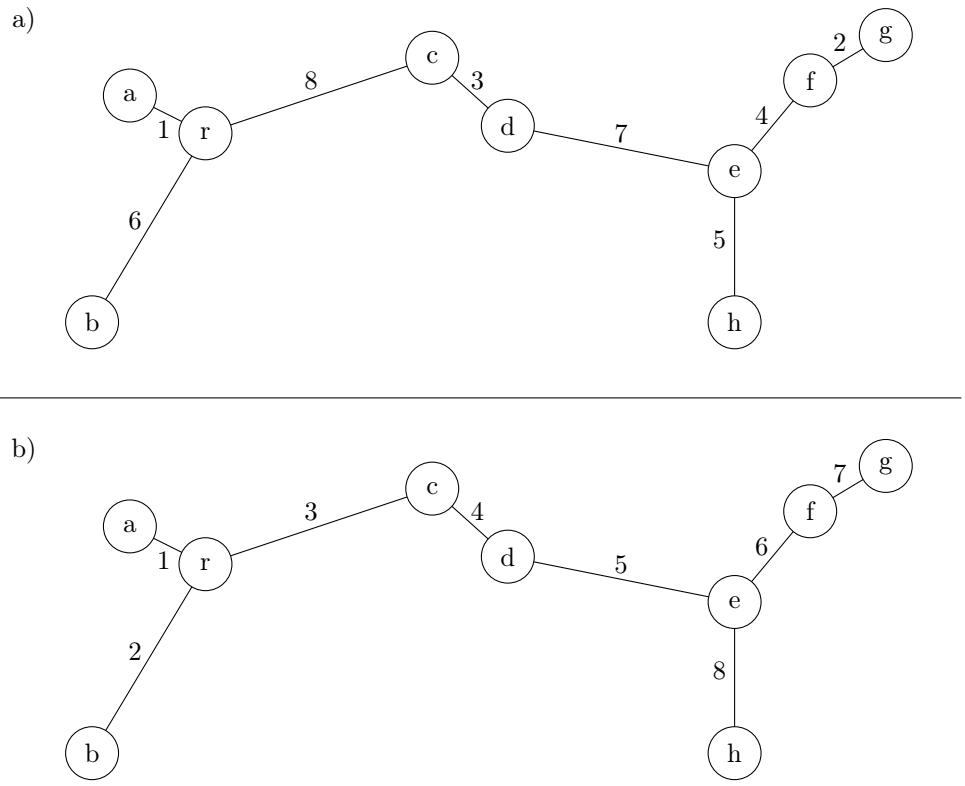


Abbildung 9.1: der minimale Spannbaum zum vollständigen Graphen mit der Euklidischen Distanz als Kantenkostenfunktion.

vollständiger Induktion verifizieren. Es folgt direkt, dass T nach Inspektion aller Kanten aus E ein Spannwald mit nur einem Baum (also ein Spannbaum) ist.
Beim Algorithmus von Prim gilt zu jedem Zeitpunkt: $T = (S, E')$ ist ein Baum und $E' \subseteq E$. Diese Invarianzbedingung lässt sich leicht mit vollständiger Induktion verifizieren. Es folgt direkt, dass $T = (S, E')$ nach Erreichen der Abbruchbedingung $S = V$ ein Spannbaum für G ist.
Beim Reverse-Delete Algorithmus gilt für $T = (V, E')$ zu jedem Zeitpunkt: (V, E') ist zusammenhängend und, falls T einen Kreis enthält, dann ist bisher noch keine Kante dieses Kreises inspiziert worden. Diese Invarianzbedingung lässt sich leicht mit vollständiger Induktion verifizieren. Es folgt direkt, dass $T = (S, E')$ nach Inspektion aller Kanten ein Spannbaum für G ist. •

Lemma 9.0.2 *Nimm an, dass die Kantenkosten für verschiedene Kanten verschieden sind. Es sei* $\emptyset \subset S \subset V$ *und* $e \in E \cap (S \times (V \setminus S))$ *sei die billigste Kante aus* E*, die einen Knoten aus* S *mit einem Knoten aus* $V \setminus S$ *verbindet. Dann gilt: jeder minimale Spannbaum enthält die Kante* e*.*

Beweis Zur Illustration der folgenden Ausführungen s. Abb. 9.2. Wir führen den Beweis indirekt, d.h., wir gehen aus von einem Spannbaum $T = (V, E')$ mit $e \notin E'$ und zeigen, dass T kein minimaler Spannbaum sein kann. Die Kante e verbindet einen Knoten $v \in S$ mit einem Knoten $w \in V \setminus S$. Es sei P der Pfad in T, welcher v mit w verbindet. Da P in $v \in S$ startet und in $w \in V \setminus S$ endet, muss er eine Kante $e' \in E \cap (S \times (V \setminus S))$ enthalten. Wenn wir in T die Kante e' durch e ersetzen, erhalten wir einen Baum, der billiger als T ist. Somit ist T kein minimaler Spannbaum. •

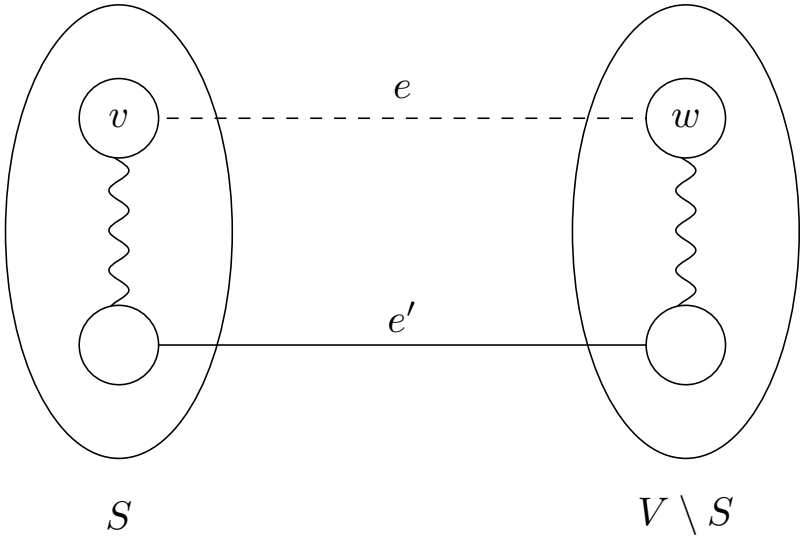


Abbildung 9.2: Ausschnitt aus dem Baum T, der die (gestrichelt gezeichnete) Kante $e = \{v, w\}$ nicht enthält, wohl aber den (geschlängelt gezeichneten) Baumpfad von v nach w, welcher eine Kante e' enthalten muss, die S mit $V \setminus S$ verbindet.

Lemma 9.0.3 *Nimm an, dass die Kantenkosten für verschiedene Kanten verschieden sind. Es sei C ein Kreis in G und e die teuerste Kante in C. Dann gilt: e ist in keinem minimalen Spannbaum enthalten.*

Beweis Wir führen den Beweis indirekt, d.h., wie gehen aus von einem Spannbaum $T = (V, E')$ mit $e \in E'$ und zeigen, dass T kein minimaler Spannbaum sein kann. Wenn wir $e = (v, w)$ aus E' entfernen, dann zerfällt T in zwei Komponenten, sagen wir T_1 und T_2. Der Kreis C besteht aus der Kante $e = (v, w)$ und einem Pfad P, der von w zurück nach v führt. P muss eine Kante e' enthalten, die die Komponenten T_1 und T_2 miteinander verbindet. Wenn wir in T die Kante e durch e' ersetzen, erhalten wir einen Baum, der billiger als T ist. Somit ist T kein minimaler Spannbaum. •

Satz 9.0.4 *Die Algorithmen von Kruskal und Prim sowie der Reverse-Delete Algorithmus berechnen alle einen minimalen Spannbaum. Darüberhinaus gilt: falls die Kantenkosten für verschiedene Kanten verschieden sind, so ist der minimale Spannbaum eindeutig bestimmt (sodass alle drei Algorithmen denselben Baum berechnen).*

Beweis Wir führen den Beweis zunächst unter der Voraussetzung, dass die Kosten für verschiedene Kanten verschieden sind und skizzieren am Ende, wie sich der Beweis ohne diese Voraussetzung aufrecht erhalten lässt.
Beginnen wir mit dem Algorithmus von Prim. Hier ist unmittelbar klar, dass der Algorithmus nur Kanten in E' aufnimmt, welche gemäß Lemma 9.0.2 zu jedem minimalen Spannbaum gehören. Da wir bereits wissen, dass der Algorithmus einen Spannbaum konstruiert, muss dieser der minimale sein.
Kommen wir nun zum Algorithmus von Kruskal und betrachten den Moment, in welchem eine Kante $e = (i, j)$ inspiziert wird, deren Knoten i, j in verschiedenen Komponenten von (V, E') liegen. Es sei S die Komponente mit $i \in S$. Die Invarianzbedingung (s. Beweis von Lemma 9.0.1) zu Kruskal's Algorithmus besagt, dass bisher noch keine Kante aus $E \cap (S \times (V \setminus S))$ inspiziert wurde. Daher ist die aktuell inspizierte Kante $e = (i, j)$ die billigste solche Kante. Mit Lemma 9.0.2 ergibt sich, dass Kruskal's Algorithmus (wie auch zuvor schon Prim's Algorithmus) nur Kanten in E' aufnimmt, welche zu jedem minimalen Spannbaum gehören. Hieraus ergibt sich die Minimalität des von Kruskal's Algorithmus konstruierten Spannbaumes.

In einer ähnlichen Weise kann mit Hilfe der Lemmas 9.0.3 und 9.0.1 gefolgert werden, dass der Reverse-Delete Algorithmus 1. nur Kanten aus $E' = E$ eliminiert, welche zu keinem minimalen Spannbaum gehören und 2. der von diesem Algorithmus berechnete Spannbaum minimal sein muss.
Aus dem bisher geführten Beweis (unter der Annahme paarweise verschiedener Kantenkosten) ergibt sich leicht, dass es nur einen minimalen Spannbaum gibt. Bleibt noch zu skizzieren, wie wir den Beweis erweitern, wenn die Kosten für verschiedene Kanten nicht notwendig verschieden sind. Es sei A einer der drei betrachteten Algorithmen. Es ist nicht schwer zu sehen, dass die Kostenwerte $w(\cdot)$ der Kanten infinitesimal[2] zu Werten $\hat{w}(\cdot)$ verändert werden können, sodass Folgendes gilt;

1. $\hat{w}$ ordnet verschiedenen Kanten verschiedene Kostenwerte zu.

2. Der bezüglich $\hat{w}$ eindeutige minimale Spannbaum ist identisch mit dem von A konstruierten Baum.

3. Die Differenz der $\hat{w}$- von den w-Werten ist so klein, dass aus
   „Spannbaum T_1 ist echt billiger als Spannbaum T_2 bezüglich w“
   stets folgt, dass T_1 auch bezüglich $\hat{w}$ billiger ist als T_2.

Dies impliziert dann, dass der eindeutige bezüglich $\hat{w}$ minimale Spannbaum auch bezüglich w ein minimaler Spannbaum ist (wenn auch i.A. nicht der Einzige). •

Wir nehmen im Folgenden an, dass der Graph $G = (V, E)$ in Adjazenzlistendarstellung gegeben ist. $N(u)$ bezeichnet die Nachbarschaftsliste zum Knoten u. In der Liste $N(u)$ findet sich beim Eintrag zu $v \in N(u)$ auch das Gewicht der Kante $e = \{u, v\}$, welches wir aber weiterhin als $w(e) = w(u, v)$ notieren.

Der Algorithmus von Kruskal liefert eine schöne Anwendung für die Datenstrukturen „Priority Queue“ und „Union-Find“:

1. Initialisiere eine Liste E' als leere Menge.

2. Initialisiere eine Priority Queue Q mit der Kantenmenge E und den Schlüsselwerten $w(e)$ für $e \in E$.

[2] also um einen Wert nahe 0

3. Für $V = \{v_1, \ldots, v_n\}$ initialisiere die Union-Find Datenstruktur mit der Partition $\{v_1\}, \ldots, \{v_n\}$.

4. Solange Q nicht leer ist, mache Folgendes:

   (a) Wähle eine Kante $e = (i, j)$ mit minimalen Kosten aus Q aus und setze $Q \leftarrow Q \setminus \{e\}$.

   (b) Falls Find$(i) \neq$ Find(j): Union(i, j, i); $E' \leftarrow E' \cup \{e\}$.

5. Gib E' bzw. $T = (V, E')$ als Spannbaum von G aus.

Die Rechenzeit wird dominiert durch die zur Verwaltung von Q benötigten Zeit $O(m \log n)$ (für jeweils m INSERT-, m MIN- und m DELETE_MIN-Operationen). Es genügt daher die Union-Find-Implementierung mit Listen (oder mit Bäumen aber ohne Pfadkompression) zu verwenden, sodass der Gesamtaufwand für $2m$ Find- und $n - 1$ Union-Operationen sich ebenfalls mit $O(m \log n)$ nach oben abschätzen lässt. Es ergibt sich der

Satz 9.0.5 *Kruskal's Algorithmus kann in Zeit $O(m \log n)$ implementiert werden.*

Es kommt vor, dass eine Priority Queue Q neben den Operationen INSERT, MIN und DELETE_MIN auch eine Operation DECREASE_KEY unterstützen soll:

- Ein Element $v \in Q$ hat einen neuen Schlüsselwert zugewiesen bekommen, welcher kleiner ist als der vorangehende.

- Q soll auch mit dem neuen Schlüsselwert für v als Priority Queue organisiert sein.

Wenn wir Q als MIN_HEAP implementieren, dann lässt sich DECREASE_KEY leicht realisieren: wir lassen v solange aufsteigen, bis er seine richtige Position in Q erreicht hat. Eine analoge Bemerkung gilt natürlich für INCREASE_KEY, angewendet auf $v \in Q$, und das Einsinken lassen von v.

Der Algorithmus von Prim hat bei geeigneter Implementierung dieselbe Laufzeit wie der Algorithmus von Kruskal, wie wir im Folgenden erarbeiten werden. Die Knoten aus V identifizieren wir mit den Nummern von 1 bis n. Als Startknoten wählen wir $s = 1$ aus. Unsere Implementierung von Prim's Algorithmus verwendet darüberhinaus folgende Datenstrukturen:

- einen Zähler z mit $z = |S|$.
- ein Array $B[1:n]$ mit $B[v] = \text{TRUE}$ genau dann wenn $v \in V \setminus S$.
- ein Array $D[1:n]$, wobei für alle $v \in V \setminus S$ die Bedingung $D[v] = \min\{w(u,v) | u \in S, \{u,v\} \in E\}$ erfüllt ist (bzw. $D[v] = \infty$, falls kein $u \in S$ mit $\{u,v\} \in E$ existiert).
- ein Array $P[1:n]$ mit $P[v] = u$ für den Knoten u mit $D[v] = w(u,v)$ bzw. $P[v] = 0$, falls $D[v] = \infty$.
- eine Priority Queue Q, welche die Knoten $v \in V \setminus S$ mit den Schlüsselwerten $D[v]$ enthält.

Das Array B gibt an, welche Knoten aktuell zu $V \setminus S$ gehören. Wenn u der Knoten in Q mit minimalem Schlüsselwert ist, dann ist $\{u, P(u)\}$ die aktuell billigste Kante, die einen Randknoten in S und den anderen in $V \setminus S$ hat. In dem folgenden Pseudocode werden die Datenstrukturen so initialisiert und fortlaufend aktualisiert, dass sie obiger Beschreibung entsprechen:

1. $s \leftarrow 1$; $B[s] \leftarrow \text{FALSE}$; $z \leftarrow 1$; $E' \leftarrow \emptyset$.
2. Für $v = 2, \ldots, n$: $B[v] = \text{TRUE}$; $D[v] \leftarrow \infty$; $P[v] = 0$.
3. Für alle $v \in N(s)$: $D[v] \leftarrow w(s,v)$; $P[v] \leftarrow s$.
4. Initialisiere Q als Priority Queue mit den Elementen $v \in V \setminus \{s\}$ und deren Schlüsselwerten $D[v]$.
5. Solange $z < n$, mache Folgendes:
   (a) Entnimm Q das Element u mit minimalem Schlüsselwert (MIN und DELETE_MIN).
   (b) $z \leftarrow z + 1$; $B[u] \leftarrow \text{FALSE}$; $e \leftarrow \{P[u], u\}$; $E' \leftarrow E' \cup \{e\}$.
   (c) Für alle $v \in N(u)$ mit $B[v] = \text{TRUE}$:
   Falls $w(u,v) < D[v]$: $P[v] \leftarrow u$; $D[v] \leftarrow w(u,v)$
   (letzteres als eine auf Q anzuwendende
   DECREASE_KEY-Operation)
6. Gib E' bzw. $T = (V, E')$ als Spannbaum von G aus.

Wie wir in der Vorlesung näher ausführen werden, wird die Laufzeit dominiert durch die DECREASE_KEY-Operationen in Schritt 5(c). Jede Kante aus E löst eine solche Operation aus. Die Rechenzeit beträgt daher $O(m \log n)$ und wir gelangen zu folgendem Resultat:

Satz 9.0.6 *Prim's Algorithmus kann in Zeit $O(m \log n)$ implementiert werden.*

Kruskal's und Prim's Algorithmus geben der Einfachheit halber die Kantenmenge E' des minimalen Spannbaumes als Liste aus. Die Algorithmen lassen sich aber (ohne wesentlichen Effizienzverlust) so umschreiben, dass der Spannbaum $T = (V, E')$ in Adjazenzlistendarstellung ausgegeben wird.

Kapitel 10

Ein Clustering-Problem

Informell besteht ein k-Clustering-Problem darin, n gegebene Datenpunkte $p_1, \ldots, p_n$ so in k disjunkte Teilmengen, genannt Cluster, zu zerlegen, dass sich Punkte desselben Clusters möglichst ähnlich, und Punkte in verschiedenen Clustern möglichst unähnlich sind. Die Begriffe „ähnlich“ und „unähnlich“ können mit Hilfe einer Distanzfunktion $d(\cdot, \cdot)$ mit nicht-negativen Werten präzisiert werden: ein kleiner Wert von $d(p_i, p_j)$ spricht dann für eine große Ähnlichkeit von p_i und p_j. Die Wahl der Funktion d ist auf die jeweilige Anwendung zugeschnitten. In diesem Kapitel wollen wir lediglich voraussetzen, dass $d(p, p) = 0$ und $d(p, p') = d(p', p)$ ist, d.h. ein Punkt hat zu sich selber die Distanz 0 und die Distanz zwischen p und p' ist gleich der Distanz zwischen p' und p.

Es sei $V = \{p_1, \ldots, p_n\}$. Die Distanz zwischen zwei Punktmengen $C, C' \subseteq V$ definieren wir wie folgt:

$$d(C, C') = \min\{d(p, p') | p \in C, p' \in C'\} \ .$$

Bei einem k-Clustering $\mathcal{C} = (C_1, \ldots, C_k)$ der Punkte in V ist es wünschenswert, dass für alle Wahlen von $1 \leq i < j \leq n$ die Cluster C_i und C_j eine große Distanz zueinander aufweisen. Wir definieren daher (zur Illustration s. Abb. 10.1) den „garantierten Cluster-Abstand“ in $\mathcal{C}$ als

$$\rho(\mathcal{C}) = \min\{d(C_i, C_j) | 1 \leq i < j \leq n\}$$

und stellen uns die Aufgabe, ein k-Clustering mit einem möglichst großen garantierten Cluster-Abstand zu berechnen.

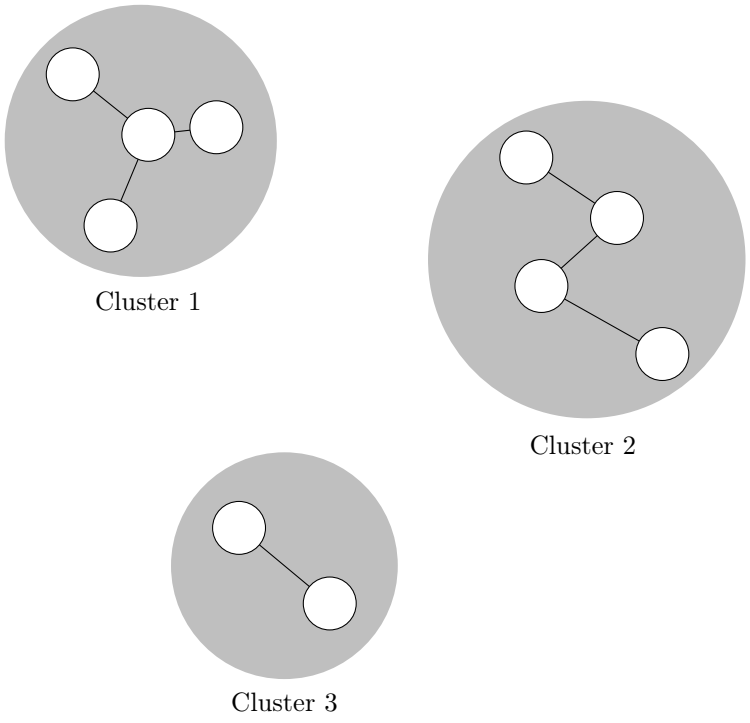


Abbildung 10.1: Ein 3-Clustering einer Punktmenge mit maximalem garantierten Cluster-Abstand (bezogen auf die Euklidische Distanz).

Im Folgenden sei $G = (V, E)$ der Graph mit Knotenmenge V und der Kantenmenge E bestehend aus allen $\binom{n}{2}$ Mengen $\{p_i, p_j\}$ mit $1 \leq i < j \leq n$. Wir ordnen einer Kante $\{p, p'\}$ die Kosten $d(p, p')$ zu. Im Folgenden spielt d die Rolle der Funktion w beim Kapitel über das MST-Problem. Der folgende Satz setzt das MST-Problem in Beziehung zu dem hier diskutierten Clustering-Problem:

Satz 10.0.1 *Es sei $\mathcal{C}^*$ das k-Clustering, welches aus dem minimalen Spannbaum T für G bezüglich d entsteht, indem wir aus T die $k-1$ teuersten Kanten löschen. Dann ist $\mathcal{C}^*$ ein k-Clustering mit einem maximalen garantierten Cluster-Abstand.*

Beweis Es seien $e_1, \ldots, e_{k-1}$ die $k-1$ teuersten Kanten aus T, sagen wir mit Kosten $w_1 \geq \ldots \geq w_{k-1}$. $T_1, \ldots, T_k$ seien die Bäume, die aus T durch

Löschen der Kanten $e_1, \ldots, e_{k-1}$ hervorgehen. Aus der Arbeitsweise von Kruskal's Algorithmus folgt, dass e_{k-1} die billigste Kante ist, die zwei der Bäume $T_1, \ldots, T_k$ miteinander verbindet. D.h., dass die von diesen Bäumen repräsentierten Cluster den garantierten Abstand w_{k-1} haben. Es sei $\mathcal{C} = (C_1, \ldots, C_k)$ ein von $\mathcal{C}^*$ verschiedenes k-Clustering. Wir müssen die Bedingung $\rho(\mathcal{C}) \leq w_{k-1}$ nachweisen. Da $\mathcal{C}$ und $\mathcal{C}^*$ verschiedene Partitionen von V sind, muss es ein Cluster T_ℓ geben, dessen Knoten sich auf zwei (oder mehr) Cluster aus $\mathcal{C}$ verteilen. Daher muss es in T_ℓ eine Kante $e = \{p, p'\}$ geben, deren Randknoten verschiedenen Clustern $C_r \neq C_s \in \mathcal{C}$ angehören. Zur Illustration s. Abb. 10.2. Da Kruskal's Algorithmus billigere Kanten vor teureren inspiziert, müssen die Kosten $d(p, p')$ durch w_{k-1} nach oben beschränkt sein. Somit gilt

$$\rho(\mathcal{C}) \leq d(C_r, C_s) \leq d(p, p') \leq w_{k-1} \ ,$$

was den Beweis abschließt. •

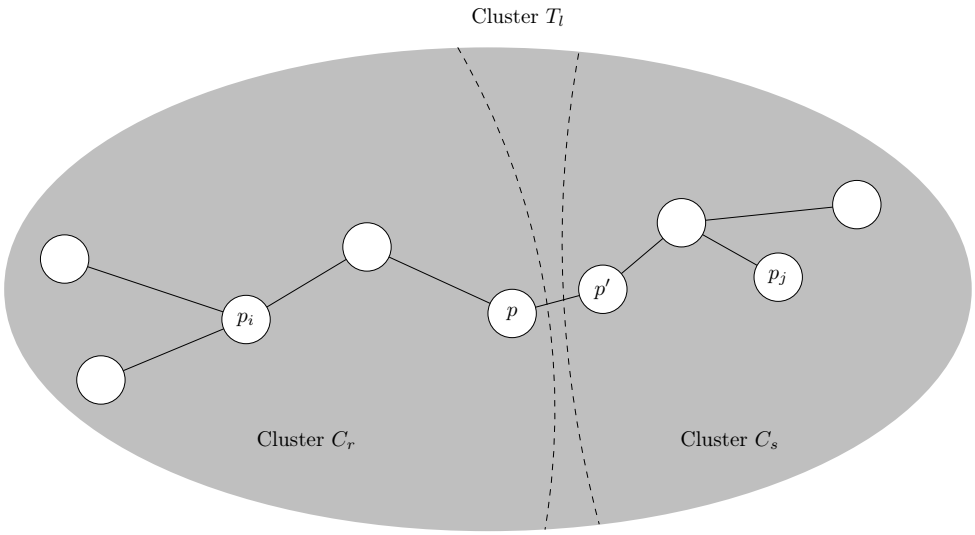


Abbildung 10.2: Eine Kante $\{p, p'\}$ in T_ℓ, die verschiedene Cluster in $\mathcal{C}$ miteinander verbindet.

Die $k-1$ teuersten Kanten im minimalen Spannbaum $T = (V, E')$ sind natürlich die $k-1$ Kanten, die von Kruskal's Algorithmus zum Schluss in E' eingefügt werden. Anstatt diese Kanten einzufügen, um sie dann gleich wieder zu entfernen, ist es gescheiter, Kruskal's Algorithmus vorzeitig zu stoppen. Mit anderen Worten:

Folgerung 10.0.2 *Wenn wir den Algorithmus von Kruskal genau dann stoppen, wenn der aktuelle Spannwald aus nur noch k Bäumen besteht, dann repräsentieren diese k Bäume ein k-Clustering mit einem maximalen garantierten Cluster-Abstand.*

Wie wir wissen, kann der Algorithmus von Kruskal so implementiert werden, dass er Rechenzeit $O(m \log n)$ benötigt. In unserer Anwendung hier hat der Graph $m = \binom{n}{2} = \theta(n^2)$ Kanten. Wir erhalten daher das folgende Ergebnis:

Folgerung 10.0.3 *Das in diesem Kapitel diskutierte k-Clustering-Problem ist in $O(n^2 \log n)$ Schritten lösbar.*

Kapitel 11

Bestimmung kürzester Pfade

In Abschnitt 11.1 betrachten wir das Problem, alle von einem festen Startknoten ausgehenden kürzesten Pfade (und ihre Längen) zu bestimmen und besprechen den Algorithmus von Dijkstra. In Abschnitt 11.2 betrachten wir das entsprechende Problem für alle Wahlen von Start- und Zielknoten und besprechen den Algorithmus von Floyd. In beiden Abschnitten ist ein Digraph mit nicht-negativen Kantenkosten gegeben. Die Länge $\ell(P)$ eines Pfades P ist dann definiert als die Summe der Gewichte auf den Kanten von P. Bei der Suche nach kürzesten Pfaden kann man sich bei nicht-negativen Kantenkosten stets auf kreisfreie Pfade beschränken. Aus diesem Grund setzen wir oBdA im gesamten Abschnitt 11 voraus, dass der gegebene Graph schleifenfrei ist (also keine Kanten der Form (v, v) enthält).

