Kapitel 9

Ausgewählte Themen

9.1 Suchen in Texten

In zahlreichen Anwendungen von Computern spielen Texte eine dominierende Rolle. Man denke etwa an Texteditoren, Literaturdatenbanken, Bibliothekssysteme und Systeme zur Symbolmanipulation. Der Begriff *Text* wird hier meistens in einem sehr allgemeinen Sinne benutzt. Texte sind nicht weiter strukturierte Folgen beliebiger Länge von Zeichen aus einem endlichen Alphabet. Das Alphabet kann Buchstaben, Ziffern und zahlreiche Sonderzeichen enthalten.

Der diesen Anwendungen zugrundeliegende Datentyp ist der Typ *string* (*Zeichenkette*). Wir lassen offen, wie dieser Datentyp programmtechnisch realisiert wird. Als Möglichkeiten kommen z.B. die Bereitstellung des Datentyps string als einer der Grundtypen der Sprache in Frage oder die Realisierung als File_of_characters, als Array_of_characters oder als verkettete Liste von Zeichen. Unabhängig von der programmtechnischen Realisierung soll jeder Zeichenkette eine nichtnegative, ganzzahlige Länge zugeordnet werden können und der Zugriff auf das i-te Zeichen einer Zeichenkette für jedes $i \geq 1$ möglich sein. Algorithmen zur Verarbeitung von Zeichenketten (*string processing*) umfassen ein weites Spektrum. Dazu gehören das Suchen in Texten und allgemeiner das Erkennen bestimmter Muster (*pattern matching*), das Verschlüsseln und Komprimieren von Texten, das Analysieren (*parsing*) und Übersetzen von Texten und viele andere Algorithmen.

Wir wollen in diesem Abschnitt nur das Suchen in Texten behandeln und einige klassische Algorithmen zur Lösung dieses Problems angeben. Das Suchproblem kann genauer wie folgt formuliert werden:

Gegeben sind eine Zeichenkette (*Text*) $a_1 \dots a_N$ von Zeichen aus einem endlichen Alphabet Σ und eine Zeichenkette, das *Muster* (*pattern*), $b_1 \dots b_M$, mit $b_i \in \Sigma$, $1 \leq i \leq M$. *Gesucht* sind ein oder alle Vorkommen von $b_1 \dots b_M$ in $a_1 \dots a_N$, d.h. Indizes i mit $1 \leq i \leq (N - M + 1)$ und $a_i = b_1$, $a_{i+1} = b_2$, $\dots$, $a_{i+M-1} = b_M$.

In der Regel ist die Länge N des Textes sehr viel größer als die Länge M des Musters. Als Beispiel verweisen wir auf das Oxford English Dictionary (OED): Die zweite, im Jahre 1989 publizierte Ausgabe des OED umfaßt etwa 616500 definierte Stichworte

und beansprucht 540 Mb Speicherplatz bzw. 20 Bände mit insgesamt 21728 Seiten in der gedruckten Version.

Das OED ist ein Beispiel für *statischen* Text; Änderungen sind verhältnismäßig selten und im Verhältnis zum Gesamtumfang geringfügig. Demgegenüber ist der durch Texteditoren manipulierte Text *dynamisch*; Änderungen sind häufig und erheblich. Will man das Suchproblem für statischen Text lösen, so kann es sich lohnen, den Text durch Hinzufügen von geeigneter Information (einem *Index*) so aufzubereiten, daß die Suche für verschiedene Muster gut unterstützt und insbesondere nicht das Durchsuchen des gesamten Textes erforderlich wird. Bei dynamischem Text lohnt sich eine aufwendige Vorverarbeitung in der Regel nicht. Es kann sich in diesem Fall aber auszahlen, Suchalgorithmen von der Struktur des Musters und vom zugrundeliegenden Alphabet abhängig zu machen.

Wir geben im folgenden eine Reihe von Algorithmen zur Lösung in dynamischen Texten an und verweisen für den anderen Fall (statische Texte) und neueste Ergebnisse über Algorithmen zur Textsuche auf [11].

9.1.1 Das naive Verfahren zur Textsuche

Am einfachsten läßt sich das Problem, ein Vorkommen des Musters $b_1 \dots b_M$ im Text $a_1 \dots a_N$ zu finden, wie folgt lösen: Man legt das Muster, beginnend beim ersten Zeichen des Textes, der Reihe nach an jeden Teilstring des Textes mit Länge M an und vergleicht zeichenweise von links nach rechts, ob eine Übereinstimmung zwischen Muster und Text vorliegt oder nicht (ein *Mismatch*), solange, bis man ein Vorkommen des Musters im Text gefunden oder das Ende des Textes erreicht hat. In Pascal-ähnlicher Notation kann das Verfahren so beschrieben werden:

```
for i := 1 to N − M + 1 do
  begin
    found := true;
    for j := 1 to M do
      if a[i+j−1] ≠ b[j] then found := false;
    if found then write (`B kommt vor von Position', i,
      `bis Position', i + M − 1);
  end
```

Bei diesem Verfahren muß das Muster B offensichtlich $(N-M+1)$-mal an den Text A angelegt und dann jeweils ganz durchlaufen werden. Das bedeutet, daß stets $(N-M+1)\cdot M$ Vergleiche ausgeführt werden. Die Laufzeit des Verfahrens ist also von der Größenordnung $\Theta(N \cdot M)$. Eine Verbesserung ist möglich, wenn man das Muster jeweils nur bis zum ersten Mismatch durchläuft:

```
function bruteforce (a, b : string; M, N: integer) : integer;
{liefert den Beginn des Musters b[1..M] im Text a[1..N] oder
  einen Wert > N, falls b in a nicht vorkommt}
var i, j : integer;
begin
```

```
i := 1;
j := 1;
repeat
  if a_i = b_j
    then
      begin
        i := i + 1;
        j := j + 1
      end
    else
      begin
        i := i − j + 2;
        j := 1
      end
until (j > M) or (i > N);
if j > M
  then bruteforce := i − M
  else bruteforce := i
end
```

Jetzt werden in vielen praktischen Fällen nur noch $O(M+N)$ Vergleiche zwischen Zeichen im Text und Zeichen im Muster durchgeführt. Einen solchen Fall zeigt das Beispiel in Abbildung 9.1; hier wird in einem Text mit Länge 50 (einschließlich Leerzeichen und Komma) nach einem Muster mit Länge 4 gesucht. Nach insgesamt 51 Vergleichen wird ein Vorkommen des Musters im Text entdeckt. Der Grund dafür ist, daß in den meisten Fällen ein Mismatch bereits beim ersten Buchstaben auftritt und daher das Muster sofort an die nächste Textposition verschoben werden kann. Andererseits ist es natürlich nicht schwer, Beispiele zu finden, in denen das naive Verfahren mindestens (NM) Schritte benötigt, um ein Vorkommen des Musters im Text zu finden: Man wähle als Text eine Zeichenfolge bestehend aus $N-1$ Nullen und einer 1 als letztem Zeichen. Das Muster sei ähnlich aufgebaut, d.h. auf $M-1$ Nullen folge eine 1. Dann wird stets erst beim Vergleich des letzten Zeichens im Muster mit einem Zeichen im Text ein Mismatch entdeckt. Bis man das Vorkommen des Musters im Text gefunden hat, werden also $(N-M)\cdot M+M=\Omega(MN)$ Zeichen verglichen.

Das naive Verfahren ist gedächtnislos in folgendem Sinne: Dieselbe Textstelle wird unter Umständen mehrfach inspiziert; das Verfahren merkt sich nicht, welche Zeichen im Text bereits mit einem Anfangsstück des Musters übereingestimmt haben, bis ein Mismatch auftrat. Das im folgenden Abschnitt dargestellte Verfahren von Knuth-Morris-Pratt nutzt diese Information. Es kann erreicht werden, daß der Zeiger i auf die nächste Textstelle, anders als beim naiven Algorithmus, niemals zurückgesetzt werden muß.

9.1.2 Das Verfahren von Knuth-Morris-Pratt

Dem Verfahren liegt folgende Idee zugrunde: Tritt beim Vergleich des Musters mit dem Text an der j-ten Stelle des Musters ein Mismatch auf, so haben die vorangehenden

```
er sprach abrakadabra, es bewegte sich aber nichts
[a]ber
 [a]ber
  [a]ber ...
          ab[e]r
           [a]ber ...
                                          [a]ber
                                           aber
```

Abbildung 9.1

$j-1$ Zeichen im Muster und Text übereingestimmt. Wir nutzen jetzt diese Information, um das Muster nach dem Mismatch nicht stets um eine Position, wie beim naiven Verfahren, sondern so weit wie möglich nach rechts zu verschieben. Betrachten wir nun ein Beispiel für ein binäres Alphabet:

		i
Text:	...	010110101 ...
Muster:		0101[0]1

Beim Vergleich des fünften Zeichens im Muster mit dem darüberstehenden i-ten Zeichen im Text tritt ein Mismatch auf. Die vorangehenden vier Zeichen 0101 des Musters haben also mit den darüberstehenden Zeichen im Text übereingestimmt. Wird das Muster um nur eine Position nach rechts verschoben, so tritt mit Sicherheit wieder ein Mismatch auf, und zwar schon an der ersten Stelle. Wie weit kann man das Muster nach rechts verschieben, ohne ein Vorkommen im Text zu übersehen? Offenbar kann man das Muster gleich um zwei Positionen nach rechts verschieben und erneut das i-te Zeichen im Text mit dem darunterstehenden Zeichen im Muster vergleichen. Im vorliegenden Beispiel weiß man, daß keine Übereinstimmung vorliegen kann, da die 0 an der fünften Stelle im Muster nicht mit dem darüberstehenden Zeichen im Text übereingestimmt hat. Das den Mismatch verursachende Zeichen im Text muß also eine 1 gewesen sein. Sie führt abermals zu einem Mismatch beim Vergleich mit dem dritten Zeichen im Muster. Im allgemeinen, d.h. für Texte über beliebigen Alphabeten, kann man aber so nicht argumentieren. Wir bestimmen dann die maximal mögliche Verschiebung des Musters nach rechts allein unter Ausnutzung der Kenntnis der Zeichen im Muster, die mit den darüberstehenden Zeichen im Text übereingestimmt haben, bis ein Mismatch auftrat.

Die allgemeine Situation ist in Abbildung 9.2 dargestellt und kann folgendermaßen beschrieben werden. Nehmen wir an, beim Vergleich des j-ten Zeichens im Muster mit dem i-ten Zeichen im Text tritt ein Mismatch auf, d.h.:

1. Die letzten $j-1$ gelesenen Zeichen im Text stimmen mit den ersten $j-1$ Zeichen des Musters überein.

2. Das gerade gelesene i-te Zeichen im Text ist verschieden vom j-ten Zeichen im Muster.

Mit welchem Zeichen im Muster kann man das i-te Textzeichen als nächstes vergleichen, so daß man kein Vorkommen des Musters im Text übersieht?

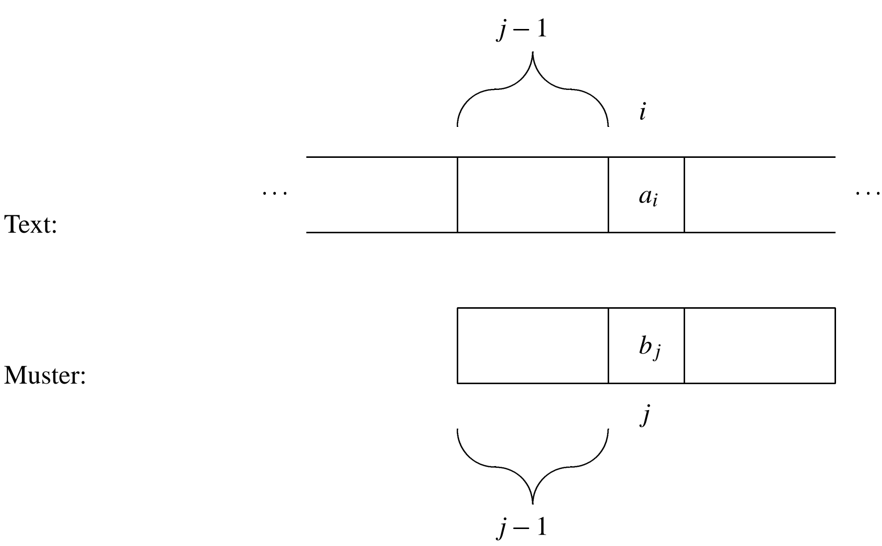


Abbildung 9.2

Dazu muß man offenbar von dem Anfangsstück des Musters mit Länge $j-1$ ein Endstück maximaler Länge l bestimmen, das ebenfalls Anfangsstück des Musters ist. Dann ist die Position $l+1$ im Muster, die wir $next[j]$ nennen wollen, die von rechts her nächste Stelle im Muster, die man mit dem i-ten Zeichen im Text mit der Chance auf Übereinstimmung vergleichen muß.

