Informatik-Spektrum 1, 81–89 (1978)

Informatik Spektrum

Effiziente Algorithmen: Ein Beispiel

Kurt Mehlhorn

FB 10, Angewandte Mathematik und Informatik der Universität des Saarlandes, Saarbrücken

Zusammenfassung. Die Entwicklung und Analyse effizienter Algorithmen wird am Beispiel der Konstruktion von Suchbäumen verdeutlicht. Fünf Algorithmen werden vorgestellt und analysiert. Die Analyse weist jeweils auf die Schwachstellen der Algorithmen hin und führt so zu Verbesserungen.

Summary. The design and analysis of algorithms is explained by means of an example: the construction of search trees. Five algorithms are described and analyzed. The analysis points at the weak spots of the algorithms and suggests improvements.

1. Einleitung

Wir wollen in diesem Beitrag die Entwicklung und Analyse effizienter Algorithmen an einem Beispiel erläutern. Als Beispiel wird uns dienen: die Konstruktion (nahezu) optimaler Suchbäume.

Algorithmen dienen zur Lösung von Problemen. Ein Problem ist etwa die Berechnung des Maximums einer endlichen Menge von reellen Zahlen oder die Berechnung des Produkts zweier ganzer Zahlen. Ein Problem P existiert meistens in unendlich vielen Fragestellungen. Eine Fragestellung des Maximumproblems ist: Bestimme die größte der 5 Zahlen 2, 7, 3, 9, 8. Eine Fragestellung des Multiplikationsproblems ist: Berechne das Produkt der Zahlen 257 und 123. Jeder Fragestellung $p \in P$ messen wir eine natürliche Zahl $g(p)$ als Größe zu, manchmal auch ein Tupel von natürlichen Zahlen. Die Größe der Fragestellung 2, 7, 3, 9, 8 können wir als 5 definieren, also als Kardinalität der Menge, die Größe von 257×123 können wir als 6, also als Summe der Längen der Dezimaldarstellungen definieren. Die Definition der Größe einer Fragestellung ist willkürlich, ergibt sich aber meist in natürlicher Weise.

Die Ausführung eines Programms in einer Rechenanlage benötigt Betriebsmittel, z. B. Rechenzeit und Speicherplatz. Der Verbrauch an Ressourcen hängt von der Fragestellung ab. Für einen Algorithmus A zur Lösung des Problems P sei $T_A(p)$ der Rechenzeitverbrauch an der Fragestellung $p \in P$. Wir können $T_A(p)$ in Millisekunden messen. Die Laufzeit $T_A(p)$ kann durch ein Experiment bestimmt werden.

Interessanter als der Rechenzeitverbrauch für jede einzelne Fragestellung ist eine globale Aussage über den Rechenzeitverbrauch an einer beliebigen Eingabe der Größe n, die nun nicht mehr durch das Experiment bestimmt werden kann. Zwei Abstraktionen bieten sich an: Verhalten im schlechtesten Fall und Verhalten im Mittel.

Das Verhalten im schlechtesten Fall definieren wir als die maximale Laufzeit an einer Fragestellung der Größe n. Wir benutzen dafür die Bezeichnung $T_A(n)$.

$$T_A(n) = \sup\{T_A(p);\ p \in P \text{ und } g(p) = n\}.$$

Das Verhalten im schlechtesten Fall greift für jede Größe n diejenige Problemstellung heraus, die den Algorithmus A am schlechtesten aussehen läßt.

Wir sind also an der Laufzeit von Algorithmen interessiert. Auf welcher Maschine? Natürlich auf der, die Sie gerade benutzen. Leider kann ich darüber nichts sagen, da ich diese Maschine nicht kenne. Um einen gemeinsamen Maßstab zu haben, sollten wir daher eine Rechenanlage definieren, die die typischen Fähigkeiten einer modernen Anlage hat. Das würde aber den Rahmen dieses Beitrags sprengen. Daher schlage ich vor: Die Maschinensprache unserer Referenzmaschine umfaßt die Grundzüge von ALGOL: Einfache und indizierte Variable, arithmetische und boolesche Ausdrücke, Wertzuweisungen, Fallunterscheidungen und Sprünge. Laufanweisungen und while-Schleifen betrachten wir als Abkürzungen.

for i from a to b do S steht für

```
        i←a;
Loop:   if i>b then goto Exit;
        S;
        i←i+1;
        goto Loop
Exit:
```

und

while P do S steht für Loop: if $\neg P$ then goto Exit;
S;
goto Loop;
Exit:

Wir nehmen an, daß jede Wertzuweisung, jeder Test und jeder Sprung eine Zeiteinheit erfordert. Die Anweisung

if $x=0$ then $x\leftarrow x+1$ else begin $x\leftarrow x-1$; goto Exit end

verbraucht demnach 2 oder 3 Zeiteinheiten, je nachdem ob $x=0$ ist oder nicht. Die obige Laufanweisung verbraucht (a, b ganze Zahlen, $b-a+1\geq 0$)

$$3+3(b-a+1)+\sum_{l=a}^{b} \text{(Zeitbedarf für } S \text{ mit } i=l)$$

Zeiteinheiten. Prozeduren sind in unserer Sprache nicht erlaubt. Natürlich sind unsere Annahmen über den Befehlsvorrat und die Ausführungszeiten unrealistisch. Wir laden den Leser ein, die folgenden Betrachtungen für seine Lieblingsmaschine durchzuführen. Die Ergebnisse werden im wesentlichen die gleichen bleiben.

Der folgende Beitrag enthält keine Beweise. Der Leser findet sie in [7].

2. Suchbäume

Sei $(U,\leq)$ eine linear geordnete Menge und $S=\{x_1,\ldots,x_n\}\subseteq U$.

Beispiel

1. $(U,\leq)$ ist die Menge der reellen Zahlen mit der normalen $\leq$-Relation. $S=\{1.5, 7, 13\}$.

2. $(U,\leq)$ ist die Menge der Worte über dem deutschen Alphabet und $\leq$ ist die lexikographische Ordnung. $S=\{$Ena, Kurt, Steffi, Uli, Zenzi$\}$

Ein Suchbaum für die Menge $S=\{x_1,\ldots,x_n\}$ besteht aus einem binären Baum (jeder Knoten hat 2 Nachfolger, ein Blatt hat keinen Nachfolger) mit n Knoten. (Wir unterscheiden Knoten und Blätter. Insbesondere sind Blätter keine Knoten. Knoten werden als Kreise, Blätter als Rechtecke gezeichnet). Dieser Baum hat dann $n+1$ Blätter. Die n Knoten sind von links nach rechts mit den Elementen von S in aufsteigender Reihenfolge beschriftet. (Abb. 1.)