Den Algorithmus von Floyd stellen wir in diesem Skriptteil ohne illustrierendes Beispiel vor. In der Vorlesung wird das Skriptum aber (an der Tafel) durch weitere illustrierende Beispiele ergänzt werden.

11.1 Kürzeste Pfade mit festem Startknoten

Folgendes Berechnungsproblem ist im englischen Sprachraum unter dem Namen „Single Source Shortest Path“ bekannt:

Eingabe: Ein Digraph $G = (V, E)$ mit Kantenkosten $d(e) = d(u, v) \geq 0$ für jede Kante $e = (u, v) \in E$ sowie ein ausgezeichneter Startknoten $s \in V$.

Voraussetzung: s ist eine Wurzelknoten in G, d.h., von s aus ist jeder

andere Knoten mit einem Pfad in G erreichbar.

Aufgabe: Berechne zu jedem Knoten $v \in V$ einen kürzesten Pfad von s nach v sowie seine Länge. Letztere notieren wir als $D[v]$.

Wir konzentrieren uns als Erstes auf die Bestimmung der Werte $D[v]$ und beschreiben einen von Dijkstra stammenden Algorithmus. Dieser verwendet eine Menge S bereits explorierter Knoten. Anfangs wird S mit $\{s\}$ und $D[s]$ mit 0 initialisiert. Die D-Werte aller Nachbarn v von s, also aller Knoten $v \in N(s) = \{v \in V | (s,v) \in E\}$, werden mit $d(s,v)$ initialisiert. Die D -Werte der verbleibenden Knoten werden auf ∞ gesetzt. Solange $S \neq V$ (Hauptschleife) wird Folgendes gemacht:

- Es wird ein Knoten $u \in V \setminus S$ mit minimalem D-Wert ausgewählt und in S aufgenommen.
- Für alle Nachbarn v von u wird $D[v]$ gemäß

$$D[v] \leftarrow \min\{D[v] \ , \ D[u] + d(u,v)\} \tag{11.1}$$

  aktualisiert.

Beispiel 11.1.1 *Wir starten den Algorithmus von Dijkstra auf dem Digraphen, der (mitsamt der Kantenkosten) in Teil (a) von Abb. 11.1 zu sehen ist. Die folgende Tabelle zeigt, wie sich die Komponenten $S, u, D(u)$ sowie die weiteren D-Werte im Laufe der insgesamten 4 Iterationen der Haupschleife entwickeln:*

Iteration	S	u	$D[u]$	$D[2]$	$D[3]$	$D[4]$	$D[5]$
0	$\{1\}$	–	–	2	∞	∞	10
1	$\{1,2\}$	2	2	2	5	∞	9
2	$\{1,2,3\}$	3	5	2	5	9	9
3	$\{1,2,3,4\}$	4	9	2	5	9	9
4	$\{1,2,3,4,5\}$	5	9	2	5	9	9

Bevor wir Implementierungsdetails besprechen, wollen wir uns davon überzeugen, dass dieses Verfahren die D-Werte der Knoten korrekt berechnet. Zu diesem Zweck definieren wir S-Pfade und verifizieren anschließend Invarianzbedingungen zur Hauptschleife von Dijkstra's Algorithmus:

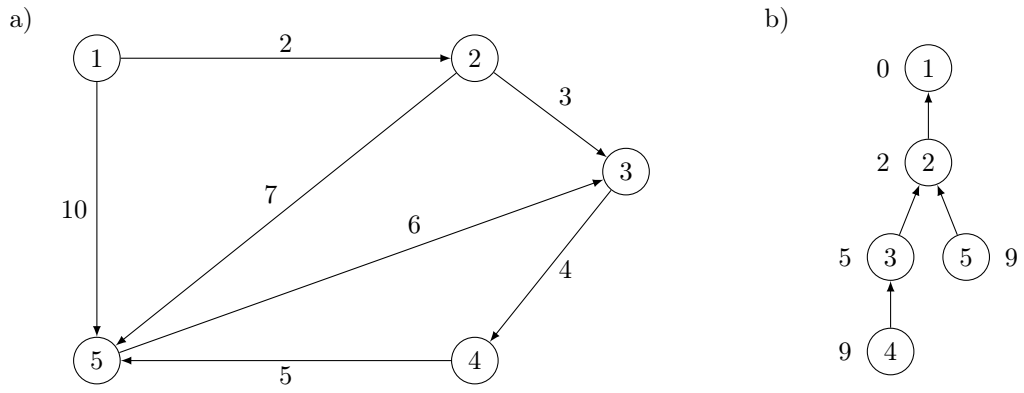


Abbildung 11.1: (a) Ein Digraph mit Kantenkosten (b) der zugehörige Kürzeste-Pfade-Baum (Knotenlabel = Länge des kürzesten Pfades).

Definition 11.1.2 *Es sei S eine echte Teilmenge von V, die den Startknoten s enthält. Es sei v ein Knoten in V. Dann ist ein* S-Pfad nach v *definiert als ein Pfad von s nach v, dessen Zwischenknoten alle in S liegen.*

Lemma 11.1.3 *Vor bzw. nach jedem Durchlauf der Hauptschleife von Dijkstra's Algorithmus gilt Folgendes:*

1. *Für jeden Knoten $v \in V \setminus S$ enthält $D[v]$ die Länge eines S-Pfades P_v nach v, der unter allen S-Pfaden nach v der kürzeste ist (bzw. $D[v] = \infty$, wenn kein S-Pfad nach v existiert).*

2. *Für jeden Knoten $v \in S$ enthält $D[v]$ die Länge eines S-Pfades P_v nach v, der unter allen Pfaden von s nach v der kürzeste ist.*

Beweis Nach der Initialisierung sind die obigen zwei Bedingungen offensichtlich erfüllt. Wir nehmen induktiv an, dass sie vor einem Durchlauf der Hauptschleife erfüllt sind und haben zu zeigen, dass sie nach dem nächsten Durchlauf immer noch gelten. Wir haben dabei zu berücksichtigen, dass der explorierte Teil S von V um einen Knoten u zu $S' = S \cup \{u\}$ erweitert und D gemäß (11.1) modifiziert wurde. Die modifizierte Version von D notieren wir im Beweis als D'. Der Induktionsbeweis wird komplett sein, wenn wir folgende Behauptungen bewiesen haben.

Behauptung 1: Zu jedem $v \in V$ existiert ein S'-Pfad P'_v nach v mit Länge $D'[v]$.

Behauptung 2: Es sei $v \in V \setminus S'$. Dann existiert kein S'-Pfad nach v, dessen Länge kleiner als $D'[v]$ ist.

Behauptung 3: Es sei $v \in S'$. Dann existiert kein Pfad von s nach v, dessen Länge kleiner als $D'[v]$ ist.

Nach Induktionsvoraussetzung gelten die drei Behauptungen mit D anstelle von D' und P_v anstelle von P'_v.
Wir beweisen nun die Behauptung 1. Falls $D'[v] = D[v]$, dann können wir einfach $P'_v = P_v$ setzen. Falls $D'[v] \neq D[v]$, dann gilt $D'[v] = D[u] + d(u, v) < D[v]$, und wir können P'_v wählen als die Komposition des Pfades P_u mit der Kante (u, v). Damit wäre die erste Behauptung bewiesen.
Weiter geht es mit einem (Widerspruchs-)Beweis der Behauptung 2. Wir machen die Annahme, es existiere ein S'-Pfad P nach v, dessen Länge $\ell(P)$ kleiner als $D'[v]$ ist. Es gilt $D'[v] = \min\{D[v], D[u] + d(u, v)\}$. Hieraus folgt 1. dass der neue Knoten u in dem Pfad P vorkommen muss (denn sonst wäre $\ell(P) \geq D[v]$) und 2. dass P nicht mit (u, v) endet (denn sonst wäre $\ell(P) \geq D[u] + d(u, v)$). Demnach muss P ein S'-Pfad der Form

$$P = s, \ldots, u, \ldots, u', v$$

mit $u' \neq u$ sein. Wegen $u' \in S' \setminus \{u\} = S$ muss es (nach Induktionsvoraussetzung) einen S-Pfad P' nach u' geben, der unter *allen* Pfaden von s nach u' der kürzeste ist. Es sei P^* der um die Kante (u', v) verlängerte Pfad P'. Dann gilt

$$D'[v] \leq D[v] \leq \ell(P^*) \leq \ell(P) \ ,$$

wobei die zweite Ungleichung induktiv geschlossen werden kann, weil P^* ein S-Pfad nach v ist. Es hat sich ein Widerspruch zu der Annahme $\ell(P) < D'[v]$ ergeben.
Der Beweis des Lemmas wird jetzt abgeschlossen mit dem Beweis der Behauptung 3. Für Knoten $v \in S$ folgt induktiv, dass kein Pfad von s nach v mit einer Länge kleiner als $D[v]$ existiert (und somit erst recht kein solcher Pfad einer Länge kleiner als $D'[v]$). Es genügt daher, Behauptung 3 für den neuen Knoten $u \in S'$ zu zeigen. Gemäß Induktionsvoraussetzung ist P_u ein S-Pfad nach u der Länge $D[u]$, der unter allen S-Pfaden nach u der kürzeste ist. Wir machen einen Widerspruchsbeweis und nehmen an, dass ein Pfad P von s nach u der Länge $\ell(P) < D[u]$ existiert. Da P kein S-Pfad sein kann, muss es auf P einen vom Zielknoten u verschiedenen Knoten aus $V \setminus S$ (und somit aus $V \setminus S'$) geben. Es bezeichne u' den frühesten solchen Knoten auf dem

Pfad P. Das Anfangsstück von P bis zum Knoten u' ist ein S-Pfad nach u'. Daher gilt $\ell(P) \geq D[u']$. Wegen der Wahl von u als Knoten aus $V \setminus S$ mit minimalem D-Wert folgt $D[u'] \geq D[u]$. Insgesamt hat sich $D[u] \leq D[u'] \leq \ell(P)$ ergeben, was im Widerspruch zu der Annahme $\ell(P) < D[u]$ steht. •

Da die Hauptschleife erst verlassen wird, wenn die Abbruchbedingung $S = V$ erfüllt ist, ergibt sich aus Lemma 11.1.3 direkt:

Folgerung 11.1.4 *Nach dem Terminieren von Dijkstra's Algorithmus gibt $D[v]$ für jeden Knoten $v \in V$ die Länge eine kürzesten Pfades von s nach v an.*

Wenn wir nicht nur die D-Werte berechnen wollen sondern auch die zugehörigen von s ausgehenden Pfade, dann ist folgende (offensichtliche) Beobachtung recht nützlich:

- Wenn P ein kürzester Pfad von s nach $v \neq s$ ist, der mit der Kante (u, v) endet, dann ist das von s nach u führende Anfangsstück P' von P ein kürzester Pfad von s nach u.

Wir können nun vereinbaren, die kürzesten Pfade so auszuwählen, dass die Wahl eines kürzesten Pfades $P = s, \ldots, u, v$ stets impliziert, dass das Anfangsstück $P' = s, \ldots, u$ von P als kürzester Pfad von s nach u gewählt wird. Die ausgewählten Pfade bilden dann einen „Kürzesten-Pfade-Baum“ mit der Wurzel s. In G sind die kürzesten Pfade von der Wurzel s weg orientiert. Man spricht daher von einem „Aus-Baum (outtree)“. Man kann aber stattdesssen auch den entsprechenden „In-Baum (intree)“ berechnen, bei dem alle Kanten zur Wurzel hin orientiert sind. Ein solcher In-Baum lässt sich bequem mit Vater-Zeigern darstellen. In Verbindung mit Dijkstra's Algorithmus, kann der Vater-Zeiger $Z[v] \leftarrow u$ gesetzt werden, wann immer bei Aktualisierung (11.1) die Bedingung $D[u] + d(u, v) < D[v]$ erfüllt ist. Teil (b) von Abb. 11.1 zeigt den Kürzesten-Pfade-Baum (als In-Baum) zu dem Digraphen aus Teil (a).

Wir besprechen zwei Implementierungen von Dijkstra's Algorithmus. Bei der ersten Implementierung setzen wir voraus, dass der Digraph G, inklusive der Kantenkosten $(d(e))_{e \in E}$, in Adjazenzlistendarstellung gegeben ist. Die Knoten aus V identifizieren wir mit den Nummern von 1 bis n und nummerieren so, dass der Startknoten s die Nummer 1 erhält. Ähnlich wie beim Algorithmus von Prim kommen folgende weitere Datenstrukturen zum Einsatz:

- Ein Zähler z mit $z = |S|$.
- Ein Array $B[1:n]$ mit $B[v] = \text{TRUE}$ genau dann wenn $v \in V \setminus S$.
- Ein Array $D[1:n]$, dessen Werte $D[v]$ die in Lemma 11.1.3 genannten Bedingungen erfüllen.
- Ein Array $Z[1:n]$ mit $Z[v] = u$, falls $u \in S$, $v \in N(u)$ und der aktuell kürzeste S-Pfad nach v endet mit der Kante (u, v).
- Eine Priority Queue Q, welche die Knoten $v \in V \setminus S$ mit den Schlüsselwerten $D[v]$ enthält.

Das Array B gibt an, welche Knoten aktuell zu $V \setminus S$ gehören. Die Zeiger im Array Z repräsentieren — nach Terminieren des Algorithmus— den Kürzesten-Pfade-Baum mit Wurzel s (als In-Baum) in der oben besprochenen Weise. In dem folgenden Pseudocode werden die Datenstrukturen so initialisiert und fortlaufend aktualisiert, dass sie obiger Beschreibung entsprechen:

1. $Z[s] \leftarrow \textbf{nil}$; $D[s] \leftarrow 0$; $B[s] \leftarrow \text{FALSE}$; $z \leftarrow 1$.
2. Für alle $v \in V \setminus \{s\}$: $D[v] \leftarrow \infty$; $B[v] \leftarrow \text{TRUE}$.
3. Für alle $v \in N(s)$: $D[v] \leftarrow d(s, v)$; $Z[v] \leftarrow s$.
4. Initialisiere Q als Priority Queue mit den Elementen $v \in V \setminus \{s\}$ und deren Schlüsselwerten $D[v]$.
5. Solange $z < n$ mache Folgendes:
   (a) Entnimm Q das Element u mit minimalem Schlüsselwert (MIN und DELETE_MIN).
   (b) $z \leftarrow z + 1$; $B[u] \leftarrow \text{FALSE}$.
   (c) Für alle $v \in N(u)$ mit $B[v] = \text{TRUE}$ und $D[u] + d(u, v) < D[v]$:
   $Z[v] \leftarrow u$; $D[v] \leftarrow D[u] + d(u, v)$
   (letzteres als eine auf Q anzuwendende DECREASE_KEY-Operation).

Wie wir in der Vorlesung näher ausführen werden, wird die Laufzeit dominiert durch die DECREASE_KEY-Operationen in Schritt 5(c). Jede Kante aus E löst eine solche Operation aus. Die Rechenzeit beträgt daher $O(m \log n)$ und wir gelangen zu folgendem Resultat:

Satz 11.1.5 *Dijkstra's Algorithmus kann in Zeit $O(m \log n)$ implementiert werden.*

Bei der zweiten Implementierung von Dijkstra's Algorithmus setzen wir voraus, dass G und $d(\cdot,\cdot)$ durch eine $(n \times n)$-Kostenmatrix (ein 2-dimensionales Array) A mit

$$A[i,j] = \begin{cases} d(i,j) & \text{falls } i \neq j \text{ und } (i,j) \in E \\ \infty & \text{falls } i \neq j \text{ und } (i,j) \notin E \\ 0 & \text{falls } i = j \end{cases} \tag{11.2}$$

gegeben sind. Wir können obigen Pseudocode weitgehend übernehmen. Lediglich die Phase 5(c) muss geändert werden wie folgt:

5(c') Für alle $v \in \{2, \ldots, n\}$ mit $B[v] = \text{TRUE}$ und $D[u] + A[u,v] < D[v]$:
$Z[v] \leftarrow u$; $D[v] \leftarrow D[u] + A[u,v]$.

Die Laufzeit wird dann dominiert durch das $(n-1)$-malige Durchlaufen der Phase 5(c'). Da jeder Durchlauf offensichtlich Zeit $O(n)$ erfordert, erhalten wir die

Folgerung 11.1.6 *Dijkstra's Algorithmus kann in Zeit $O(n^2)$ implementiert werden.*

Die zweite Implementierung ist demnach der ersten dann vorzuziehen, wenn die Kantenanzahl m als Funktion in der Knotenanzahl n die Bedingung

$$m(n) = \omega\left(\frac{n^2}{\log n}\right)$$

erfüllt.

11.2 Kürzeste Pfade für alle Paare von Knoten

Folgendes Berechnungsproblem ist im englischen Sprachraum unter dem Namen „All Pair Shortest Path“ bekannt:

Eingabe: Ein Digraph $G = (V, E)$ mit Kantenkosten $d(e) = d(u,v) \geq 0$ für jede Kante $e = (u,v) \in E$.

Aufgabe: Berechne zu jedem Knotenpaar $u \neq v \in V$ die Länge eines kürzesten Pfades von u nach v in G. Letztere notieren wir als $D[u, v]$ (wobei $D[u, v] = \infty$, falls in G kein Pfad von u nach v existiert).

Wir gehen davon aus, dass G und $d(\cdot, \cdot)$ als Kostenmatrix A gemäß (11.2) gegeben sind. Wir identifizieren wieder die Knoten von G mit den Nummern von 1 bis n und beschreiben einen von Floyd stammenden (auf dynamischem Programmieren beruhenden) Algorithmus:

1. Setze die Matrix D^0 gleich der Kostenmatrix A.

2. Für $k = 1, \ldots, n$:
   für $i = 1, \ldots, n$:
   für $j = 1, \ldots, n$:

$$D^k[i, j] \leftarrow \min\{D^{k-1}[i, j] \ , \ D^{k-1}[i, k] + D^{k-1}[k, j]\} \ . \tag{11.3}$$

3. Gib $D = D^n$ als Ergebnis aus.

Da die Laufzeit von der dreifach geschachtelten Laufanweisung dominiert wird, terminiert der Algorithmus von Floyd nach $O(n^3)$ Schritten (konstante Rechenzeit für jedes Tripel $(k, i, j) \in [n]^3$).

In einer realen Implementierung würde man ein zwei-dimensionales Array D „recyclen“, d.h., man würde alle Matrizen $D^0, D^1, \ldots, D^n$ in demselben Array D abspeichern.[1]
Die Korrektheit des Verfahrens ergibt sich aus folgender Invarianzbedingung:

Lemma 11.2.1 *Wenn die äußere Schleife mit Laufindex k den Wert $k_0 \geq 0$ erreicht hat, dann gilt nach dem Durchlauf der zwei inneren Schleifen (bzw. nach der Initialisierung im Falle $k_0 = 0$) für alle $i, j \in [n]$:*
$D^{k_0}[i, j]$ ist gleich der Länge des kürzesten $\{1, \ldots, k_0\}$-Pfades[2] von i nach j.

Beweis Wir führen den Beweis durch vollständige Induktion nach k_0. Für $k_0 = 0$ gilt $\{1, \ldots, k_0\} = \emptyset$, d.h. es sind keine Zwischenknoten erlaubt und somit nur Punktpfade und Pfade mit exakt einer Kante zugelassen. Dann

[1] Im Folgenden beweisen wir die Korrektheit des Algorithmus ohne „Recycling“. Es ist aber nicht schwer, sich zu überlegen, dass der Algorithmus korrekt bleibt, wenn alle D^k-Matrizen im selben Array D gespeichert werden.

[2] mit $1, \ldots, k_0$ als den erlaubten Zwischenknoten

enthält aber $D^0 = A$ die dazu passenden Distanzwerte.
Kommen wir zum Induktionsschritt von $k_0 - 1$ nach k_0. Die Matrix D wird gemäß (11.3) aktualisiert. Die kreisfreien $\{1, \ldots, k_0\}$-Pfade von i nach j zerfallen in zwei Klassen:

- Pfade, die keinen Gebrauch von dem Zwischenknoten k_0 machen (Kategorie 1),
- Pfade, in denen k_0 genau einmal als Zwischenknoten vorkommt (Kategorie 2).

Die Länge der kürzesten Pfade der Kategorie 1 (bzw. der Kategorie 2) beträgt nach Induktionsvoraussetzung $D^{k_0-1}[i,j]$ (bzw. $D^{k_0-1}[i,k_0]+D^{k_0-1}[k_0,j]$. Somit liefert die Aktualisierung (11.3) die Länge des kürzesten $\{1, \ldots, k_0\}$-Pfades von i nach j. •

Da k nach Terminieren des Algorithmus den Wert n erreicht hat, ergibt sich aus Lemma 11.2.1 sofort die

Folgerung 11.2.2 *Der Algorithmus von Floyd löst das „All Pair Shortest Path" Problem in $O(n^3)$ Rechenschritten.*

Eine Erweiterung des Algorithmus von Floyd. Es sei $K[i,j]$ der größte Wert des Laufindex k, für welchen bei der Aktualisierung (11.3) die Bedingung

$$D^{k-1}[i,k] + D^{k-1}[k,j] < D^{k-1}[i,j]$$

erfüllt ist (bzw. $K[i,j] = 0$, falls diese Bedingung für kein $k \in [n]$ erfüllt ist). Es ist nicht schwer,

1. den Algorithmus von Floyd so zu modifizieren, dass er auch die $K[\cdot,\cdot]$-Werte bestimmt und
2. eine rekursive Prozedur Path zu entwerfen, die mit Hilfe von dem Array K zu gegebenen Knotenindizes i und j in Zeit $O(n)$ den kürzesten Pfad von i nach j berechnet.

Es wäre eine gute Übung, die entsprechende Modifikation von Floyd's Algorithmus und die rekursive Prozedur Path selber zu entwerfen.

Berechnung der (reflexiven-)transitiven Hülle. Es sei $G = (V, E)$ ein Digraph mit Knotenmenge $V = [n]$ und mit Adjazenzmatrix A. Wir nennen einen Pfad *nichttrivial*, falls er kein Punktpfad ist und definieren:

$$\begin{aligned} E^+ &= \{(i,j) \in V \times V | \text{ in } G \text{ existiert ein nichttrivialer Pfad von } i \text{ nach } j\} \ . \\ E^* &= \{(i,j) \in V \times V | \text{ in } G \text{ existiert ein Pfad von } i \text{ nach } j\} \ . \end{aligned}$$

Der Digraph $G^+ = (V, E^+)$ heißt die *transitive Hülle* und der Digraph $G^* = (V, E^*)$ heißt die *reflexive transitive Hülle* von G. Eine leichte Abänderung von Floyd's Algorithmus bekommt als Eingabe die Adjazenzmatrix A und berechnet die Adjazenzmatrix A^+ bzw. A^* des Digraphen G^+ bzw. G^*. Der resultierende Algorithmus ist unter dem Namen „Warshall's Algorithmus" bekannt. Es wäre eine nette Übung, sich die entsprechenden Versionen von Floyd's Algorithmus selber auszudenken.

Kapitel 12

Bestimmung starker Komponenten

Lesen Sie Abschnitt 5.7 in [2]!

Kapitel 13

Sortieren mit Schlüsselvergleichsverfahren

Wir haben in einer der vorangehenden Vorlesungen bereits das rekursive Verfahren namens „MergeSort“ kennengelernt. Es benötigt zum Sortieren von n Daten $O(n \log n)$ Schlüsselvergleiche (und die Laufzeit wird durch die Schlüsselvergleiche dominiert). Nach einer kurzen Besprechung naiver Sortierverfahren mit quadratischer Laufzeit machen wir in diesem Abschnitt Bekanntschaft mit den folgenden Algorithmen:

QuickSort:
ein rekursives Verfahren mit $O(n \log n)$ Schlüsselvergleichen im „average-case“.

HeapSort:
ein Verfahren mit $O(n \log n)$ Schlüsselvergleichen im „worst-case“.

Zudem beweisen wir, dass jedes Schlüsselvergleichsverfahren (sogar im average-case) mindestens $\Omega(n \log n)$ Schlüsselvergleiche benötigt.

Im Folgenden sei eine Sequenz $s_1, \ldots, s_n$ von Elementen einer linear geordneten Menge gegeben. Die Aufgabe besteht im Sortieren der Sequenz, d.h., gesucht ist eine Umordnung (Permutation) σ von $1, \ldots, n$ mit

$$s_{\sigma(1)} \leq \ldots \leq s_{\sigma(n)} \quad \text{oder} \quad s_{\sigma(1)} \geq \ldots \geq s_{\sigma(n)} \ .$$

- Bei realen Anwendungen werden Records sortiert, und s_i ist die Schlüsselkomponente des i-ten Records.

- Wir setzen voraus, dass die unsortierte Sequenz in Form eines Arrays $S[1:n]$ mit $S[i] = s_i$ vorliegt. Nach Terminieren des Sortierverfahrens soll das Array S die gleichen Schlüsselwerte in auf- oder absteigend sortierter Form enthalten.

13.1 Naive Verfahren mit quadratischer Laufzeit

Es ist eine naheliegende Idee, das Problem in n Iterationen zu lösen, wobei gilt:

- Nach der i-ten Iteration sind die Komponenten im Indexbereich $\{1, \ldots, i\}$ sortiert und im Indexbereich $\{i+1, \ldots, n\}$ unsortiert.

Zum Vorgehen in der i-ten Iteration betrachten wir zwei Verfahren:

SelectionSort: Finde einen Index $j \in \{i, \ldots, n\}$ mit $S[j] = \min\{S[i], \ldots, S[n]\}$ und vertausche $S[i]$ und $S[j]$.
Nach Iteration i enthält $S[1:i]$ die i kleinsten Elemente der ursprünglich gegebenen Folge in sortierter Form.

InsertionSort: Füge $S[i]$ an der richtigen Stelle im Indexbereich $\{1, \ldots, i\}$ ein, und zwar durch iteriertes Vertauschen mit dem linken Nachbarn, solange dieser einen Schlüsselwert oberhalb von $S[i]$ hat.
Nach Iteration i enthält $S[1:i]$ die ersten i Elemente der ursprünglich gegebenen Folge in sortierter Form.