Falls im Vergleich des i-ten Zeichens im Text mit dem Zeichen an Position $next[j]$ im Muster kein Mismatch mehr auftritt, verschiebt man den Zeiger im Text und im Muster um eine Position nach rechts und vergleicht die Zeichen an den Positionen $i+1$ und $next[j]+1$ in Text und Muster.

Falls im Vergleich des i-ten Zeichens im Text mit dem Zeichen an Position $next[j]$ im Muster jedoch erneut ein Mismatch auftritt, gehen wir entsprechend vor: Wir bestimmen die Länge l' des längsten echten Endstücks des Anfangsstücks mit Länge $next[j]-1$, das zugleich Anfangsstück des Musters ist, und vergleichen das i-te Zeichen im Text mit dem Zeichen an Position $l'+1 = next[next[j]]$. Falls immer noch ein Mismatch auftritt, muß man wie beschrieben fortfahren, d.h.

$$next[next[\dots next[j]\dots]]$$

bestimmen. Es müssen also immer wieder für Anfangsstücke des Musters Endstücke bestimmt werden, die selbst Anfangsstücke maximaler Länge des Musters sind. Das i-te Zeichen im Text muß der Reihe nach mit den Zeichen an den Positionen j, $next[j]$, $next[next[j]]\ldots$ im Muster verglichen werden. Das geschieht solange, bis erstmals kein Mismatch mehr auftritt oder man an der Position 1 im Muster angekommen ist. Im letzten Fall kann man offenbar den Textzeiger i um eine Position nach rechts verschieben und das Zeichen an Position $i+1$ mit dem ersten Zeichen im Muster vergleichen.

Es gilt also für jedes j mit $2 \leq j \leq M$, M Länge des Musters:

$next[j] = 1+$ Länge des längsten echten Endstücks der ersten $j-1$ Zeichen,
das zugleich Anfangsstück des Musters ist.

Wir setzen noch $next[1] = 0$. Nehmen wir nun an, daß das *next*-Array bekannt ist, so kann das Verfahren von Knuth-Morris-Pratt wie folgt beschrieben werden:

```
function kmp_search (a, b : string; M, N : integer) : integer;
var i, j : integer;
begin
  i := 1;
  j := 1;
  repeat
    if a_i = b_j or j = 0
      then
        begin
          i := i + 1;
          j := j + 1
        end
      else
        j := next[j]
  until (j > M) or (i > N);
  if j > M
    then kmp_search := i − M
    else kmp_search := i
end
```

Man kann aus dieser Formulierung des Verfahrens unmittelbar ablesen, daß der Zeiger i, der auf die jeweils nächste zu inspizierende Stelle im Text weist, nie zurückgesetzt wird. Der Zeiger j kann natürlich zurückgesetzt werden. Mit jeder Zuweisung $j := next[j]$ verringert sich der Wert von j um wenigstens 1; für $j = 1$ wird $next[j] = 0$ und $j = 0$ und damit beim nächsten Durchlauf der **repeat**-Schleife sowohl i als auch j um 1 erhöht. j kann natürlich insgesamt nur so oft herabgesetzt werden, wie es erhöht wurde. Da jedoch i und j innerhalb der **repeat**-Schleife stets gemeinsam erhöht werden, kann j insgesamt nur so oft herabgesetzt werden, wie i heraufgesetzt wurde. Weil die **repeat**-Schleife für $i > N$ abbricht, folgt, daß die Anweisung $j := next[j]$ insgesamt höchstens N-mal ausgeführt wird. Nimmt man also an, daß das *next*-Array bekannt ist, so benötigt das Verfahren $O(N)$ Schritte.

Wir müssen jetzt noch angeben, wie man die Belegung des *next*-Arrays berechnet. Das geschieht durch ein Programm, das eine ganz ähnliche Struktur hat wie das bereits angegebene Verfahren *kmp_search*. Darin kommt zum Ausdruck, daß wir das Muster mit sich selbst vergleichen.

```
procedure initnext;
  var i, j : integer;
  begin
    i := 1;
    j := 0;
    next[i] := 0;
      repeat
        if b_i = b_j or j = 0
          then
            begin
              i := i + 1;
              j := j + 1;
              next[i] := j
            end
          else
            j := next[j]
      until i > M
  end
```

Das *next*-Array muß für alle i, $2 \leq i \leq M$, M Länge des Musters $B = b_1 \dots b_M$, so belegt werden, daß gilt: Ist $next[i] = j$, so ist $j-1$ die Länge des längsten echten Endstücks des Anfangsstücks mit Länge $i-1$, das zugleich Anfangsstück des Musters ist. Zunächst wird $next[1] = 0$ gesetzt, wie wir das im Verfahren *kmp_search* verlangt haben. Nehmen wir jetzt an, daß $next[1], \dots, next[i]$ im angegebenen Sinne bereits richtig belegt wurden. Nach Ausführung der Zuweisung $next[i] := j$ ist also $j-1$ die Länge des längsten echten Endstücks des Anfangsstücks mit Länge $i-1$, das zugleich Anfangsstück des Musters ist. Wir vergleichen nun b_i und b_j.

Fall 1: $[b_i = b_j]$
Dann kennen wir das längste echte Endstück des Musters im Anfangsstück mit Länge i, das zugleich Anfangsstück des Musters ist. Es enthält das nächste Zeichen b_i bzw. b_j und hat damit die Länge j. Nach Definition ist folglich $next[i+1] = j+1$.

Fall 2: $[b_i \neq b_j]$
Zur Bestimmung des längsten echten Endstücks des Anfangsstücks mit Länge i des Musters, das zugleich Anfangsstück des Musters ist, müssen wir genauso vorgehen, wie wir das beim Vergleich des Musters mit dem Text getan haben (i ist dabei Textzeiger, j Musterzeiger). Wir vergleichen der Reihe nach b_i mit den Zeichen an den Positionen $next[j]$, $next[next[j]] \dots$, bis erstmals eine Übereinstimmung mit b_i erreicht wurde oder der Zeiger j bei 0 angekommen ist. Im letzten Fall wissen wir, daß das leere Wort das längste echte Endstück des Anfangsstücks des Musters mit Länge i ist, das zugleich Anfangsstück des Musters ist, und wir können $next[i+1] = 1$ setzen. Sonst sei j' die Position, für die erstmals eine Übereinstimmung bei b_i festgestellt wurde. Dann müssen wir setzen: $next[i+1] = j'+1$.

Wieviele Schritte benötigt das oben angegebene Verfahren zur Bestimmung des *next*-Arrays? i und j durchlaufen Positionen im Muster, dabei wird i nur erhöht, während j entweder erhöht oder wieder herabgesetzt wird. j kann natürlich insgesamt nur so oft herabgesetzt werden, wie es erhöht wurde. Da j jedoch stets nur gemeinsam mit i erhöht wird, die Schleife aber bei $i > M$ abbricht, gilt: Die Gesamtzahl aller Ausführungen der Anweisung $j := next[j]$ in allen Schleifendurchläufen der **repeat**-Schleife ist höchstens M. Damit folgt, daß das *next*-Array in $O(M)$ Schritten bestimmt werden kann. Das Verfahren von Knuth-Morris-Pratt benötigt also insgesamt höchstens $O(M+N)$ Schritte, um ein Muster mit Länge M in einem Text mit Länge N zu finden.

Die Existenz eines Verfahrens zur Textsuche mit Zeitkomplexität $O(M+N)$, statt $\Theta(M \cdot N)$ wie beim naiven Verfahren, folgt aus einem allgemeinen Satz von S. Cook über die Simulierbarkeit von gewissen Automaten [31]. Die wichtigste Referenz für die von uns angegebene Version des Verfahrens ist [91]. Man kann das Verfahren auch in einem automatentheoretischen Gewand präsentieren: Zu einem gegebenen Muster wird ein endlicher Automat konstruiert, der den gegebenen Text liest und genau dann in einen ausgezeichneten Endzustand übergeht, wenn ein Vorkommen des Musters im Text gefunden wurde. Die Zustände des Automaten repräsentieren, welches Anfangsstück des Musters bereits entdeckt wurde. Diese Darstellung des Verfahrens wurde z.B. in [6] gewählt. Eine Erweiterung auf die gleichzeitige Suche nach mehreren Mustern findet man in[2].

9.1.3 Das Verfahren von Boyer-Moore

Bei dem Verfahren von Boyer und Moore [22] werden die Zeichen im Muster nicht von links nach rechts, sondern von rechts nach links mit den Zeichen im Text verglichen. Man legt das Muster zwar der Reihe nach an von links nach rechts wachsende Textpositionen an, beginnt aber einen Vergleich zwischen Zeichen im Text und Zeichen im Muster immer beim letzten Zeichen im Muster. Tritt dabei kein Mismatch auf, hat man ein Vorkommen des Musters im Text gefunden. Tritt jedoch ein Mismatch auf, so wird eine Verschiebung des Musters berechnet, d.h. eine Anzahl von Positionen, um die man das Muster nach rechts verschieben kann, bevor ein erneuter Vergleich zwischen Muster und Text, wieder beginnend mit dem letzten Zeichen im Muster, durchgeführt wird. In vielen Fällen ist es möglich, das Muster um große Distanzen nach rechts zu verschieben und so nur einen Bruchteil der Textzeichen zu inspizieren. Betrachten wir als Beispiel noch einmal den Text aus Abbildung 9.1. Wird das Muster an die erste Textposition angelegt, so wird zuerst das Textzeichen `s` mit dem letzten Zeichen des Musters verglichen. Es tritt ein Mismatch auf. Da das Textzeichen `s` im Muster überhaupt nicht vorkommt, kann man das Muster gleich um die Musterlänge, also um vier Positionen nach rechts verschieben. Falls das den Mismatch verursachende Zeichen doch im Muster auftritt, wie z.B. in folgender Situation

```
... abrakadabra ...
    aber
```

kann man das Muster so weit nach rechts schieben, bis erstmals das Textzeichen und das Zeichen im Muster übereinanderstehen. Die gesamte Folge der Vergleiche und Verschiebungen des Musters ist in Abbildung 9.3 dargestellt. Bis das Muster gefunden ist, werden nur insgesamt 17 Zeichen des Textes inspiziert.

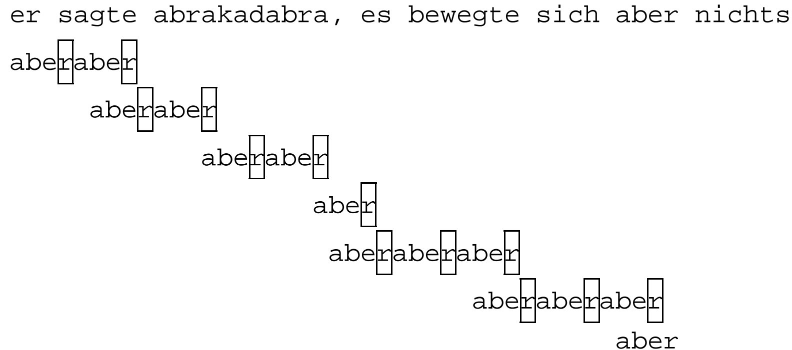
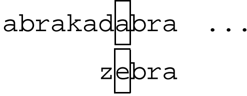


Abbildung 9.3

Das Beispiel in Abbildung 9.3 ist insofern durchaus typisch, als insbesondere bei kurzen Mustern die meisten Textzeichen im Muster überhaupt nicht vorkommen. In diesem Beispiel tritt darüberhinaus ein Mismatch stets bereits beim Vergleich des letzten Zeichens im Muster mit dem darüberstehenden Textzeichen auf. Das ist natürlich im allgemeinen nicht so, wie wir bereits gesehen haben und auch folgendes Beispiel nochmals zeigt.



In jedem Fall kann man aber eine mögliche Verschiebung des Musters nach rechts berechnen. Man kann diese Verschiebung *nur* davon abhängig machen, welches Zeichen *im Text* für den Mismatch verantwortlich war, und davon, ob dieses Zeichen und gegebenenfalls an welcher Position es im Muster auftritt. Diese Heuristik zur Berechnung der Verschiebung wird als *Vorkommens-Heuristik* bezeichnet. Das den Mismatch verursachende Zeichen c im Text bestimmt die Weite der Sprünge bei der Suche nach dem Muster $B = b_1 \dots b_M$ im Text $A = a_1 \dots a_N$.