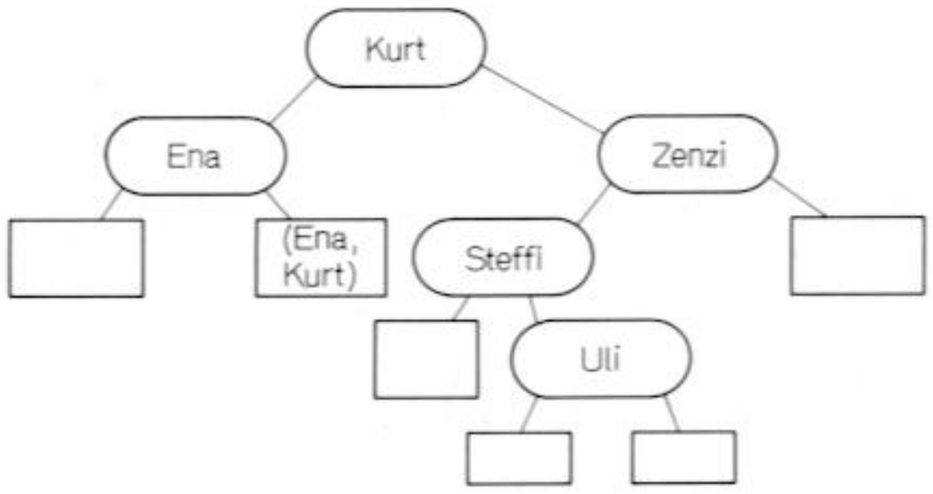


Abb. 1.

Um einen Namen X in diesem Suchbaum zu suchen, vergleicht man X mit dem Inhalt der Wurzel (hier Kurt) und geht nach links (stoppt, geht nach rechts), wenn $X <$ Kurt ($X=$Kurt, $X>$Kurt).

Sucht man einen Namen, der nicht in S liegt (etwa Helmut), so endet man in einem Blatt (im Blatt zwischen Ena und Kurt). Dies gilt nicht nur für den Namen Helmut, sondern für jedes Wort, das zwischen Ena und Kurt in der alphabetischen Ordnung steht. Wir beschriften daher dieses Blatt mit dem offenen Intervall (Ena, Kurt). Das linkste Blatt steht für alle Namen vor Ena. Wir bezeichnen es mit ($\square$, Ena).

Im allgemeinen Fall sind also die Knoten mit $x_1,\ldots,x_n$ und die Blätter mit $(\square,x_1),(x_1,x_2),\ldots,(x_n,\square)$ beschriftet. Statt $(\square,x_1)$ bzw. $(x_n,\square)$ schreiben wir manchmal (x_0,x_1), (x_n,x_{n+1}). Wenn wir nach $X=x_i$ suchen, dann vergleichen wir X mit allen Knoten auf dem Weg von der Wurzel zum Knoten x_i einschließlich x_i. Wenn wir nach X mit $x_j<X<x_{j+1}$ suchen, dann vergleichen wir X mit allen Knoten auf dem Weg von der Wurzel zum Blatt (x_j,x_{j+1}). Sei daher b_i die Tiefe des Knoten x_i und a_j die Tiefe des Blattes (x_j,x_{j+1}). (Die Tiefe eines Knotens bzw. Blattes ist die Anzahl der Kanten auf dem Pfad von der Wurzel zu dem Knoten bzw. Blatt). Dann führen wir im ersten Fall b_i+1 und im zweiten Fall a_j Vergleiche aus.

Im Baum von Abb. 2. ist $b_1=1$, $b_2=2$, $b_3=0$, $b_4=1$, $a_0=a_3=a_4=2$ und $a_1=a_2=3$.

Oft wird nach den Elementen von S nicht gleich häufig gesucht. Wir brauchen dann noch Angaben über die Häufigkeit, mit der nach den Elementen von S gesucht wird. Sei etwa β_i, $1\leq i\leq n$, die Häufigkeit mit

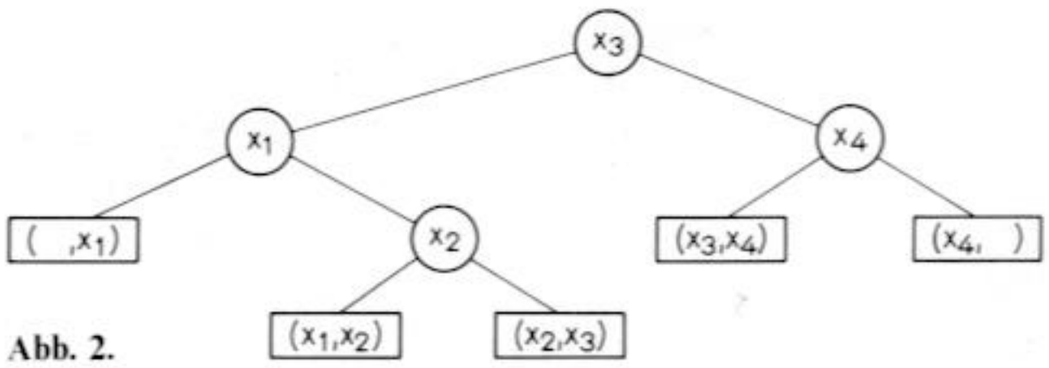


Abb. 2.

der nach $X=x_i$ gesucht wird, und α_j, $0\leq j\leq n$, die Häufigkeit mit der nach X, $x_j<X<x_{j+1}$, gesucht wird ($\beta_i\geq 0$, $\alpha_j\geq 0$). Dann ist

$$P=\sum_{i=1}^{n}\beta_i(b_i+1)+\sum_{j=0}^{n}\alpha_j\,a_j$$

ein Maß für die mittlere Suchzeit im Baum. P heißt gewichtete Weglänge.

Für unser Beispiel sei $(\alpha_0,\beta_1,\alpha_1,\beta_2,\alpha_2,\beta_3,\alpha_3,\beta_4,\alpha_4)=(4, 1, 0, 3, 0, 3, 3, 0, 10)$. Dann hat obiger Baum die gewichtete Weglänge 48. (Abb. 2)

Es ist erwähnenswert, daß auch die Extremfälle (alle $\beta_i=0$ bzw. alle $\alpha_j=0$) interessant sind. Falls $\alpha_j=0$ für alle j, dann sind alle Suchvorgänge erfolgreich, falls $\beta_i=0$ für alle i, dann dienen die Knoten nur als „Wegweiser".

Durch die Definition der gewichteten Weglänge weisen wir einem Baum eine einzige reelle Zahl als Maß seiner Güte zu. Wir können daher nach dem Baum fragen, der die gewichtete Weglänge minimiert. Dieser Baum optimiert dann auch die mittlere Suchzeit.

Definition

Sei $S=\{x_1,\ldots,x_n\}$ eine Menge und $(\alpha_0,\beta_1,\ldots,\beta_n,\alpha_n)$ eine Zugriffsverteilung. Ein Suchbaum T *für* S heißt *optimaler Suchbaum*, wenn seine gewichtete Weglänge minimal ist unter allen Suchbäumen für S. Wir bezeichnen mit T_{opt} stets einen optimalen Suchbaum und mit P_{opt} seine gewichtete Weglänge.

Wie können wir nun einen optimalen Suchbaum finden. Ein erster naiver Ansatz, nennen wir den Algorithmus Alg 0, wäre es, *alle* binären Bäume mit n Knoten zu konstruieren und zu vergleichen. Davon gibt es aber ungefähr 4^n. Wenn wir für die Konstruktion eines Baumes und Bestimmung seiner gewichteten Weglänge nur eine Zeiteinheit rechnen, dann verbraucht Alg 0 also mindestens 4^n Zeiteinheiten. Dies ist horrend. Wir verfolgen daher Alg 0 nicht weiter. Alg 0 wird üblicherweise als Exhaustionsalgorithmus bezeichnet.