SelectionSort benötigt in der i-ten Iteration $n-i$ Schlüsselvergleiche zur Minimumsbestimmung. Die Gesamtanzahl der Schlüsselvergleiche beträgt daher

$$\sum_{i=1}^{n}(n-i) = n^2 - \frac{1}{2}n(n+1) = O(n^2) \ .$$

InsertionSort benötigt in der i-ten Iteration bis zu $i-1$ Schlüsselvergleiche. Die Gesamtanzahl der Schlüsselvergleiche beträgt daher

$$\sum_{i=1}^{n}(i-1) = \frac{1}{2}(n-1)n = O(n^2) \ .$$

Bei beiden Verfahren wird eine quadratische Anzahl von Schlüsselvergleichen im worst-case tatsächlich auch benötigt (wie man sich leicht überlegen könnte). Beide Verfahren arbeiten „in situ“, d.h., sie benötigen neben dem Array S (fast) keinen weiteren Speicherplatz.

Beispiel 13.1.1 *Es sei* $(8, 20, 15, 6, 4, 12)$ *die im Array* $S[1:6]$ *gespeicherte Folge. Die iterative Veränderung von* S *im Rahmen von SelectionSort gestaltet sich wie folgt:*

1	2	3	4	5	6
8	20	15	6	$\underline{4}$	12
4	20	15	$\underline{6}$	8	12
4	**6**	15	20	$\underline{8}$	12
4	**6**	**8**	20	15	$\underline{12}$
4	**6**	**8**	**12**	$\underline{15}$	20
4	**6**	**8**	**12**	**15**	$\underline{20}$
4	**6**	**8**	**12**	**15**	**20**

In dieser Darstellung wurde die Teilfolge $S[1:i]$ *in Fettdruck gesetzt, und das Minimum in der Teilfolge* $S[i+1:6]$ *wurde durch Unterstreichen kenntlich gemacht.*
Die iterative Veränderung von S *im Rahmen von InsertionSort liest sich wie folgt:*

1	2	3	4	5	6
8	20	15	6	4	12
8	**20**	15	6	4	12
8	**15**	**20**	6	4	12
6	**8**	**15**	**20**	4	12
4	**6**	**8**	**15**	**20**	12
4	**6**	**8**	**12**	**15**	**20**

13.2 QuickSort

Die Grundidee von QuickSort besteht darin, mit Hilfe eines sogenannten „Split-Elementes“ $S[k]$ die Folge $S = (S[1], \ldots, S[n])$ in zwei Teilfolgen S' und S'' aufzuteilen: S' (bzw. S'') ist die Teilfolge von S bestehend aus allen $S[i]$ mit $S[i] < S[k]$ (bzw. aus allen $S[j]$ mit $S[j] \geq S[k]$). Danach erfolgen rekursive Aufrufe zum Sortieren von S' und S''. Die sortierte Gesamtfolge ergibt sich schließlich aus der Konkatenation der (sortierten) Folgen S' und

S''. Natürlich benötigt man eine Bedingung zum Abbruch der Rekursion und das Split-Element sollte nicht ausgerechnet das kleinste Element der Folge sein. Eine „High-Level“ Beschreibung von QuickSort liest sich wie folgt:

1. Falls $S[1] = \ldots = S[n]$: Gib S aus und stoppe.
   Andernfalls: Ermittle einen Index k mit $S[k] \neq \min\{S[1], \ldots, S[n]\}$
   und mache weiter.

2. Zerlege S in die Teilfolgen

   S' mit den S-Komponenten, die kleiner als $S[k]$ sind, sowie

   S'' mit den S-Komponenten, die größer als oder gleich $S[k]$ sind.

3. Rufe QuickSort rekursiv auf S' und S'' auf und erhalte als Ergebnis S' und S'' in jeweils sortierter Form.

4. Belege $S[1], \ldots, S[n]$ mit der Konkatenation von S' und S'' und gib das (nunmehr sortierte) Array S aus.

Die Korrektheit des QuickSort-Verfahrens ist offensichtlich. Kommen wir zur Laufzeitanalyse. In den Phasen 1 und 2 genügen jeweils $n-1$ Schlüsselvergleiche. Phase 4 ist trivial und kommt ohne Schlüsselvergleiche aus. Wenn $T(n)$ die Anzahl der Schlüsselvergleiche bei Eingabelänge n bezeichnet und wenn S in Teillisten S' und S'' der Längen ℓ und $n-\ell$ zerfällt, so benötigen die beiden rekursiven Aufrufe $T(\ell) + T(n-\ell)$ Schlüsselvergleiche. Im worst-case wird das Split-Element jedes Mal so gewählt, dass eine Liste einer Länge r in zwei Listen mit Längen 1 und $r-1$ aufgeteilt wird. Dies führt zu einer Rekursionsgleichung vom Typ

$$T(1) = a \quad \text{und} \quad T(n) = T(n-1) + T(1) + cn = T(n-1) + cn + a \ .$$

Expansion der Rekursion liefert

$$T(n) = cn + c(n-1) + \ldots + c2 + an = O(n^2) \ .$$

Im worst-case benötigt QuickSort demnach quadratisch viele Schlüsselvergleiche (so wie die zuvor diskutierten naiven Verfahren).
Wir betreiben jetzt eine average-case-Analyse von QuickSort unter der Annahme, dass

1. die eingangs gegebene Folge $S[1], \ldots, S[n]$ aus paarweise verschiedenen Schlüsselwerten besteht,

2. jedes Split-Element, das verschieden vom Minimum der Folge ist, mit gleicher Wahrscheinlichkeit ausgewählt wird.

In diesem Fall ergibt sich für $T(n)$ eine Rekursionsgleichung vom Typ $T(1) = a$ und

$$T(n) = cn + \frac{1}{n-1}\sum_{i=1}^{n-1}(T(i) + T(n-i)) = cn + \frac{2}{n-1}\sum_{i=1}^{n-1}T(i) \ .$$

Diese Rekursionsgleichung hat die (den Hörerinnen und Hörern der Vorlesung „Diskrete Mathematik I“ bekannte) Lösung $T(n) = O(n \log n)$.

Die High-Level-Beschreibung von QuickSort suggeriert, dass wir neben dem Array S noch weitere Arrays S' und S'' benötigen. Dieser Eindruck trügt: QuickSort besitzt eine „in situ“-Implementierung. Kernstück dabei ist eine Prozedur PARTITION(i, j, x) mit Indizes $1 \leq i < j \leq n$ und einem Vergleichselement $x \in \{S[i], \ldots, S[j]\} \setminus \{\min\{S[i], \ldots, S[j]\}\}$, welche Folgendes leistet:

1. Sie liefert als Ergebnis einen Index $r \in \{i+1, \ldots, j\}$ zurück.

2. Sie ordnet (während der Berechnung von r) die Komponenten von $S[i : j]$ so um, dass die Komponenten $S[i], \ldots, S[r-1]$ kleiner als x und die Komponenten $S[r], \ldots, S[j]$ größer als oder gleich x sind.

Auf dem obersten Rekursionslevel würde PARTITION mit den Parametern $i = 1$, $j = n$ und $x = S[k]$ aufgerufen, und der Aufruf würde einen entsprechenden Index $r \in \{2, \ldots, n\}$ zurückliefern. Dann ist die Teilfolge S' durch $S[1], \ldots, S[r-1]$ und die Teilfolge S'' durch $S[r], \ldots, S[n]$ gegeben.

Die Idee zur Implementierung von PARTITION ist einfach. Wir installieren zwei Zeiger ℓ und r, die anfangs die Werte $\ell = i$ und $r = j$ haben. Der Zeiger ℓ wandert nach rechts (d.h. ℓ wird schrittweise inkrementiert) bis die Bedingung $S[\ell] \geq x$ erfüllt ist. Entsprechend wandert der Zeiger r nach links (d.h. r wird schrittweise dekrementiert) bis er entweder auf ℓ trifft oder bis die Bedingung $S[r] < x$ erfüllt ist. Wenn beide Zeiger stationär werden, ohne sich zu treffen, dann werden die Schlüsselwerte $S[\ell]$ und $S[r]$ vertauscht. Die Prozedur terminiert, wenn beide Zeiger sich treffen. Etwas stärker formalisiert liest sich der Rumpf der Prozedur PARTITION wie folgt:

1. $\ell \leftarrow i$; $r \leftarrow j$.

2. Solange $\ell < r$ mache Folgendes:
   (a) Solange $S[\ell] < x$: $\ell \leftarrow \ell + 1$.
   (b) Solange $r > \ell$ und $S[r] \geq x$: $r \leftarrow r - 1$.
   (c) Falls $\ell < r$: Vertausche $S[\ell]$ und $S[r]$.
3. Gib r als Ergebnis aus.

Es ist nicht schwer zu zeigen, dass diese Prozedur (bezüglich ihrer Hauptschleife) folgende (in $S[i:j]$ gültigen) Invarianzbedingungen erfüllt:

- Alle Elemente links von ℓ sind kleiner als x.
- Alle Elemente rechts von r sind größer als oder gleich x.
- $\ell \leq r$ und $S[r] \geq x$.

Damit hat PARTITION die von uns gewünschten Eigenschaften.

Wir fassen das Hauptergebnis dieses Abschnittes in einem Satz zusammen:

Satz 13.2.1 *Die Prozedur Quicksort besitzt eine „in situ"-Implementierung. Die Anzahl der Schlüsselvergleiche (sowie die Laufzeit) zum Sortieren von n Daten beträgt $O(n^2)$ im worst-case und $O(n \log n)$ im average-case.*

Beispiel 13.2.2 *Wir stellen uns die Aufgabe, das Array $S[1:8]$ mit den Komponenten $8, 20, 15, 6, 4, 12, 19, 3$ mit QuickSort zu sortieren. Dabei wählen wir das Split-Element immer als das Größere der ersten beiden verschiedenen S-Komponenten. Unser Split-Element für das Array $S[1:8]$ wäre demnach $S[2] = 20$. Wir starten den Aufruf PARTITION$(1, 8, 20)$ und machen ein paar „Schnappschüsse" von dem resultierenden Rechenprozess (Split-Element durch Unterstreichen kenntlich gemacht, Zeiger ℓ und r als Pfeile visualisiert):*

8	$\underline{20}$	15	6	4	12	19	3
↑							↑
8	$\underline{20}$	15	6	4	12	19	3
	↑						↑
8	3	15	6	4	12	19	$\underline{20}$
	↑						↑
8	3	15	6	4	12	19	$\underline{20}$
							↑↑

- $S[1:8]$ *zerfällt in* $S[1:7] = (8, 3, 15, 6, 4, 12, 19)$ *und* $S[8:8] = (20)$.

Wir betrachten den rekursiven Aufruf von Quicksort auf $S[1:7]$. *Dieser führt zu dem Aufruf* $PARTITION(1, 7, 8)$ *mit Split-Element* 8*:*

$\underline{8}$	3	15	6	4	12	19
↑						↑
$\underline{8}$	3	15	6	4	12	19
↑				↑		
4	3	15	6	$\underline{8}$	12	19
↑				↑		
4	3	15	6	$\underline{8}$	12	19
		↑	↑			
4	3	6	15	$\underline{8}$	12	19
		↑	↑			
4	3	6	15	$\underline{8}$	12	19
			↑↑			

- $S[1:7]$ *zerfällt in* $S[1:3] = (4, 3, 6)$ *und* $S[4:7] = (15, 8, 12, 19)$.

Dies sorgt für rekursive Aufrufe auf den Teil-Arrays $S[1:3]$ *und* $S[4:7]$. *Wir kürzen den weiteren Verlauf etwas ab (indem wir die Arbeit der PARTITION-Aufrufe nur im Geiste mitvollziehen):*

- $S[1:3] = (4, 3, 6)$ *wird mit Hilfe von Split-Element* 4 *zerlegt in* $S[1:1] = (3)$ *und* $S[2:3] = (4, 6)$.
- $S[2:3] = (4, 6)$ *wird mit Hilfe von Split-Element* 6 *zerlegt in* $S[2:2] = (4)$ *und* $S[3:3] = (6)$.
- $S[4:7] = (15, 8, 12, 19)$ *wird mit Hilfe von Split-Element* 15 *zerlegt in* $S[4:5] = (12, 8)$ *und* $S[6:7] = (15, 19)$.
- $S[4:5] = (12, 8)$ *wird mit Hilfe von Split-Element* 12 *zerlegt in* $S[4:4] = (8)$ *und* $S[5:5] = (12)$.
- $S[6:7] = (15, 19)$ *wird mit Hilfe von Split-Element* 19 *zerlegt in* $S[6:6] = (15)$ *und* $S[7:7] = (19)$.

Die Konkatenation aller Teilfolgen führt schließlich zu der sortierten Sequenz $(3, 4, 6, 8, 12, 15, 19, 20)$.

13.3 HeapSort

Wir erinnern daran, dass ein (Min-)Heap der Größe n in einem Array $S[1:n]$ dargestellt werden kann. Dabei muss die Heap-Bedingung

$$S[j] \leq S[2j] \quad \text{bzw.} \quad S[j] \leq S[2j+1]$$

gelten, sofern $2j \leq n$ bzw. $2j+1 \leq n$. Falls $2j \leq n$, dann ist $2j$ die Adresse des linken Kindes von Knoten j. Eine entsprechende Bemerkung gilt für $2j+1$ und das rechte Kind von j. In diesem Abschnitt benötigen wir das Konzept eines „Teilheaps“:

Definition 13.3.1 *Es sei $1 \leq i \leq k \leq n$. Dann heißt $S[i:k]$ ein* Teilheap, *falls für alle $j = i, \ldots, k$*

$$S[j] \leq S[2j] \quad \textit{bzw.} \quad S[j] \leq S[2j+1]$$

gilt, sofern $2j \leq k$ bzw. $2j+1 \leq k$.

Wir nehmen an, dass das Array $S[1:n]$ anfangs n Schlüsselwerte enthält, ohne notwendigerweise eine Heap-Bedingung zu erfüllen. Beachte, dass dennoch $S[\lfloor n/2 \rfloor + 1 : n]$ ein Teilheap ist, weil für $j > \lfloor n/2 \rfloor$ die Heap-Bedingung in Definition 13.3.1 leer ist. Die folgende „High-Level-Prozedur“ zum Heap-Aufbau erweitert in jeder Iteration diesen Teilheap um eine weitere Adresse mit Hilfe der Ihnen bereits bekannten Technik des „Einsinken-Lassens“:

Heap-Aufbau: Für $i = \lfloor n/2 \rfloor, \ldots, 1$:
Erweitere den Teilheap $S[i+1:n]$ zum Teilheap $S[i:n]$ durch Einsinken-Lassen von $S[i]$.

Beispiel 13.3.2 *Es sei $(8, 20, 15, 6, 4, 12, 19, 3)$ die im Array $S[1:8]$ abgespeicherte Folge. Die iterative Veränderung von S im Rahmen des Heap-*

Aufbaus gestaltet sich wie folgt:

1	2	3	4	5	6	7	8
8	20	15	6	**4**	**12**	**19**	**3**
8	20	15	**3**	**4**	**12**	**19**	**6**
8	20	15	**3**	**4**	**12**	**19**	**6**
8	20	**12**	**3**	**4**	**15**	**19**	**6**
8	20	**12**	**3**	**4**	**15**	**19**	**6**
8	3	**12**	**20**	**4**	**15**	**19**	**6**
8	**3**	**12**	**6**	**4**	**15**	**19**	**20**
8	**3**	**12**	**6**	**4**	**15**	**19**	**20**
3	**8**	**12**	**6**	**4**	**15**	**19**	**20**
3	**4**	**12**	**6**	**8**	**15**	**19**	**20**

In dieser Darstellung wurde der aktuelle Teilheap $S[i:n]$ in Fettdruck gesetzt. Dabei ist $S[i:n]$ erst dann komplett, wenn $S[i]$ vollständig eingesunken ist. Das aktuell einsinkende Element haben wir durch Unterstreichen kenntlich gemacht. Wenn der Einsinkvorgang abgeschlosssen (und damit der Teilheap $S[i:n]$ komplett) ist, zeigen wir das durch eine horizontale Trennlinie an. Der vollständige Heap, den wir nach Beendigung der Heap-Aufbau-Prozedur erhalten ist in Abb. 13.1 in Form eines Binärbaumes zu sehen.

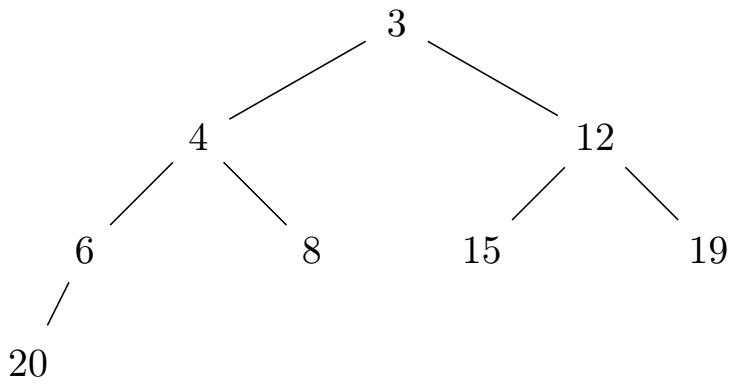


Abbildung 13.1: Der Heap zu den Daten aus Beispiel 13.3.2.

Satz 13.3.3 *Die obige Prozedur baut einen Heap in $O(n)$ Schritten auf.*

Beweis Die Zeit zum Einsinken-Lassen ist proportional zur Einsinktiefe. Die Laufzeit ergibt sich asymptotisch aus folgender Überlegung:

1. $n/4$ Knoten haben Einsinktiefe 1
2. $n/8$ Knoten haben Einsinktiefe 2.
3. $n/16$ Knoten haben Einsinktiefe 3. Et cetera.

Weiterhin gilt

$$\sum_{i=1}^{\infty} i \cdot \frac{n}{2^{i+1}} = \frac{n}{2} \sum_{i=1}^{\infty} \frac{i}{2^i} = n \ .$$

Im letzten Schritt wurde $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$ ausgenutzt. Aus dieser Diskussion ergibt sich die Zeitschranke $O(n)$ für den Heap-Aufbau. •

Die Grundidee zum Sortieren mit Hilfe eines gegebenen Heaps ist einfach. Man sorgt dafür, dass für $i = n, \ldots, 1$ Folgendes gilt:

1. $S[i+1 : n]$ enthält die $n - i$ kleinsten Schlüsselwerte in absteigend sortierter Form.
2. $S[1 : i]$ ist ein Teil-Heap.

Hier ist die „High-Level-Beschreibung“ des zugehörigen Sortierverfahrens:

Sortieren: Für $i = n, \ldots, 2$ mache Folgendes:

1. Vertausche die Schlüsselwerte $S[1]$ und $S[i]$.
2. Reproduziere den Teilheap $S[1 : i-1]$ durch Einsinken-Lassen von $S[1]$.

Da die Einsinktiefe durch $\log n$ nach oben beschränkt ist, benötigt die obige Prozedur zum Sortieren Zeit $O(n \log n)$.

Beispiel 13.3.4 *Wir setzen Beispiel 13.3.2 fort und steigen ein mit dem Heap* $S[1:8] = (3, 4, 12, 6, 8, 15, 19, 20)$. *Die iterative Veränderung von S im*

Rahmen des oben geschilderten Sortierverfahrens gestaltet sich wie folgt:

1	2	3	4	5	6	7	8
3	4	12	6	8	15	19	20
$\underline{20}$	4	12	6	8	15	19	**3**
4	$\underline{20}$	12	6	8	15	19	**3**
4	6	12	$\underline{20}$	8	15	19	**3**
$\underline{19}$	6	12	20	8	15	**4**	**3**
6	$\underline{19}$	12	20	8	15	**4**	**3**
6	8	12	20	$\underline{19}$	15	**4**	**3**
$\underline{15}$	8	12	20	19	**6**	**4**	**3**
8	$\underline{15}$	12	20	19	**6**	**4**	**3**
$\underline{19}$	15	12	20	**8**	**6**	**4**	**3**
12	15	$\underline{19}$	20	**8**	**6**	**4**	**3**
$\underline{20}$	15	19	**12**	**8**	**6**	**4**	**3**
15	$\underline{20}$	19	**12**	**8**	**6**	**4**	**3**
$\underline{19}$	$\underline{20}$	**15**	**12**	**8**	**6**	**4**	**3**
$\underline{20}$	**19**	**15**	**12**	**8**	**6**	**4**	**3**
20	**19**	**15**	**12**	**8**	**6**	**4**	**3**

In dieser Darstellung wurde die bereits sortierte Teilfolge $S[i+1:n]$ in Fettdruck gesetzt. Die komplementäre Teilfolge $S[1:i]$ repräsentiert den aktuellen Teilheap. Das aktuell einsinkende Element haben wir wieder durch Unterstreichen kenntlich gemacht. Den Abschluss eines Einsinkvorganges zeigen wir wieder durch eine horizontale Trennlinie an.

Das Gesamtverfahren HeapSort verläuft in zwei Phasen: Heapaufbau (Phase 1) und Sortieren bei gegebenem Heap (Phase 2). Wir fassen das Hauptergebnis noch einmal zusammen:

Satz 13.3.5 *HeapSort sortiert n Schlüsselwerte in Zeit $O(n \log n)$, wobei der Heap-Aufbau lediglich Zeit $O(n)$ benötigt.*

Sowohl in Phase 1 als auch in Phase 2 wird eine Hilfsprozedur reheap(i, k) zum Einsinken-Lassen eines Schlüssels in einem Teil-Heap benötigt. Diese Prozedur hat Zugriff auf das Array $S[1:n]$ (lässt aber den Arraybereich außerhalb $\{i, \ldots, k\}$ unangetastet) und leistet Folgendes:

Eingabe: Ein Teilheap $S[i+1:k]$.

Ausgabe: der Teilheap $S[i:k]$.

Es wäre leicht, diese Prozedur mit den entsprechenden Array-Manipulationen zu implementieren (aber wir verzichten an dieser Stelle auf die Angabe weiterer Implementierungsdetails).

13.4 Eine Barriere für Schlüsselvergleichsverfahren

Ein *Schlüsselvergleichsverfahren* ist ein Verfahren, dass auf die Eingabedaten $S[1:n]$ nicht direkt, sondern nur über Schlüsselvergleiche der Form „$S[i] \leq S[j]$“ oder „$S[i] < S[j]$“ zugreifen kann. Solche Verfahren sind mit einem binären Entscheidungsbaum T modellierbar. Abb. 13.2 zeigt einen solchen Baum im Falle $n = 3$. Jeder innere Knoten von T entspricht einem Schlüsselvergleich. Jedes Blatt von T entspricht einer der ($n!$ vielen) Umordnungen von $1, \ldots, n$. Das Verfahren beginnt bei der Wurzel und beschreibt danach einen Pfad zu einem Blatt. Wenn es sich an einem inneren Knoten u befindet, passiert Folgendes:

1. Der Schlüsselvergleich am Knoten u wird ausgewertet.

2. Falls die Antwort darauf „Nein“ (bzw. „Ja“) lautet, so wird das Verfahren beim linken (bzw. rechten) Kind von u fortgesetzt.

Wenn der resultierende Berechnungspfad ein Blatt erreicht, dann wird die entsprechende Umordnung der Komponenten von $S[1:n]$ durchgeführt. Offensichtlich gilt:

1. Die Anzahl der Schlüsselvergleiche im worst-case ist gleich der Höhe von T (= Länge eines längsten Pfades von der Wurzel zu einem Blatt).

2. Die Anzahl der Schlüsselvergleiche im average-case ist gleich der mittleren Blatttiefe in T.

3. Unter der Voraussetzung, dass die n Eingabedaten paarweise verschieden sind, hat T insgesamt $n!$ Blätter.

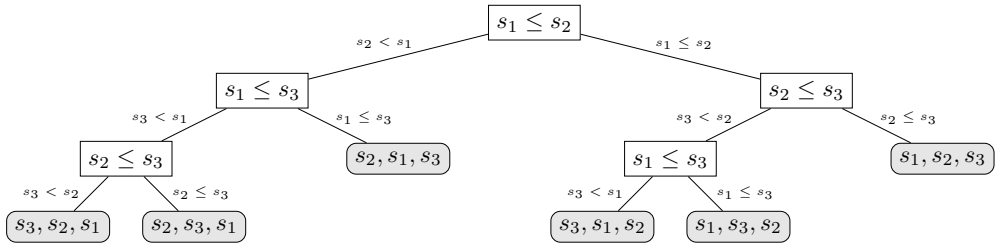


Abbildung 13.2: Modellierung eines Schlüsselvergleichsverfahrens mit Hilfe eines Entscheidungsbaumes.

Da ein binärer Baum mit $N = n!$ Blättern eine Höhe von mindestens

$$\log N = \log n! \geq \log \left(\frac{n}{2}\right)^{\lfloor n/2 \rfloor} = \Omega(n \log n)$$

hat, benötigt jedes Schlüsselvergleichsverfahren zum Sortieren von n Daten im worst-case mindestens $\Omega(n \log n)$ Schlüsselvergleiche. Die Sortierverfahren MergeSort und HeapSort sind daher im worst-case (im Sinne der O, Ω-Notation) optimal.

Zu einem Binärbaum T mit N Blättern $w_1, \ldots, w_N$ bezeichne d_i die Tiefe von w_i in T. Es ist nicht schwer, die Ungleichung

$$\sum_{i=1}^{N} d_i \geq N \log N$$

zu beweisen (Induktion über N). Damit gilt für die mittlere Blatttiefe

$$\frac{1}{N} \sum_{i=1}^{N} d_i \geq \log N \ .$$

Im Falle von $N = n!$ ergibt sich eine mittlere Blatttiefe von $\Omega(n \log n)$. Daher benötigt jedes Schlüsselvergleichsverfahren zum Sortieren von n Daten bereits im average-case mindestens $\Omega(n \log n)$ Schlüsselvergleiche. QuickSort ist demzufolge ein im average-case (im Sinne der O, Ω-Notation) optimales Sortierverfahren.

Kapitel 14

Sortieren durch Fachverteilung

In Abschnitt 14.1 besprechen wir das Verfahren BucketSort zum Sortieren einer gegebenen Folge von Zahlen (oder, allgemeiner, von Objekten einer linear geordneten Menge). In Abschnitt 14.2 besprechen wir das Konzept der lexikographischen Ordnung auf Strings über einem linear geordneten Alphabet. Diese Strings lassen sich auch als Zahlentupel auffassen. Abschnitt 14.3 ist dem Verfahren RadixSort zum Sortieren von Strings (oder Zahlentupeln) gleicher Länge gewidmet. In Abschnitt 14.4 besprechen wir zwei Varianten von RadixSort. Die erste Variante sortiert weiterhin Zahlentupel (oder Strings) gleicher Länge, erzielt aber eine bessere Laufzeit. Die zweite Variante erlaubt das effiziente Sortieren von Zahlentupeln (oder Strings) von i.A. verschiedener Länge.

14.1 Sortieren von Zahlen mit BucketSort

Eine linear geordnete Menge Σ der Größe m kann auf die offensichtliche Weise mit der Zahlenmenge $\{0, 1 \ldots, m-1\}$ identifiziert werden. Das Sortieren einer gegebenen Folge von Elementen aus M entspricht dann dem Sortieren einer Folge $a_1, \ldots, a_n$ von Zahlen aus dem Bereich $\{0, 1 \ldots, m-1\}$. Dies führt uns zu folgendem Sortierproblem:

Problem 1: Sortiere eine gegebene Folge $a_1, \ldots, a_n$ von Zahlen aus dem Bereich $\{0, 1, \ldots, m-1\}$.