Abhängig vom Muster und vom Alphabet wird eine *delta-1-Tabelle* erstellt, die für alle im Text eventuell vorkommenden Zeichen des Alphabets die mögliche Verschiebung des Musters nach rechts nach Auftreten eines durch das Zeichen c verursachten Mismatches enthält.

$$\mathit{delta}\text{-}1(c) = \begin{cases} M, & \text{falls } c \text{ in } b_1 \dots b_M \text{ nicht vorkommt} \\ M - j, & \text{falls } c = b_j \text{ und } c \neq b_k \\ & \text{für } j < k \leq M \end{cases}$$

Für die meisten Zeichen c des Alphabets ist $delta\text{-}1(c) = M$. Falls c im Muster $B = b_1 \dots b_M$ vorkommt, ist $delta\text{-}1(c)$ der Abstand des rechtesten Vorkommens von c in B vom Musterende.

Natürlich ist man eigentlich nicht an der Verschiebung des Musters, sondern an der möglichen Verschiebung des Textzeigers nach rechts interessiert. Ferner möchte man nach Auftreten eines Mismatches den Textzeiger auf jeden Fall über die Position hinaus nach rechts verschieben, an der man zuletzt begonnen hat, Zeichen in Muster und Text von rechts nach links zu vergleichen. In jedem Fall kann man die Verschiebung des Textzeigers aus dem *delta*-1-Wert berechnen. Sei c das den Mismatch verursachende Zeichen, und seien i und j die aktuellen Positionen im Text und im Muster.

Fall 1: $[M - j + 1 > delta\text{-}1(c)$, siehe Abbildung 9.4]

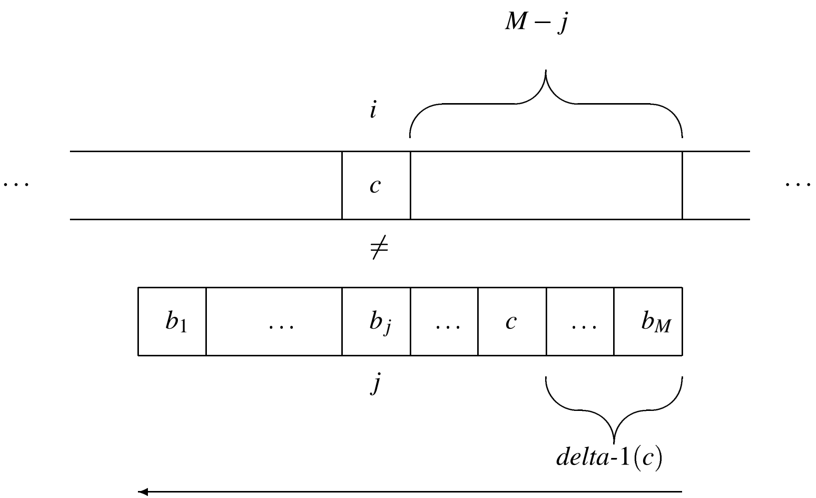


Abbildung 9.4

Wir setzen $i := i + M - j + 1$; $j := M$. Dies ist eine besonders einfache Version des Verfahrens von Boyer-Moore. Denn durch die Ersetzung $i := i + M - j + 1$ wird das Muster gegenüber seiner vorherigen Position ja nur um eine Position nach rechts verschoben erneut an den Text angelegt. Offensichtlich könnte man sich noch zusätzlich die Information zunutze machen, daß das den Mismatch verursachende Zeichen c rechts von dem im Muster auftretenden c an Position $delta\text{-}1(c)$ von rechts nicht vorkommt. Daher könnte man i sogar auf den größeren Wert $i + M - j + delta\text{-}1(c)$ setzen.

Fall 2: $[M - j + 1 \leq delta\text{-}1(c)$, siehe Abbildung 9.5]
Setze $i := i +\ delta\text{-}1(c)$; $j := M$.

Denken wir uns die *delta*-1-Tabelle gegeben, so kann eine dieser Vorkommens-Heuristik folgende, vereinfachte Version des Verfahrens von Boyer-Moore wie folgt beschrieben werden:

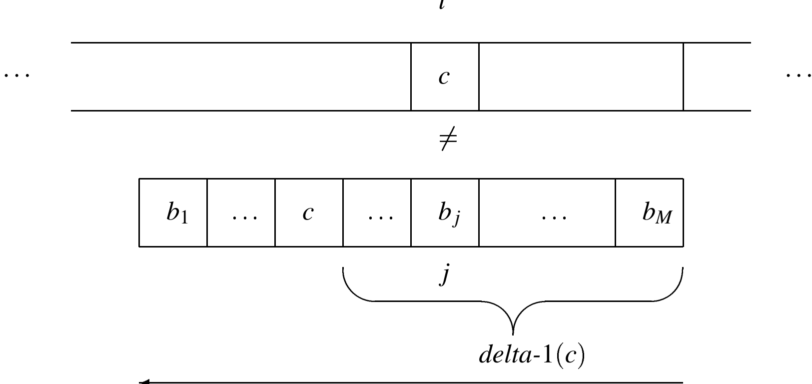


Abbildung 9.5

function *bmeinfach* $(a, b : string; M, N : integer) : integer$;
var $i, j : integer$;
begin
  $i := M$;
  $j := M$;
  repeat
    if $a_i = b_j$
      then
        begin
          $i := i - 1$;
          $j := j - 1$
        end
    else {*Mismatch verursacht durch a_i; Textzeiger entsprechend Fall 1 oder Fall 2 heraufsetzen; Musterzeiger an das Ende des Musters*}
      begin
        if $M - j + 1 > delta\text{-}1(a_i)$
          then $i := i + M - j + 1$
          else $i := i + delta\text{-}1(a_i)$;
        $j := M$
      end
  until $(j < 1)$ **or** $(i > N)$;
  $bmeinfach := i + 1$
end

Es ist leicht zu sehen, daß diese vereinfachte Version des Verfahrens von Boyer-Moore im schlechtesten Fall nicht besser ist als das naive Verfahren zur Textsuche, also $\Omega(NM)$ Schritte benötigt. (Man betrachte ein Muster $10\ldots0$ mit Länge M und durchsuche einen aus lauter Nullen bestehenden Text nach diesem Verfahren.)

Von Boyer und Moore wurde daher in [22] eine zweite Heuristik zur Berechnung der möglichen Verschiebung des Musters benutzt, die sogenannte *Match-Heuristik*. Ähnlich wie beim Verfahren von Knuth-Morris-Pratt nutzt diese Heuristik die Information über den bis zum Auftreten des Mismatch bereits inspizierten, mit einem Endstück des Musters übereinstimmenden Text. Betrachten wir dazu folgendes Beispiel:

```
Text:     orange ananas banana ...
Muster:          ban
```

Nehmen wir also an, daß die letzten m Zeichen im Muster mit den darüberstehenden m Zeichen im Text übereinstimmen und an der Position j der von rechts her erste Mismatch auftritt. Wir wollen die letzten m Zeichen das *Submuster* des Musters (für dieses m und j) nennen. Wir suchen dann von rechts her im Muster nach einem weiteren Vorkommen des Submusters. Haben wir ein solches Vorkommen gefunden, so können wir das Muster so weit nach rechts verschieben, daß das weitere Vorkommen des Submusters im Muster dem Vorkommen des Submusters im Text gegenübersteht. Diesem zweiten Vorkommen des Submusters im Muster darf natürlich nicht das gleiche Zeichen vorangehen wie dem ersten, denn sonst würde dieses Zeichen sicher wieder einen Mismatch verursachen.

Im oben angegebenen Beispiel kommt das Submuster `ana` im Muster `banana` noch einmal vor, und das dem zweiten Vorkommen vorangehende Zeichen `b` ist verschieden von dem Zeichen `n`, das dem ersten Vorkommen vorangeht:

```
banana
  banana
```

Wir können das Muster also um zwei Positionen nach rechts verschieben und fortfahren, von rechts her Zeichen im Muster mit Zeichen im Text zu vergleichen, beginnend mit dem letzten Zeichen im Muster.

Es bezeichne $wrw(j)$ die Position, an der das von rechts her nächste Vorkommen des Submusters beginnt. Dabei ist j die Position, an der der erste Mismatch auftrat, also $b_{j+1} \dots b_M$ das Submuster. Es wird angenommen, daß das dem zweiten Vorkommen des Submusters vorangehende Zeichen, also das Zeichen an Position $wrw(j) - 1$, vom Zeichen b_j verschieden ist.

Im obigen Beispiel ist $j = 3$, denn das dritte Zeichen `n` im Muster `banana` hat den Mismatch verursacht. $wrw(3) = 2$, denn das von rechts her nächste Vorkommen des Submusters `ana` beginnt an Position 2 im Muster.

Eine Funktion $delta\text{-}2(j)$ gibt an, um wieviele Positionen der Zeiger i auf das aktuelle Zeichen im Text nach rechts verschoben werden kann, wenn der erste Mismatch im Muster an Position j auftrat. Der Vergleich der Zeichen in Muster und Text beginnt nach jedem Mismatch jeweils neu mit dem letzten Zeichen des Musters. Es muß daher jeweils nur berechnet werden, um welche Distanz der Zeiger i im Text bewegt werden kann. Durch Verschieben nach rechts um $m = M - j$ Positionen wird der Zeiger i an das dem letzten Zeichen im Muster gegenüberliegende Zeichen im Text bewegt; das Muster kann jetzt um $j + 1 - wrw(j)$ Positionen nach rechts bewegt werden. Der Textzeiger muß noch um denselben Betrag erhöht werden. Insgesamt ergibt sich also, daß nach Auftreten eines Mismatches an Position j der Textzeiger um $delta\text{-}2(j)$ Positionen nach rechts bewegt werden kann, mit

$$delta\text{-}2(j) = M + 1 - wrw(j).$$

Wir berechnen die Werte von $wrw(j)$ und $delta\text{-}2(j)$ für $j = 5,4,3,2,1$ und das oben angegebene Beispiel des Musters `banana`. Sei zunächst $j = 5$; das an Position $j+1 = 6$ beginnende Submuster `a` tritt im Muster noch zwei weitere Male auf. Dem von rechts her nächsten `a` geht aber das gleiche Zeichen voran wie dem `a` an Position 6. Es ist daher $wrw(5) = 2$ und $delta\text{-}2(5) = 5$.

Sei nun $j = 4$; das an Position 5 beginnende Submuster `na` kommt noch einmal vor; beiden Vorkommen geht aber dasselbe Zeichen a voran. Innerhalb des Musters gibt es also überhaupt kein weiteres Vorkommen des Submusters mit der verlangten Eigenschaft. Denkt man sich aber das Muster nach links um „don't care"-Symbole fortgesetzt und setzt $wrw(4) = -1$, so ergibt sich für $delta\text{-}2(4)$ der Wert 8, also genau der Wert, um den man den Textzeiger nach rechts verschieben muß, wenn man das Muster um M Positionen nach rechts verschieben kann und an Position $j = 4$ ein Mismatch auftrat. Den Wert $wrw(3) = 2$ haben wir schon begründet. Damit ergibt sich $delta\text{-}2(3) = 5$. Sei schließlich $j = 2$. Das an Position 3 beginnende Submuster kommt im Muster nicht noch einmal vor. Es ist $wrw(j) = -3$:

Position j:		-4	-3	-2	-1	0	1	2	3	4	5	6
Muster:	...	*	*	*	*	*	b	a	n	a	n	a
Submuster:			n	a	n	a						

Damit ergibt sich $delta\text{-}2(2) = 10$. Auf ähnliche Weise erhält man $wrw(1) = -4$ und $delta\text{-}2(1) = 11$.

Offenbar hängt die *delta*-2-Tabelle nur vom Muster ab und kann ganz ähnlich berechnet werden wie im Verfahren von Knuth-Morris-Pratt, indem man das Muster gewissermaßen über sich selbst hinwegschiebt. Für jedes j, $1 \leq j < M$, enthält $delta\text{-}2(j)$ als Wert die Distanz, um die man den Textzeiger i nach rechts schieben muß, wenn beim Vergleich des Zeichens an Position i im Text ein Mismatch mit dem Zeichen an Position j im Muster aufgetreten ist.