3. Zwei Algorithmen zur Bestimmung des optimalen Baumes

Sei nun T_{opt} ein optimaler Suchbaum für S. Dann hat T_{opt} irgendeinen Knoten zur Wurzel, etwa x_i. Dann ist der linke Unterbaum T_l von T_{opt} ein Suchbaum für die Menge $\{x_1,x_2,\ldots,x_{i-1}\}$. T_l ist selbstverständlich optimal. Anderenfalls könnten wir ihn durch einen besseren Baum ersetzen und so T_{opt} insgesamt verbessern. Also gilt

Unterbäume optimaler Bäume sind optimal.

Diese Beobachtung erlaubt uns, den optimalen Suchbaum mit Hilfe von dynamischem Programmieren zu finden. Dabei konstruiert man auf systematische Weise Lösungen für immer größere Teilprobleme. In unserem Fall bedeutet das die Konstruktion der optimalen Bäume für alle Paare, Tripel, ... von nebeneinanderliegenden Knoten.

Sei T_{ij}, $1\leq i\leq j\leq n$, ein optimaler Suchbaum für die Knoten $\{x_i,x_{i+1},\ldots,x_j\}$ und die Blätter $\{(x_{i-1},x_i),(x_i,x_{i+1}),\ldots,(x_j,x_{j+1})\}$ mit der Verteilung $(\alpha_{i-1},\beta_i,\alpha_i,\ldots,\beta_j,\alpha_{j+1})$, sei P_{ij} die gewichtete Weglänge von T_{ij} und sei

$$w_{ij}=\alpha_{i-1}+\beta_i+\alpha_i+\ \ldots\ +\beta_j+\alpha_j$$

das „Gewicht" von T_{ij}.

Als Grenzfall betrachten wir noch den Baum $T_{i+1,i}$, der nur aus dem Blatt (x_i,x_{i+1}) besteht, also $P_{i+1,i}=0$ und $w_{i+1,i}=\alpha_i$.

Nach unserer Beobachtung über Unterbäume von optimalen Bäumen hat in unserem Beispiel vom vorigen Abschnitt der Baum $T_{1,4}$ eine der folgenden 4 Gestalten:

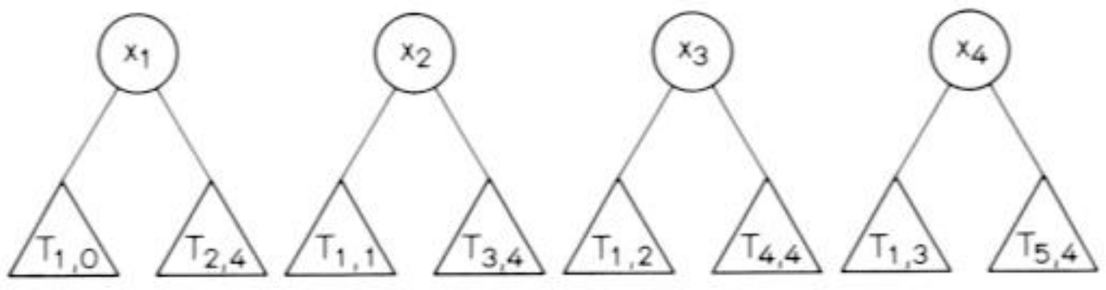


Abb. 3.

Die gewichtete Weglänge eines Baumes:

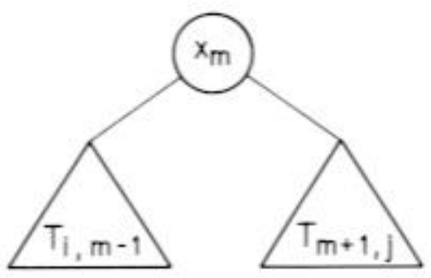


Abb. 4.

ist $P_{i,m-1}+w_{i,m-1}+\beta_m+P_{m+1,j}+w_{m+1,j}=P_{i,m-1}+P_{m+1,j}+w_{i,j}$. Die Knoten und Blätter in $T_{i,m-1}$ und $T_{m+1,j}$ sind sämtlich um 1 Niveau nach unten gerutscht und x_m kam neu hinzu.

Um also von den obigen 4 Bäumen den besten auszusuchen, brauchen wir nur die 4 Größen

$$P_{1,0}+P_{2,4}+w_{1,4},\ P_{1,1}+P_{3,4}+w_{1,4},\ P_{1,2}+P_{4,4}+w_{1,4}$$

und $P_{1,3}+P_{5,4}+w_{1,4}$ zu vergleichen und die kleinste auszusuchen. Dies führt zu folgendem Programm. Wir benutzen zusätzlich die Größen r_{ij}. Dabei ist r_{ij} der Index der Wurzel von T_{ij}.

comment wir berechnen zunächst die relevanten Größen für die Bäume $T_{i+1,i}$ und $T_{i,i}$.

(1) $P_{1,0} \leftarrow 0$; $w_{1,0} \leftarrow \alpha_0$
(2) for i from 1 to n do
(3) begin $P_{i+1,i} \leftarrow 0$; $w_{i+1,i} \leftarrow \alpha_i$;
$w_{i,i} \leftarrow \alpha_{i-1} + \beta_i + \alpha_i$; $P_{i,i} \leftarrow w_{i,i}$; $r_{i,i} \leftarrow i$
end
comment wir bestimmen nun die optimalen Bäume mit 2, 3, 4, ..., n Knoten
(4) for k from 1 to $n-1$ do
(5) for i from 1 to $n-k$ do
(6) begin $j \leftarrow i+k$
$w_{ij} \leftarrow w_{i,j-1} + \beta_j + \alpha_j$
comment wir bestimmen nun die Wurzel von T_{ij};
(8) $m \leftarrow i$; $s \leftarrow P_{i,m-1} + P_{m+1,j}$
(9) for l from $i+1$ to j do
(10) if $P_{i,l-1} + P_{l+1,j} < s$
(11) then begin $m \leftarrow l$; $s \leftarrow P_{i,m-1} + P_{m+1,j}$ end;
(12) $r_{ij} \leftarrow m$
(13) $P_{ij} \leftarrow P_{i,m-1} + w_{ij} + P_{m+1,j}$
end

Algorithmus 1: [3]

Für unser Beispiel ergeben sich die folgenden Werte:

W

	0	1	2	3	4
1	4	5	8	14	24
2		0	3	9	19
3			0	6	16
4				3	13
5					10

P

	0	1	2	3	4
1	0	5	11	25	48
2		0	3	12	31
3			0	6	22
4				0	13
5					0

r

	0	1	2	3	4
1		1	1	2	3
2			2	3	4

r

	0	1	2	3	4
3				3	4
4					4
5					

Im Feld r liegt nun der optimale Baum implizit vor. Die Wurzel des Baumes ist $x_3 (3 = r_{1,4})$. Damit ist der linke Unterbaum $T_{1,2}$ und der rechte $T_{4,4}$ mit Wurzeln $x_1 (r_{1,2} = 1)$ und $x_4 (r_{4,4} = 4)$. Der Baum aus dem vorigen Abschnitt ist demnach optimal.