Methode: BucketSort

1. Stelle leere Queues $Q_0, Q_1, \ldots, Q_{m-1}$, genannt „Buckets“, bereit (Zeit: $O(m)$).
2. Inspiziere die Folge $a_1, \ldots, a_n$ von links nach rechts und füge a_i in Q_{a_i} ein (Zeit: $O(n)$).
3. Produziere die sortierte Ausgabeliste durch Konkatenation von $Q_0, Q_1, \ldots, Q_{m-1}$ (Zeit: $O(m)$).

Offensichtlich gilt:

Satz 14.1.1 *BucketSort löst Problem 1 in $O(m+n)$ Schritten.*

Beachte, dass die Schranke $m+n$ asymptotisch unterhalb von $n\log(n)$ liegt, sofern $m = o(n\log(n))$.

Frage: Wieso steht Satz 14.1.1 nicht im Widerspruch zu der früher bewiesenen unteren Schranke $n/2\log(n/2)$ für die Anzahl der zum Sortieren benötigten Schlüsselvergleiche?

14.2 Die lexikographische Ordnung auf Zahltupeln

Es sei „$<$“ eine (irreflexive) lineare Ordnung auf einem Alphabet[1] Σ. Weiter sei Σ^* die Menge aller Wörter (= endlichen, evtl. leeren, Folgen) über Σ. Dann können wir „$<$“ zu einer linearen Ordnung auf Σ^*, genannt *lexikographische Ordnung*, fortsetzen wie folgt:

Definition 14.2.1 *Die Folge $u = (u_1, \ldots, u_r)$ ist kleiner als die Folge $v = (v_1, \ldots, v_s)$, notiert als $u < v$, falls eine der folgenden beiden Bedingungen erfüllt ist:*

(i) u ist ein echter Präfix von v, d.h., $r < s$ und die ersten r Komponenten von u und v stimmen überein.

(ii) Es existiert ein $i \in \{1, \ldots, \min\{r,s\}\}$, so dass $u_i < v_i$ und die ersten $i-1$ Komponenten von u und v stimmen überein.

[1] Alphabet = endliche Menge.

Wenn Σ das lateinische Alphabet ist, versehen mit der natürlichen Ordnung von a bis z, dann liefert Definition 14.2.1 die in einem Lexikon verwendete Ordnung.

Beispiel 14.2.2 *„aal<aalglatt“ (Bedingung (i)) und „tangente<tango“ (Bedingung (ii)).*

Bemerkung 14.2.3 *1. Wie früher bereits erwähnt, kann eine linear geordnete Menge Σ der Größe $|\Sigma| = m$ mit der Menge $\{0, 1, \ldots, m-1\}$ identifiziert werden.*

2. Wörter aus Σ^k (feste Wortlänge k) sind interpretierbar als m-näre Zahlendarstellungen:

(a) Die kleinste darstellbare Zahl ist die Null. $(0, \ldots, 0)$ ist das zugehörige k-Tupel.

(b) Die größte darstellbare Zahl ist $m^k - 1$; $(m-1, \ldots, m-1)$ ist das zugehörige k-Tupel.

(c) Allgemein gilt: das k-Tupel $A = (a_{k-1}, \ldots, a_1, a_0)$ stellt die Zahl $\sum_{i=0}^{k-1} a_i m^i$ dar.

Bei fester Wortlänge k stimmt die lexikographische Ordnung überein mit der natürlichen Ordnung auf Zahlen.

3. Das Konzept der Lexikographischen Ordnung ist (auf die offensichtliche Weise) verallgemeinerbar auf den Fall verschiedener Alphabete für verschiedene Positionen in der Folge. S. dazu das folgende Beispiel 14.2.4.

Beispiel 14.2.4 *Die Tripel (Jahr,Monat,Tag)*$\in \{1900, \ldots, 2100\} \times \{1, \ldots, 12\} \times \{1, \ldots, 31\}$ *benutzen in jeder Komponente ein anderes Alphabet.*

14.3 Sortieren von Zahltupeln mit RadixSort

In diesem Abschnitt beschäftigen wir uns mit dem folgenden

Problem 2: Sortiere gemäß der lexikographischen Ordnung eine gegebene Folge $A_1, \ldots, A_n$ von k-Tupeln mit Komponenten aus dem Bereich $\{0, 1, \ldots, m-1\}$.

Notation: $A_i = (a_{i1}, \ldots, a_{ik}) \in \{0, 1, \ldots, m-1\}^k$.

Plan: Wende BucketSort komponentenweise an in Richtung von rechts (also Komponente k) nach links (also Komponente 1).

Resultierende Methode: RadixSort

1. Initialisiere eine Queue Q mit $A_1, \ldots, A_n$.
2. Für $j = k, k-1, \ldots, 1$ mache Folgendes:
   (a) Stelle $Q_0, Q_1, \ldots, Q_{m-1}$ als (zunächst) leere Queues bereit.
   (b) Solange Q nicht leer ist, entnimm Q das vorderste Element, sagen wir A_i, und füge es in $Q_{a_{ij}}$ ein.
   (c) Rekonfiguriere Q neu als die Queue, die sich aus der Konkatenation von $Q_0, Q_1, \ldots, Q_{m-1}$ ergibt.

Beispiel 14.3.1 *Wir machen die folgenden „Schnappschüsse“ von einem Beispiellauf von RadixSort mit den Parametern $n = 8$, $m = 10$ und $k = 3$. Die Eingabe $A_1, \ldots, A_n$ ergibt sich aus der Initialisierung von Q (s. unten).*

Anfangs: *$Q_0, Q_1, \ldots, Q_{m-1}$ sind leer.*

$$Q = 087, 754, 750, 532, 501, 001, 539, 437 \ .$$

Nach Iteration j=3:

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
750	501	532	–	754	–	–	087	–	539
	001						437		

$$Q = 750, 501, 001, 532, 754, 087, 437, 539 \ .$$

Nach Iteration j=2:

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
501	–	–	532	–	750	–	–	087	–
001			437		754				
			539						

$$Q = 501, 001, 532, 437, 539, 750, 754, 087 \ .$$

Nach Iteration j=1:

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
001	–	–	–	437	501	–	750	–	–
087					532		754		
					539				

$$Q = 001, 087, 437, 501, 532, 539, 750, 754 \ .$$

Dass sich im obigen Beispiellauf am Ende eine lexikographische Sortierung ergibt, ist kein Zufall:

Satz 14.3.2 (Korrektheit von RadixSort) *Nach Terminieren von RadixSort befinden sich in Q die Eingangsdaten $A_1, \ldots, A_n$ in lexikographischer Sortierung.*

Beweis Die lexikographische Ordnung bezüglich der Komponenten $j, \ldots, k$ von Zahlen-k-Tupeln notieren wir als $<_j^k$. Beachte, dass $<_k^k$ mit der Ordnung bezüglich der k-ten Komponente und dass $<_1^k$ mit der gewöhnlichen lexikographischen Ordnung übereinstimmt. Nach jeder Iteration der Hauptschleife von RadixSort gilt die folgende Bedingung:

- Q enthält die Eingangsdaten $A_1, \ldots, A_n$ in Sortierung gemäß $<_j^k$.

Aus dieser Invarianzbedingung ergibt sich die Korrektheit: nach der k-ten Iteration mit $j = 1$ sind die Daten bezüglich aller Komponenten $1, \ldots, k$ lexikographisch sortiert. Die Bedingung selbst ist leicht verifizierbar mit Induktion über die Anzahl der Iterationen:

Induktionsanfang: Nach der ersten Iteration mit $j = k$ sind die Daten offensichtlich gemäß der k-ten Komponente (und somit gemäß $<_k^k$) sortiert.

Induktionsschritt: Wir betrachten eine Iteration mit $j < k$. Wir können induktiv voraussetzen, dass vor dieser Iteration die Daten gemäß $<_{j+1}^k$ sortiert sind. Die aktuelle Iteration hat den folgenden Effekt:

- Falls $A_i <_j^k A_{i'}$ wegen $a_{ij} < a_{i'j}$, dann landet in Phase 2(b) A_i in $Q_{a_{ij}}$ und $A_{i'}$ in $Q_{a_{i'j}}$. In Phase 2(c) wird somit A_i vor $A_{i'}$ in Q platziert.

- Falls $A_i <_j^k A_{i'}$ und $a_{ij} = a_{i'j}$, so gilt auch $A_i <_{j+1}^k A_{i'}$. Dann wird A_i in Phase 2(b) vor $A_{i'}$ in $Q_{a_{ij}}$ eingefügt und dementsprechend steht A_i nach Phase 2(c) vor $A_{i'}$ in Q.

Aus der obigen Diskussion ergibt sich die Korrektheit von RadixSort. •

RadixSort besteht im Wesentlichen aus k-maliger Anwendung von BucketSort. Anstatt k-Tupel in Queues einzufügen (oder gar Records R, bei denen das k-Tupel nur eine Komponente R.key darstellt) fügen wir stets nur einen Zeiger auf das betreffende k-Tupel ein. Daher kann eine Iteration der Hauptschleife in $O(n+m)$ Schritten durchgeführt werden und es ergibt sich folgendes Resultat:

Satz 14.3.3 *RadixSort benötigt zum lexikographischen Sortieren von n k-Tupeln mit Komponenten aus dem Bereich $\{0, 1, \ldots, m-1\}$ eine Rechenzeit der Größenordnung $O(k \cdot (n+m))$.*

Falls die Zahlen in den j-ten Komponenten der k-Tupel aus dem Bereich $\{0, 1, \ldots, m_j - 1\}$ stammen (vgl. mit Beispiel 14.2.4), dann benötigt die betreffende Iteration der Hauptschleife von RadixSort $O(n+m_j)$ Rechenschritte. Es ergibt sich daher die folgende Verallgemeinerung von Satz 14.3.3:

Satz 14.3.4 *RadixSort benötigt zum lexikographischen Sortieren von n k-Tupeln mit j-ten Komponenten aus dem Bereich $\{0, 1, \ldots, m_j - 1\}$ eine Rechenzeit der Größenordnung $O\left(kn + \sum_{j=1}^{k} m_j\right)$.*

Eine Zahl $a \in \{0, 1, \ldots, n^k - 1\}$ kann via

$$a = \sum_{i=0}^{k-1} a_i n^i \text{ mit } 0 \leq a_i \leq n-1$$

als k-Tupel $A = (a_{k-1}, \ldots, a_1, a_0)$ dargestellt werden. Zudem kann das k-Tupel A zu gegebenem a, oder umgekehrt a aus A, leicht in $O(k)$ Schritten berechnet werden. Damit ergibt sich folgende Anwendung von RadixSort:

Satz 14.3.5 *n Zahlen im Bereich $\{0, 1, \ldots, n^k - 1\}$ können in $O(kn)$ Schritten sortiert werden.*

14.4 Verbesserte Varianten von RadixSort

In diesem Abschnitt stellen wir uns zwei Ziele:

1. Laufzeit $O(kn + m)$ statt $O(k(n + m))$ bei der Sortierung von Strings fester Wortlänge k (also von k-Tupeln) .

2. Laufzeit $O(L+m)$ bei der Sortierung von Strings verschiedener Länge, wobei L die Gesamtlänge aller Strings bezeichnet.

Zu Ziel 1 merken wir Folgendes an:

Beobachtung: In den Phasen 2(a) und 2(c) verschwenden wir Rechenzeit durch die (sinnlose) Manipulation von leeren Buckets.

Plan: Berechne im Voraus für jedes $j \in \{0, 1, \ldots, m-1\}$ eine sortierte Liste NONEMPTY[j], welche die Zahlen aus

$$\{a_{ij} |\ i = 1, \ldots, n\}$$

enthält. Das sind nämlich exakt die Adressen der in dieser Iteration nichtleeren Buckets.

Die Vorteile dieser Vorausberechnung sind, ein paar leichte Modifikationen in der Hauptschleife vorausgesetzt, wie folgt:

1. In Phase 2(a) der Hauptschleife brauchen (in den Iterationen mit $j < k$) nur die nichtleeren Buckets der vorangegangenen Iteration (also die Q_i mit $i \in$ NONEMPTY[$j + 1$]) geleert werden. (Die anderen sind bereits leer.)

2. In Phase 2(c) der Hauptschleife werden nur die nichtleeren Buckets Q_i mit $i \in$ NONEMPTY[j] der Reihe nach konkateniert.

Eine Iteration dauert dann nur noch $O(n)$ statt $O(m)$ Schritte und wir gelangen zur Zeitschranke $O(kn)$ plus die Zeit zur Vorausberechnung der NONEMPTY-Listen. Das folgende Verfahren berechnet diese Listen in $O(kn+m)$ Rechenschritten:

1. Initialisiere in $O(kn)$ Schritten eine Queue Q' mit den Elementen der Multimenge
$$\{(j, a_{ij}) |\ 1 \leq i \leq n, 1 \leq j \leq k\} \ .$$

2. Sortiere in $O(kn + m)$ Schritten die Paare aus Q' lexikographisch mit Hilfe von RadixSort, so dass das (sortierte) Ergebnis wieder in Q' steht.

3. Arbeite in $O(kn)$ Schritten die Paare in Q' der Reihe nach ab und baue parallel dazu die sortierten NONEMPTY-Listen auf.

Wir fassen das Ergebnis zusammen:

Satz 14.4.1 *Die oben beschriebene verbesserte Variante von RadixSort sortiert n k-Tupel mit Komponenten aus $\{0, 1, \ldots, m-1\}$ in $O(kn+m)$ Schritten.*

Beispiel 14.4.2 *Gegeben seien die folgenden 8 Tripel mit Komponenten aus $\{0, 1, \ldots, 9\}$ (die selben Eingabedaten wie in Beispiel 14.3.1):*

$$087, 754, 750, 532, 501, 001, 539, 437$$

Diese Liste können wir von links nach rechts durchlesen und parallel dazu die entsprechenden Paare in Q' aufnehmen:

$$\begin{aligned} Q' \;=\; & (1,0),(2,8),(3,7),(1,7),(2,5),(3,4),(1,7),(2,5),(3,0),(1,5),(2,3),(3,2),\\ & (1,5),(2,0),(3,1),(1,0),(2,0),(3,1),(1,5),(2,3),(3,9),(1,4),(2,3),(3,7) \end{aligned}$$

Nach Anwendung von RadixSort enthält Q' die selben Paare wie zuvor, aber in sortierter Form:

$$\begin{aligned} Q' \;=\; & (1,0),(1,0),(1,4),(1,5),(1,5),(1,5),(1,7),(1,7),(2,0),(2,0),(2,3),(2,3),\\ & (2,3),(2,5),(2,5),(2,8),(3,0),(3,1),(3,1),(3,2),(3,4),(3,7),(3,7),(3,9) \end{aligned}$$

Hieraus ergeben sich die folgenden NONEMPTY-Listen:

$$\begin{aligned} \mathit{NONEMPTY}[1] &= 0, 4, 5, 7\\ \mathit{NONEMPTY}[2] &= 0, 3, 5, 8\\ \mathit{NONEMPTY}[3] &= 0, 1, 2, 4, 7, 9 \end{aligned}$$

Wenden wir uns nun dem zweiten Ziel dieses Abschnittes zu: der lexikographischen Sortierung von Strings bzw. Zahlentupeln verschiedener Länge.

Problem 3: Sortiere gemäß der lexikographischen Ordnung eine gegebene Folge $A_1, \ldots, A_n$ von Tupeln der Längen $\ell_1, \ldots, \ell_n$ mit Komponenten aus dem Bereich $\{0, 1, \ldots, m-1\}$. (Somit gilt $A_i \in \{0, 1 \ldots, m-1\}^{\ell_i}$.)

Wir setzen $k := \max_{i\in[n]} \ell_i$. Für die Gesamtlänge $L := \sum_{i\in[n]} \ell_i$ aller Tupel gilt dann $L \leq kn$, aber L kann i.A. sehr viel kleiner als kn sein (zum Beispiel, wenn sich unter den A_i ein sehr langes und viele kleine Tupel befinden).

Bemerkung 14.4.3 *Eine Sortierung mit RadixSort bzw. verbessertem RadixSort würde in $O(k(n+m))$ bzw. $O(kn+m)$ Schritten ablaufen.*

Dies ist jedoch im Falle $L << kn$ nicht zufriedenstellend. Wir streben vielmehr eine Laufzeit $O(L+m)$ an. Dies kann erreicht werden

- durch Verwendung der uns bereits bekannten NONEMPTY-Listen,
- durch Verwendung von LENGTH-Listen, die uns helfen werden, für Tupel einer Länge ℓ in den Iterationen mit $j = k, k-1, \ldots, \ell+1$ keine Zeit aufzuwenden. Vielmehr werden diese Tupel erst in der Iteration mit $j = \ell$ ins Spiel gebracht.

Diese Überlegungen führen zu der folgenden Variante von RadixSort:

1. Bestimme zu jedem Tupel A_i seine Länge ℓ_i und baue anschließend die LENGTH-Listen auf, wobei LENGTH[j] alle A_i mit $\ell_i = j$ enthält.[2] Dies lässt sich in $O(L)$ Schritten bewerkstelligen.
2. Initialisiere in $O(L)$ Schritten eine Queue Q' mit den Elementen der Multimenge
$$\{(j, a_{ij}) |\ 1 \leq i \leq n, 1 \leq j \leq \ell_i\}\ .$$
(Diese Multimenge hat die Größe L.)
3. Sortiere in $O(L+m)$ Schritten die Paare aus Q' lexikographisch mit Hilfe von RadixSort, so dass das (sortierte) Ergebnis wieder in Q' steht.
4. Arbeite die Paare in Q' der Reihe nach ab und baue parallel dazu in $O(L)$ Schritten die NONEMPTY-Listen auf (in der gleichen Weise wie bei der uns schon bekannten Verbesserung von RadixSort).

Die gesamte Vorausberechnung der LENGTH- und NONEMPTY-Listen kann in Zeit $O(L+m)$ abgeschlossen werden.

Die Vorteile dieser Vorausberechnung sind, ein paar leichte Modifikationen in den Phasen 1 und 2 von RadixSort vorausgesetzt, wie folgt:

[2] In einer realen Implementierung würde man die Adressen der A_i enthaltenden Records statt der A_i in die LENGTH-Listen einfügen.

1. In Phase 1 von RadixSort wird Q als leere Queue initialisiert.
2. In Phase 2(a) der Hauptschleife brauchen (in den Iterationen mit $j < k$) nur die nichtleeren Buckets der vorangegangenen Iteration (also die Q_i mit $i \in \text{NONEMPTY}[j+1]$) geleert werden. (Die anderen sind bereits leer.)
3. In Phase 2(b) der Hauptschleife fügen wir die in LENGTH[j] enthaltenen Tupel A_i *vorne* in Q ein. Q enthält zu diesem Zeitpunkt die Daten A_i mit $\ell_i = j$, gefolgt von den Daten A_i mit $\ell_i \geq j+1$ (letztere in der Sortierung gemäß „$<^k_{j+1}$“)[3].
4. In Phase 2(c) der Hauptschleife werden nur die nichtleeren Buckets, also die Buckets Q_i mit $i \in \text{NONEMPTY}[j]$, der Reihe nach konkateniert.

Satz 14.4.4 (Korrektheit der „Problem 3“-Variante von RadixSort) *Nach Terminieren von RadixSort stehen die Daten $A_1, \ldots, A_n$ in Q in lexikographischer Sortierung.*

Den Beweis, der ähnlich zum Beweis von Satz 14.3.2 verläuft, lassen wir aus.

Satz 14.4.5 (Laufzeit der „Problem 3“-Variante von RadixSort) *RadixSort terminiert nach $O(L+m)$ Rechenschritten.*

Beweis Wir hatten bereits früher angemerkt, dass die Vorausberechnung der NONEMPTY- und LENGTH-Listen in Zeit $O(L+m)$ erfolgen kann. Die weitere Anzahl der Rechenschritte wird durch Phase 2 (die Hauptschleife) von RadixSort dominiert. Der Gesamtaufwand für die Phasen 2(a) und 2(c) lässt sich abschätzen durch die Gesamtlänge der NONEMPTY-Listen. Da jedes Tupel A_i einen Beitrag von maximal ℓ_i zu dieser Gesamtlänge leistet, erhalten wir die Schranke $O(L)$. Um den Gesamtaufwand für die Phase 2(b) abzuschätzen, verwenden wir folgende Buchhaltung: wann immer ein A_i aus Q inspiziert wird (um es in $Q_{a_{ij}}$ einzufügen), belasten wir A_i mit Kosten 1. Da ein A_i nur während der Iterationen $j = \ell_i, \ldots, 2, 1$ in Q verweilt, wird jedes A_i mit Kosten ℓ_i belastet. Daher beträgt der Gesamtaufwand für die Phase 2(b) ebenfalls $O(L)$. Aus dieser Diskussion ergibt sich die Gesamtlaufzeit $O(L+m)$. •

[3]Da Tupel einer Länge j in den Komponenten $j+1, \ldots, k$ eine leere Sequenz repräsentieren, und diese kleiner ist als jede nichtleere Sequenz, sind *alle* in Q befindlichen Daten (also auch die aus LENGTH[j] am Anfang von Q) vor Phase 2(c) gemäß „$<^k_{j+1}$“ sortiert. Diese Beobachtung wäre wichtig für einen Korrektheitsbeweis.

Teil II

Grundlagen

Kapitel 15

Laufzeitanalyse und Asymptotik

Eine penible Bestimmung der Laufzeit eines Algorithmus A könnte darin bestehen, zu jeder potenziellen Eingabeinstanz x die exakte Anzahl von Rechenschritten zu bestimmen, die A auf Eingabe x bis zum Terminieren der Rechnung durchführt. Dies ist aber ein Ansatz, der viel zu detailliert ist, um als Grundlage einer schlüssigen Theorie dienen zu können. Um zu einfachen und dennoch aussagekräftigen Aussagen zu gelangen, betrachten wir das Laufzeitverhalten von Algorithmen aus der folgenden Perspektive:

- Wir fassen die Eingabeinstanzen x derselben Länge n in einer Menge X_n zusammen.[1]
- Wir betrachten die Laufzeit von A als Funktion $T(n)$ in der Eingabelänge n (anstatt $T(x)$ individuell für jede Instanz $x \in X_n$ zu bestimmen).
- Wir identifizieren Funktionen in n miteinander, wenn sie die gleiche asymptotische Wachstumsordung haben.

Zur Definition von $T(n)$ gibt es verschiedene Ansätze:

Worst Case:	$T(n) = \max_{x \in X_n} T(x)$
Best Case:	$T(n) = \min_{x \in X_n} T(x)$
Average Case:	$T(n) = \frac{1}{\lvert X_n \rvert} \sum_{x \in X_n} T(x)$

[1] Für „Länge“ verwenden wir meist eine natürliches Maß wie zum Beispiel die Anzahl n der Knoten eines Graphen.

Da wir in der Vorlesung Algorithmen behandeln, die für alle Eingabeinstanzen eine akzeptable Laufzeit haben, werden wir in der Regel eine Worst Case Analyse durchführen.[2]

Die asymptotische Wachstumsordnung einer Funktion $f : \mathbb{N} \to \mathbb{N}$ wird durch die Landau'sche $O, \Omega, \Theta, o, \omega$-Notation erfasst:

$$\begin{aligned}
O(f) &= \{g | \exists c > 0, n_0 \in \mathbb{N}, \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \ . \\
\Omega(f) &= \{g | \exists c > 0, n_0 \in \mathbb{N}, \forall n \geq n_0 : g(n) \geq c \cdot f(n)\} \ . \\
\Theta(f) &= O(f) \cap \Omega(f) \\
o(f) &= \left\{ g \,\middle|\, \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \right\} \\
\omega(f) &= \left\{ g \,\middle|\, \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \right\}
\end{aligned}$$

Falls der Limes existiert, können $O(f), \Omega(f), \Theta(f)$ auch definiert werden wie folgt:

$$\begin{aligned}
O(f) &= \left\{ g \,\middle|\, \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \right\} \\
\Omega(f) &= \left\{ g \,\middle|\, \lim_{n\to\infty} \frac{g(n)}{f(n)} > 0 \right\} \\
\Theta(f) &= \left\{ g \,\middle|\, 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \right\}
\end{aligned}$$

Aus historischen Gründen schreibt man „$g = O(f)$“ statt „$g \in O(f)$“ und, für eine Menge G von Funktionen von $\mathbb{N}$ nach $\mathbb{N}$, „$G = O(f)$“ statt „$G \subseteq O(f)$“. Analoge Schreibweisen gelten auch für die anderen Landau-Symbole. Ein Beispiel: wir schreiben

$$3n^2 + 7n = 3n^2 + O(n) = \Theta(n^2) = O(n^2) = O(n^3) \tag{15.1}$$

statt

$$3n^2 + 7n \ \in \ 3n^2 + O(n) \ \subseteq \ \Theta(n^2) \subseteq O(n^2) \subseteq O(n^3) \ .$$

Gleichungen der Art (15.1) gelten immer nur von links nach rechts gelesen (und sind unsinnig in der umgekehrten Leserichtung).

[2] In der Vorlesung gehen wir auch kurz auf Best und Average Case Analysen ein.

Lemma 15.0.1 *Es bezeichne $c > 0$ eine von n unabhängige Konstante, f und g seien Funktionen in n, wobei $g = O(f)$. Dann gelten folgende Regeln:*

$$c \cdot f = O(f) \quad \textit{und} \quad f + g = O(f) \ .$$

Beide Regeln gelten auch mit Θ anstelle von O.

Salopp formuliert besagen diese Regeln, dass die O-Notation multiplikative Konstanten und subdominante Terme verschluckt. (Wegen $g = O(f)$ ist f der dominante Term und g der subdominante.) Bei einem Polynom $p(n) = \sum_{i=0}^{k} a_i x^i$ vom Grade k ist $a_k x^k$ (mit $a_k \neq 0$) der dominante Term und die Konstante a_k wird von O verschluckt:

Folgerung 15.0.2 *Es sei p ein Polynom vom Grad k. Dann gilt $p = \Theta(n^k)$ (und somit auch $p = O(n^k)$).*

Vorteile der asymptotischen Laufzeitanalyse. Wenn $g = o(f)$ und daher $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$, dann steht f in der asymptotischen Hackordnung oberhalb von g (d.h. f strebt auf Dauer schneller gegen unendlich). Das schließt nicht aus, dass $g(n)$ erst für sehr große Werte von n kleiner ist als $f(n)$. Warum sind wir bei der Laufzeitanalyse an dieser Art von Asymptotik interessiert? Die Antwort lautet: der Hauptzweck beim Design von zeiteffizienten Algorithmen besteht darin, diese Algorithmen für große Eingabeinstanzen (big data) fit zu machen. Zudem wird unser Maschinenmodell, nämlich die RAM[3], eine grobe Vereinfachung eines realen Computers sein, sodass eine auf der RAM beruhende Analyse die reale Laufzeit sicher nur bis auf eine multiplikative Konstante approximiert. Schließlich hat die asymptotische Laufzeitanalyse den Charme, dass die ermittelten Zeitschranken i.A. eine sehr einfache Form haben (wie etwa $\Theta(n^k)$ für Polynome vom Grad k).

[3]= Random Access Machine (s. Abschnitt 16).