Das Verfahren von Boyer und Moore in der ursprünglich angegebenen Version benutzt beide Heuristiken zur Berechnung der Verschiebung des Musters und folgt jeweils der, die den größeren Wert liefert. Denken wir uns also die *delta*-1-Tabelle und die *delta*-2-Tabelle gegeben, so kann man das Verfahren wie folgt formulieren:

```
function boyermoore (a, b : string; M, N : integer) : integer;
var i, j : integer;
begin
  i := M;
  j := M;
  repeat
    if a_i = b_j
      then
        begin
          i := i − 1;
          j := j − 1
        end
      else {Mismatch; Muster verschieben}
```

begin
$i := i + max\{delta\text{-}1(a_i) + 1,\ delta\text{-}2(j)\}$;
$j := M$
end
until $(j < 1)$ **or** $(i > N)$;
boyermoore $:= i + 1$
end

Man kann sich leicht überlegen, daß nach Auftreten eines Mismatches das Muster stets um wenigstens eine Position nach rechts verschoben wird, also der Textzeiger i um wenigstens $M - j + 1$ Positionen, wenn ein Mismatch an Position j im Muster auftrat. Die bei der vereinfachten Version des Verfahrens von Boyer-Moore gemachte Fallunterscheidung ist also jetzt entbehrlich.

Die verwendeten Tabellen *delta*-1 und *delta*-2 hängen nur vom Alphabet und vom gegebenen Muster ab. Wie in [81] gezeigt wurde, trägt die *delta*-2-Tabelle zur Schnelligkeit des Algorithmus in der Praxis kaum etwas bei. Der einzige Zweck dieser Tabelle ist es, Muster mit mehrfach auftretenden Submustern optimal zu nutzen und eine Laufzeit des Verfahrens von $\Theta(M \cdot N)$ im schlechtesten Fall zu verhindern. Weil Muster mit wiederholt auftretenden Submustern aber relativ selten vorkommen, insbesondere, wenn die Muster kurz sind, kann man auf die *delta*-2-Tabelle auch ganz verzichten.

Wir haben das Verfahren von Boyer-Moore so formuliert, daß der Algorithmus hält, wenn das erste Vorkommen des Musters im Text gefunden wurde. Die Laufzeit dieses Verfahrens beträgt $O(M + N)$. Natürlich ist es einfach, das Verfahren so zu verändern, daß es alle r Vorkommen des Musters im Text findet. Die Laufzeit beträgt dann $O(N + rM)$.

In der Praxis hat sich die vereinfachte Version des Verfahrens von Boyer-Moore ausgezeichnet bewährt. Man kann erwarten, daß das Verfahren für genügend kurze Muster und hinreichend große Alphabete etwa $O(N/M)$ Schritte durchführt, d.h. das Verfahren inspiziert nur jedes M-te Textzeichen und das Muster kann nahezu immer um die gesamte Musterlänge nach rechts verschoben werden.

9.1.4 Signaturen

Die von uns angegebenen Verfahren zur Suche in Texten benutzen als einzige Grundoperation den Vergleich von Zeichen im Muster und Zeichen im Text. Man kann Zeichen und Zeichenketten aber auch Zahlen zuordnen und Algorithmen entwerfen, die diese Zuordnung nutzen und arithmetische Operationen verwenden. Eine sehr einfache Möglichkeit besteht darin, *jedem* Teilstring des Textes mit Länge M durch eine Hashfunktion h eine Zahl zuzuordnen. Ist dann h so beschaffen, daß Adreßkollisionen sehr unwahrscheinlich sind, so hat man ein Vorkommen des Musters gefunden, wenn der Wert der Hashfunktion h für einen Teilstring mit Länge M gleich dem Wert $h(b_1 \ldots b_M)$ des Musters ist. Man berechnet also zu jedem Teilstring mit Länge M ein h-Bild als *Signatur* des Textes. Weil man nur einen einzigen h-Wert sucht, muß man die h-Werte, also die Hashtafel, natürlich nicht speichern. Attraktiv wird ein Verfahren zur Textsuche über die Berechnung von Signaturen natürlich erst dann, wenn die Berechnung der

Signatur einfach und zwar inkrementell möglich ist. D.h. der h-Wert von zwei aufeinanderfolgenden Teilstrings mit Länge M sollte sich wie folgt berechnen lassen:

$$\ldots \overline{a_{i+1}\underline{a_{i+2}\ldots a_{i+M}}}\underline{a_{i+M+1}}\ldots$$

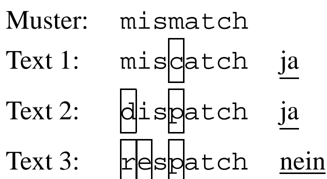
$h(a_{i+2}\ldots a_{i+M+1})$ ist eine einfache Funktion von $h(a_{i+1}\ldots a_{i+M})$. Ein Verfahren dieser Art wurde erstmals von Karp und Rabin angegeben [84]. Sie fassen eine Zeichenkette mit Länge M als d-adische Zahl auf, wobei d die Alphabetgröße ist, und benutzen als Hashfunktion die Funktion $h(k) = k \bmod p$ für eine geeignet gewählte, große Primzahl p. Man kann dann zeigen, daß das Verfahren von Karp und Rabin mit hoher Wahrscheinlichkeit nur $O(M+N)$ Schritte benötigt.

Gonnet und Baeza-Yates [11] haben Verfahren zur Textsuche angegeben, bei denen die Berechnung der Signatur nur noch vom gegebenen Muster abhängt. Ihre Verfahren lassen sich leicht über die reine Textsuche (exact matching) hinaus ausdehnen auf den Fall, daß auch „don't care"-Symbole, Komplementärsymbole (wie z.B. $\bar{c}$ zur Bezeichnung aller Zeichen, die von c verschieden sind) und mehrfache Muster in Suchanfragen vorkommen.

9.1.5 Approximative Zeichenkettensuche

Das Problem, in einem gegebenen Text alle Vorkommen eines gegebenen Musters zu finden, kann auf naheliegende Weise zum *k-Mismatch-Problem* verallgemeinert werden: Gegeben sind ein Text $a_1\ldots a_N$, ein Muster $b_1\ldots b_M$ und eine Zahl k, $0 \leq k < M$. Gesucht sind alle Vorkommen von Mustern $b'_1\ldots b'_M$ der Länge M im Text derart, daß sich $b_1\ldots b_M$ und $b'_1\ldots b'_M$ an höchstens k Positionen unterscheiden.

Für $k = 0$ ist dies das uns bereits bekannte Textsuchproblem, das wir mit Hilfe verschiedener, in den vorangehenden Abschnitten vorgestellter Algorithmen lösen können. Als Beispiel für den Fall $k = 2$ betrachten wir verschiedene Textstücke mit acht Buchstaben, die mit dem Muster `mismatch` verglichen werden. Ein Vergleich des jeweiligen Textstücks mit dem Muster führt zu einem positiven Ergebnis, wenn das Muster und das jeweilige Textstück an höchstens zwei Stellen verschiedene Buchstaben haben.



Das naive Verfahren zur Textsuche kann leicht auf diesen allgemeineren Fall ausgedehnt werden: Man legt das Muster der Reihe nach an jeder Position des Textes beginnend an, vergleicht zeichenweise von links nach rechts, ob eine Übereinstimmung zwischen Muster und Text vorliegt, und zählt die Anzahl der aufgetretenen Nichtübereinstimmungen (Mismatches). In Pascal-ähnlicher Notation kann das Verfahren so beschrieben werden:

```
procedure mismatch (a,b : string; N,M,k : integer);
{liefert alle Positionen im Text a[1 .. N], an denen ein Vorkommen des
 Musters b[1 .. M] mit höchstens k Mismatches beginnt}
var i, j, m : integer;
begin
  for i := 1 to N − M + 1 do
    begin
      m := 0;
      for j := 1 to M do
        if a_{i+j−1} ≠ b_j then m := m + 1;
      if m ≤ k
        then write(`höchstens', m, `Mismatches an Position', i)
    end
end
```

Es ist offensichtlich, daß das Verfahren Zeit $\Theta(M \cdot N)$ benötigt.

Wie im Falle der exakten Zeichenkettensuche, also wie für den Spezialfall des 0-Mismatch-Problems, kann man auch das k-Mismatch-Problem für $k > 0$ dadurch effizienter zu lösen versuchen, daß man etwa die Verfahren von Knuth-Morris-Pratt oder Boyer-Moore geeignet verallgemeinert. Überlegungen dazu findet man beispielsweise in [11].

Für Anwendungen bei Texteditoren oder bei der „Dekodierung" von DNA-Sequenzen in der Biologie viel wichtiger ist aber eine andere Verallgemeinerung des Textsuchproblems: Statt einfach die Anzahl der Buchstaben zu zählen, die verschieden sind, prüft man, wieviele Buchstaben eingefügt, gelöscht oder geändert werden müssen, um eine Übereinstimmung zwischen Text und Muster herzustellen. Das führt zum Begriff der *Editier-* (oder: *Evolutions-*)distanz und zu Algorithmen für die approximative Zeichenkettensuche, die auf dem algorithmischen Prinzip des *dynamischen Programmierens* beruhen. Das ist die Methode, immer größere optimale Teillösungen eines Problems iterativ „von unten nach oben" zu berechnen, d.h. angefangen bei optimalen Lösungen von „trivialen" Anfangsproblemen bis zur optimalen Gesamtlösung. Wir haben diese Methode bereits für die Konstruktion optimaler Suchbäume im Abschnitt 5.7 benutzt.

Editierdistanz

Wir wollen die folgenden *Editier-Operationen* zur Veränderung von Zeichenreihen zulassen: *Löschen*, *Einfügen* und *Ändern* eines einzelnen Symbols an einer bestimmten Stelle. Wir können diese Operationen als „Ersetzungsregeln" in der Form $\alpha \to \beta$ mitteilen, wobei α und β Buchstaben des zugrunde liegenden Alphabets Σ oder aber das Zeichen ε für das leere Wort sind. Die Veränderung einer Zeichenkette A durch eine Editier-Operation $\alpha \to \beta$ bedeutet dann, daß ein Vorkommen von α in A durch β ersetzt wird. Da das leere Wort ε „überall" in A vorkommt, heißt das insbesondere, daß eine Einfüge-Operation $\varepsilon \to a$ das Einfügen eines Zeichens a an jeder Position von A erlaubt.

Jeder Editier-Operation $\alpha \to \beta$ werden nichtnegative Kosten $c(\alpha \to \beta)$ zugeordnet. Man interessiert sich insbesondere für den Fall, daß die Kosten jeder Editier-Operation einheitlich gleich 1 gewählt werden (Einheitskosten-Modell). Im Einheitskosten-

Modell gilt also für zwei beliebige Zeichen a, $b \in \Sigma$, $a \neq b$: $c(a \to b) = c(\varepsilon \to b) = c(a \to \varepsilon) = 1$ und natürlich $c(a \to a) = 0$.

Sind nun zwei Zeichenketten $A = a_1 \dots a_m$ und $B = b_1 \dots b_n$ gegeben, so definieren wird als *Editierdistanz* $D(A,B)$ die minimalen Kosten, die eine Folge von Editier-Operationen hat, die A in B überführt.

Beispiel für eine Folge von Editier-Operationen, die `auto` in `rad` überführt:

`auto`	Operation $u \to \varepsilon$ an Position 2 liefert
`ato`	Operation $t \to d$ an Position 2 liefert
`ado`	Operation $o \to \varepsilon$ an Position 3 liefert
`ad`	Operation $\varepsilon \to r$ an Position 0 liefert
`rad`	

Im Einheitskosten-Modell hat diese Folge von Editier-Operationen die Kosten 4. Es ist nicht schwer zu sehen, daß es keine Folge von Editier-Operationen mit geringeren Kosten gibt, die `auto` in `rad` überführt. Daher ist $D(\texttt{auto}, \texttt{rad}) = 4$.

Es ist üblich anzunehmen, daß ein durch eine Editier-Operation einmal eingefügtes, gelöschtes oder geändertes Zeichen nicht nochmals verändert, also gelöscht, eingefügt oder geändert wird. Diese Annahme gilt für die Folge der Editier-Operationen mit minimal möglichen Kosten sicher dann, wenn die Kostenfunktion für die Editier-Operationen eine „Dreiecksungleichung“ erfüllt, d.h. wenn gilt

$$c(\alpha \to \gamma) \leq c(\alpha \to \beta) + c(\beta \to \gamma),$$

falls $\alpha \neq \beta \neq \gamma$, und $c(\alpha \to \beta) > 0$, falls $\alpha \neq \beta$. Das ist insbesondere im Einheitskosten-Modell erfüllt.

Zwei Probleme sind im Zusammenhang mit Editierdistanzen von besonderem Interesse:

Problem 1: (Berechnung der Editierdistanz)
Berechne für zwei gegebene Zeichenketten A und B möglichst effizient die Editierdistanz $D(A,B)$ und eine kostenminimale Folge von Editier-Operationen, die A in B überführt.