Wie aufwendig ist nun dieser Algorithmus. Dazu bestimmen wir seine Laufzeit auf der ALGOL-Maschine. Einmaliges Ausführen der Zeilen 10 und 11 kostet entweder 1 oder 3 Zeiteinheiten. Wir rechnen im folgenden mit 3 Zeiteinheiten, da wir eine obere Schranke für die Laufzeit berechnen wollen. Demnach kosten uns die Zeilen 9, 10 und 11 für festes i und j

$$3 + 3(j - (i+1) + 1) + \sum_{l=i+1}^{j} 3 \;=\; 3 + 6(j-i)$$

Zeiteinheiten und demnach einmaliges Ausführen von 6 – 13 für festes i und k $9 + 6k$ Zeiteinheiten. Für festes k kostet uns demnach die Schleife 5 – 13

$$3 + 3(n-k-1+1) + \sum_{i=1}^{n-k} (9 + 6k) \;=\; 3 + (n-k)(12+6k)$$

Zeiteinheiten. Schließlich erhalten wir als Kosten für die Schleife 4 – 13

$$3 + 3(n-1-1+1) + \sum_{k=1}^{n-1} [3 + (n-k)(12+6k)] =$$
$$= n^3 + 6n^2 - n - 3.$$

Für die Initialisierung kommen noch $5 + 8n$ Zeiteinheiten hinzu, so daß die Gesamtlaufzeit höchstens $n^3 + 6n^2 + 7n + 2$ ist. Beachten Sie, daß wir die Laufzeit nur an einer Stelle nach oben abgeschätzt haben: die 3 Einheiten für Zeile 10 und 11. Rechnet man statt dessen mit 1 Zeiteinheit, so erhält man $(2n^3 + 18n - 2n - 9)/3$ als Mindestlaufzeit.

Alg. 1 braucht mindestens $(2n^3 + 18n - 2n - 9)/3$ und höchstens $n^3 + 6n^2 + 7n + 2$ Zeiteinheiten, um einen optimalen Suchbaum für eine Menge mit n Elementen zu konstruieren.

Die kritische Stelle in diesem Algorithmus ist die Suchschleife 8 – 11. Hier entsteht der kubische Aufwand. Daran müssen wir also arbeiten.

Inspiziert man in unserem Beispiel das Feld r der Wurzelindizes genauer, so sieht man, daß die Einträge in jeder Zeile von links nach rechts und in jeder Spalte von oben nach unten zunehmen. Die Wurzel von $T_{i,j}$ liegt also nicht links von der Wurzel von $T_{i,j-1}$ (betrachte die i-te Zeile) und nicht rechts von der

Wurzel $T_{i+1,j}$ (betrachte die j-te Spalte). Dies gilt nicht nur in unserem Beispiel, sondern stets (Knuth, 1971). Man kann daher die Suche nach $r_{i,j}$ auf die Indizes zwischen $r_{i,j-1}$ und $r_{i+1,j}$ beschränken. Wir inkorporieren diese Idee in unser Programm, indem wir die Zeilen (8) und (9) abändern in:

(8′) $m \leftarrow r_{i,j-1}$; $s \leftarrow P_{i,m-1} + P_{m+1,j}$
$t \leftarrow r_{i+1,j}$

(9′) for l from $m+1$ to t do

In unserem Beispiel werden nun im Fall $i=1$, $j=4$ in den Zeilen 8′ und 9′ für m nur noch die Werte 2, 3, 4 probiert anstatt der Werte 1, 2, 3, 4 im ursprünglichen Algorithmus.

Wie gut ist nun dieser modifizierte Algorithmus Alg 2? Zählen wir wie zuvor (siehe Darstellung unten)

Ausführung von	≤ Zeiteinheiten
9′,10,11 für festes i und j	$3+6(r_{i+1,j}-r_{i,j-1})$
6,7,8′, 9′; 10–13 für festes i und k	$10+6(r_{i+1,i+k}-r_{i,i+k-1})$
5–13 für festes k	$3+3(n-k)+\sum_{i=1}^{n-k}(10+6(r_{i-1,i-k}-r_{i,i-k-1}))$ $\leq 3+13(n-k)+6(n-1)$
4–13	$3+3(n-1)+\sum_{k=1}^{n-1}(3+13(n-k)+6(n-1))$ $=\frac{25}{2}n^2-\frac{25}{2}n+3$
1–13	$\frac{25}{2}n^2-\frac{9}{2}n+8$

Beachten Sie dabei

$$\sum_{i=1}^{n-k}(r_{i+1,i+k}-r_{i,i+k-1})=(r_{2,k+1}-r_{1,k})+(r_{3,k+2}-r_{2,k+1})+\cdots+(r_{n-k,n-1}-r_{n-k-1,n-2})+(r_{n-k+1,n}-r_{n-k,n-1})=r_{n-k+1,n}-r_{1,k}\leq n-1.$$

Wie vorher könnten wir auch eine untere Schranke für die Laufzeit von Alg 2 bestimmen. Wir verzichten darauf.

Alg 2 braucht höchstens $\frac{25}{2}n^2-\frac{9}{2}n+8$ Zeiteinheiten, um einen optimalen Suchbaum für eine Menge mit n Elementen zu konstruieren.

Der Speicherplatzbedarf von Alg 1 und Alg 2 ist identisch; vor allem sind da die 3 quadratischen Felder r, P und w zu nennen, von denen man allerdings nur das obere Dreieck braucht. Vom Feld w braucht man zu jedem Zeitpunkt nur eine Nebendiagonale. Trägt man dieser Tatsache in der Programmierung Rechnung, so ergibt sich ein Platzbedarf von etwa n^2 Zellen.

Wir vergleichen die beiden Algorithmen im Schlußabschnitt.

4. Zwei Algorithmen zur Bestimmung nahezu optimaler Bäume

Unsere Algorithmen zur Bestimmung des optimalen Baumes haben mindestens quadratischen Zeit- und Platzbedarf. Ihre Anwendung verbietet sich daher für größere n. Auch sind die Zugriffshäufigkeiten meist nur ungefähr bekannt und es ist daher eigentlich unsinnig, einen optimalen Baum konstruieren zu wollen. Wir wenden uns deswegen nun der Konstruktion nahezu optimaler Bäume zu.

Tragen wir uns dazu die Verteilung aus unserem Beispiel auf der Zahlengeraden auf:

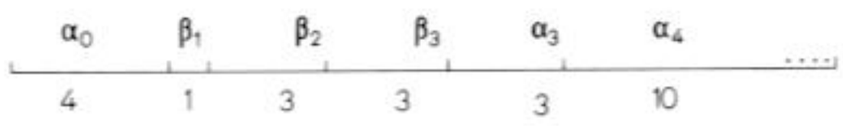


Abb. 5.