Kapitel 16

Ein einfaches Maschinenmodell

Im Jahre 1945 entwarf John von Neumann eine Rechnerarchitektur, die einfach und dennoch aussagekräftig ist. Obwohl die Hardwaretechnologie sich seit 1945 rasant weiterentwickelt und stark ausdifferenziert hat, ist die von-Neumann-Architektur weiterhin eine gute Grundlage zur Vorhersage des (asymptotischen) Laufzeitverhaltens von Algorithmen. Eine Variante dieser Architektur ist die 1963 von Shepardson und Sturgis vorgeschlagene Random Access Maschine, die wir in diesem Abschnitt skizzieren wollen.

Random Access Maschine (RAM). Wir stellen uns einen Rechner mit einem Programm- und einem Rechenspeicher vor.[1] Beide Speichermodule sind in (unendlich viele) Zellen unterteilt. Die Rechenspeicherzellen notieren wir als $S[0], S[1], S[2], \ldots$. Eine Zelle im Programmspeicher enthält einen Befehl (d.h. eine vom Rechner ausführbare elementare Instruktion); eine Zelle im Rechenspeicher enthält eine „kleine“ ganze Zahl.[2] Auf Zellen kann mit nicht-negativ ganzzahligen Adressen in konstanter Zeit zugegriffen werden (wie bei einem Array). Ganze Zahlen liegen in binärer Kodierung vor und können auch als Bitstring interpretiert werden.
Unser Rechner verfügt weiterhin über eine kleine Anzahl von Registern $R_1, R_2,$

[1] Da ein Programm selber aus Daten besteht, ist diese Zweiteilung eigentlich überflüssig und ignoriert zudem die Möglichkeit, dass Rechner ihr Programm, während der Ausführung desselben, eigenständig modifizieren könnten. Andererseits ist es für unsere Zwecke übersichtlicher, uns Programm- und Rechenspeicher als zwei separate Module vorzustellen (und von der Möglichkeit, dass ein Programm durch seine Ausführung modifiziert wird, werden wir keinen Gebrauch machen.

[2] Der Begriff „klein” wird später präzisiert.

$R_3 \ldots$ (die kleine ganze Zahlen speichern können), einen Befehlszähler und eine arithmetisch-logische Einheit (ALU). Der Befehlszähler enthält stets die Adresse des als nächstes auszuführenden Befehls. Die ALU wird es erlauben, Registerinhalte arithmetisch-logisch miteinander zu verknüpfen.
Bleibt zu klären, welche Befehle wir zulassen und wie der Rechner diese interpretiert.

Transportbefehle: $R_i \leftarrow S[R_j]$ lädt den Inhalt der Speicherzelle, deren Adresse im Register R_j steht, in das Register R_i.
$S[R_j] \leftarrow R_i$ speichert den Inhalt von Register i in der Speicherzelle, deren Adresse in Register R_j steht.
$R_i \leftarrow c$ weist dem Register i die Konstante c zu (Transport einer im Programmspeicher befindlichen Konstante in ein Register).

Rechenbefehle: $R_i \leftarrow R_j + R_k$ weist dem Register R_i die Summe aus den Inhalten der Registern R_j und R_k zu.
Entsprechende Befehle gibt es für andere binäre arithmetische oder logische Operationen (Subtraktion „$-$“, Multiplikation „$*$“, ganzzahlige Division „DIV“ und „MOD“, Vergleiche „$=, \neq, <, \leq, >, \geq$,“, bitweise logische Verundung „$\wedge$“ oder Veroderung „$\vee$“). Die arithmetischen Operationen liefern ganzzahlige Ergebnisse; das Ergebnis eines Vergleichs oder einer logischen Operation lautet TRUE oder FALSE (die Booleschen Wahrheitswerte).
$R_i \leftarrow -R_j$ weist dem Register R_i den mit Faktor -1 negierten Inhalt des Registers R_j zu.
$R_i \leftarrow \neg R_j$ weist R_i den Bitstring zu, der durch bitweise Negation aus dem in R_j gespeicherten Bitstring hervorgeht.

Sprungbefehle: JUMP j weist dem Befehlszähler den Wert j zu (unbedingter Sprung).
JUMP_ZERO j, R_i weist dem Befehlszähler den Wert j zu, sofern das Register R_i den Inhalt 0 hat.
Wenn kein Sprung erfolgt, wird der Befehlszähler stets um 1 inkrementiert.

Stoppbefehl: STOP führt zum Terminieren des Programms.

Wir müssen noch klären, was eine „kleine“ ganze Zahl ist:

Konvention: Bei einer Eingabe der Länge n gilt eine ganze Zahl der Bitlänge $O(\log n)$ als „klein“.

Beachte, dass man mit $k \log n$ Bits Zahlen bis zur Größe $\approx n^k$ repräsentieren kann.

In realen Rechnern haben die Speicherzellen eine feste konstante Wortlänge (wie zum Beispiel 64 Bit). Allerdings wird diese Wortlänge von Zeit zu Zeit erhöht. Reale Rechner (selbst sequentielle) betreiben bis zu einem gewissen Grad Parallelverarbeitung und besitzen eine sogenannte Speicherhierarchie (und entsprechende Paging-Strategien). Solche Details sind in dem einfachen Modell der RAM nicht aufgehoben. Dennoch wird die von der RAM ermittelte asymptotische Wachstumsordnung der Laufzeit $T(n)$ für reale Rechner Aussagekraft haben. In der Vorlesung werden wir ein paar weitere Details zu den Unterschieden zwischen RAM und realen Rechnern kurz ansprechen.

Kapitel 17

Problemorientiert beschriebene Algorithmen

RAM-Befehle bieten zum problemorientierten Algorithmenentwurf bzw. zum problemorientierten Programmieren wenig Komfort, da sie zu hardware-nah und zu simpel gestrickt sind. Höhere Programmiersprachen wie zum Beispiel C++ oder JAVA

- haben mächtigere und bequemer zu nutzende syntaktische Konstrukte,
- fördern strukturierte Programmierung (zum Beispiel Verwendung von Schleifen anstelle eines aus Sprungbefehlen zusammengesetzten „Spaghetti-Codes“)
- und entlasten einen von lästigen Details wie dem Austausch von Daten zwischen Registern und Rechenspeicher.

Der Programmierer kann sich daher auf die Problemlösung konzentrieren, weshalb höhere Programmiersprachen mitunter auch „problem-orientierte Sprachen“ genannt werden. In der Vorlesung werden wir die syntaktischen Konstrukte höherer Sprachen ebenfalls benutzen, wenn auch weniger formal und mehr umgangssprachlich. Es ist daher lehrreich sich klarzumachen, dass sich die Konstrukte höherer Programmiersprachen (eine breite Palette von Datentypen und mächtigere Befehle) implizit auch im RAM-Modell darstellen lassen. Dabei verfolgen wir das

RAM-Simulations-Ziel: Eine Zeitanalyse, welche (ohne den Umweg über die RAM) direkt an einer problemorientierten Beschreibung des Al-

gorithmus ansetzt, wird die Rechenzeit eines entsprechenden RAM-Programmes zwar etwas unterschätzen, aber (in der Regel) maximal um eine multiplikative Konstante (welche von der O-Notation ohnehin unterdrückt wird).

Wir werden die Erreichung dieses Zieles nicht in allen Details formal nachweisen, weil wir dazu einen Compiler schreiben müssten, der eine höhere Programmiersprache in RAM-Code übersetzt. Dies würde den Rahmen der Vorlesung sprengen. Wir werden aber zu Plausibilitätsüberlegungen ausholen und skizzieren, wie eine Laufzeitanalyse ohne permanente Reduktion auf das RAM-Modell ablaufen kann.

17.1 Die RAM-Darstellung verschiedener Datentypen

Typische elementare Datentypen in höheren Programmiersprachen sind zum Beispiel:

integer: Die Daten vom Typ **integer** sind ganze Zahlen. Da in realen Computern die Wortlänge beschränkt ist, deckt sich das so in etwa mit dem Konzept der „kleinen ganzen Zahlen“ bei der RAM.

real: Die Daten vom Typ **real** sind reelle Zahlen (streng genommen rationale Zahlen). Wenn reelle Arithmetik eine große Rolle spielt, benötigt man ein daran besser angepasstes Modell der RAM. In der Vorlesung behandeln wir in der Regel kombinatorische Probleme, bei denen reelle Arithmetik keine große Rolle spielt.

boolean: Die Daten vom Typ **boolean** nehmen die Wahrheitswerte TRUE und FALSE an. Wir können TRUE mit 1 und FALSE mit 0 identifizieren. Dies sind unbestreitbar kleine ganze Zahlen.

char: Die Daten vom Typ **char** sind ASCII-Zeichen[1] (also im Wesentlichen die Zeichen auf einer Tastatur) oder Teilmengen davon. Jedes Zeichen wird im ASCII-Code durch ein Byte (= 8 Bit) repräsentiert und ist somit als kleine ganze Zahl darstellbar.[2]

[1] ASCII = American Standard Code for Information Interchange

[2] Jedes Datum von einem Typ mit endlich vielen Werten ist, via Nummerierung der letzteren, im Prinzip auch als kleine ganze Zahl repräsentierbar.

Interessanter wird unsere Diskussion bei zusammengesetzten Datentypen. Wir starten die Diskussion mit den Arrays. Da der Rechenspeicher der RAM ein unendlich großes 1-dimensionales Array ist, bereiten uns Arrays keine großen Probleme:

ein 1-dimensionales Array: Wir weisen einem Array $A[1:m]$ (also mit den Komponenten $A[1],\ldots,A[m]$) im Rechenspeicher einen Block der Größe m mit einer geeigneten Basisadresse a zu. Dann wird die Komponente $A[j]$ im Rechenspeicher in der Zelle $S[a+j]$ verwaltet.

mehrere 1-dimensionale Arrays: Wir weisen jedem Array einen entsprechenden Block im Rechenspeicher zu. Die Basisadressen werden so gewählt, dass die Blöcke sich nicht überlappen.[3]

mehrdimensionales Array: Ein 2-dimensionales Array $A[1:m,1:n]$ kann „zeilenweise“ in ein 1-dimensionales Array $B[1:mn]$ eingebettet werden, und zwar mit der Gleichsetzung $A[i,j]=B[(i-1)n+j]$. Nach dem gleichen Prinzip können wir bei k-dimensionalen Arrays mit $k\geq 3$ vorgehen.

Wir nehmen als Nächtes Listen ins Visier. Diese kollabieren bei einer FIFO-Verwaltung[4] zu einer QUEUE (= Warteschlange wie an der Kasse im Supermarkt) und bei einer LIFO-Verwaltung[5] zu einem STACK (= Kellerspeicher wie bei einem Bücherstapel). Wir vermerken hier nur kurz, dass Listen, Stacks und Queues mit Hilfe von Arrays implementierbar sind. Wir gehen darauf in Abschnitt 4 näher ein.

Einen großen Raum werden später in der Vorlesung Graphen und Graphalgorithmen einnehmen. Wir vermerken hier nur kurz, dass ein Graph (egal ob gerichtet oder ungerichtet) leicht als Adjazenzmatrix (ein speziell definiertes 2-dimensionale Boolesches Array) oder mit Hilfe von Adjazenzlisten dargestellt werden kann.

Résumée. Elementare Datentypen lassen sich im RAM-Modell darstellen (mit Abstrichen beim Datentyp **real**). Arrays (sogar mehrdimensionale) lassen sich im RAM-Modell darstellen. Listen, Queues und Stacks sind als Arrays implementierbar und daher auch im RAM-Modell darstellbar. Ein

[3] In dieses Konzept sind auch Variable integrierbar. Ist eine Variable von einem elementaren Datentyp, so reicht es, genau eine Zelle im Rechenspeicher dafür vorzusehen.

[4] FIFO = First in First out.

[5] LIFO = Last in First out.

Graph kann als ein 2-dimensionales Array (nämlich als Adjazenzmatrix) gegeben sein oder in Adjazenzlistendarstellung. Daher sind Graphen ebenfalls im RAM-Modell, darstellbar.

Wir skizzieren abschließend, dass die RAM-Darstellung von mehrdimensionalen Arrays dem RAM-Simulations-Ziel genügt:

- Bei unseren Laufzeitanalysen werden wir so tun als würde ein Zugriff auf $A[i,j]$ nur 1 Rechenschritt kosten. Bei der RAM-Darstellung von $A[1:m,1:n]$ als $B[1:mn]$ greifen wir statt auf $A[i,j]$ auf $B[(i-1)n+j]$ zu. Wenn B die Basisadresse b hat, dann befindet sich der Inhalt von $A[i,j]$ in $S[b+(i-1)n+j]$. Dies kostet neben dem eigentlichen Zugriff auf die betreffende Speicherzelle konstant viel Rechenschritte zur Auswertung des arithmetischen Ausdrucks $b+(i-1)n+j$.

- Bei einem k-dimensionalen Array verhält es sich ähnlich, vorausgesetzt wir können k als eine Konstante ansehen, die nicht mit der Eingabelänge n wächst. Dies ist aber der Fall, da a) k im Algorithmus spezifiziert sein muss und b) der Algorithmus durch einen endlichen Text spezifiziert ist, der sich nicht bei Vergrößerung der Eingabelänge ändert.

Wir werden später zu ähnlichen Ergebnissen gelangen, was die RAM-Darstellung von Listen, Queues, Stacks oder Graphen betrifft.

17.2 Die RAM-Simulation komplexerer Befehle

Allgemeinere Wertzuweisungsbefehle. Wir werden so tun, als wären Zuweisungen der Form $x \leftarrow A$ mit einer Variable x und einem Rechenausdruck A in 1 Rechenschritt ausführbar. Somit sollte eine RAM-Simulation dafür auch nur konstante Zeit (konstant $= \mathrm{O}(1)$) beanspruchen. Wir betrachten als Beispiel die Zuweisung $x \leftarrow (a+b)\mathrm{DIV}c$. Wir nehmen an, dass die Variablen a,b,c,x von der RAM in den Speicherzellen mit Adressen i,j,k,ℓ verwaltet werden. Eine RAM-Simulation könnte dann aussehen wie folgt:

$$R_1 \leftarrow S[i]; R_2 \leftarrow S[j]; R_1 \leftarrow R_1 + R_2; R_2 \leftarrow S[k]; R_1 \leftarrow R_1 \,\mathrm{DIV}\, R_2; S[\ell] \leftarrow R_1 \ .$$

Da Computerprogramme stets eine konstante (nicht von der Eingabelänge n abhängige) Größe haben, haben auch im Programm vorkommende arithmetische Ausdrücke A eine konstante Größe. Es sollte daher klar sein, dass

sich beliebige Zuweisungen der Form $x \leftarrow A$ von einer RAM in Zeit $O(1)$ simulieren lassen.

if-then-else Anweisungen. Es sei C eine Bedingung und P_1 und P_2 seien Programme. Wir nehmen an, dass ein RAM-Programm zum Testen von C erarbeitet wurde und dass dieses den Wahrheitswert von C in einer Speicherzelle $S[c]$ abgelegt hat. Wir nehmen weiter an, dass RAM-Programme Q_1 und Q_2 zur Simulation von P_1 und P_2 vorliegen. Es seien j_1 und j_2 die Adressen im Programmspeicher, unter denen wir auf die Startzeilen dieser Programme zugreifen können. Wir werden so tun als würde (nachdem C bereits ausgewertet wurde) die Verzweigung zum Programm P_1 oder P_2 gemäß der Anweisung „**if** C **then** P_1 **else** P_2“ in 1 Rechenschritt möglich sein. Hier ist die entsprechende RAM-Simulation mit 4 Rechenschritten:

$$R \leftarrow c;\ R' \leftarrow S[R];\ \text{JUMP\_ZERO}\ j_2, R';\ \text{JUMP}\ j_1\ .$$

Beachte dabei, dass R' den Wert 0=FALSE genau dann hat, wenn C *nicht* erfüllt ist.

Schleifen. Eine ähnliche Technik kann zur RAM-Simulation von Schleifen verwendet werden. Betrachte die folgende umgangssprachliche Version einer **while**-Schleife:

- Solange Bedingung C erfüllt ist führe P aus.[6]

Es sollte klar sein, dass wir auch diese Kontrollstruktur durch Einsatz der Befehle JUMP und JUMP_ZERO auf einer RAM simulieren können.

Prozeduren. Die Prozedurtechnik ist ein wichtiger Bestandteil höherer Programmiersprachen. Hier geht es zum einen

(a) um die Verzweigung aus dem Hauptprogramm P in eine Prozedur P' und, nach dem Terminieren von P',

(b) um die Rückkehr aus P' an die richtige Stelle in P.

[6]Dies muss so interpretiert werden, dass die Bedingung C nach jeder *vollständigen* Ausführung von P kontrolliert wird. Wenn C weiterhin gilt, wird die nächste *vollständige* Ausführung von P gestartet.

Das entsprechende Verzweigen zum richtigen Block im Programmspeicher der RAM, kann mit den RAM-Sprungbefehlen geregelt werden. Zum anderen findet aber bei dem Aufruf von P und der Rückkehr aus P' jeweils eine Parameterübergabe statt. Daher müssen die betreffenden Blöcke im RAM-Rechenspeicher Platz für diese Parameter vorsehen. Die Parameterübergabe kann mit Transportbefehlen geregelt werden. Eine besondere Herausforderung sind *rekursive* Prozeduren (also Prozeduren, die sich selber aufrufen bzw. sich wechselseitig aufrufen). Eine RAM muss dann im Rechenspeicher einen Block vorsehen, in dem ein sogenannter Rekursions-STACK verwaltet wird. Jeder Prozeduraufruf vergrößert den STACK um einen Teilblock. Jedes Terminieren eines Prozeduraufrufes verkleinert ihn wieder. Zwischen aufeinander folgenden Teilblöcken finden die Parameterübergaben statt. Die Administration von (evtl. rekursiven) Prozeduren seitens der RAM kann an dieser Stelle nicht präzise besprochen werden. Es soll der Hinweis genügen, dass der administrative Overhead den Aufwand nicht um mehr als eine multiplikative Konstante vergrößert. D.h., wenn wir bereits wissen, dass eine Prozedur P eine Laufzeitschranke $O(T_P(n))$ hat, dann wird diese Schranke nicht überschritten, wenn die Prozedur in ein größeres Hauptprogramm eingebettet ist und von dort (oder, im rekursiven Fall, von sich selbst) aufgerufen wird.

Die obigen Überlegungen führen zu einer Reihe von Faustregeln:

1. Die Komposition $P_1; P_2$ zweier Programme mit Laufzeiten $O(T_1)$ und $O(T_2)$ hat die Laufzeit $O(T_1 + T_2)$.

2. Wenn T_i eine Laufzeitschranke für das Programm P_i ist (mit $i = 1, 2$), und der Wahrheitswert der Bedingung C in Zeit T_C ermittelt werden kann, dann kann die Anweisung

   if C **then** P_1 **else** P_2

   in Zeit $O(T_C + \max\{T_1, T_2\}) = O(T_C + T_1 + T_2)$ ausgeführt werden.

3. Wenn ein Programmteil P solange durchlaufen wird, bis eine Bedingung C verletzt ist (**while**-Schleife) und T_i ist die Laufzeit für die i-te Iteration der Schleife (inklusive dem Testen von C), dann kann die Anweisung

   „Solange C gilt führe P aus“ (*)

   in Zeit $O(T_1 + \ldots + T_m)$ ausgeführt werden, wobei m die Anzahl der Iterationen bis zum Ausstieg aus der Schleife bezeichnet. Falls T eine obere Schranke von $T_1, \ldots, T_m$ ist, können wir die Zeit für (*) auch mit $O(mT)$ ansetzen.[7]

[7] Das ist evtl. eine schlechte Abschätzung, insbesondere wenn es viele kleine und wenig

4. Es seien a, b, c von n unabhängige Konstanten. Dann gilt: wenn eine rekursive Prozedur P, angewendet auf eine Eingabe der Länge n,

   - im Falle $n = 1$ maximal a Schritte rechnet,
   - im Falle $n \geq 2$ sich selber b-mal auf Eingaben der Länge n/c aufruft und außerhalb dieser rekursiven Aufrufe weitere $h(n)$ Schritte rechnet,

   dann hat die Laufzeit die gleiche asymptotische Größenordnung wie die Lösung der Rekursionsgleichung

   $$T(1) = a \quad \text{und} \quad T(n) = b \cdot T(n/c) + h(n) \ . \tag{17.1}$$

   Dabei steht natürlich $b \cdot T(n/c)$ für die Rechenzeit, die durch die rekursiven Aufrufe verursacht wird; der Term $h(n)$ steht für den „Overhead“ außerhalb der rekursiven Aufrufe.[8]

Die Bemerkung über die **while**-Schleife gilt sinngemäß auch für andere Schleifenkonstrukte (wie zum Beispiel die **for** ... **do**-Schleife oder die **repeat** ... **until**-Schleife). Warum setzen wir für verschiedene Iterationen einer Schleife verschiedene Laufzeiten $T_1, T_2, \ldots$ an? Ein häufiger Grund besteht darin, dass sich verschiedene Iterationen mit verschiedenen, und unterschiedlich großen, Teilen der Eingabeinstanz beschäftigen. Betrachten wir als Beispiel einen Graphen mit n Knoten und m Kanten, bei welchem wir in der i-ten Iteration alle Nachbarn des Knotens i durchmustern. Nehmen wir an, dass die Laufzeit einer Iteration linear mit der Anzahl der Nachbarn des betreffenden Knotens wächst. Wenn dann $d_1, d_2, \ldots$ die Knotengrade (Knotengrad von i = Anzahl der Nachbarn von i) bezeichnen, dann erhalten wir für die i-te Iteration die Laufzeit $O(d_i)$. Die Knotengrade können von 0 bis $n - 1$ variieren. Eine Abschätzung von d_i durch die obere Schranke $n - 1$ würde zur Abschätzung $\sum_{i=1}^{n} T_i = O(n^2)$ führen. Aus der Graphtheorie ist bekannt, dass die Summe der Knotengrade das Doppelte der Kantenanzahl

große Terme innerhalb $T_1, \ldots, T_m$ gibt.

[8] Und mit „Overhead“ ist nicht der zusätzliche (aber asymptotisch vernachlässigbare) Aufwand gemeint, der durch die Simulation mit einer RAM verursacht wird, sondern die Zeit, die es braucht, um ein Problem der Länge n in b Teilprobleme der Länge n/c zu zerlegen und, nach Rückkehr aus der Rekursion, die Lösungen der Teilprobleme zu einer Lösung des Gesamtproblems zusammenzufügen.

liefert, d.h. $\sum_{i=1}^{n} d_i = 2m$. Dies führt zu der i.A. viel besseren[9] Laufzeitschranke $\sum_{i=1}^{n} T_i = O(m)$. Das Beispiel mit der Graphdurchmusterung ist eine gute Gelegenheit, sich mit einem „Buchhaltertrick“ vertraut zu machen. Es ist i.A. nicht trivial, eine gute Abschätzung für $T_1 + T_2 + \ldots$ zu finden, insbesondere wenn die Terme T_i sehr unterschiedliche Größe haben. Manchmal hilft es, die Laufzeit als Kosten zu interpretieren, die bestimmten kombinatorischen Objekten angelastet werden. Die Gesamtkosten, summiert über die kombinatorischen Objekte, entsprechen dann der Laufzeit. Wenn wir bei Anwendung dieses Buchhaltertricks geschickt waren, dann sind am Ende alle kombinatorischen Objekte mit den gleichen Kosten k belastet. Falls es m solcher Objekte gibt, dann erhalten wir die (scharfe!) Laufzeitschranke $O(km)$. Im obigen Beispiel mit $O(d_i)$ Rechenschritten in der i-ten Iteration, können wir die Kosten $O(d_i)$ so verteilen, dass jede der d_i von i ausgehenden Kanten mit Kosten $O(1)$belastet wird. Über die n Iterationen hinweg wird jede Kante des Graphen mit Kosten $O(1)$ belastet. Also ist die Laufzeit von der Größenordnung $O(m)$. Der Buchhaltertrick hat uns zu demselben Ergebnis geführt wie die Überlegung mit der Summe der Knotengrade zuvor. Der Punkt ist aber, dass der Buchhaltertrick oft auch dann anwendbar ist, wenn keine schöne Formel (wie im Beispiel $\sum_{i=1}^{n} d_i = 2m$) zur Verfügung steht.

Die obige Rekursionsgleichung (17.1) zur Bestimmung der Laufzeit einer rekursiven Prozedur P findet ihre Anwendung bei den sogenannten „Divide&Conquer“-Verfahren (die wir später noch genauer kennenlernen werden). Oftmals gilt insbesondere $h(n) = O(n)$ und $b = c$, d.h. ein Problem der Größe n wird in b Teilprobleme der Größe n/b zerlegt und der Overhead außerhalb der rekursiven Aufrufe ist linear. In diesem Fall hat die Rekursionsgleichung die Lösung $O(n \log n)$.[10] Eine allgemeinere Aussage macht der folgende

Satz 17.2.1 (Master-Theorem) *Es seien a, b, c, d von n unabhängige Konstanten und $h(n) = dn$. Dann hat die Rekursion (17.1) die Lösung*

$$T(n) = \begin{cases} O(n) & \text{falls } b < c \\ O(n \log n) & \text{falls } b = c \\ O\left(n^{\log_c b}\right) & \text{falls } b > c \end{cases} .$$

Hierbei bezeichnet $\log_c b$ den Logarithmus von b zur Basis c.

[9] Die Kantenanzahl m ist zwar im Extremfall quadratisch in der Knotenanzahl n, aber i.A. gilt $m << n^2$.

[10] mit ein Grund, warum extrem viele effiziente Algorithmen die Laufzeit $O(n \log n)$ haben

Ein Beweis dieses Satzes wird üblicherweise in der Vorlesung „Diskrete Mathematik I“ geführt.
Mehr oder weniger offensichtlich[11] ist, dass die Rekursion

$$T(1) = a \quad \text{und} \quad T(n) = T\left(\frac{n}{2}\right) + b \tag{17.2}$$

die Lösung $T(n) = O(\log n)$ hat. Eine Rekursion dieses Typs finden wir zum Beispiel bei der Binärsuche, bei welcher ein Suchraum der Größe n in konstanter Zeit halbiert wird. Das Thema der Binärsuche wird später noch einmal aufgegriffen werden.

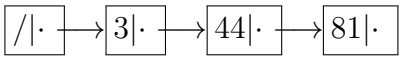
[11] $\log n = k$ gibt gerade an, wie oft $n = 2^k$ halbiert werden muss, bis wir beim Ergebnis 1 anlangen. Expansion der Rekursion liefert $T(n) = b + T(n/2) = 2b + T(n/4) = \ldots = b\log(n) + T(1) = b\log n + a = O(\log n)$.

Kapitel 18

Listen als Arrays

In höheren Programmiersprachen können Listen mit Hilfe von Zeigern (englisch: Pointern) und Verbundtypen (englisch: Records) verwaltet werden. Ein Listenelement vom Verbundtyp besteht aus zwei Teilen: dem eigentlichen Wert und einem Zeiger auf das in der Liste folgende Element (= eine Adresse unter der das nächste Listenelement zu finden ist). In der Vorlesung werden wir diese Art der Listendarstellung veranschaulichen durch sogenannte Box-Pointer-Diagramme.