Problem 2: (Approximative Zeichenkettensuche)
Gegeben seien ein Text A und ein Muster B sowie eine Zahl $k \geq 0$. Gesucht sind alle Vorkommen von Zeichenreihen B' in A, so daß $D(B,B') \leq k$ ist.

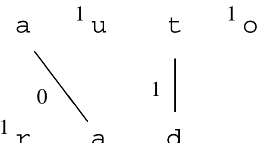
Für $k = 0$ ist Problem 2 natürlich wieder das gewöhnliche Zeichenketten-Suchproblem. Das k-Mismatch-Problem kann als Spezialfall von Problem 2 aufgefaßt werden, wenn man nur Änderungen von Zeichen, also weder Einfügen noch Löschen von Zeichen zuläßt.

Wir behandeln zunächst Verfahren zur Lösung von Problem 1 und werden dann sehen, daß dabei verwendete Methoden auch zur Lösung von Problem 2 benutzt werden können. Dabei setzen wird zur Vereinfachung stets das Einheitskosten-Modell voraus und überlassen es dem Leser, sich zu überlegen, wie Verfahren auf den Fall unterschiedlicher Kosten ausgedehnt werden können.

Berechnung der Editierdistanz

Eine Folge von Editier-Operationen mit minimalen Kosten, die eine Zeichenreihe A in eine andere Zeichenreihe B überführt, ändert jedes von einer Operation betroffene Zeichen höchstens einmal. Wir können uns daher auch vorstellen, daß die Operationen nicht nacheinander, sondern alle gleichzeitig ausgeführt werden. Das führt zum Begriff der *Spur* (englisch: *trace*), die A in B transformiert. Wir verzichten auf eine formal exakte Definition dieses Begriffs und verweisen dazu auf [186, 187]. Stattdessen teilen wir Spuren in folgender Weise graphisch mit: Wird auf ein Zeichen a in A eine Änderungsoperation $a \to b$ ausgeführt, so verbinden wir das Zeichen a in A (an der Position, an der diese Operation ausgeführt wird) mit dem entsprechenden Zeichen b in B durch eine Kante; die Kante wird mit 1 beschriftet, wenn $a \neq b$ ist, und mit 0 sonst. Ein Zeichen a in A, auf das eine Lösch-Operation $a \to \varepsilon$ angewandt wird, erhält einen linken oberen Index 1; ein Zeichen b in B, das durch eine Einfüge-Operation $\varepsilon \to b$ entstanden ist, erhält einen linken oberen Index 1. Die Summe der Indizes und Kantenbeschriftungen sind die Kosten der Spur.

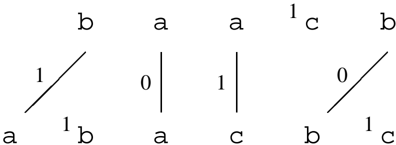
Beispiel: Die oben angegebene Folge von vier Editier-Operationen, die $A = \texttt{auto}$ in $B = \texttt{rad}$ transformiert, kann zur folgenden Spur zusammengefaßt werden:



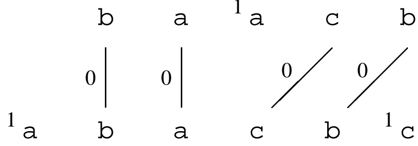
Aus der Annahme, daß zur Transformation von A in B jedes Zeichen höchstens einmal geändert werden darf, folgt, daß eine Spur keine sich kreuzenden Kanten enthalten kann. Statt alle Folgen von Editier-Operationen zu betrachten genügt es also, alle Spuren ohne sich kreuzende Kanten zu betrachten. Die Editierdistanz $D(A,B)$ ist gleich den Kosten einer *optimalen* Spur, also einer Spur mit minimalen Kosten, die A in B transformiert. Aus einer Spur kann man leicht eine Folge von Editier-Operationen ablesen, die A in B transformiert und die genau die Kosten der Spur hat.

Beispiel: Seien $A = \texttt{baacb}$ und $B = \texttt{abacbc}$.

Dann ist

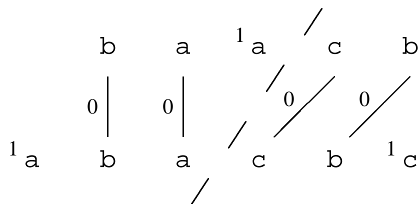


eine Spur mit Kosten 5. Das ist keine optimale Spur. Eine optimale Spur mit den Kosten 3 ist:



Aus dieser Spur kann man ablesen, daß Löschen von `a` an der Position 3 in A, dann Einfügen von `a` am Anfang und Einfügen von `c` am Ende A in B transformiert. Offenbar kann man aus jeder Spur, die A in B transformiert, auch sofort eine Spur erhalten, die umgekehrt B in A transformiert und dieselben Kosten hat. Man muß dazu nur alle Operationen, die A in B transformieren, umkehren. Daher ist klar, daß (im Einheitskosten-Modell) $D(A,B) = D(B,A)$ gilt.

Offenbar kann man Spuren in der Regel auf vielfältige Art teilen, so daß die Teile selbst wieder Spuren zur Transformation kürzerer Zeichenreihen sind. Beispielsweise kann man die (optimale) Spur



entlang der gestrichelten Linie teilen und erhält zwei Spuren, die `baa` in `aba` und `cb` in `cbc` transformieren.

Der Schlüssel zur Berechnung einer optimalen Spur nach der Methode der dynamischen Programmierung besteht nun in der Beobachtung, daß jede durch Teilung einer optimalen Spur entstandene Spur selbst wieder optimal sein muß. Denn wäre das nicht der Fall, dann könnte man durch Ersetzen eines nicht optimalen Teils einer optimalen Spur durch einen Teil mit geringeren Kosten die Gesamtkosten verringern, so daß die ursprünglich gegebene Spur nicht optimal gewesen sein kann. Daher kann man immer „längere“ optimale Spuren nach der Methode des dynamischen Programmierens aus „kürzeren“ berechnen. Genauer besteht das Verfahren zur Berechnung der Editierdistanz $D(A,B)$, also der Kosten einer optimalen Spur zur Transformation einer Zeichenreihe $A = a_1 \dots a_m$ in eine Zeichenreihe $B = b_1 \dots b_n$, darin, für jedes Paar (i, j) mit $0 \leq i \leq m$ und $0 \leq j \leq n$ die Kosten $D_{i,j}$ einer optimalen Spur zu berechnen, die $a_1 \dots a_i$ in $b_1 \dots b_j$ transformiert. Wir berechnen also für alle i, j mit $0 \leq i \leq m$ und $0 \leq j \leq n$

$$D_{i,j} = D(a_1 \dots a_i, b_1 \dots b_j).$$

Dabei ist das erste Argument von D das leere Wort, falls $i = 0$, und das zweite Argument von D das leere Wort, falls $j = 0$. Dann ist offenbar die gesuchte Editierdistanz $D(A,B) = D_{m,n}$.

Zunächst gilt offensichtlich

$$\begin{aligned} D_{0,0} &= D(\varepsilon,\varepsilon) = 0, \\ D_{0,j} &= D(\varepsilon, b_1 \dots b_j) = j, \text{ für } 1 \leq j \leq n, \end{aligned}$$

da genau j Einfüge-Operationen in die (anfangs) leere Zeichenreihe erforderlich sind, um $b_1 \dots b_j$ zu erzeugen. Ferner ist

$$D_{i,0} = D(a_1 \dots a_i, \varepsilon) = i, \text{ für } 1 \leq i \leq m,$$

da genau i Lösch-Operationen nötig sind, um aus $a_1 \dots a_i$ das leere Wort zu erzeugen.

Nun überlegen wir uns, wie wir für $i \geq 1$ und $j \geq 1$ den Wert $D_{i,j}$ aus $D_{i-1,j}$, $D_{i,j-1}$ und $D_{i-1,j-1}$ berechnen können. Dazu betrachten wir eine optimale Spur, die $a_1 \dots a_i$ in $b_1 \dots b_j$ transformiert. Am rechten Ende dieser Spur liegt dann einer der folgenden drei Fälle vor.

Fall 1: [Ändern: a_i und b_j sind durch eine Kante miteinander verbunden, die mit 1 beschriftet ist, falls $a_i \neq b_j$ ist, und mit 0, falls $a_i = b_j$ ist]
Läßt man diese Kante weg, so erhält man eine Spur mit minimalen Kosten $D_{i-1,j-1}$, die $a_1 \dots a_{i-1}$ in $b_1 \dots b_{j-1}$ transformiert:

$$\begin{array}{ccc} \cdots\cdots\ a_{i-1} & | & a_i \\ & | & | \\ \underbrace{\cdots\cdots\ b_{j-1}}_{\text{Spur mit Kosten } D_{i-1,j-1}} & | & b_j \end{array}$$

Für die Kosten $D_{i,j}$ gilt in diesem Fall

$$D_{i,j} = D_{i-1,j-1} + \begin{cases} 1; & \text{falls } a_i \neq b_j \\ 0; & \text{falls } a_i = b_j \end{cases}$$

Fall 2: [Einfügen: b_j ist nicht durch eine Kante mit einem Zeichen aus A verbunden]
Läßt man b_j weg, so erhält man eine Spur mit minimalen Kosten $D_{i,j-1}$, die $a_1 \dots a_i$ in $b_1 \dots b_{j-1}$ transformiert:

$$\begin{array}{ccc} \cdots\ a_{i-1}\ \ a_i & | & \\ & | & \\ \underbrace{\cdots\cdots\ b_{j-1}}_{\text{Spur mit Kosten } D_{i,j-1}} & | & {}^1b_j \end{array}$$

Für die Kosten $D_{i,j}$ gilt in diesem Fall

$$D_{i,j} = D_{i,j-1} + 1.$$

Fall 3: [Löschen: a_i ist nicht durch eine Kante mit einem Zeichen aus B verbunden]
Läßt man a_i weg, so erhält man eine Spur mit minimalen Kosten $D_{i-1,j}$, die $a_1 \dots a_{i-1}$ in $b_1 \dots b_j$ transformiert:

$$\begin{array}{ccc} \cdots\cdots\ a_{i-1} & | & {}^1a_i \\ & | & \\ \underbrace{\cdots\ b_{j-1}\ \ b_j}_{\text{Spur mit Kosten } D_{i-1,j}} & | & \end{array}$$

Für die Kosten $D_{i,j}$ gilt in diesem Fall

$$D_{i,j} = D_{i-1,j} + 1.$$

Wir überlegen uns noch, daß dies alle zu betrachtenden Fälle sind. Weil eine Spur kreuzungsfrei ist, kann es nicht vorkommen, daß a_i und b_j auf zwei verschiedenen Kanten liegen. Schließlich liegt wegen der Optimalität der Spur, die $a_1 \ldots a_i$ in $b_1 \ldots b_j$ transformiert, wenigstens eines der beiden Zeichen a_i und b_j auf einer Kante (andernfalls wäre ein Kante von a_i nach b_j billiger). Damit ist unsere Fallunterscheidung vollständig und eindeutig.

Wir erhalten insgesamt also die folgende Rekursionsformel für die gesuchten Werte $D_{i,j}$, $0 \leq i \leq m$, $0 \leq j \leq n$:

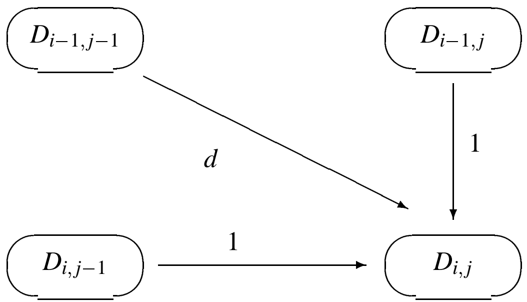
$$\begin{array}{lcll} D_{0,0} & = & 0 & \\ D_{0,j} & = & j, & \text{für } 1 \leq j \leq m, \\ D_{i,0} & = & i, & \text{für } 1 \leq i \leq n, \end{array}$$

und für $0 < i \leq m$ und $0 < j \leq m$:

$$D_{i,j} = \min\{\ D_{i-1,j-1} + \left\{ \begin{array}{ll} 1; & \text{falls } a_i \neq b_j \\ 0; & \text{falls } a_i = b_j \end{array} \right. ,\quad D_{i,j-1}+1,\ D_{i-1,j}+1\ \}$$

Diese Darstellung zeigt, daß man die Werte $D_{i,j}$ z.B. zeilenweise oder spaltenweise und daher in Zeit $O(m \cdot n)$ und Platz $O(m)$ oder $O(n)$ berechnen kann. Die Editierdistanz $D(A,B) = D(a_1 \ldots a_m, b_1 \ldots b_n) = D_{m,n}$ kann man dann in der rechten unteren Ecke der Matrix $(D_{i,j})$ ablesen.