Wenn wir den Knoten x_1 als Wurzel wählen, so erhalten wir einen linken Unterbaum vom Gewicht 4 und einen rechten vom Gewicht 19. Analog, wenn wir $x_2[x_3,x_4]$ als Wurzel wählen, so erhalten wir Unterbäume vom Gewicht (5,16) [(8,13), (14,10)]. Es ist intuitiv einleuchtend, daß ein guter Suchbaum „im Gleichgewicht" sein sollte; d. h. linker und rechter Unterbaum sollten möglichst gleich schwer sein. Diese Überlegung führt uns dazu, x_4 als Wurzel zu wählen: Der Gewichtsunterschied ist 4, bei anderer Wahl der Wurzel mindestens 5. Wir müssen nun noch einen Baum für x_1, x_2, x_3 konstruieren. Wählen wir x_1 [x_2, x_3] als Wurzel, so ist die Gewichtsverteilung (4,9) [(5,6), (8,3)]. Wir wählen also x_2.

Insgesamt erhalten wir den Baum

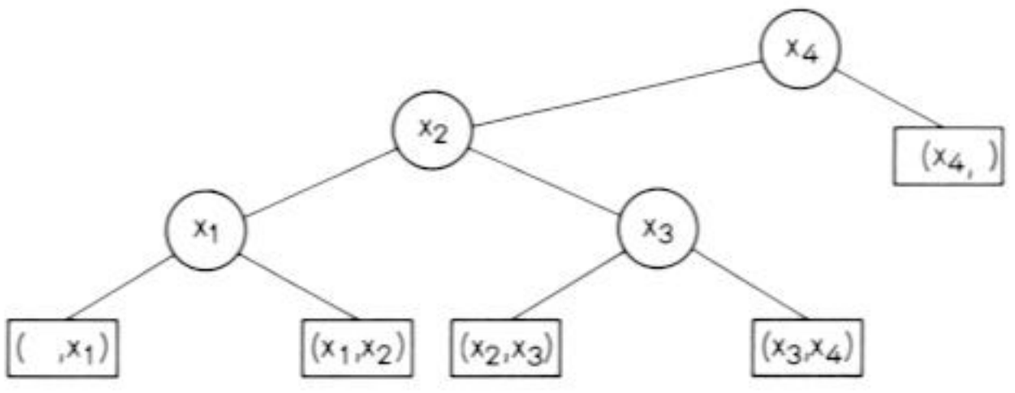


Abb. 6.

mit gewichteter Weglänge
$4\cdot 3+1\cdot 3+0\cdot 3+3\cdot 2+0\cdot 3+3\cdot 3+3\cdot 3+0\cdot 1+10\cdot 1$
$=49$.

Die gewichtete Weglänge des optimalen Baumes war 48. Wir formulieren also folgende Konstruktionsregel.

Regel

Wähle die Wurzel so, daß sich das Gewicht der Unterbäume möglichst wenig unterscheidet. Dann verfahre rekursiv mit den Unterbäumen.

Sei wie oben $w_{ij}=\alpha_{i-1}+\beta_i+\alpha_i+\ \ldots\ +\alpha_{j-1}+\beta_j+\alpha_j$ das Gewicht eines Baumes mit den Knoten $x_i,\ldots,x_j$. Dann ist

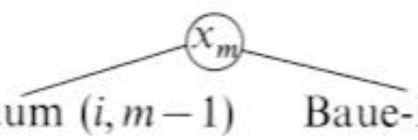
$$w_{ij}=w_{1j}-w_{1,i-1}+\alpha_{i-1}.$$

Wir können nun die obige Regel als folgendes Programm formulieren

```
(1) begin w_{1,0} ← α_0
(2)       for j from 1 to n do w_{1,j} ← w_{1,j-1} + β_j + α_j;
(3)       Baue − Baum (1,n)
    end
```

wo Baue − Baum folgende rekursive Prozedur ist:

```
(4) procedure Baue − Baum (i,j);
    co 1 ≤ i, j ≤ n, i ≤ j+1.
    Wir wollen einen Baum für die Knoten x_i, x_{i+1}, ..., x_j konstruieren, bzw. nur das Blatt
    (x_i, x_{i+1}), falls i = j+1;
    begin
(5)       if i = j+1
(6)       then konstruiere den Baum [(x_i, x_{i+1})]
(7)       else finde m, i ≤ m ≤ j, so daß
                abs(w_{i,m-1} − w_{m+1,j}) ≤ abs(w_{i,k-1} − w_{k+1,j})
                für alle k, i ≤ k ≤ j;
(8)             konstruiere den Baum
```




```
(9)       Baue-Baum (i, m−1)    Baue-Baum (m+1, j)
(10) end
```

Bevor wir dieses Programm analysieren können, müssen wir natürlich noch viele Details einfügen, Was bedeutet „konstruiere den Baum . . .“? Wir nehmen dazu an, daß wir den Baum in Standardspeicherung ablegen wollen, d. h. pro Knoten die Verweise auf linken und rechten Sohn. Die Phrase „konstruiere . . .“ bedeutet dann Setzen dieser Verweise. Den Suchvorgang in Zeile 7 spezifizieren wir später, ebenso eliminieren wir die Rekursion später.

Zunächst einige Aussagen über die Güte des konstruierten Baumes, sei dazu T_{WB} der von Baue − Baum konstruierte Baum und P_{WB} seine gewichtete Weglänge, P_{opt} die gewichtete Weglänge des optimalen Baumes und

$$W=w_{1,n}=\alpha_0+\beta_1+\ \ldots\ +\beta_n+\alpha_n$$

das Gewicht der Verteilung. Es gilt dann (Bayer)

$$P_{\text{opt}}\leqq P_{WB}\leq P_{\text{opt}}+\log P_{\text{opt}}+3{,}44\cdot W,$$

die Abweichung ist also nie sehr groß. Beachten Sie, daß P_{opt} im allgemeinen groß gegenüber W ist. Ein Experiment mit 200 Verteilungen ergab als mittleren (maximalen) Wert für P_{WB}/P_{opt} 1.077 (1.286) (Güttler, et al.). Die Prozedur Baue − Baum konstruiert demnach sehr gute Bäume.