Wir diskutieren ein längeres Beispiel. Das Box-Pointer-Diagramm zu der Liste $L = (3, 44, 81)$ sieht, bei einfacher Verkettung, aus wie folgt:



Die erste Box (mit leerem Inhalt) repräsentiert den Listenkopf. Danach folgen drei Boxen mit den Inhalten $3, 44, 81$. Der Zeiger der letzten Box ist verkümmert. Man bzeichnet ihn als den Nullzeiger und seinen Wert als **nil**. Er markiert das Ende einer Liste.
Die Array-Darstellung dieser Liste benutzt die Arrays W (für „Wert“) und

NF (für „Nachfolger“) und sieht aus wie folgt:

	W	NF
1		
2	44	3
3	81	0
$L = 4$	/	7
5		
6		
7	3	2
8		
9		
10		
11		
12		

Die Liste L ist durch die Anfangsadresse 4 gegeben. Dort befindet sich der Listenkopf. Mit den Adressen des Arrays NF kann man sich durch die Liste hangeln. NF[3] spielt die Rolle des Nullzeigers. Der Nulleintrag in $NF[3]$ entspricht dem Wert **nil** des Nullzeigers.
Die Pointe bei Listen L ist, dass wir mitten in L ein Element löschen oder einfügen können, ohne alle Elemente der Liste um 1 Position zu verrutschen: wir müssen einfach nur ein paar Zeiger umsetzen. Hätten wir die Liste $L = (3, 44, 81)$ in einem geordneten Array W zusammenhängend unter den Adressen $1, 2, 3$ gespeichert und wollten wir L zu $L = (3, 20, 44, 81)$ aktualisieren, so müssten wir die Einträge 44 und 81 um eine Position nach unten schieben. Beachte, dass bei sehr langen Listen evtl. sehr viele Einträge im Array verschoben werden müssten. Deswegen wird die oben angezeigte Lösung mit den Arrays W und NF gewählt, wobei die benutzten Adressen (im Beispiel $2, 3, 4, 7$) i.A. kein zusammenhängendes Segment mehr bilden. Das Einfügen des Elementes 20 hinter 3 (also hinter der Array-Komponente mit Adresse 7) könnte (unter Verwendung einer freien Speicherzelle wie zum Beispiel die mit Adresse 9) realisiert werden wie folgt:

$$W[9] \leftarrow 20;\ \mathrm{NF}[9] \leftarrow \mathrm{NF}[7];\ \mathrm{NF}[7] \leftarrow 9\ .$$

Es wird also ein neues Array-Element mit der Adresse 9 präpariert und durch Umänderung von ein paar Adressen an die gewünschte Stelle in der Liste eingehängt. Woher aber konnten wir wissen, dass unter der Adresse 9 eine

freie Speicherzelle zu finden war? Dieser Frage wenden wir uns im nächsten Absatz zu.

Wenn eine Liste um einen Eintrag erweitert werden soll, so wird dies in einer höheren Programmiersprache durch einen Befehl vom Typ NEW vorbereitet. Dieser bewirkt, dass hinreichend viele freie Zellen des Rechenspeichers für den neuen Eintrag bereitgestellt werden. Umgekehrt kann auch Speicherplatz, der nicht mehr benötigt wird, freigegeben werden, nämlich mit einem Befehl vom Typ DISPOSE. Eine RAM, die das Programm einer höheren Programmiersprache simuliert, wird i.A. mehrere Listen gleichzeitig verwalten. Zudem muss sie, wegen der Befehle vom Typ NEW und DISPOSE einen hinreichend großen Vorrat an freien Speicherzellen bereithalten. Wir illustrieren die entsprechende Vorgehensweise wieder an unserem obigen Beispiel. Die Belegung der Arrays W und NF bei Vorliegen einer zweiten Liste $L' = (13, 19)$ und einer Liste FREI für die Verwaltung der freien Speicherzellen könnte aussehen wie folgt:

	W	NF
FREI = 1	/	9
2	44	3
3	81	0
$L = 4$	/	7
5		10
6	19	0
7	3	2
8	13	6
9		12
10		0
$L' = 11$	/	8
12		5

Ein Hangeln durch die Liste FREI zeigt, dass die Zellen mit den Adressen $1, 9, 12, 5, 10$ derzeit frei sind, wobei 1 mit dem Listenkopf der Liste FREI belegt ist. Nehmen wir an, dass L (wie oben schon mal durchexerziert) um einen neuen Eintrag mit Wert 20 erweitert werden soll. Wenn wir aus der Liste FREI (ohne den Listenkopf zu ändern) die Zelle mit Adresse NF[FREI] = NF[1] = 9 entfernen, kann diese zur Erweiterung von L um den Eintrag 20 genutzt werden. NF[FREI] sollte anschließend mit NF[NF[$FREI$]] = 12 belegt sein. Dies leisten die folgenden (leicht RAM-simulierbaren) Befehle:

1. $R \leftarrow \mathrm{NF}[\mathrm{FREI}]$; $W[R] \leftarrow 20$; $\mathrm{NF}[\mathrm{FREI}] \leftarrow \mathrm{NF}[R]$.
2. $\mathrm{NF}[R] \leftarrow \mathrm{NF}[7]$; $\mathrm{NF}[7] \leftarrow R$.

In der Phase 1 wird die Array-Komponente mit Adresse $\mathrm{NF}[\mathrm{FREI}] = 9$ aus der Liste FREI ausgehängt, und der Wert 20 wird in ihr vorsorglich schon mal eingetragen. In Phase 2 wird diese Array Komponente in die Liste L hinter der L-Komponente mit Adresse 7 (und Wert 3) eingehängt. Die neue Belegung der Arrays W und NF sieht aus wie folgt:

	W	NF
FREI = 1	/	12
2	44	3
3	81	0
$L = 4$	/	7
5		10
6	19	0
7	3	9
8	13	6
9	20	2
10		0
$L' = 11$	/	8
12		5

Doppelt verkettete Listen. Bei diesen hat ein Listenelement zwei Zeiger. Einer zeigt (wie bisher) auf das nächste Listenelement, der andere auf das vorhergehende. In der Array-Implementierung benötigen wir dann neben den Arrays W und NF ein weiteres Array VG (VG = Vorgänger). Der Vorteil der doppelten Verkettung ist, dass sich die Liste in beide Richtungen (nach rechts und nach links) durchmustern lässt (auf Kosten der Administration von zwei Zeigern statt nur einem). Die obige Diskussion von einfach verketteten Listen und ihrer Array-Implementierung überträgt sich sinngemäß auf doppelt verkettete Listen.

Stacks (Kellerspeicher). Ein STACK ist eine Liste, die nach dem LIFO[1]-Prinzip verwaltet wird. Die Operationen INSERT (Einfügen eines neuen Listenelementes) und DELETE (Löschen eines Listenelementes) passieren im

[1] Last in First Out

STACK immer „oben“, d.h. am Ende der Liste. Mit der Operation TOP kann man das oberste Element des STACK auslesen. Zur Array-Verwaltung eines STACK der Maximalgröße n genügt ein Array $W[1:n]$ und eine Variable TOP, die stets die Adresse des obersten STACK-Elementes enthält. Wenn der STACK aktuell die Größe $s \leq n$ hat, dann befinden sich die im STACK gespeicherten Werte in den Array-Komponenten $W[1], \ldots, W[s]$; $W[1]$ liegt ganz unten im STACK, $W[s]$ ganz oben und TOP enthält die Adresse s. Bei STACKS wird oft PUSH in der Bedeutung von INSERT und POP in der Bedeutung von DELETE verwendet.

Queues (Warteschlangen). Eine QUEUE ist eine Liste, die nach dem FIFO[2]-Prinzip verwaltet wird. Die Operation INSERT (Einfügen eines neuen Listenelementes) passiert immer hinten in der Queue, die Operation DELETE (Löschen eines Listenelementes) immer vorne. Mit der Operation FRONT kann man das vorderste Element der Queue auslesen, mit der Operation REAR das hinterste. Zur Array-Verwaltung einer QUEUE der Maximalgröße n genügt ein Array $W[0:n-1]$ sowie die Variablen FRONT und REAR, die stets die Adresse des vordersten bzw. hintersten QUEUE-Elementes enthalten. Wenn die QUEUE aktuell die Größe $s \leq n$ hat, dann befinden sich die in der QUEUE gespeicherten Werte in einem zusammenhängenden Segment der Größe s des Array, sagen wir in den Array-Komponenten $W[i], \ldots, W[i+s-1 \bmod n]$. $W[i]$ ist das vorderste und $W[i+s-1 \bmod n]$ das hinterste Element der QUEUE. FRONT enthält die Adresse i und REAR die Adresse $i+s-1 \bmod n$. Wird ein neues Element eingefügt, so geschieht dies hinten, d.h. in Zelle $i+s \bmod n$ (und REAR wird modulo n um 1 inkrementiert). Wird ein Element gelöscht, so geschieht dies vorne, d.h. in Zelle i (und FRONT wird modulo n um 1 inkrementiert). Durch diese Art der Verwaltung wandert das die QUEUE enthaltende Array-Segment gewissermaßen im Kreis herum.[3] Bei QUEUES wird oft ENQUEUE in der Bedeutung von INSERT und DEQUEUE in der Bedeutung von DELETE verwendet.

Listen, Stacks, Queues und Laufzeitanalyse. Wir werden bei dem Umgang mit Listen, Queues und Stacks so tun, als ob die folgenden Operationen jeweils nur 1 Rechenschritt erfordern:

- Auslesen des ersten oder letzten Elementes einer Liste

[2]First in First Out

[3]Der Kreis ist das System $\{0, 1, \ldots, n-1\}$ der kleinsten Reste modulo n.

- Von einem Listenelement ausgehend zugreifen auf das nächste (bei doppelter Verkettung auch vorige) Listenelement
- Löschen eines Listenelementes, sofern die Adresse des vorangehenden Listenelementes gegeben ist.
- Einfügen eines Listenelementes hinter ein schon vorhandenes Listenelement, dessen Adresse gegeben ist.
- die Operationen TOP, PUSH, POP bei einem STACK
- die Operationen REAR, FRONT, ENQUEUE, DEQUEUE bei einer QUEUE

Aus den Ausführungen in diesem Abschnitt sollte klar geworden sein, dass eine RAM-Simulation der genannten Operationen in Zeit $O(1)$ möglich ist.

Wir nennen abschließend noch ein paar wichtige Anwendungen von Stacks und Queues in der Informatik. Beginnen wir mit den Stacks:

- Implementierung rekursiver Prozeduren
- Auswertung arithmetischer Ausdrücke
- kontextfreie Analyse

Nun zu typischen Anwendungen von Queues:

- Realisierung von Warteschlangen in Betriebssystemen (Prozesse, Nachrichten, Druckaufträge, et cetera)
- Verwaltung von Warteschlangen in Informationsnetzwerken (Pakete, die in einem Puffer darauf warten, zu ihrer Zieladresse weitergeroutet zu werden)

Kapitel 19

Korrektheitsnachweise

Es verschafft dem Designer von Algorithmen stets eine tiefe Befriedigung, wenn sein Algorithmus genau das macht, was er machen soll. Wir werden in diesem Abschnitt Methoden kennenlernen, das korrekte Verhalten eines Algorithmus mit einem mathematischen Beweis sicherzustellen. Das schließt nicht aus, dass eine Software, welche den Algorithmus implementiert, fehlerhaft ist. Es bedeutet aber: wenn wir bei der Implementierung darauf achten, den Algorithmus nicht durch Programmierfehler abzuändern, dann ist auch das resultierende Computerprogramm korrekt.

19.1 Schleifeninvarianten und Fortschrittsmaße

Betrachten wir als einfaches Beispiel die Binärsuche in einem geordneten Array $A[0:n]$ mit $A[0] < A[1] < \ldots < A[n]$. Gegeben ist eine Zahl x mit $A[0] < x \leq A[n]$. Gesucht ist der eindeutige Index $u \in [n]$ mit $A[u-1] < x \leq A[u]$. Unsere Voraussetzung $A[0] < x \leq A[n]$ stellt sicher, dass der gesuchte Index sich in dem Bereich $[n] = \{1, \ldots, n\}$ befindet. Das Wesen der Binärsuche besteht darin, mit Hilfe eines einzigen Schlüsselvergleichs, einen Suchraum der Größe n zu halbieren. Diese Vorgehensweise kann iteriert werden, bis der Suchraum nur noch aus einem Element u besteht, welches dann der gesuchte Index sein muss. Hier ist der entsprechende Algorithmus:

Eingabe: Gegeben sind zwei Zahlen $n \geq 1$ und x sowie ein geordnetes Array $A[0:n]$.

Voraussetzung: $A[0] < x \leq A[n]$.

Aufgabe: Finde den Index $u \in [n]$ mit $A[u-1] < x \leq A[u]$.

Methode: Wir verwenden **Binärsuche (BS)**.

1. $l \leftarrow 0$; $u \leftarrow n$.
2. Solange $l \neq u - 1$, mache folgendes:
   (a) $m \leftarrow \lceil \frac{1}{2}(l+u) \rceil$.
   (b) Falls $x > A[m]$: $l \leftarrow m$;
   andernfalls: $u \leftarrow m$.
3. Gib u aus.

Konvention: Eingaben, die eine unter „Voraussetzung“ genannte Bedingung erfüllen, nennen wir von nun an „legitim“.

Der Algorithmus **BS** verwendet die Parameter l und u, um den Suchraum $\{l+1, \ldots, u\}$ einzugrenzen. Die folgende Bedingung gilt vor und nach jedem Lauf durch die Hauptschleife (Schritt 2):

$$A[l] < x \leq A[u] \ . \tag{19.1}$$

Der Nachweis solcher „Invarianzbedingungen“ geschieht über vollständige Induktion nach der Anzahl der Schleifendurchläufe.

Induktionsanfang: Anfangs gilt $l = 0$, $u = n$ (wegen Schritt 1) und $A[0] < x \leq A[n]$ (wegen der Legitimität der Eingabe). Vor dem ersten Schleifendurchlauf gilt demnach die Bedingung (19.1).

Induktionsschritt: Wir nehmen an, dass vor einem Schleifendurchlauf die Bedingung $A[l] < x \leq A[u]$ erfüllt ist. Aus $x > A[m]$ folgt $A[m] < x \leq A[u]$. Somit bleibt die Bedingung (19.1) korrekt, wenn wir l auf den neuen Wert m aktualisieren. Aus $x \leq A[m]$ folgt $A[l] < x \leq A[m]$. Somit bleibt die Bedingung (19.1) korrekt, wenn wir u auf den neuen Wert m aktualisieren.

Wir wollen nun nachweisen, dass **BS** den korrekten Index u (mit $A[u-1] < x \leq A[u]$) berechnet. Wenn **BS** aus der Hauptschleife aussteigt, muss die Abbruchbedingung $l = u - 1$ und somit $A[l] = A[u-1]$ erfüllt sein. Zudem, wie eben induktiv nachgewiesen, muss $A[l] < x \leq A[u]$ gelten. Wegen

$A[l] = A[u-1]$ ist u in der Tat der gesuchte Index. Aus dieser kleinen Diskussion können wir Folgendes schließen: falls **BS** terminiert (und daher irgendwann aus der Hauptschleife aussteigt), dann liefert **BS** das korrekte Ergebnis.

Schleifeninvarianten verweisen auf Ordnungsmerkmale, die im Laufe eines dynamischen Prozesses aufrecht erhalten werden. Daneben muss es aber noch ein anderes, dynamisches, Element geben: ein Maß für den Fortschritt, der in jedem Durchlauf der Schleife erzielt wird. Da jede Schleife irgendwann wieder verlassen werden sollte, muss der Fortschritt sich auch darin äußern, dass man sich in jeder Iteration auf eine messbare Weise dem Abbruchkriterium annähert.

Wie verhält sich das am Beispiel der Binärsuche? Wir betrachten als Maß für Fortschritt die Größe des Suchraumes: der gesuchte Index befindet sich wegen der Invarianzbedingung $A[l] < x \leq A[u]$ stets im Suchraum $\{l+1, \ldots, u\}$ der Größe $u-l$. Da wir m stets als das (gerundete) arithmetische Mittel von l und u wählen, wird der Suchraum in jeder Iteration (fast) halbiert.[1] Hieraus folgt, dass die Abbruchbedingung $l = u-1$ nach spätestens $\lceil \log n \rceil$ Iterationen erfüllt sein muss. Denn:

- Anfangs hat der Suchraum die Größe n.
- Somit hat er nach höchstens $\lceil \log n \rceil$ Iterationen die Größe 1 (d.h. $l = u-1$ und der Suchraum ist identisch zu $\{u\}$).

Fassen wir zusammen: **BS** verlässt nach maximal $\lceil \log n \rceil$ Iterationen die Hauptschleife und liefert dann in Schritt 3 den gesuchten Index. Da pro Iteration nur $O(1)$ Rechenschritte erfolgen, gelangen wir zu dem

Satz 19.1.1 *Zu einem gegebenen geordneten Array $A[0:n]$ und einem Schlüsselwert x mit $A[0] < x \leq A[n]$ liefert der Algorithmus* **BS** *nach $O(\log n)$ Schritten den Index u mit $A[u-1] < x \leq A[u]$.*

Nehmen wir an A wäre ein unendlich großes Array mit den Komponenten $A[1], A[2], A[3] \ldots$. A sei wieder geordnet, d.h. $A[1] < A[2] < A[3] < \ldots$. Es sei x ein Schlüsselwert, der die Bedingung $A[1] < x$ erfüllt. Wir suchen den Index $u^* \geq 2$ mit $A[u^*-1] < x \leq A[u^*]$. Bei diesem Problem können wir nicht unmittelbar Binärsuche verwenden, da uns erst einmal keine Schranke u

[1] Ein Suchraum einer geraden Größe wird exakt halbiert; ein Suchraum einer ungeraden Größe, sagen wir der Größe $2g+1$, hat nach der nächsten Iteration die Größe g oder $g+1$.

bekannt ist, welche den Suchraum nach oben begrenzt. Wir stellen uns also das Ziel, eine solche obere Schranke zu bestimmen. (Danach kann wieder Binärsuche eingesetzt werden.) Hier ist der Algorithmus zur Ermittlung von u mit $x \leq A[u]$.

Eingabe: Gegeben ist eine Zahl x sowie ein geordnetes Array A mit den Komponenten $A[1], A[2], A[3], \ldots$.

Voraussetzung: $A[1] < x$.

Aufgabe: Finde einen geradzahligen Index $u \in \mathbb{N}$ mit $A[u/2] < x \leq A[u]$.

Methode: Wir verwenden **Exponentialsuche (ES)**.

1. $u \leftarrow 2$.
2. Solange $x > A[u]$: $u \leftarrow 2u$.
3. Gib u aus.

Es bezeichne u^* den Index mit

$$A[u^* - 1] < x \leq A[u^*] \ . \tag{19.2}$$

Es ist leicht zu sehen, dass die von **ES** verwendete Variable u nach s Iterationen der Hauptschleife (Schritt 2) den Wert 2^{s+1} hat. Außerdem wird die Hauptschleife verlassen, sowie die Abbruchbedingung $A[u] \geq x$ erfüllt ist. Daher hat u nach Verlassen der Hauptschleife den Wert 2^s mit

$$A[2^{s-1}] < x \leq A[u^*] \leq A[2^s] \ .$$

Demzufolge ist 2^s ist die kleinste Zweierpotenz, die nicht kleiner als u^* ist, und wir verlassen die Hauptschleife nach weniger als $\log u^*$ Iterationen. Pro Iteration erfolgen nur $O(1)$ viele Rechenschritte. Unsere kleine Diskussion lässt sich zusammenfassen wie folgt:

Satz 19.1.2 *Zu einem unendlichen geordneten Array A mit den Komponenten $A[1], A[2], A[3], \ldots$ und einem Schlüsselwert $x > A[1]$ liefert der Algorithmus* **ES** *nach $O(\log u^*)$ Rechenschritten einen Index u mit $A[u/2] < x \leq A[u^*] \leq A[u]$. Hierbei bezeichnet u^* den in (19.2) spezifizierten Index.*

Gute Programmierpraxis versus Fokus auf das Wesentliche. In der Vorlesung werden wir oft voraussetzen, dass die Eingabe eine bestimmte Form hat und daher legitim ist. S. zum Beispiel die Voraussetzungen, die wir bei den Algorithmen **BS** und **ES** getätigt haben. Es ist gute Programmierpraxis, zu Anfang eines Computerprogrammes mit etwas Extra-Code sicherzustellen, dass die Eingabedaten diese Voraussetzungen erfüllen. In der Vorlesung werden wir uns aber nicht mit den Details einer solchen Kontrolle belasten, da uns diese von den zentralen Aspekten eines algorithmischen Verfahrens ablenken würden.
Es ist weiterhin gute Programmierpraxis, Schleifeninvarianten bei der Programmdokumentation anzugeben.

19.2 Rekursive Verfahren

Die Algorithmen **BS** und **ES** können leicht als rekursive (sich selbst aufrufende) Prozeduren angegeben werden. Wir gehen dabei von denselben Eingabedaten und Voraussetzungen aus wie zuvor. Das geordnete Array stehe unter der Bezeichnung A zur Verfügung.[2] Die Parameter l und u, die den Suchraum abstecken, müssen bei Prozeduraufrufen als Parameter übergeben werden, da sie ihre Werte von Aufruf zu Aufruf dynamisch ändern. Den Schlüsselwert x betrachten wir ebenfalls als Prozedurparameter (obschon er seinen Wert während der Rechnung nicht ändert). Es folgt die rekursive Variante der Binärsuche.

Prozedur: BS(l, u, x)

lokale Variable: m mit nicht-negativ ganzzahligen Werten

Voraussetzung: $A[l] < x \leq A[u]$.

Vorgehensweise: Falls $l = u - 1$, so gib u aus;
andernfalls mache weiter wie folgt:

1. $m \leftarrow \lceil \frac{1}{2}(l + u) \rceil$.
2. Falls $x > A[m]$: gib BS(m, u, x) aus;
   andernfalls: gib BS(l, m, x) aus.

[2] In einer höheren Programmiersprache wäre A eine sogenannte „globale“ Variable.

Konvention: Aufrufe einer rekursiven Prozedur, die eine unter „Voraussetzung“ genannte Bedingung erfüllen, nennen wir von nun an „legitim“.

Im Hauptprogramm würde der Aufruf $\mathrm{BS}(0, n, x)$ erfolgen.

Wir können $\mathrm{BS}(l, u, x)$ mit dem Ausgabeparameter der Prozedur identifizieren.[3] D.h. $\mathrm{BS}(l, u, x)$ ist der Index, den die Prozedur irgendwann ausgibt. Der Korrektheitsbeweis würde mit vollständiger Induktion folgendermaßen ablaufen:

- Beim Induktionsanfang beschäftigen wir uns mit dem Fall $l = u - 1$, der zum Ausstieg aus der Rekursion führt. Wegen der Voraussetzung $A[l] < x \leq A[u]$ ist hier die Ausgabe von u gerechtfertigt.
- Beim Induktionsschritt dürfen wir induktiv voraussetzen, dass die rekursiven Aufrufe (sofern sie legitim sind!) ein korrektes Ergebnis liefern. (Die Legitimität der rekursiven Aufrufe wird in unserem Beispiel der Binärsuche natürlich mit Hilfe der Abfrage „$x > A[m]$?“ sichergestellt.)

Es folgt, dass die Prozedur korrekt ist, sofern sie terminiert. Die Frage der Terminierung wird durch eine Laufzeitanalyse beantwortet. Die rekursive Prozedur BS ist ein ''Divide and Conquer“-Verfahren, dass ein Problem der Größe n in konstanter Rechenzeit auf ein Teilproblem der Größe $\approx n/2$ reduziert. Daher hat die Laufzeit die gleiche asymptotische Größenordnung wie die Lösung der Rekursionsgleichung

$$T(1) = 1 \quad \text{und} \quad T(n) = T(n/2) + 1 \ . \tag{19.3}$$

Wie wir bereits wissen, ist die Lösung von der Größenordnung $O(\log n)$.

Es ist leicht, eine rekursive Variante der Exponentialsuche anzugeben. Wir wollen aber stattdessen lieber eine rekursive Prozedur für die Methode „Iteriertes Quadrieren (IQ)“ zur Berechnung einer Potenz a^n angeben.

Prozedur: $\mathrm{IQ}(a, q)$

Voraussetzung: $q \geq 0$, a und q sind nicht beide 0.

Vorgehensweise: Falls $q = 0$: gib 1 aus.
andernfalls mache weiter wie folgt:

[3] In höheren Programmiersprachen spricht man von einer Funktionsprozedur. Mathematisch gesehen wird BS wie eine Funktion betrachtet, welche die Eingabe- in die Ausgabedaten transformiert.

- Falls q ungerade: gib $a \cdot IQ(a, q-1)$ aus;
  andernfalls: gib das Quadrat von $IQ(a, q/2)$ aus.

Im Hauptprogramm würde der Aufruf IQ(a, n) erfolgen. Unter Verwendung der Rechenregeln

$$a^0 = 1, a^q = a \cdot a^{q-1} \text{ und } (a^{q/2})^2 = a^q$$

lässt sich induktiv leicht zeigen, dass der Aufruf $IQ(a, q)$ die Potenz a^q korrekt berechnet. Wir beobachten, dass spätestens jeder zweite rekursive Aufruf den Exponenten q halbiert. Der Aufruf IQ(a, n) inszeniert offensichtlich ein „Divide and Conquer"-Verfahren, das ein Problem der Größe n in konstant vielen Rechenschritten auf ein Teilproblem der Größe $n/2$ bzw. $(n-1)/2$ reduziert. Die Laufzeit lässt sich daher asymptotisch abschätzen durch eine Rekursionsgleichung vom Typ (19.3) mit der Lösung $O(\log n)$.

Schlussbemerkung. Der Nachweis einer Invarianzbedingung mit vollständiger Induktion ist (mit etwas Übung) meist relativ leicht zu bewerkstelligen. Der kreative Teil ist das Auffinden von Invarianzbedingungen bzw. das Design eines Algorithmus, welches zu diesen Bedingungen führt.

Kapitel 20

Binäre Suchbäume ohne Rebalancierung

Wir gehen den ganzen Abschnitt über davon aus, dass ein Binärbaum T in der Form $(r, \mathrm{LS}, \mathrm{RS}, E, \mathrm{KEY})$ mit hinreichend großen Arrays $(\mathrm{LS}, \mathrm{RS}, E, \mathrm{KEY})$ gegeben ist. Dabei werden Knoten in T durch Indizes im Adressraum der Arrays repräsentiert. Die Variable r gibt die Wurzel von T an. Wenn i ein Knoten in T ist, so geben $\mathrm{LS}[i]$ und $\mathrm{RS}[i]$ das linke und rechte Kind von i an (wobei der Wert 0 das Fehlen eines Kindes anzeigt). $E[i]$ ist das im Knoten i gespeicherte Element und $\mathrm{KEY}[i]$ ist der zugehörige Schlüsselwert.

Es bezeichne T_i den Unterbaum von T mit der Wurzel i. Wenn j das linke (bzw. rechte) Kind von i ist, so heißt T_j der linke (bzw. rechte) Teilbaum von T_i. T heißt *binärer Suchbaum*, falls für alle Knoten i gilt:

- Alle Knoten im linken Teilbaum von T_i haben einen Schlüssel, der kleiner als $\mathrm{KEY}[i]$ ist.
- Alle Knoten im rechten Teilbaum von T_i haben einen Schlüssel, der größer als $\mathrm{KEY}[i]$ ist.