Man kann sich eine vollständige Übersicht über alle möglichen Spuren, die A in B transformieren, und über alle möglichen Wege zur Berechnung der $(m+1) \cdot (n+1)$ Werte $D_{i,j}$ mit Hilfe der angegebenen Rekursionsformel verschaffen. Dazu ordnet man jedem Paar (i, j) mit $0 \leq i \leq m$ und $0 \leq j \leq n$ einen (mit dem Wert $D_{i,j}$ beschrifteten) Knoten eines (planaren) Graphen zu; die Knoten werden in Form einer Matrix mit $m+1$ Zeilen und $n+1$ Spalten angeordnet. Um nicht zu viele Bezeichnungen einführen zu müssen, bezeichnen wir auch den Knoten an Position (i, j) mit $D_{i,j}$. Es ist aus dem Kontext stets eindeutig zu entnehmen, ob $D_{i,j}$ den Knoten an der Position (i, j) oder dessen Wert bezeichnet. Der Knoten in der linken oberen Ecke erhält den Wert 0. Die Knoten in der 0-ten Zeile sind jeweils durch eine waagerechte, mit 1 beschriftete Kante miteinander verbunden. Jede Kante repräsentiert eine Einfüge-Operation, die ausgehend vom anfangs leeren Wort das jeweils nächste Zeichen von B liefert. Daher erhalten die Knoten der 0-ten Zeile auch der Reihe nach die Werte $1, 2, \ldots, n$. Entsprechend sind die Knoten der 0-ten Spalte jeweils durch eine senkrechte, mit 1 beschriftete Kante miteinander verbunden. Jede Kante repräsentiert eine Lösch-Operation, die das jeweils nächste Zeichen von A löscht. Daher erhalten die Knoten der 0-ten Spalte der Reihe nach die Werte $1, 2, \ldots, m$. Alle anderen Knoten werden nach folgendem Schema durch mit 0 oder 1 beschriftete Kanten miteinander verbunden:



Die Diagonalkante ist mit $d = 1$ beschriftet, falls $a_i \neq b_j$, und mit $d = 0$, falls $a_i = b_j$ ist. Sie repräsentiert also eine Änderungsoperation $a_i \to b_j$. Entsprechend repräsentiert die horizontale Kante eine Einfüge-Operation $\varepsilon \to b_j$ und die senkrechte Kante eine Lösch-0peration $a_i \to \varepsilon$. Abbildung 9.6 zeigt als Beispiel einen Graphen für $A =$ `baac` und $B =$ `abac`.

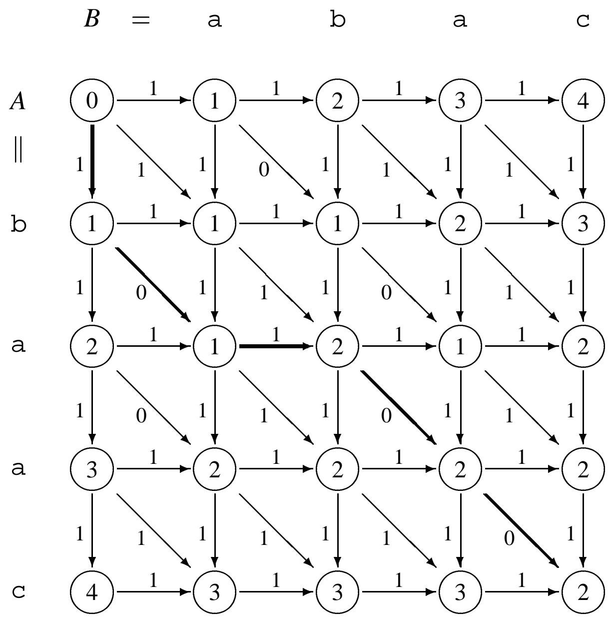
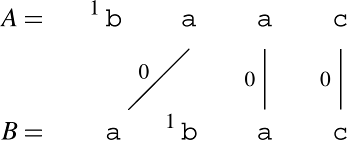


Abbildung 9.6

Man beachte, daß für $0 < i \leq m$ und $0 < j \leq n$ $D_{i,j}$ das Minimum der Werte $D_{i,j-1} + 1$, $D_{i-1,j} + 1$ und $D_{i-1,j-1} + d$ ist. Jedem Weg von der linken oberen zur rechten unteren Ecke entspricht eine Spur, die A in B transformiert. Umgekehrt entspricht auch jeder Spur ein solcher Weg. Wir nennen den resultierenden Graphen daher auch *Spurgraphen.*

Beispielsweise entspricht dem fett gezeichneten Weg des Spurgraphen in Abbildung 9.6 die folgende Spur:



Falls es sich, wie in diesem Beispiel, um eine optimale Spur handelt, sind die Werte der Knoten längs eines solchen Weges jeweils genau die Summen der Kantenbeschriftungen. Genau die Wege mit dieser Eigenschaft repräsentieren daher die optimalen Spuren und sämtliche Möglichkeiten zur Berechnung von $D_{m,n} = D(A,B)$.

In jedem Fall sind die Kosten einer Spur die Summe der Kantenbeschriftungen des die Spur repräsentierenden Weges im Spurgraphen.

Betrachten wir jetzt das Problem, für eine gegebene Zahl s festzustellen, ob $D_{m,n} \leq s$ ist. Natürlich kann man dieses Problem lösen, indem man alle $(m+1) \cdot (n+1)$ Werte $D_{i,j}$ im Spurgraphen berechnet und nachsieht, ob $D_{m,n} \leq s$ ist. Das ist aber nicht immer nötig. Denn jede horizontale und jede vertikale Kante eines eine Spur repräsentierenden Weges im Spurgraphen liefert den Beitrag 1 zu den Kosten der Spur. Die Gesamtkosten können also höchstens dann unterhalb der vorgegebenen Schranke s bleiben, wenn der Weg höchstens s horizontale und vertikale Kanten insgesamt enthält. Weil die Zahl der horizontalen und vertikalen Kanten, die man mindestens durchlaufen muß, um vom Knoten $D_{0,0}$ im Spurgraphen zum Knoten $D_{i,j}$ zu gelangen, gleich $|i-j|$ ist, folgt: Sobald $|i-j| > s$ ist, kann der Knoten $D_{i,j}$ auf keinem Weg von $D_{0,0}$ zu $D_{m,n}$ liegen, dessen Kosten $\leq s$ bleiben. Zur Prüfung, ob $D_{m,n} \leq s$ ist, genügt es also, alle $D_{i,j}$ zu berechnen, für die $|i-j| \leq s$ bleibt. Sie liegen auf einem Streifen links und rechts von der Diagonalen durch $D_{0,0}$, vgl. Abbildung 9.7.

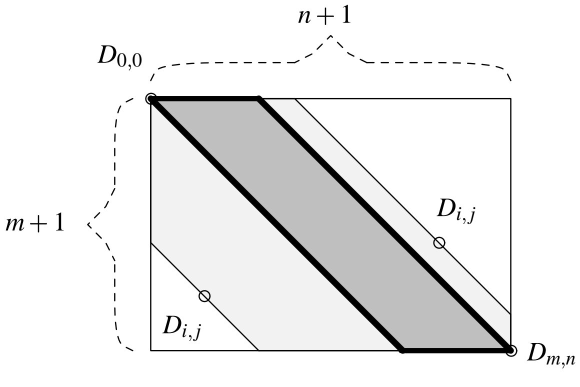


Abbildung 9.7

Insbesondere folgt natürlich, daß $D_{m,n} \leq s$ nur möglich ist, wenn $|m-n| \leq s$ ist. Wie groß kann der Wert $D_{m,n}$ höchstens sein? Offenbar nicht länger als die Länge eines Weges von $D_{0,0}$ nach $D_{m,n}$ mit minimaler Kantenzahl. Nehmen wir (wie

in Abbildung 9.7 geschehen) ohne Einschränkung an, daß $n \geq m$ ist, so haben alle Wege mit $n-m$ horizontalen und m Diagonalkanten die minimal mögliche Kantenzahl. Sie verlaufen im dunkel schraffierten Bereich von Abbildung 9.7. Es ist also $D_{m,n} = D(A,B) \leq m + (n-m) = n$. Diese Beobachtung entspricht natürlich beispielsweise der Möglichkeit, A in B dadurch zu transformieren, daß man die ersten m Buchstaben von B durch Ändern der m Buchstaben von A erzeugt und anschließend die noch fehlenden $n-m$ Buchstaben von B durch Einfüge-Operationen erzeugt.

Falls $s < n-m$ ist, gibt es sicher keinen Weg im Spurgraphen, der $D_{0,0}$ mit $D_{m,n}$ verbindet und Kosten $\leq s$ hat, weil man auf jeden Fall wenigstens $n-m$ horizontale Kanten durchlaufen muß, um von $D_{0,0}$ nach $D_{m,n}$ zu gelangen. Sonst genügt es, die $n-m+1$ Diagonalen der Länge m im stark schraffierten Bereich von Abbildung 9.7 auszuwerten und je $1/2(s-(n-m))$ kürzere Diagonalen unterhalb der Diagonalen durch $D_{0,0}$ und oberhalb der Diagonalen durch $D_{m,n}$. Denn nur Wege in diesem Diagonalband können Spuren entsprechen mit Gesamtkosten, die s nicht übersteigen. Der Aufwand zur Berechnung der Werte $D_{i,j}$ in diesem Bereich kann daher nach oben abgeschätzt werden durch

$$\begin{aligned}&(n-m+1)\cdot m + (s-(n-m))(m-1)\\ &= sm - s + n \leq sm - (n-m) + n = O(s\cdot m)\end{aligned}$$

In [187] ist gezeigt, daß man mit Platz $O(min(s,m,n))$ auskommt, um diese Rechnung durchzuführen.

Man erhält so insgesamt:

Satz 9.1 *Für zwei Zeichenreihen $A = a_1 \ldots a_m$ und $B = b_1 \ldots b_n$ mit $m \leq n$ und eine gegebene Zahl s kann man in Zeit $O(s \cdot m)$ und Platz $O(min(s,m))$ feststellen, ob $D(A,B) \leq s$ ist.*

Die besonders regelmäßige Struktur des Spurgraphen läßt noch weitere Verbesserungen, d.h. eine weitere Reduzierung des Zeit- und Platzbedarfs zur Berechnung der Editierdistanz zu. Dazu vergleiche man z.B. [186, 187].

Approximative Zeichenkettensuche

Um in einem gegebenen Text $A = a_1 \ldots a_n$ für ein gegebenes $k \geq 0$ und ein Muster $B = b_1 \ldots b_m$ alle Vorkommen von Zeichenreihen B' in A zu finden, für die $D(B,B') \leq k$ ist, kann man natürlich wie folgt vorgehen: Man betrachtet für jedes Paar (j,j') mit $1 \leq j \leq j' \leq n$ das Teilstück $a_j a_{j+1} \ldots a_{j'}$ von A und berechnet die Editierdistanz $D(a_j a_{j+1} \ldots a_{j'}, B)$. Falls sie kleiner oder gleich k ist, hat man ein approximatives Vorkommen von B in A gefunden.

Wieviele Schritte benötigt dies *naive Verfahren* zur approximativen Zeichenkettensuche? Da es $n(n-1)/2$ Teilstücke $a_j a_{j+1} \ldots a_{j'}$ von A gibt, die betrachtet werden, und die Prüfung, ob für die Editierdistanz $D(a_j a_{j+1} \ldots a_{j'}, B) \leq k$ gilt, nach Satz 9.1 in Zeit $O(k \cdot \min(j'-j,m))$ durchgeführt werden kann, folgt: Das naive Verfahren findet alle approximativen Vorkommen von B in A in Zeit $O(n(n-1)/2 \cdot k \cdot \min(j'-j,m)) = O(n^2 \cdot k \cdot m)$. Das ist wenig praktikabel, weil im allgemeinen n sehr groß im Vergleich zu m und k ist.