Wie implementiert man nun den Suchvorgang in Zeile 7? Eine wesentliche Beobachtung ist, daß für $i\leq m\leq j$ der Ausdruck $w_{i,m-1}-w_{m+1,j}$ eine wachsende Funktion von m darstellt. (Abb. 7.)

$$\underbrace{\alpha_{i-1}\,\beta_i\,\alpha_{i+1}\ldots\alpha_{m-2}\,\beta_{m-1}\,\alpha_{m-1}}_{w_{i,m-1}}\ \beta_m\ \underbrace{\alpha_m\,\beta_{m+1}\ldots\beta_j\,\alpha_{j+1}}_{w_{m+1,j}}$$

Wir brauchen daher nur den kleinsten Index p zu finden, so daß $w_{i,p-1}-w_{p+1,j}$ positiv ist. Dann ist entweder p oder $p-1$ der gewünschte Wert. Eine mögliche Vorgehensweise für die Suche nach p ist Binärsuche. Man probiert zunächst den Wert $p=\lfloor(i+j)/2\rfloor$ und je nachdem, ob $w_{i,p-1}-w_{p+1,j}$ positiv oder negativ ist, sucht man im linken oder rechten Teilintervall weiter. Wir ersetzen also die Zeile 7 in obigem Programm durch: (Wir schreiben dabei immer $w_{i,p-1}-w_{p+1,j}$ als Abkürzung für

$$(w_{1,p-1}-w_{1,i-1}+\alpha_{i-1})-(w_{1j}-w_{1,p}+\alpha_p))$$

```
7.1  unten ← i; oben ← j; p ← ⌊(i+j)/2⌋
7.2  if w_{i,j-1} − w_{j+1,j} ≤ 0 or i = j
     then co j ist der gesuchte Wert;
7.3  begin m ← j; goto Ende end;
7.4  while oben − unten ≥ 2
7.5  do begin if w_{i,p-1} − w_{p+1,j} > 0
7.6            then oben ← p
```

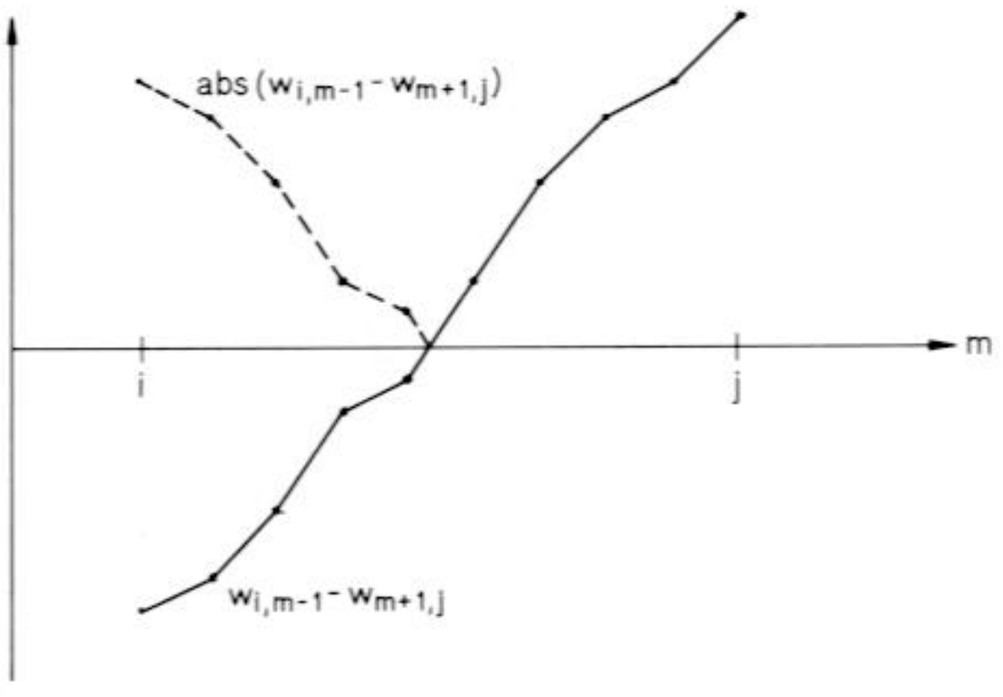


Abb. 7.

7.7 else unten $\leftarrow p$;
7.8 $p \leftarrow \lfloor(\text{oben}+\text{unten})/2\rfloor$
end;
co nun ist oben = unten +1 und entweder oben oder unten ist der gesuchte Wert;
7.9 if $\text{abs}(w_{i,\text{oben}-1} - w_{\text{oben}+1,j})$
$> \text{abs}(w_{i,\text{unten}-1} - w_{\text{unten}+1,j})$
7.10 then $m \leftarrow$ unten else $m \leftarrow$ oben
Ende:

Analysieren wir dieses Programmstück:

Sei dazu $t = \lfloor \log(j-i) \rfloor$. Dann gilt nach dem k-ten Ausführen des Rumpfes 7.5–7.8 der while-Schleife oben – unten $\leq 2^{t-k}$ wie man leicht durch Induktion über k nachprüft. Ferner sieht man leicht ein, daß der Kommentar am Ende der while-Schleife korrekt ist. Also muß für die Anzahl k_0 der Ausführungen des Rumpfes der while-Schleife gelten

$1 = 2^0 \leq 2^{t-k_0}$ also $k_0 \leq \lceil \log(j-i) \rceil \leq 1 + \log(j-i)$.

Insgesamt ist die Laufzeit der Zeile 7 für festes i und j damit $\leq 7 + 5k_0 \leq 11 + 5\log(j-i)$ Zeiteinheiten.

Es bleibt uns nun noch die Rekursion zu eliminieren. Wir tun dies mit der Standardmethode, indem wir den Keller explizit programmieren. Der Keller wird durch ein Feld K simuliert. Die Prozedur Baue – Baum hat 2 Parameter i, j und 4 lokale Variable oben, unten, p und m. Daher weisen wir jedem Aufruf von Baue – Baum einen Block von 7 aufeinander folgenden Speicherzellen zu: die ersten 6 Zellen nehmen die Parameter und die lokalen Variablen auf, die 7-te Zelle die Rücksprungadresse.

Ferner haben wir eine globale Variable AKTIV, die auf den Speicherbereich der augenblicklich aktiven Prozedur Baue – Baum zeigt, d. h. i entspricht der Zelle K[AKTIV+1], p der Zelle K[AKTIV+5] und die Rücksprungadresse steht in K[AKTIV+7]. Damit erhalten wir schließlich unser Programm, indem wir den Aufruf in Zeile 3 des Hauptprogramms ersetzen durch

(3.1) AKTIV $\leftarrow$ 0; K[AKTIV+1] $\leftarrow$ 1; K[AKTIV+2] $\leftarrow n$;
K[AKTIV+7] $\leftarrow$ „HP“;
(3.2) goto Baue – Baum;
(3.3) HP:

Die Zeile 4 ersetzen wir durch die Marke Baue – Baum: In den Zeilen 5–8 ersetzen wir die Parameter und lokalen Variablen durch Zugriffe auf die entsprechenden Feldelemente und den ersten Aufruf in Zeile 9 ersetzen wir durch

(9.1) K[AKTIV+7+1] $\leftarrow$ K[AKTIV+1];
(9.2) K[AKTIV+7+2] $\leftarrow$ K[AKTIV+6] − 1;
(9.3) K[AKTIV+7+7] $\leftarrow$ „Rückkehr 1-ter Aufruf“;
(9.4) AKTIV $\leftarrow$ AKTIV + 7;
(9.5) goto Baue – Baum;
(9.6) Rückkehr 1-ter Aufruf: AKTIV $\leftarrow$ AKTIV − 7

und analog den 2-ten Aufruf. Schließlich ersetzen wir das end in Zeile 10 durch

(10) goto K[AKTIV+7].

Jetzt endlich verwenden wir nur noch primitive Befehle in unserem Programm. Es erscheint nun aber sehr schwierig, die Laufzeit dieses Programms zu bestimmen. Aber man muß nur richtig zählen!