Diese Organisation von Suchbäumen wird es ermöglichen, Binärsuchen nach einem Schlüsselwert k zu betreiben (wie wir später noch genauer ausführen werden).

In diesem Abschnitt werden wir demonstrieren, wie sich die Wörterbuch-Operationen MEMBER, FIND, INSERT und DELETE auf einem Suchbaum implementieren lassen. Wir machen dabei Gebrauch von einer Freispeicherverwaltung, welche uns die Operationen NEW und DISPOSE zur

Verfügung stellen. Mit dem Kommando NEW erhalten wir (in konstanter Zeit) die Adresse einer aktuell freien Array-Komponente. Mit dem Kommando DISPOSE(j) können wir eine nicht mehr benötigte Adresse j an die Freispeicherverwaltung zurückgeben. Wie unschwer zu erraten ist, wird NEW bei der Implementierung von INSERT und DISPOSE bei der Implementierung von DELETE zum Einsatz kommen.

Es wird sich zeigen, dass jede Wörterbuch-Operation in Zeit $O(d(T))$ implementiert werden kann, wobei $d(T)$ die Tiefe von T (also die maximale Anzahl von Kanten auf einem Weg von der Wurzel zu einem Blatt) bezeichnet. Wir werden keine Anstrengung unternehmen, die Tiefe von T mit Hilfe von Rebalancierungen zu kontrollieren. In einem späteren Abschnitt werden wir eine Rebalancierungs-Technik kennenlernen, mit der $d(T)$ nach oben durch $O(\log n)$ beschränkt werden kann, wobei n die maximale Anzahl der Knoten in T bezeichnet.

Im Folgenden beschreiben wir Implementierungen der Wörterbuch-Operationen dadurch, dass wir präzise sagen, wie die Arrays LS,RS,E,KEY manipuliert werden. In der Vorlesung werden wir diese Implementierungen nicht so formal angeben. Stattdessen visualisieren wir die Array-Manipulationen, indem wir an der Tafel aufmalen, wie sich in dem Baum T ein Suchpfad aufbaut und wie sich, im Falle von INSERT und DELETE, der Baum T dynamisch verändert.

Wir beginnen nun unseren Streifzug durch die Wörterbuchoperationen mit der Prozedur MEMBER.

Prozedur: MEMBER(j, k) — eine Boolesche Funktionsprozedur —

Parameter: ein Knoten j in T und ein Schlüsselwert k

Frage: Enthält der Unterbaum T_j ein Element mit Schlüsselwert k?

Methode: Binärsuche

1. Solange $j \neq 0$ mache Folgendes:
   - Falls $k <$ KEY$[j]$: $j \leftarrow LS[j]$.
     Falls $k >$ KEY$[j]$: $j \leftarrow RS[j]$.
     Falls $k =$ KEY$[j]$: gib TRUE aus und stoppe.
2. Gib FALSE aus und stoppe.

Mit dem Aufruf MEMBER(r, k) könnten wir testen, ob der Gesamtbaum T ein Element mit Schlüsselwert k enthält.

Beispiel 20.0.1 *Es bezeichne r die (Nummer der) Wurzel des in Abb. 20.1 links oben gegebenen Baumes. Durch den Prozeduraufruf MEMBER$(r, 52)$ würde der Suchpfad über die Schlüsselwerte $7, 14, 33, 52$ aufgebaut und das Ergebnis wäre TRUE. Durch das Kommando MEMBER$(r, 26)$ wäre der Suchpfad entsprechend $7, 14, 33, 19$. Dann würde auffallen, dass der Knoten mit Schlüssel 19 kein rechtes Kind hat, und das Ergebnis der Prozedur wäre FALSE.*

Die Prozedur FIND ist ein enger Verwandter von MEMBER.

Prozedur: FIND(j, k) — eine E-wertige Funktionsprozedur —

Parameter: ein Knoten j in T und ein Schlüsselwert k

Voraussetzung: T_j enthält ein Element mit Schlüsselwert k.

Aufgabe: Ermittle das betreffende Element.

Methode: Binärsuche

Mache bis zum Erreichen des Stopp-Befehls Folgendes:

- Falls $k <$ KEY$[j]$: $j \leftarrow LS[j]$.
  Falls $k >$ KEY$[j]$: $j \leftarrow RS[j]$.
  Falls $k =$ KEY$[j]$: gib $E[j]$ aus und stoppe.

Mit dem Aufruf FIND(r, k) könnten wir das Element mit Schlüsselwert k im Gesamtbaum T aufspüren.

Um ein neues Element in T einzufügen, präparieren wir einen entsprechenden neuen Eintrag für unsere Arrays, betreiben dann Binärsuche nach der Stelle in T, wo eingefügt werden sollte, und fügen schließlich dort ein. Hier ist die dazu passende Prozedur:

Prozedur: INSERT(x, k)

Parameter: ein Element x und ein Schlüsselwert k

lokale Variable: ein Array-Index i

Voraussetzung: T enthält kein Element mit Schlüsselwert k.

Aufgabe: Füge x mit Schlüsselwert k (an der richtigen Stelle) in T ein.

Methode: neuen Record präparieren (Schritt 1), Binärsuche (Schritte 2a,2b) und Einfügen (Schritt 2c)

1. $n \leftarrow$ NEW; $E[n] \leftarrow x$; KEY$[n] \leftarrow k$; LS$[n] \leftarrow 0$; RS$[n] \leftarrow 0$.
2. Falls $r = 0$: $r \leftarrow n$.
   Andernfalls mache weiter wie folgt:
   (a) $j \leftarrow r$.
   (b) Solange $j \neq 0$ mache Folgendes:
      i. $i \leftarrow j$.
      ii. Falls $k <$ KEY$[j]$: $j \leftarrow LS[j]$.
      Falls $k >$ KEY$[j]$: $j \leftarrow RS[j]$.
   (c) Falls $k <$ KEY$[i]$: LS$[i] \leftarrow n$.
   Falls $k >$ KEY$[i]$: RS$[i] \leftarrow n$.

Die Abfrage $r = 0$ ist dem Sonderfall eines anfänglich leeren Baumes T geschuldet. In diesem Fall enthält T nach INSERT nur den Knoten n mit Element x und Schlüssel k. Im Falle $r \neq 0$ läuft der Index i immer einen Schritt hinter dem Index j her. Wenn j den Wert 0 hat (die Abbruchbedingung für die Hauptschleife), dann ist i das Blatt in T, dessen Kind der neue Knoten n werden sollte.

MEMBER, FIND und INSERT hätten auch als rekursive Prozeduren entworfen werden können. Für enttäuschte Liebhaber von Rekursion, werden wir weiter unten für DELETE ein rekursives Design wählen.

Das Entfernen eines Elementes mit Schlüsselwert k aus T bedarf einer Fallunterscheidung:

- Falls k in einem Blatt von T gespeichert ist, so kann dieses einfach entfernt (und mit DISPOSE dem Freispeicher zugeeignet) werden. (Wenn dadurch der Baum T leer wird, setzen wir $r = 0$.)

- Falls k in einem Knoten v mit nur einem Kind w gespeichert ist, dann können wir den auf v gerichteten Zeiger des Vaters von v auf w umlenken (womit v aus T „ausgehängt“ wird und mit DISPOSE wieder freigegeben werden kann). Der Spezialfall, dass v die Wurzel von T ist, muss dabei gesondert behandelt werden.

- Falls k in einem Knoten v mit zwei Kindern, sagen wir linkem Kind v_0 und rechtem Kind v_1, gespeichert ist (der komplizierteste Fall), dann gehen wir vor wie folgt:
  - Wir suchen den Knoten q mit minimalem Schlüsselwert in T_{v_1}. Hierbei verwenden wir eine Hilfsprozedur SEARCH_MIN, deren Details wir zu gegebener Zeit enthüllen.
  - Wir überschreiben den Schlüsselwert von v mit KEY$[q]$ und $E[v]$ mit $E[q]$.
  - Anschließend eliminieren wir q aus T. Da q (als Knoten mit minimalem Schlüsselwert in T_{v_1}) kein linkes Kind haben kann, ist q entweder ein Blatt oder ein Knoten mit nur einem Kind (und zwar dem rechten). Daher rutschen wir mit der Elimination von q in einen der zuvor diskutierten einfacheren Fälle.

Beispiel 20.0.2 *In Abb. 20.1 sehen wir, wie sich ein Baum durch Ausführung einer INSERT und zweier DELETE-Operationen sukzessive verändert.*

Die folgende detaillierte Prozedur für DELETE ist recht lang, weil alle Sonderfälle (sowie alle „links–rechts–symmetrischen“ Fälle) berücksichtigt werden müssen.

Prozedur: DELETE(i, j, k) — rekursiv —

Parameter: Knoten i, j und ein Schlüsselwert k

lokale Variable: Knoten q

Voraussetzung: j ist ein Knoten in T, der Unterbaum T_j enthält ein Element mit Schlüsselwert k und i ist der Vater von j in T (bzw. $i = 0$, falls $j = r$).

Aufgabe: Entferne das Element mit Schlüsselwert k aus T_j (und damit auch aus T).

Methode: — gemäß Fallunterscheidung (wie oben besprochen) —

1. Falls $k <$ KEY$[j]$: DELETE$(j, \text{LS}[j], k)$ und stoppe.
2. Falls $k >$ KEY$[j]$: DELETE$(j, \text{RS}[j], k)$ und stoppe.
   (**Kommentar:** Ab jetzt gilt $k =$ KEY$[j]$.
   Schritte 3–5 behandeln den Fall, dass j ein Blatt ist.)

3. Falls $\mathrm{LS}[j] = \mathrm{RS}[j] = 0$ und $i = 0$:
   $r \leftarrow 0$; DISPOSE(j) und stoppe.
4. Falls $\mathrm{LS}[j] = \mathrm{RS}[j] = 0$, $i \neq 0$ und $j = \mathrm{LS}[i]$:
   $\mathrm{LS}[i] \leftarrow 0$; DISPOSE($j$) und stoppe.
5. Falls $\mathrm{LS}[j] = \mathrm{RS}[j] = 0$, $i \neq 0$ und $j = \mathrm{RS}[i]$:
   $\mathrm{RS}[i] \leftarrow 0$; DISPOSE($j$) und stoppe.
   (**Kommentar:** Schritte 6–11 behandeln den Fall, dass j genau ein Kind hat.)
6. Falls $\mathrm{LS}[j] = 0$, $\mathrm{RS}[j] \neq 0$ und $i = 0$:
   DISPOSE(r); $r \leftarrow \mathrm{RS}[j]$ und stoppe.
7. Falls $\mathrm{LS}[j] \neq 0$, $\mathrm{RS}[j] = 0$ und $i = 0$:
   DISPOSE(r); $r \leftarrow \mathrm{LS}[j]$ und stoppe.
8. Falls $\mathrm{LS}[j] = 0$, $\mathrm{RS}[j] \neq 0$, $i \neq 0$ und $j = \mathrm{LS}[i]$:
   $\mathrm{LS}[i] \leftarrow \mathrm{RS}[j]$; DISPOSE($j$) und stoppe.
9. Falls $\mathrm{LS}[j] = 0$, $\mathrm{RS}[j] \neq 0$, $i \neq 0$ und $j = \mathrm{RS}[i]$:
   $\mathrm{RS}[i] \leftarrow \mathrm{RS}[j]$; DISPOSE($j$) und stoppe.
10. Falls $\mathrm{LS}[j] \neq 0$, $\mathrm{RS}[j] = 0$, $i \neq 0$ und $j = \mathrm{LS}[i]$:
    $\mathrm{LS}[i] \leftarrow \mathrm{LS}[j]$; DISPOSE($j$) und stoppe.
11. Falls $\mathrm{LS}[j] \neq 0$, $\mathrm{RS}[j] = 0$, $i \neq 0$ und $j = \mathrm{RS}[i]$:
    $\mathrm{RS}[i] \leftarrow \mathrm{LS}[j]$; DISPOSE($j$) und stoppe.
    (**Kommentar:** Der abschließende Schritt 12 behandelt den Fall, dass j zwei Kinder hat.)
12. Falls $\mathrm{LS}[j] \neq 0$ und $\mathrm{RS}[j] \neq 0$, dann mache weiter wie folgt:
    (a) $p \leftarrow$ SEARCH_MIN($\mathrm{RS}[j]$).
    (b) Falls $p = 0$: $q \leftarrow \mathrm{RS}[j]$; $\mathrm{RS}[j] \leftarrow \mathrm{RS}[q]$.
        andernfalls: $q \leftarrow \mathrm{LS}[p]$; $\mathrm{LS}[p] \leftarrow \mathrm{RS}[q]$.
    (c) $\mathrm{KEY}[j] \leftarrow \mathrm{KEY}[q]$; $E[j] \leftarrow E[q]$; DISPOSE(q).

In jedem der unterschiedenen Fälle muss berücksichtigt werden, dass j evtl. auch die Wurzel von T sein könnte (was durch $i = 0$ angezeigt wird). Um Schritt 12 zu verstehen, müssen wir uns mit der nun folgenden Hilfsprozedur SEARCH_MIN vertraut machen.

Prozedur: SEARCH_MIN(j) — Knoten-wertige Funktionsprozedur —

Parameter: j ist ein Knoten in T

Ausgabe: der Knoten in T_j, dessen linkes Kind den minimalen Schlüsselwert in T_j hat (bzw. der Wert 0, falls $\mathrm{LS}[j] = 0$)

Methode: Suche entlang einer LS-Zeiger-Kette.

- Falls $\mathrm{LS}[j] = 0$, dann gib 0 aus.
  Andernfalls mache weiter wie folgt:
  1. Solange $\mathrm{LS}[j] \neq 0$: $p \leftarrow j; j \leftarrow \mathrm{LS}[j]$.
  2. Gib p aus.

Man erkennt nun, dass in Schritt 12 von DELETE die Zuweisung $p \leftarrow$ SEARCH_MIN(RS[j]) und die anschließende Wertzuweisung an q dafür sorgen, dass q der Knoten mit minimalem Schlüsselwert in T_j ist.

Es ist nicht schwer sich zu überlegen, dass die Prozeduren INSERT und DELETE den Baum T so manipulieren, dass die Suchbaum-Eigenschaft nicht verloren geht. Weiterhin lässt sich beobachten, dass alle obigen Implementierungen von Wörterbuchoperationen im Wesentlichen einen Suchpfad inspizieren, der von der Wurzel ausgeht und blattwärts gerichtet ist. Auf jedem unterwegs angetroffenen Knoten muss nur konstante Rechenzeit aufgewendet werden. Somit erhalten wir folgendes Resultat:

Satz 20.0.3 *Die oben beschriebenen Prozeduren für die Wörterbuch-Operationen benötigen Rechenzeit $O(d(T))$, wobei $d(T)$ die Tiefe des Suchbaums T bezeichnet.*

Die *totale Pfadlänge* in einem Suchbaum T ist definiert als

$$\mathrm{TPL}(T) = \sum_{x \in T} (1 + d_T(x)) \ ,$$

wobei $d(x)$ die Tiefe des Knotens x im Baum T (also die Länge des Baumpfades von der Wurzel zu x) bezeichnet. Sie ist proportional zu der Anzahl der Rechenschritte, die der Aufbau von T mit INSERT-Operationen (ohne Rebalancierung) aus einem anfangs leeren Baum erfordert hätte. Die *mittlere Pfadlänge (Average Path Length)* in einem Suchbaum T mit n Knoten ist definiert als

$$\mathrm{APL}(T) = \frac{1}{n} \cdot \mathrm{TPL}(T) \ .$$

Sie ist proportional zur mittleren Suchzeit nach einem Knoten in T.

Beispiel 20.0.4 *Der Baum T links oben in Abb. 20.1 hat* 6 *Knoten. Ihre Tiefen (aufgelistet) sind* $0, 1, 1, 2, 3, 3$. *Daher gilt* $\mathrm{TPL}(T) = 10$ *und* $\mathrm{APL}(T) = \frac{5}{3}$.

Im „worst case“ ist ein Baum T der Größe n zu einem Pfad degeneriert. In diesem Fall gilt

$$\mathrm{TPL}(T) = 1 + 2 + \ldots + n = \frac{1}{2}n(n+1) = \theta(n^2)$$

und daher

$$\mathrm{APL}(T) = \frac{1}{2}(n+1) = \theta(n) \ .$$

Im „best case“ ist T vollständig balanciert, d.h. es gibt 2^i Knoten der Tiefe i in T für $i = 0, 1, \ldots, h$, wobei h die Höhe von T bezeichnet.[1] In diesem Fall erfüllt die Anzahl n der Knoten die Gleichung

$$n = 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$$

und es gilt

$$\mathrm{TPL}(T) = \sum_{i=0}^{h}(1+i)2^i = \sum_{i=0}^{h} 2^i + \sum_{i=1}^{h} i2^i = h2^{h+1} + 1 \ ,$$

wobei der Nachweis der letzten Gleichung eine nette Übung im Umgang mit Summenausdrücken wäre. Für APL gilt dann

$$\mathrm{APL}(T) = \frac{h2^{h+1}+1}{2^{h+1}-1} = \theta(h) = \theta(\log n) \ .$$

Betrachten wir abschließend den „average case“. Dabei nehmen wir an, dass beim Einfügen von n Daten $x_1, x_2, \ldots, x_n$ mit den Schlüsselwerten $k_1 < k_2 < \ldots < k_n$ alle Umordnungen (Permutationen) gleichwahrscheinlich sind. Zu einer festen Permutation σ von 1 bis n würden wir die Daten $x_1, \ldots, x_n$ in der Reihenfolge $x_{\sigma(1)}, \ldots, x_{\sigma(n)}$ einfügen. Damit würde $x_{\sigma(1)}$ die Wurzel okkupieren. Wir beobachten Folgendes:

[1] Die Tiefe eines Knotens i in einem Baum T ist die Länge des Pfades von i zur Wurzel. Die Höhe eines Knotens i in T ist die maximale Anzahl von Kanten auf einem Weg von i zu einem Blatt. Die Höhe von T, oft notiert als $h(T)$, ist definiert als die Höhe der Wurzel (und somit identisch zu $d(T)$).

1. Aus Symmetriegründen hat jeder Schlüssel k_i die gleiche Chance, in der Wurzel von T zu landen.

2. Gegeben, dass k_i in der Wurzel landet, so landen $k_1, \ldots, k_{i-1}$ unweigerlich im linken Unterbaum T_0 und $k_{i+1}, \ldots, k_n$ unweigerlich im rechten Unterbaum T_1 der Wurzel.

3. Was T_0 und T_1 betrifft, sind wieder alle Permutationen von $i-1$ bzw. $n-i$ Schlüsseln gleichwahrscheinlich.

Anhand dieser Überlegungen ist es nicht schwer, folgendes Resultat zu beweisen (Details zum Beweis weiter unten):

Lemma 20.0.5 *Die totale Pfadlänge eines Suchbaumes der Größe n im „average case“ ist die Lösung zu folgender Rekursionsgleichung:*

$$f(1) = 1 \quad \textit{und} \quad f(n) = n + \frac{1}{n}\sum_{i=1}^{n}(f(i-1) + f(n-i)) = n + \frac{2}{n}\sum_{i=0}^{n-1} f(i) \ .$$

Dies ist eine Rekursion vom Typ „Quicksort“ (einem rekursiven Sortierverfahren). Sie hat die Lösung $\theta(n \log n)$.[2] Da das Lösen von Rekursionsgleichungen in der Vorlesung DiMa I intensiv trainiert wird, lassen wir den Beweis für die Schranke $O(n \log n)$ hier aus.
Wir fassen unsere Ergebnisse zum „average case“ jetzt in folgendem Resultat zusammen:

Satz 20.0.6 *Im statistischen Mittel (average case) hat die totale Pfadlänge die Größenordnung $\theta(n \log n)$ und die mittlere Pfadlänge die Größenordnung $\theta(\log n)$.*

Beweis[Lemma 20.0.5] Es sei T_σ der Baum, der sich durch iteriertes Einfügen von $x_{\sigma(1)}, \ldots, x_{\sigma(n)}$ in einen anfangs leeren Baum ergibt. Definiere $L_\sigma = \mathrm{TPL}(T_\sigma)$. Da σ zufällig ist, ist L_σ eine Zufallsvariable. Dann ist $f(n) = E[L_\sigma]$, also der Erwartungswert von L_σ, die gesuchte totale Pfadlänge im „average case“. Es bezeichne E_i das Ereignis „$i = \sigma(1)$“. Die Aufspaltung des Erwartungswertes von L_σ in bedingte Erwartungswerte zu den Ereignissen $E_1, \ldots, E_n$ liefert

$$f(n) = E[L_\sigma] = \frac{1}{n}\sum_{i=1}^{n} E[L_\sigma | E_i] \ . \tag{20.1}$$

[2] wie Hörerinnen und Hörer von DiMa I wohl wissen

Weiterhin gilt

$$E[L_\sigma|E_i] = 1 + (n-1) + f(i-1) + f(n-i) \ . \tag{20.2}$$

Der Term 1 ist der Anteil der Wurzel i am bedingten Erwartungswert. Die Terme $f(i-1)$ und $f(n-i)$ sind die Anteile der Unterbäume der Wurzel. Der „Korrektur-Term“ $n-1$ berücksichtigt, dass ein von der Wurzel verschiedener Knoten im Gesamtbaum eine um 1 größere Tiefe hat als in seinem Unterbaum. Wenn wir (20.1) und (20.2) kombinieren, erhalten wir die Aussage des Lemmas. •

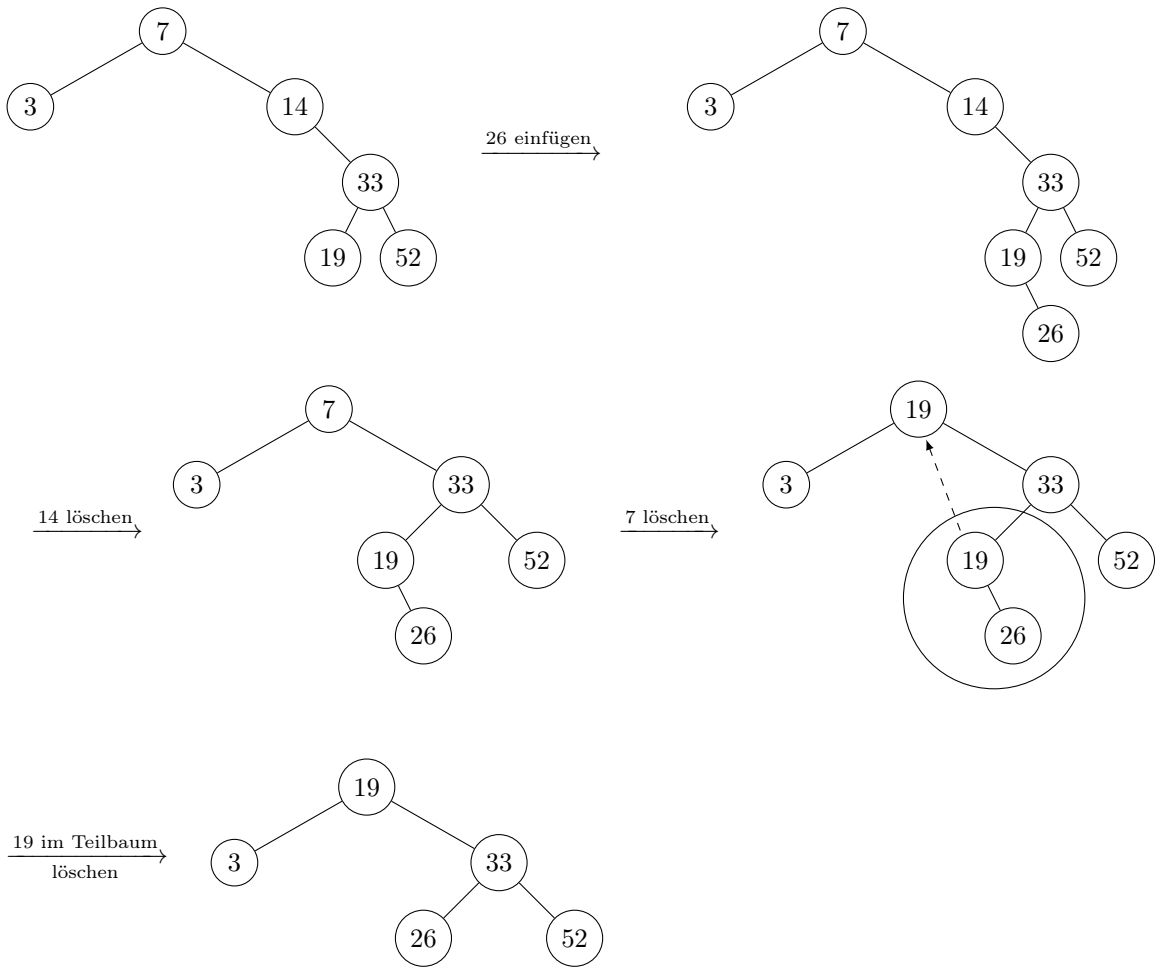


Abbildung 20.1: Der Effekt von INSERT- und DELETE-Operationen auf einen gegebenen Binärbaum. Die Zahlen an den Knoten geben nicht die Knotennummern sondern die am Knoten gespeicherten Schlüsselwerte an.

Kapitel 21

AVL-Bäume

Lesen Sie Abschnitt 4.2.4 **in [2]!**

Kapitel 22

Union-Find-Datenstruktur

Lesen Sie Abschnitt 4.6 **und** 4.7 **in [1]!**

Kapitel 23

Graphexploration

23.1 Exploration geordneter Wurzelbäume

Ein *geordneter Wurzelbaum* ist ein Baum, bei welchem ein Knoten als Wurzel ausgezeichnet wird und bei welchem die Kinder eines Knotens linear, sagen wir von links nach rechts, geordnet werden.[1] Die rekursive Definition eines geordneten Wurzelbaumes ist wie folgt:

1. Ein Baum mit einem einzigen Knoten (wie in Abb. 23.1(a) dargestellt) ist ein geordneter Wurzelbaum.

2. Wenn $T_1, \ldots, T_k$ geordnete Wurzelbäume sind, dann ist auch der Baum mit Wurzel r und „Unterbäumen“ $T_1, \ldots, T_k$ (wie in Abb. 23.1(b) dargestellt) ein geordneter Wurzelbaum.

Wenn jeder innere Knoten maximal zwei Kinder hat (ein linkes und/oder ein rechtes), dann spricht man auch von einem binären Baum. Ein binärer Baum T heißt „Suchbaum“, wenn der numerische Wert, der in einem Knoten v von T gespeichert ist, größer ist als alle im linken Teilbaum von v gespeicherten numerischen Werte und kleiner als alle im rechten Teilbaum von v gespeicherten numerischen Werte. Ein Beispiel für einen Suchbaum ist in Abb. 23.2 zu sehen. Die jeweils gespeicherten numerischen Werte werden auch als „Schlüssel“ oder als „Schlüsselwerte“ bezeichnet. Die Organisation eines Suchbaumes ermöglicht ein effizientes Aufspüren eines Knotens mit gegebenem Schlüssel.