Um zu effizienteren Verfahren für die approximative Zeichenkettensuche zu kommen, ist es zunächst vernünftig, die Problemstellung leicht zu verändern. Anstatt alle Paare (j, j') von Indizes mit $1 \leq j \leq j' \leq n$ zu finden, für die $D(a_j a_{j+1} \ldots a_{j'}, B) \leq k$ ist, bestimmt man für jede Stelle j im Text A ein *ähnlichstes,* bei j endendes Teilstück von A. Das ist ein Teilstück von A, das an der Position j endet und die minimal mögliche Editierdistanz zum Muster B hat. Wir lösen also das folgende Problem $2'$ anstelle des oben formulierten Problems 2:

Problem $2'$: (Bestimmung ähnlichster Teile)
Gegeben sind ein Text $A = a_1 \ldots a_n$ und ein Muster $B = b_1 \ldots b_m$. Gesucht ist für jedes j, $1 \leq j \leq n$, ein j' mit $1 \leq j' \leq j$, so daß für jedes j'' mit $1 \leq j'' \leq j$ gilt: $D(a_{j'} \ldots a_j, B) \leq D(a_{j''} \ldots a_j, B)$. (Das Teilstück $a_{j'} \ldots a_j$ von A ist also ein zu B ähnlichstes Teilstück, das an Position j endet.)

Wir werden Problem $2'$ so lösen, daß wir zu jeder Textstelle nicht nur ein dort endendes, dem Muster möglichst ähnliches Teilstück finden, sondern auch die Editierdistanz zwischen diesem Teilstück und dem Muster B bestimmen. Daher können wir eine Lösung von Problem $2'$ auch als eine Lösung von Problem 2 auffassen: Für jede Textstelle j können wir feststellen, ob es überhaupt ein an der Stelle j endendes Teilstück gibt, das eine Editierdistanz von höchstens k zum Muster B hat; und wenn das der Fall ist, kennen wir ein dort endendes, zu B ähnlichstes Stück von A. Die übrigen Teilstücke von A mit Editierdistanz kleiner oder gleich k zu B lassen sich daraus durch Verlängern oder Verkürzen gewinnen.

Bestimmung ähnlichster Teile

Wir werden uns jetzt überlegen, daß das Problem $2'$ auf ganz ähnliche Weise gelöst werden kann wie das Problem 1, nämlich durch sukzessive Berechnung aller Werte einer $(m+1) \cdot (n+1)$-Matrix wie folgt:

$$\begin{array}{lcll} D_{0,j} & = & 0, & \text{für } 0 \leq j \leq n, \\ D_{i,0} & = & i, & \text{für } 0 \leq i \leq m, \end{array}$$

und für $0 < i \leq m$, $0 < j \leq m$

$$D_{i,j} = \min\{\, D_{i-1,j-1} + \begin{cases} 1; & \text{falls } a_i \neq b_j \\ 0; & \text{falls } a_i = b_j \end{cases},\quad D_{i,j-1} + 1,\ D_{i-1,j} + 1 \,\}.$$

Diese Matrix unterscheidet sich also von der Matrix zur Berechnung der Editierdistanz zwischen A und B nur durch die Initialisierung der 0-ten Zeile: Dort treten ausschließlich Nullen auf.

In Analogie zum Spurgraphen können wir alle Werte $D_{i,j}$ in einem *Abhängigkeitsgraphen* veranschaulichen. Das ist ein Graph mit $(m+1) \cdot (n+1)$ Knoten. Darin ist der Knoten $D_{i,j}$ mit $D_{i-1,j}$, $D_{i,j-1}$ oder $D_{i-1,j-1}$ durch eine Kante verbunden, wenn der Wert $D_{i,j}$ unter Rückgriff auf diese Werte erhalten werden kann. Genauer gilt für $i > 0$ und $j > 0$: Es gibt eine Kante zwischen $D_{i-1,j}$ und $D_{i,j}$, wenn $D_{i,j} = D_{i-1,j} + 1$ ist. Ferner gibt es eine Kante zwischen $D_{i,j-1}$ und $D_{i,j}$, wenn $D_{i,j} = D_{i,j-1} + 1$ ist; und schließlich gibt es eine Kante zwischen $D_{i-1,j-1}$ und $D_{i,j}$, wenn $D_{i,j} = D_{i-1,j-1}$ und $a_i = b_j$ ist oder wenn $D_{i,j} = D_{i-1,j-1} + 1$ und $a_i \neq b_j$ ist.

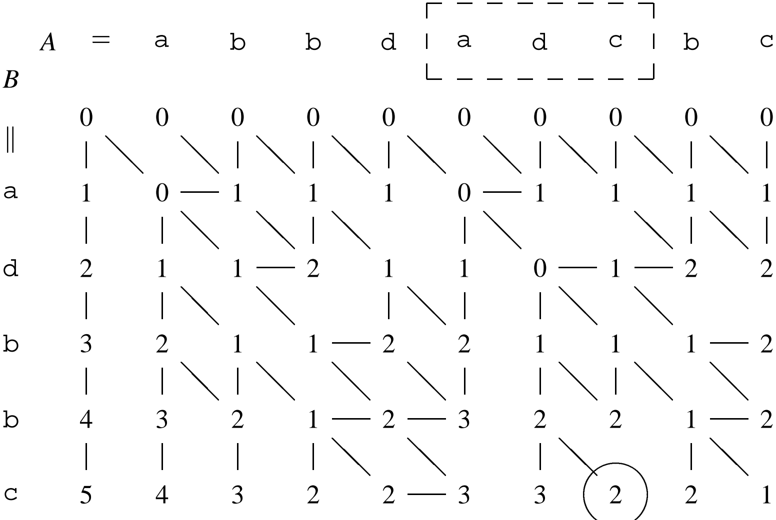


Abbildung 9.8

Abbildung 9.8 zeigt als Beispiel den Abhängigkeitsgraphen für den Text $A =$ abbdadcbc und das Muster $B =$ adbbc.

Ähnlich wie für die optimalen Wege im Spurgraphen gilt für *jeden* Weg im Abhängigkeitsgraphen, daß die Werte längs eines jeden Weges von links oben nach rechts unten nur zunehmen. Die Wege im Abhängigkeitsgraphen entsprechen optimalen Spuren in folgendem Sinne: Gibt es im Abhängigkeitsgraphen einen Weg von $D_{0,j'-1}$ nach $D_{i,j}$, so ist $a_{j'} \ldots a_j$ ein zu $b_1 \ldots b_i$ ähnlichstes, bei j endendes Teilstück von A mit $D(b_1 \ldots b_i, a_{j'} \ldots a_j) = D_{i,j}$. Das kann man leicht durch Induktion beweisen, weil jeder Weg zum Knoten $D_{i,j}$ im Abhängigkeitsgraphen über einen der Knoten $D_{i-1,j}$, $D_{i,j-1}$ oder $D_{i-1,j-1}$ führen muß. Man findet also ein zu $B = b_1 \ldots b_m$ ähnlichstes Teilstück von A, das bei Position j endet, wenn man einen Weg von $D_{m,j}$ zur Zeile 0 zurückverfolgt: Ist $D_{0,j'-1}$ durch einen (nach rechts und unten gerichteten) Weg mit $D_{m,j}$ verbunden, so ist $a_{j'} \ldots a_j$ ein gesuchtes Teilstück, vgl. Abbildung 9.9.

Der Wert $D_{m,j}$ gibt die Editierdistanz des bei j endenden, zu B ähnlichsten Teils von A an. So entnimmt man beispielsweise der Abbildung 9.8, daß adc ein an Position 7 endendes, zum Muster B ähnlichstes Teilstück von A ist, das die Editierdistanz 2 zu B hat.

Alle Stellen j in der letzten Zeile, an denen Werte $D_{m,j} \leq k$ auftreten, sind also Stellen, an denen Teile von A mit Editierdistanz höchstens k zum Muster B enden können. Insgesamt erhalten wir damit:

Satz 9.2 *Für einen Text $A = a_1 \ldots a_n$ und ein Muster $B = b_1 \ldots b_m$ kann man in Zeit und Platz $O(m \cdot n)$ zu jeder Stelle j, $1 \leq j \leq n$, im Text ein zu B ähnlichstes, bei j endendes Teilstück von A finden.*

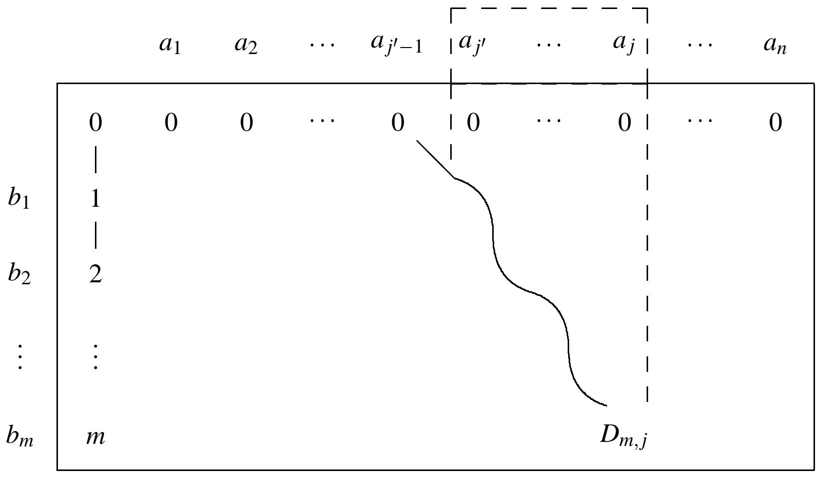


Abbildung 9.9

Dieses Ergebnis wurde von Sellers in [165] bewiesen. Es wurde später in vielfältiger Weise verbessert. Ein Ziel ist dabei, Algorithmen zu entwickeln, die in zwei Phasen arbeiten, einer ersten, nur von m und evtl. der Alphabetgröße abhängigen Aufbereitungsphase für das Muster und einer zweiten, dann nur noch von n abhängigen Textinspektionsphase. Daß das prinzipiell möglich ist, zeigt folgende Überlegung: Man kann die Spalten des Abhängigkeitsgraphen als Zustände eines (allerdings sehr großen) endlichen Automaten auffassen. Man berechnet dann zunächst alle möglichen Zustandsübergänge voraus, d.h. zu jedem möglichen Zustand und jedem Zeichen des zugrundeliegenden Alphabets berechnet man den Folgezustand. Dann inspiziert man den Text mit diesem endlichen Automaten. Diese und andere Verbesserungen des oben angegebenen Verfahrens von Sellers findet man in [186] und in der Übersicht in [68].

9.2 Parallele Algorithmen

Wir sind bisher stets davon ausgegangen, daß die Instruktionen von Programmen durch einen einzigen Prozessor sequentiell nacheinander ausgeführt werden. Als Modell eines solchen, nach dem Von-Neumann-Prinzip aufgebauten Rechners haben wir im Abschnitt 1.1 die Random-Access-Maschine (RAM) eingeführt. Eine Beschleunigung von Algorithmen für Rechner dieses Typs kann nur dadurch erfolgen, daß man die Arbeitsgeschwindigkeiten der einzelnen Systemkomponenten (Prozessor, Speicher, Datenübertragungswege) erhöht. Hier ist man inzwischen fast an der Grenze des physikalisch Möglichen angelangt. Eine weitere Steigerung der Rechengeschwindigkeit ist jedoch erreichbar, wenn man die Von-Neumann-Architektur verläßt und sogenannte

Parallelrechner mit vielen Prozessoren benutzt, die es erlauben, mehrere Verarbeitungsschritte gleichzeitig auszuführen.

Parallelität bedeutet aus algorithmischer Sicht, daß man Probleme daraufhin untersucht, ob sich mehrere zur Lösung erforderliche Teilaufgaben unabhängig voneinander und damit parallel erledigen lassen. Solche Verfahren können dann unter Umständen auf geeigneten Parallelrechnern implementiert werden.

Inzwischen wurde eine große Zahl verschiedener Parallelrechner vorgeschlagen und teilweise auch realisiert. Analog zur RAM hat man auch idealisierte Modelle von Parallelrechnern vorgeschlagen und studiert. Das wichtigste Modell ist die Parallel-Random-Access-Maschine (PRAM). Sie besteht aus p Prozessoren $P_1, \ldots, P_p$, die sämtlich auf einen gemeinsamen Speicher zugreifen können. Außer diesem gemeinsamen Speicher verfügt jeder Prozessor noch über einen privaten Arbeitsspeicher. Die p Prozessoren sind synchronisiert, d.h. sie führen Rechenschritte gleichzeitig, taktweise durch. Ein Rechenschritt eines Prozessors besteht aus drei Phasen. Zuerst kann ein Prozessor den Inhalt einer Zelle des gemeinsamen Speichers lesen, dann eine Rechnung unter Benutzung seines privaten Arbeitsspeichers ausführen und schließlich das Ergebnis der Rechnung in eine Zelle des gemeinsamen Speichers übertragen. Die Kommunikation der Prozessoren untereinander erfolgt über den gemeinsamen Speicher. Jeder Prozessor kann mit jedem anderen Daten in zwei Rechenschritten austauschen.