Bei jedem Aufruf der Prozedur Baue – Baum wird ein Knoten oder ein Blatt des Baumes konstruiert. Also wird die Prozedur genau $2n+1$ mal aufgerufen. Für jeden Aufruf von Baue – Baum (außer dem ersten) werden die Zeilen 5, 6 oder 8, 9.1–9.6 und 10 je einmal aufgeführt, beim ersten Aufruf statt 9.1–9.6 die Zeilen 3.1–3.3. Setzen wir noch willkürlich fest, daß uns Zeile 6 bzw. 8 eine Zeiteinheit kostet, so entstehen in den einzelnen Zeilen folgende Gesamtkosten:

Zeile	Kosten
1, 2	$4+4n$
3.1–3.3	6
5	$2n+1$
6 und 8	$2n+1$
9.1–9.6	$6 \cdot 2n$
10	$2n+1$
	$22n+13$.

Damit haben wir alle Kosten gezählt außer denen, die in Zeile 7 entstehen. Sei dazu $T(n)$ der maximale Aufwand, der bei der Konstruktion eines Baumes mit n Knoten durch die Prozedur Baue – Baum in Zeile 7 entsteht; d. h. wir betrachten einen Aufruf Baue – Baum (i,j) mit $j-i+1=n$.

Für $n=0$ ist $T(n)=0$, da dann Zeile 7 nie erreicht wird. Für $n=1$ verläuft der Test in 7.2 positiv und daher ist $T(1)=6$. Sei nun $n>1$, dann erwachsen zunächst Kosten $11+5\log(j-i)=11+5\log(n-1)$ in Zeile 7; ferner wird Baue – Baum $(i,m-1)$ und Baue – Baum $(m+1,j)$ aufgerufen. Also entstehen noch die zusätzlichen Kosten

$T(m-1-i+1)$ und $T(j-(m+1)+1)$

für ein m, $i \leq m \leq j$. Da wir den Wert von m nicht kennen, müssen wir das Maximum über alle möglichen Werte von m nehmen und erhalten (beachten Sie $(m-i)+(j-m)=n-1$)

$$T(n) = \max \{11+5 \log (n-1)+T(k)+T(n-k-1): 0 \leq k \leq n-1\} \text{ für } n>1$$
$$T(1) = 6$$
$$T(0) = 0.$$

Damit haben wir eine Rekursionsgleichung für die in Zeile 7 entstehenden Gesamtkosten hergeleitet. Vom Ursprung unserer Rekursion her wissen wir, daß der

Fall $k=0$ (bzw. $k=n-1$) vermutlich am ungünstigsten ist. Wir treiben dann ja einen ziemlich hohen Aufwand ($11+5\ \log(n-1)$ Zeiteinheiten), um die Wurzel zu bestimmen, verbleiben aber immer noch mit dem Problem einen Baum mit $n-1$ Knoten konstruieren zu müssen. Setzen wir also $k=0$ in der obigen Gleichung, so erhalten wir:

$$
\begin{aligned}
T(0) &= 0 \qquad T(1)=6\\
T(n) &\geq 11+5\log(n-1)+T(n-1)\\
&\geq [11+5\log(n-1)]+[11+5\log(n-2)]+T(n-2)\\
&\ \ \vdots\\
&\geq [11+5\log(n-1)]+\ \ldots\ +[11+5\log 1]+T(1)\\
&= 11(n-1)+5\log[(n-1)!]+6.
\end{aligned}
$$

Wir müssen nun nur noch verifizieren, daß $k=0$ in der Tat der ungünstigste Fall war. Dazu zeigen wir durch Induktion über n:

$$
\begin{aligned}
T(0) &= 0 \qquad T(1)=6\\
T(n) &\leq 11(n-1)+5\log[(n-1)!]+6 \text{ für } n\geq 2.
\end{aligned}
$$

Wir überlassen den einfachen Induktionsbeweis dem Leser. Damit wissen wir: In Zeile 7 entstehen die Kosten

$$11n-5+5\log(n-1)!\approx 5n\log n+3{,}785-2{,}5\log n+8{,}842$$

wegen $\log n!\approx(n+1/2)(\log n-1{,}443)+2{,}04$. Insgesamt gilt:

Alg 3 braucht höchstens $5n\log n+25{,}78n-2{,}5\log n+21{,}84$ Zeiteinheiten, um einen (nahezu optimalen) Suchbaum für eine Menge S mit n Elementen zu konstruieren.

Können wir Alg 3 noch verbessern?

Die kritische Stelle ist offensichtlich Zeile 7. Dort entstehen Kosten $\theta(n\log n)$ [lies: von der Ordnung genau $n\log n$], während die übrigen Kosten linear in n sind. Wenn wir den Suchvorgang in Zeile 7 verbessern wollen, müssen wir seine schwache Stelle untersuchen: die Wurzel liegt ganz am Rand. Nehmen wir etwa an, der Aufruf Baue – Baum (i,j) wählt x_i als Wurzel. Dann probieren wir in der Suchschleife 7.1 – 7.11 für p nacheinander die Werte $i+(j-i)/2$, $i+(j-i)/4$, $i+(j-i)/8$, $i+(j-i)/16$, usw. Wir tasten uns also in exponentiell kleiner werdenden Schritten von der Mitte an den Rand heran. Drehen wir den Spieß doch um: erforschen wir die Welt in exponentiell größer werdenden Schritten vom Rand her [2]:

(7.1′) $p\leftarrow(i+j)/2$

(7.2′) if $w_{i,p-1}-w_{p+1,j}>0$
then goto linke Hälfte else goto rechte Hälfte;
linke Hälfte: co wir wissen nun, daß m einen der Werte $i, i+1, \ldots, \llcorner(i+j)/2\lrcorner$ annimmt. Wir probieren nun $p=i+1$, $i+2$, $i+4$, $i+8$, ... bis wir ein p mit $w_{i,p-1}-w_{p+1,j}>0$ finden;

(7.3′) $t\leftarrow 0$; $p\leftarrow i+1$;

(7.4′) while $w_{i,p-1}-w_{p+1,j}<0$ do
begin $t\leftarrow t+1$; $p\leftarrow i+2^t$ end;
co es ist nun $w_{i,p-1}-w_{p+1,j}\geq 0$ und $p=i+2^t$ und $(w_{i,p'-1}-w_{p'+1,j}<0,\ p'=i+2^{t-1}$ oder $t=0)$;

Wir suchen nun nach m auf dem Intervall $p',\ldots,p$ durch Binärsuche wie vorher beschrieben;

(7.5′) unten $\leftarrow$ if $t=0$ then i else $i+2^{t-1}$;
oben $\leftarrow i+2^t$; $p\leftarrow\llcorner$(unten + oben)$/2\lrcorner$;
goto Binärsuche;
rechte Hälfte: analog (7.3′) – (7.5′)
Binärsuche: das Programmstück (7.2) – (7.10) von vorher.