[1] Die Terminologie mit „Kindern“, „Vater“, „Großvater“, „Vorfahr“, „Nachkomme“ usw. ist wie bei Stammbäumen.

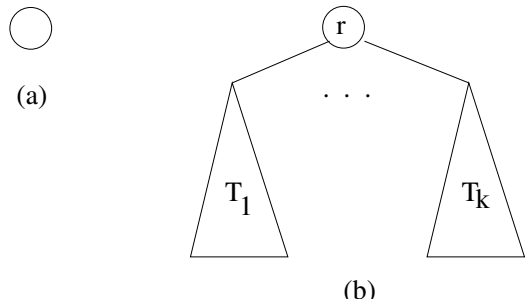


Abbildung 23.1: Illustration der rekursiven Definition eines geordneten Wurzelbaumes

In vielen Anwendungen müssen die Knoten eines geordneten Wurzelbaums systematisch, einer nach dem anderen, durchlaufen werden. Im Englischen spricht man von einem „traversal“ und unterscheidet die folgenden (rekursiv definierten) Durchlaufstrategien (vgl. mit Abb. 23.1):

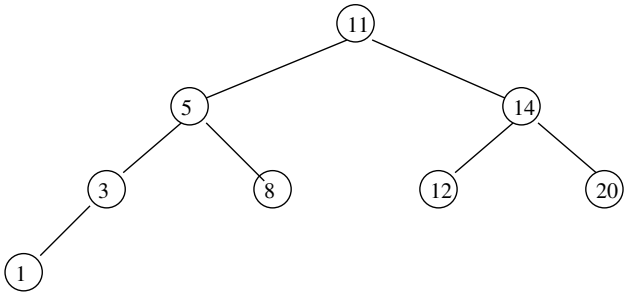
Durchlaufen in Präordnung (preorder traversal): Durchlaufe zuerst r und dann (jeweils in Präordnung) die Unterbäume $T_1, \ldots, T_k$ (in ebendieser Reihenfolge).

Durchlaufen in Postordnung (postorder traversal): Durchlaufe zuerst (jeweils in Postordnung) die Unterbäume $T_1, \ldots, T_k$ (in ebendieser Reihenfolge) und dann r.

Durchlaufen in Inordnung (inorder traversal): Diese Knotenreihenfolge ist nur für $k = 2$ erklärt: durchlaufe zuerst (in Inordnung) T_1, dann r und schließlich (in Inordnung) T_2.

Wenn zum Beispiel die Daten in einem geordneten Wurzelbaum „top down“ (von der Wurzel ausgehend blätterwärts) aktualisiert werden sollen, dann bietet sich ein „preorder traversal“ an. Werden die Daten „bottom-up“ aktualisiert (von den Blättern ausgehend wurzelwärts), so bietet sich ein „postorder traversal“ an.

Beobachtung: Inorder Traversal angewendet auf einen Suchbaum liefert eine sortierte Reihenfolge der Schlüsselwerte.



Inorder Traversal:
1 , 3 , 5 , 8 , 11 , 12 , 14 , 20

Preorder Traversal:
11 , 5 , 3 , 1 , 8 , 14 ,12 , 20

Postorder Traversal:
1 , 3 , 8 , 5 , 12 , 20 , 14 , 11

Abbildung 23.2: Ein (Such-)Baum und seine „traversals“.

23.2 Exploration von (ungerichteten) Graphen

Wir schildern zunächst ein generisches[2] Verfahren und beschreiben im Anschluss, wie die populären Explorationstechniken

- Tiefensuche (= Depth First Search = DFS)
- Breitensuche (= Breadth First Search = BFS)

daraus hervorgehen. Am Ende des Abschnittes diskutieren wir ein paar naheliegende Erweiterungen des simplen DFS- bzw. BFS-Verfahrens.

23.2.1 Exploration von einem Startknoten aus

Wie oben bereits angekündigt beginnen wir mit einem simplen generischen Verfahren.

Eingabe: Graph $G = (V, E)$, Startknoten $s \in V$

Ausgabe: Spannbaum der s enthaltenden Zusammenhangskomponente

Datenstrukturen:
- Eingabegraph in Adjazenzlistendarstellung
- Markierung der Knoten mit „alt“ (bereits besucht) oder „neu“ (noch unbesucht)
- Liste L bereits besuchter Knoten, deren Nachbarschaftsliste noch nicht vollständig durchforstet wurde

[2] allgemein gehaltenes, nicht detailliert ausimplementiertes

- Knotenzeiger zur Darstellung des auszugebenden Spannbaumes (Wurzel s, Zeiger zur Wurzel hin orientiert, **nil** bezeichnet den „Nullzeiger“)

Methode:

1. $L := \{s\}$; markiere s „alt“; Zeiger(s) := **nil**;
2. Markiere alle $v \in V \setminus \{s\}$ „neu“;
3. `--- evtl. weitere Instruktionen ---`
4. Wiederhole folgende Schritte bis L leer ist:
   (a) Sei v ein Knoten aus L.
   (b) **Fall 1:** v besitzt einen „neu“ markierten Nachbarn w
      i. Füge w in L ein und markiere w „alt“;
      ii. Zeiger(w) := v;
      iii. `--- evtl. weitere Instruktionen ---`

   Fall 2: v besitzt keinen „neu“ markierten Nachbarn w
      i. Entferne v aus L;
      ii. `--- evtl. weitere Instruktionen ---`

Die geschilderte Methode heißt *Tiefensuche*, *Depth First Search* oder kurz *DFS*, wenn wir die Liste L als STACK verwalten, d.h., L wird nach dem LIFO-Prinzip[3] organisiert.
Sie heißt *Breitensuche*, *Breadth First Search* oder kurz *BFS*, wenn wir die Liste L als QUEUE verwalten, d.h., L wird nach dem FIFO-Prinzip[4] organisiert.

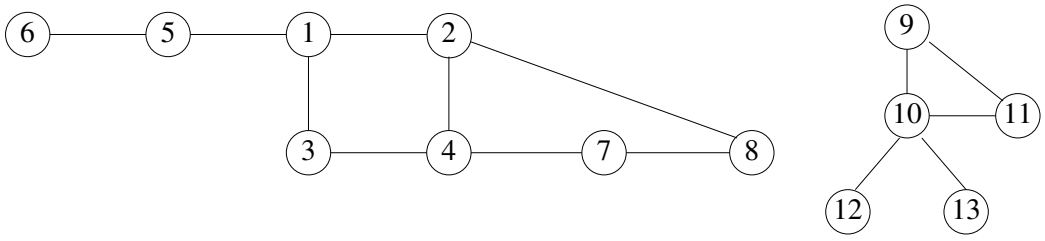


Abbildung 23.3: ein Graph mit zwei Zusammenhangskomponenten

[3] LIFO = Last In First Out
[4] FIFO = First In First Out

Technische Vereinbarung: Wir nehmen im Folgenden an, dass im Rahmen einer Graphexploration die Nachbarn eines Knotens in der Reihenfolge aufsteigender Knotennummern inspiziert werden. Dies hat den Vorteil, dass die Beispielläufe, die wir betrachten werden, stets zu einem eindeutigen Ergebnis führen.

Wir betrachten als Beispiel den Graphen in Abb. 23.3 mit Startknoten 1. DFS führt zu folgender „Evolution“ des STACK L (wobei jede Spalte einen „Schnappschuss“ des STACK darstellt und Elemente stets oben eingefügt oder entnommen werden):

						8									
			3		7	7	7								
		4	4	4	4	4	4	4				6			
	2	2	2	2	2	2	2	2	2		5	5	5		
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	STACK leer

Aus den dynamischen Veränderungen des STACK lassen sich die aktuellen Knotenmarkierungen und die aktuellen Knotenzeiger leicht ablesen:

- Zu jedem Zeitpunkt sind genau die Knoten „alt“ markiert, die bereits in den STACK aufgenommen wurden.
- Wann immer im STACK ein Knoten b auf einen Knoten a „gestapelt“ wird, muss ein Zeiger von b nach a gesetzt werden.

Die im Verlauf der DFS gesetzten Knotenzeiger sind in Abb. 23.4(a) zu besichtigen. Diese Zeiger repräsentieren den DFS-Spannbaum der Zusammenhangskomponente mit Startknoten 1. Derselbe Spannbaum ist in einem etwas ansprechenderen Layout nochmals in Abb. 23.4(b) zu sehen.

Wenn wir auf dem Eingabegraphen in Abb. 23.3 mit Startknoten 1 eine BFS durchführen, entwickelt sich die QUEUE wie folgt (wobei jede Spalte einen „Schnappschuss“ der QUEUE darstellt und Elemente stets oben eingefügt und unten entnommen werden)[5]:

						8									
			5		4	4	8		6		7				
		3	3	5	5	5	4	8	8	6	6	7			
	2	2	2	3	3	3	5	4	4	8	8	6	7		
1	1	1	1	2	2	2	3	5	5	4	4	8	6	6	QUEUE leer

[5] „oben“ entspricht also „hinten in der Warteschlange“; „unten“ entspricht „vorne in der Warteschlange“.

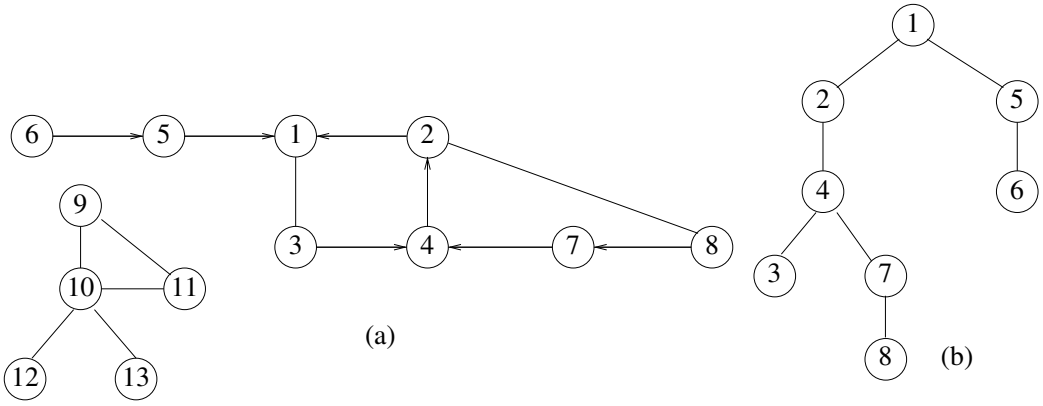


Abbildung 23.4: (a) der Eingabegraph ergänzt um die im Verlauf der DFS gesetzten Zeiger (b) anderes Layout des durch die Zeiger repräsentierten Spannbaums

Aus den dynamischen Veränderungen der QUEUE lassen sich die aktuellen Knotenmarkierungen und die aktuellen Knotenzeiger leicht ablesen:

- Zu jedem Zeitpunkt sind genau die Knoten „alt“ markiert, die bereits in die QUEUE aufgenommen wurden.
- Wann immer zuoberst ein Knoten b eingefügt wird während zuunterst ein Knoten a liegt, muss ein Zeiger von b nach a gesetzt werden.

Die im Verlauf der BFS gesetzten Knotenzeiger sind in Abb. 23.5(a) zu besichtigen. Diese Zeiger repräsentieren den BFS-Spannbaum der Zusammenhangskomponente mit Startknoten 1. Derselbe Spannbaum ist in einem etwas ansprechenderen Layout nochmals in Abb. 23.5(b) zu sehen.

23.2.2 Erweiterung 1: Vollständige Exploration eines Graphen

Bisher haben wir nur beschrieben, wie die Zusammenhangskomponente exploriert wird, welche den Startknoten s des Graphen enthält. Die Ausgabe bestand aus einem Spannbaum für diese Komponente. Falls der Graph nicht zusammenhängend ist, blieben unexplorierte Komponenten übrig. Es ist aber einfach, um das generische Verfahren herum ein Hauptprogramm

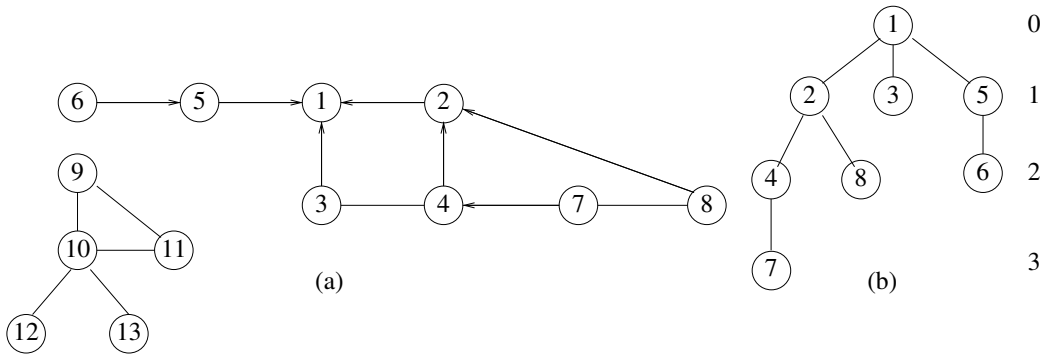


Abbildung 23.5: (a) der Eingabegraph ergänzt um die im Verlauf der BFS gesetzten Zeiger (b) anderes Layout des durch die Zeiger repräsentierten Spannbaums

zu „stricken“, welches die Knoten der Reihe nach inspiziert und auf einem „neu“ markierten Knoten u das generische Verfahren mit u als Startknoten neu startet. Auf diese Weise wird der Graph vollständig exploriert und die Ausgabe besteht aus einem „Spannwald“ mit einem Spannbaum für jede Komponente des Graphen. Wenden wir dabei DFS (BFS) an, so sprechen wir von einem DFS-Spannwald (BFS-Spannwald).

In dem Eingabegraphen aus Abb. 23.3 wären zwei Starts des generischen Verfahrens nötig: zum Beispiel einer mit Startknoten 1 und einer mit Startknoten 9. Der resultierende DFS-Spannwald (bzw. BFS-Spannwald) ist in Abb. 23.6 (bzw. in Abb. 23.7) zu sehen.

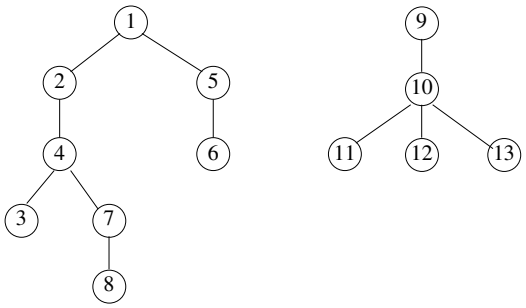


Abbildung 23.6: Der DFS-Spannwald zum Graphen aus Abb. 23.3

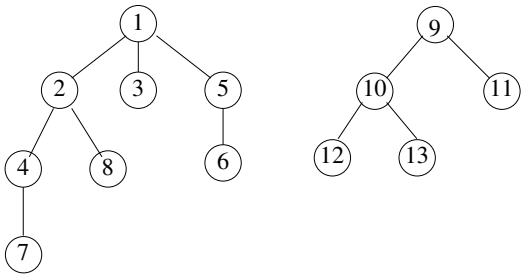


Abbildung 23.7: Der BFS-Spannwald zum Graphen aus Abb. 23.3

23.2.3 Erweiterung 2: Verteilung von DFS-Nummern

Wir sagen ein Knoten v erhält die *DFS-Eintrittsnummer* i, notiert als $N_{ein}(v) = i$, wenn er der i-te Knoten ist, der in den STACK aufgenommen wird. Analog sagen wir, v erhält die *DFS-Austrittsnummer* i, notiert als $N_{aus}(v) = i$, wenn er der i-te Knoten ist, der aus dem STACK entfernt wird. Anwendungen der DFS-Nummern werden wir sowohl in den Übungen als auch im Abschnitt über Exploration von Digraphen kennenlernen. An dieser Stelle wollen wir nur darauf hinweisen, dass die Berechnung der DFS-Nummern mit Hilfe zweier Zähler Z_{ein}, Z_{aus} leicht in das generische Verfahren integrierbar ist (und zwar an den Stellen, die wir vorsorglich mit „`evtl. weitere Instruktionen`“ gekennzeichnet hatten):

3. $N_{ein}(s) := 1$; $Z_{ein} := 1$; $Z_{aus} := 0$

4.(b), Fall 1, iii. $Z_{ein} := Z_{ein} + 1; N_{ein}(w) := Z_{ein}$

4.(b), Fall 2, ii. $Z_{aus} := Z_{aus} + 1; N_{aus}(v) := Z_{aus}$

Beide Nummern lassen sich alternativ auch absteigend vergeben. In diesem Fall würden die Zähler mit $n + 1$ (statt mit 0) initialisiert und vor jeder neu vergebenen Nummer um 1 runtergezählt (statt um 1 raufgezählt). Wenn wir nicht explizit auf absteigende Nummerierung hinweisen, wollen wir aber im Normalfall von einer aufsteigenden Nummerierung ausgehen.

Im Beipielgraphen aus Abb. 23.3 und einer DFS mit Startknoten 1 würden die Knoten den STACK in der Reihenfolge

$$1, 2, 4, 3, 7, 8, 5, 6$$

betreten und ihn in der Reihenfolge

$$3, 8, 7, 4, 2, 6, 5, 1,$$

wieder verlassen. Damit ergeben sich folgende DFS-Nummern:

v	1	2	3	4	5	6	7	8
$N_{ein}(v)$	1	2	4	3	7	8	5	6
$N_{aus}(v)$	8	5	1	4	7	6	3	2

Man überlegt sich leicht den folgenden

Satz:
1. Die Reihenfolge, in der die Knoten den STACK betreten, entspricht einem „preorder traversal“ des betreffenden DFS-Spannbaumes.
2. Die Reihenfolge, in der die Knoten den STACK verlassen, entspricht einem „postorder traversal“ des betreffenden DFS-Spannbaumes.

23.2.4 Erweiterung 3: BFS und die Distanz zum Startknoten

Die *Distanz* von Knoten u zum Knoten v in einem Graphen G ist die Länge eines kürzesten Pfades von u nach v (gemessen in der Anzahl der Kanten). Es ist leicht, eine BFS mit Startknoten s so zu erweitern, dass sie die Distanz $d(v)$ des Startknotens s zum Knoten v für jedes $v \in V$ berechnet:

3. $d(s) := 0$; $d(v) := \infty$ für alle $v \in V \setminus \{s\}$

4.(b), Fall 1, iii. $d(w) := d(v) + 1$

In Abb. 23.5(b) sind die Distanzwerte rechts am Spannbaum mit Wurzel 1 vermerkt. Die Distanz deckt sich jeweils mit der Tiefe des betreffenden Knotens im BFS-Spannbaum. Dies ist kein Zufall, denn es gilt der allgemeine

Satz: Der BFS-Spannbaum ist ein „Kürzeste-Pfade Baum“, d.h. der eindeutige Pfad im BFS-Spannbaum von der Wurzel s zu einem Knoten v ist auch in G ein kürzester Pfad von s nach v.

23.3 Exploration von Digraphen

DFS und BFS (und auch die im vorigen Abschnitt diskutierten Erweiterungen) können jeweils ohne Probleme auf Digraphen G (anstelle von ungerichteten Graphen) angewendet werden. Wenn wir von einem festen Startknoten s ausgehen, dann ist das Ergebnis ein gerichteter Baum mit Wurzel s, der alle Knoten enthält, welche in G von s aus erreichbar sind. Falls von s aus in G alle Knoten erreichbar sind, ergibt sich sogar ein Spannbaum von G. Falls das nicht der Fall ist, dann kann mit Hilfe von Neustarts (analog zur Vorgehensweise in Abschnitt 23.2.2) eine vollständige Graphexploration durchgeführt werden, deren Ergebnis ein gerichteter Spannwald von G ist.

Zur Illustration betrachten wir den Digraphen aus Abb. 23.8(a) mit Startknoten 1 (von dem aus alle Knoten des Digraphen erreichbar sind). Es sei an die Vereinbarung erinnert, dass Nachbarn eines Knotens in der Reihenfolge aufsteigender Knotennummern inspiziert werden. Der resultierende DFS-Spannbaum (BFS-Spannbaum) ist in Abb. 23.8(b) (bzw. in Abb. 23.8(c)) zu sehen. Wie schon bei ungerichteten Graphen, ist der BFS-Spannbaum wieder ein „Kürzeste-Pfade Baum“.

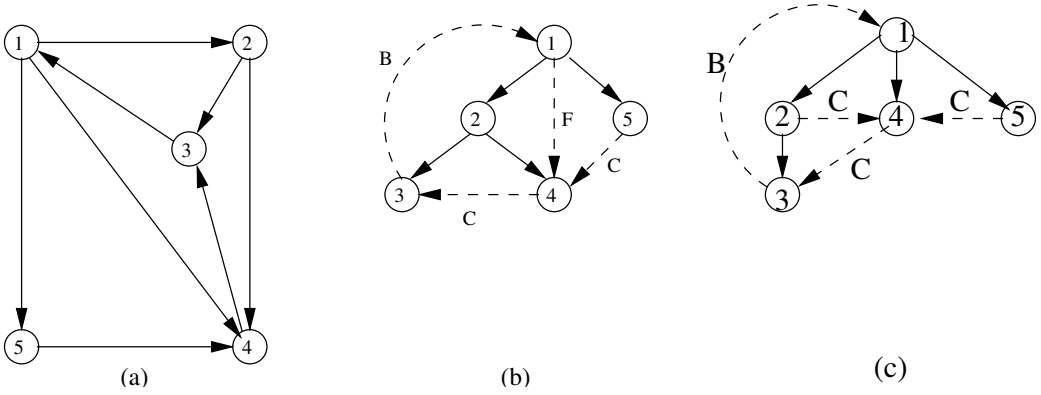


Abbildung 23.8: (a) ein Digraph (b) sein DFS-Spannbaum (c) sein BFS-Spannbaum (nicht zum Spannbaum gehörende Graphkanten gestrichelt)

Die nicht zum jeweiligen Spannbaum T gehörenden Kanten sind in Abb. 23.8 gestrichelt dargestellt. Sie zerfallen auf natürliche Weise in folgende Typen:

F-Kanten Die „Vorwärtskanten“ (kurz: F-Kanten[6]) führen von einem Kno-

[6]Englisch: Forward-edges

ten v zu einem Nachfolger in T.

B-Kanten Die „Rückwärtskanten“ (kurz: B-Kanten[7]) führen von einem Knoten v zu einem Vorgänger in T.

C-Kanten Die „kreuzenden Kanten“ (kurz: C-Kanten[8]) verbinden zwei in T „unabhängige“ Knoten (d.h. zwei Knoten von denen keiner Nachfolger des anderen ist).

Man überlegt sich leicht:

Beobachtung 1: C-Kanten in einem DFS-Baum (oder DFS-Wald) führen stets von rechts nach links. (Warum?)

Beobachtung 2: In einem BFS-Baum (oder BFS-Wald) gibt es keine F-Kanten und C-Kanten verlaufen stets von einem Knoten einer Tiefe i zu einem Knoten einer Tiefe von maximal $i + 1$. (Warum?)

Aus Beobachtung 1 ergibt sich die

Folgerung: Ein Digraph ist genau dann azyklisch (also ein DAG = Directed Acyclic Graph), wenn sein DFS-Spannwald keine B-Kanten enthält.

Der Typ (F, B oder C) der nicht zum DFS-Wald gehörenden Kanten des Digraphen ist leicht an den Eintritts- und Austrittsnummern der Knoten ablesbar.[9] Somit ist auch ein „DAG-Test“ für einen Digraphen mit (geeignet erweiterter) DFS leicht durchzuführen.

Als letzte Anwendung von DFS betrachten wir die topologische Sortierung der Knoten eines DAGs $G = (V, E)$. Die Knoten heißen dabei *topologisch sortiert*, wenn jeder Knoten erst dann durchlaufen wird, nachdem alle seine Vorgänger bereits durchlaufen wurden. Wir wollen der Einfachheit halber annehmen, dass der DAG G einen Knoten s enthält, von dem aus alle Knoten erreichbar sind und den die DFS als Startknoten einsetzt.[10] Mit Hilfe von Beobachtung 1 (s. oben) ergibt sich leicht folgender

Satz: Die absteigenden DFS-Austrittsnummern (also ein „gespiegeltes postorder traversal“) liefern eine topologische Sortierung der Knoten von G.

[7] Englisch: Backward-edges

[8] Englisch: Crossing-edges

[9] Dies zu zeigen wäre eine gute Übungsaufgabe!

[10] Ein solcher Knoten kann notfalls hinzugefügt werden.

Zur Illustration betrachte Abb. 23.9. Wie am DFS-Spannbaum erkennbar ist, gibt es keine B-Kanten. Somit handelt es sich wirklich um einen DAG. Weiterhin bringt ein „postorder traversal“ des DFS-Spannbaums die Knoten in die Reihenfolge $3, 4, 2, 5, 1$. Spiegelung liefert $1, 5, 2, 4, 3$: in der Tat eine topologische Sortierung der Knoten !

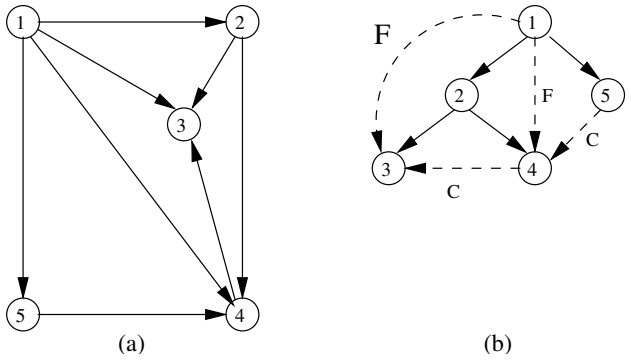


Abbildung 23.9: (a) ein DAG (b) sein DFS-Spannbaum (nicht zum Spannbaum gehörende Graphkanten gestrichelt)

23.4 Rechenzeit für DFS und BFS

Bei vernünftiger Implementierung von DFS bzw. BFS benötigen diese Verfahren auf einem (Di-)Graphen mit n Knoten und m Kanten nur $O(n + m)$ Rechenschritte (Linearzeit). Dies gilt auch für die vollständige Graphexploration (mit Neustarts bis alle Knoten exploriert sind) und für alle in diesem Manuskript geschilderten Erweiterungen. Insbesondere sind folgende Probleme in Linearzeit lösbar:

- Berechnung der Zusammenhangskomponenten eines Graphen
- Berechnung des DFS- oder BFS-Spannwaldes
- Test eines Digraphen auf Existenz von Zykeln (DAG oder nicht DAG)
- topologische Sortierung der Knoten eines DAG

Kapitel 24

Hashing

Lesen Sie Abschnitt 4.2.2 **in [2] sowie Abschnitte** 4.2 **und** 4.5 **in [4]!**

Literaturverzeichnis

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[2] Ralh H. Güting and Stefan Dieker. *Datenstrukturen und Algorithmen*. Teubner Verlag, 2003.

[3] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson international, 2006.

[4] Kurt Mehlhorn and Peter Sanders. *Algorithms and data Structures: the Basic Toolbox*. Springer, 2007.

[5] Angelika Steger. *Diskrete Strukturen*. Springer, 2001.