Nun stehen wir vor der ziemlich komplexen Aufgabe, dieses Programm zu analysieren. Sei dazu t_0 der Wert von t mit dem die Schleife (7.4) verlassen wird. Dann entstehen folgende Kosten:

Zeile	Kosten
(7.1′), (7.2′),	
(7.3′)	5
(7.4′)	$1+4\cdot t_0$
(7.5′)	5
Binärsuche:	$8+5\log(\max(2^{t_0-1},1))$
und danach	(Binärsuche auf einem Intervall der Länge 2^{t-1})
	19 falls $t_0=0$
	$14+9t_0$ falls $t_0>0$.

Welchen Bezug hat nun t_0 zu den Größen i, j und m? Offensichtlich gilt:

$$p'=\left.\begin{cases} i & \text{falls } t_0=0\\ i+2^{t_0-1} & \text{falls } t_0>0\end{cases}\right\}\leq m$$

also

$$t_0\leq\begin{cases}1+\log(m-i) & \text{falls } m>i\\ 0 & \text{falls } m=i.\end{cases}$$

Wir haben nun das obige Programm analysiert unter der Annahme, daß m in der linken Hälfte des Intervalls $i,\ldots,j$ liegt. Eine ähnliche Analyse gilt für den anderen Fall. Damit erhalten wir folgende Rekursionsgleichung für den Gesamtaufwand in Zeile 7

$$
\begin{aligned}
T(0) &= 0\\
T(n) &= \max\{19+T(n-1),\\
&\qquad 23+9\ \log k+T(k)+T(n-k-1),\\
&\qquad 23+9\ \log(n-l-1)+T(l)+T(n-l-1);\\
&\qquad 1\leq k\leq\llcorner n/2\lrcorner,\ \llcorner n/2\lrcorner\leq l<n-1\}
\end{aligned}
$$

mit einer Lösung [7], S. 95, Satz 10

$$19n\leq T(n)\leq 32n.$$

Damit gilt:

Alg 4 braucht höchstens $54n+13$ Zeiteinheiten, um

einen (nahezu) optimalen Suchbaum für eine Menge S mit n Elementen zu konstruieren.

Wir haben nun das Ende einer langen Reise erreicht: von 4^n über n^3 nach n^2, weiter nach $n \log n$ und schließlich nach n.

5. Ein Vergleich

Wir lernten nun 5 Algorithmen zur Konstruktion von (nahezu) optimalen Suchbäumen kennen mit den Laufzeiten

Alg 0 4^n
Alg 1 n^3+6n^2+7n+2
Alg 2 $(25n^2-9n+16)/2$
Alg 3 $5n \log n-25{,}78n-2{,}5 \log n+21{,}84$
Alg 4 $54n+13$.

Die nächste Tabelle gibt die Laufzeit (in Zeiteinheiten) der 5 Algorithmen für einige Problemgrößen wieder.

	Alg 0	Alg 1	Alg 2	Alg 3	Alg 4
5	1024	312	298	203	283
10	1048576	1672	1213	438	553
50	10^{30}	140352	31033	27108	2713
100	10^{60}	10^6	124558	5905	5413
500		$1.26 \cdot 10^8$	$3.1 \cdot 10^6$	35304	27013
1000		10^9	$1.25 \cdot 10^7$	75606	54013
5000		$1.25 \cdot 10^{11}$	$3.1 \cdot 10^9$	436084	270013
10000		10^{12}	$1.25 \cdot 10^9$	922174	540013
100000		10^{15}	$1.25 \cdot 10^{11}$	$10{,}9 \cdot 10^6$	$5.4 \cdot 10^6$

Man sieht sehr deutlich die Explosion der Laufzeit bei Alg 0 und den immer deutlicher werdenden Unterschied zwischen den anderen Algorithmen. Dieser Sachverhalt wird noch einprägsamer beschrieben durch folgenden Vergleich:
Nehmen wir an, wir hätten eine Anlage, die 10^4 Instruktionen/Sekunde ausführen kann und wir wären bereit 1 min Rechenzeit zu bezahlen. Bis zu welcher Größe können wir dann Suchbäume konstruieren?

	Alg 0	Alg 1	Alg 2	Alg 3	Alg 4
n_1 maximal lösbares Problem in 1 min	7–8	37	69	809	1111
n_2 maximal lösbares Problem in 10 min	9–10	32	219	6710	11110
	$n_2=n_1+2$	$n_2/n_1=2.21$	$n_2/n_1=3.17$	$n_2/n_1=8.29$	$n_2/n_1=10$

Falls diese Größe nicht ausreicht, können wir eine schnellere Maschine benutzen (etwa 10^5 Instruktionen/Sekunde) oder mehr Rechenzeit verwenden (etwa 10 Minuten). Die zweite Zeile in obiger Tabelle gibt die maximale Größe des dann lösbaren Problems wieder. Bei sehr ineffizienten Algorithmen (Alg 0) hat der Übergang zur schnellen Maschine praktisch keinen Einfluß, bei Alg 1 (Alg 2, Alg 4) gewinnen wir den Faktor

$\approx \sqrt[3]{10}=2.17 (\approx \sqrt[2]{10}=3.16,\ 10)$.

Der Übergang zum effizienten Algorithmus ist allemal vielversprechender. Eine andere Betrachtungsweise ist folgende: Nehmen wir an, wir könnten ein Problem der Größe n in t Zeiteinheiten lösen. Dann brauchen wir t' Zeiteinheiten, um ein Problem der Größe $n'=10n$ zu lösen. (Die angegebenen Größen von t' erhält man durch Einsetzen von $10n$ in die berechneten Laufzeiten).

	Alg 0	Alg 1	Alg 2	Alg 3	Alg 4
$t' \approx$	t^{10}	$1000 \cdot t$	$100 \cdot t$	$10t+166n$	$10 \cdot t$

Wir versuchten an einem Beispiel, die Entwicklung und Analyse effizienter Algorithmen zu demonstrieren. Entwicklung und Analyse beeinflußten sich gegenseitig. Die Analyse zeigte die kritischen Stellen der Algorithmen auf und legte Modifikationen nahe, die Entwicklung gab Hinweise für die Analyse.

Literatur

1. Bayer, P. J.: Improved bounds on the cost of optimal and balanced binary search trees. To appear in Acta Informatica
2. Fredman, M. L.: Two applications of a probabilistic search technique: Sorting $X+Y$ and building balanced search trees. 7th ACM Symposium on Theory of Computing, 1975, pp. 240–244
3. Gilbert, E. N., Moore, E. F.: Variable-length binary encodings. Bell System Techn. J. **38,** 933–968 (1959)
4. Güttler, R., Mehlhorn, K., Schneider, W., Wernet, N.: Binary search trees: Average and worst case behaviour. GI-Jahrestagung 1976. Informatik-Fachberichte, Bd. 5. Berlin, Heidelberg, New York: Springer 1976
5. Knuth, D. E.: The art of computer programming, Vol. 3: Sorting and searching. New York: Addison Wesley 1973
6. Knuth, D. E.: Optimum binary search trees. Acta Informatica **1,** 14–25 (1971)
7. Mehlhorn, K.: Effiziente Algorithmen. Stuttgart: Teubner 1977

Eingegangen am 11. Mai 1978

Prof. Dr. K. Mehlhorn
Fachbereich 10 der Universität
D-6600 Saarbrücken