Man unterscheidet PRAM-Modelle häufig weiter danach, ob mehrere Prozessoren gleichzeitig Daten aus derselben Zelle des gemeinsamen Speichers lesen oder dorthin schreiben dürfen. Das führt zu den EREW (exclusive read exclusive write), CREW (concurrent read exclusive write) und CRCW (concurrent read concurrent write) PRAM-Modellen. Wir zeigen im Abschnitt 9.2.1 an einigen einfachen Beispielen, welche Auswirkung auf die Laufzeit von Algorithmen der Wechsel des Maschinenmodells von der RAM zur PRAM hat.

Natürlich ist die Annahme, daß unbeschränkt viele Prozessoren auf dieselbe Speicherzelle zugreifen können und so miteinander verbunden sind, nicht sehr realistisch. Man kann stattdessen auch Parallelrechner betrachten, bei denen mehrere Prozessoren über ein sogenanntes Verbindungsnetz miteinander kommunizieren. In diesem Fall sind identische Prozessoren an den Knoten eines Graphen plaziert. Die Prozessoren kommunizieren untereinander längs der Kanten des Graphen. Eine ganze Reihe unterschiedlicher Verbindungsnetze sind studiert und zum Teil realisiert worden. Die Struktur des Verbindungsnetzes bestimmt weitgehend, für welche Aufgaben der parallele Rechner besonders geeignet ist. Insbesondere ist die Frage interessant, welche Verbindungsnetze als Basis von universellen Parallelrechnern in Frage kommen. Ein prominenter Vertreter eines Verbindungsnetzes ist der Shuffle-exchange-Graph. Wir werden im Abschnitt 9.2.2 zeigen, wie das Sortieren von Zahlen in einem Netz von Prozessoren durchgeführt werden kann, die an den Knoten eines Shuffle-exchange-Graphen plaziert sind. Eine spezielle Form der Parallelverarbeitung auf in der Regel für bestimmte Aufgaben spezialisierten Rechnern sind systolische Arrays und Algorithmen. Wir bringen einige einfache Beispiele im Abschnitt 9.2.3.

Ziel dieses Abschnittes kann es nicht sein, einen auch nur einigermaßen vollständigen Überblick über das umfangreiche Gebiet der parallelen Algorithmen und Parallelrechner zu geben. Es soll vielmehr an einigen Beispielen illustriert werden, daß ein Wechsel des Rechnermodells erhebliche Auswirkungen auf die Form und Effizienz von Lösungen für Probleme hat, die wir bisher auf Rechnern des Von-Neumann-Typs gelöst ha-

ben. Als Einstieg in die umfangreiche Literatur zum Thema Parallelität verweisen wir auf das Lehrbuch von Leighton [107], auf die zusammenfassende Übersicht [159] und auf die Bücher von Quinn [153], Akl [5], Parberry [145] und Petkov [147] über systolische Algorithmen.

9.2.1 Einfache Beispiele paralleler Algorithmen

Für manche sequentiellen Algorithmen liegt die Parallelisierbarkeit auf der Hand. Betrachten wir als erstes Beispiel die Aufgabe, das Minimum in einer gegebenen Menge von N Schlüsseln zu finden. Jeder sequentielle Algorithmus zur Bestimmung des Minimums muß wenigstens $N-1$ Schlüsselvergleiche durchführen, vgl. Abschnitt 2.1.1. Natürlich kann man das Minimum von N Schlüsseln $k_1,\ldots,k_N$ auch tatsächlich mit $N-1$ Schlüsselvergleichen auf folgende Weise finden:

> $min := k_1$;
> **for** $i := 2$ **to** N **do**
>   **if** $k_i < min$ **then** $min := k_i$;

Offensichtlich kann man jedoch auch anders vorgehen. Man bestimmt zunächst in einem ersten Durchgang $k'_1 = min(k_1,k_2)$, $k'_2 = min(k_3,k_4)$, $k'_3 = min(k_5,k_6)$,..., $k'_{N/2}$ $= min(k_{N-1},k_N)$. Dann bestimmt man in einem zweiten Durchgang $k''_1 = min(k'_1,k'_2)$, $k''_2 = min(k'_3,k'_4)$ usw. Nach $\lceil \log_2 N \rceil$ Durchgängen hat man dann das Minimum gefunden. Offenbar können sämtliche Minimumbestimmungen eines Durchgangs parallel aufgeführt werden. Nehmen wir nun an, daß wir $\lceil N/2 \rceil$ Prozessoren zur Verfügung haben und die N Schlüssel anfangs in Speicherzellen $m[1],\ldots,m[N]$ des gemeinsamen Speichers der Prozessoren stehen. In einem ersten Durchgang lesen die $\lceil N/2 \rceil$ Prozessoren gleichzeitig jeweils den Inhalt zweier aufeinanderfolgender Speicherzellen, der letzte Prozessor eventuell zweimal denselben Schlüssel, berechnen das Minimum der jeweils gelesenen Werte und schreiben es in die ersten $\lceil N/2 \rceil$ Speicherzellen zurück. Jeder Prozessor P_i, $1 \leq i \leq \lceil N/2 \rceil$, liest also $m[2i-1]$ und $m[2i]$, berechnet $min = min(m[2i-1], m[2i])$ und speichert min in Zelle $m[i]$. In einem zweiten Durchgang lesen $\lceil N/4 \rceil$ Prozessoren wiederum gleichzeitig jeweils den Inhalt zweier aufeinanderfolgender Speicherzellen, berechnen das Minimum der jeweils gelesenen Werte und schreiben es in die ersten $\lceil N/4 \rceil$ Speicherzellen zurück usw. Nach $r = \lceil \log_2 N \rceil$ Durchgängen steht dann das Minimum in der Speicherzelle $m[1]$. Folgende Tabelle 9.1 zeigt die Belegung des gemeinsamen Speichers nach jedem Durchgang für ein kleines Beispiel.

Der Inhalt der mit „—“ markierten Zellen hat sich nicht verändert. Lesekonflikte treten nicht auf. Es werden Werte des gemeinsamen Speichers überschrieben, aber Schreibkonflikte treten dabei ebenfalls nicht auf. Man kann also das Minimum von N Schlüsseln mit einer EREW-PRAM mit $\lceil N/2 \rceil$ Prozessoren in $O(\log N)$ Zeit berechnen.

Offenbar kann man diese als *binäre Fan-in-Technik* bekannte Methode des Akkumulierens von Werten in $\lceil \log N \rceil$ Schritten auf eine ganze Reihe weiterer Probleme anwenden. Wir geben einige Beispiele.

$\sum_{i=1}^{N} a_i$ kann mit Hilfe von $\lceil N/2 \rceil$ Prozessoren in Zeit $O(\log N)$ berechnet werden.

m:	1	2	3	4	5	6	7	
	15	2	43	17	4	8	47	Anfangsbelegung
	2	17	4	47	—	—	—	nach 1. Durchgang
	2	4	—	—	—	—	—	nach 2. Durchgang
	2	—	—	—	—	—	—	nach 3. Durchgang

Tabelle 9.1

Denn nehmen wir ohne Einschränkung an, daß $N = 2^r$ ist. Wir benutzen im ersten Durchgang die $N/2$ Prozessoren, um $a_{2i-1} + a_{2i}$ für $1 \leq i \leq N/2$, also die Partialsummen aus je zwei Summanden, zu berechnen. Im zweiten Durchgang werden $N/4$ Prozessoren benutzt, um die Partialsummen aus je vier Summanden zu berechnen, usw. Schließlich berechnet ein Prozessor aus den Partialsummen $a_1 + \ldots + a_{N/2}$ und $a_{N/2+1} + \ldots + a_N$ das Ergebnis. Natürlich funktioniert dasselbe Verfahren auch für die Berechnung von $\prod_{i=1}^{N} a_i$.

Das Produkt zweier $N \times N$ Matrizen kann mit N^3 Prozessoren in Zeit $O(\log N)$ berechnet werden.

Zur Berechnung von $C = A \cdot B$, mit $C = (c_{ij})$ und $c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}$, verwendet man für jedes Element der Produktmatrix N Prozessoren. Mit Hilfe dieser N Prozessoren berechnet man zunächst die N Produkte $a_{i1} \cdot b_{1j}$, $a_{i2} \cdot b_{2j}, \ldots, a_{iN} \cdot b_{Nj}$ und daraus wie oben angegeben in $O(\log N)$ Zeit das Element c_{ij} durch wiederholte Verdopplung der Anzahl der Summanden der Partialsummen. Die insgesamt N^3 Prozessoren müssen zwar dieselben Elemente der Ausgangsmatrizen A und B gleichzeitig lesen können, Schreibkonflikte sind aber vermeidbar, so daß sich das Verfahren auf einer CREW-PRAM implementieren läßt.

Falls mehrere Prozessoren gleichzeitig in dieselbe Speicherzelle des gemeinsamen Speichers schreiben dürfen, ist das Ergebnis von Schreiboperationen zunächst nur dann wohldefiniert, wenn alle Prozessoren denselben Wert in eine Zelle schreiben. Mögliche Schreibkonflikte, also der Versuch, verschiedene Werte in dieselbe Zelle zu schreiben, können nach unterschiedlichen Strategien aufgelöst werden, die uns hier nicht weiter interessieren. Wir wollen jedoch zeigen, daß das Minimum von N Schlüsseln auf einer CRCW-PRAM in konstanter Zeit berechnet werden kann, ohne daß Schreibkonflikte auftreten. Dazu nehmen wir an, daß die Schlüssel in Zellen $a[1], \ldots, a[N]$ gespeichert sind und zusätzlich N Speicherzellen $b[1], \ldots, b[N]$ des gemeinsamen Speichers genutzt werden können. Für alle i und j mit $1 \leq i, j \leq N$ führen die Prozessoren P_{ij} gleichzeitig die folgenden vier Schritte aus, die wir an einem Beispiel mit sieben Schlüsseln erläutern.

	1	2	3	4	5	6	7
a:	15	2	43	2	4	8	47

1. Schritt: P_{i1} schreibt 0 nach $b[i]$.

2. Schritt: P_{ij} liest $a[i]$ und $a[j]$ und schreibt eine 1 nach b_j genau dann, wenn $a[i] < a[j]$. Mit Ausnahme jeder Position j, an der ein minimales Element steht, wird also in b die 0 überall durch eine 1 überschrieben. Für das Beispiel ergibt sich folgende Belegung von b:

	1	2	3	4	5	6	7
b:	1	0	1	0	1	1	1

3. Schritt: P_{ij} liest $b[i]$ und schreibt eine 1 nach $b[j]$ genau dann, wenn $i < j$ und $b[i] = 0$. Dadurch bleibt nur für das kleinste i mit $b[i] = 0$ der Wert 0 erhalten; alle anderen Werte werden durch eine 1 überschrieben. In unserem Beispiel erhalten wir:

	1	2	3	4	5	6	7
b:	1	0	1	1	1	1	1

4. Schritt: P_{i1} liest $b[i]$ und schreibt $a[i]$ nach $b[1]$ genau dann, wenn $b[i] = 0$ ist. Jetzt steht das Minimum in $b[1]$.

Als letztes Beispiel wollen wir zeigen, wie ein *Verfahren zur Berechnung eines minimalen spannenden Baumes* (*MST*) eines Graphen parallelisiert werden kann. Das in Abschnitt 8.6 beschriebene Verfahren von Boruvka zur Berechnung des MST besteht darin, einen Wald von Teilbäumen des MST sukzessiv zum MST zusammenwachsen zu lassen. Man beginnt mit Teilbäumen, die sämtlich nur aus je genau einem Knoten des gegebenen Graphen bestehen. Dann werden immer wieder je zwei verschiedene Teilbäume durch Hinzunahme einer Kante minimalen Gewichtes zu einem Baum verbunden, bis ein einziger Baum, der MST, entstanden ist. Man kann versuchen, eine parallele Version dieses Verfahrens dadurch zu erhalten, daß man gleichzeitig Kanten minimalen Gewichts wählt, die verschiedene Teilbäume miteinander verbinden. Wie man leicht sieht, kann eine nicht weiter eingeschränkte Wahl aber zu Zyklen führen, wenn im Graphen Kanten gleichen Gewichts auftreten.

Nehmen wir an, daß die Knoten des gegebenen Graphen mit den natürlichen Zahlen $1,\dots,N$ bezeichnet werden. Dann kann man auf den Kanten eine lexikographische Anordnung ungeordneter Paare, die sogenannte *Min-max-Ordnung* „$\prec$“, wie folgt einführen.

> Es gilt für die ungerichteten Kanten (u,v) und (u',v')
> $(u,v) \prec (u',v')$ genau dann, wenn $min\{u,v\} < min\{u',v'\}$ oder
> $(min\{u,v\} = min\{u',v'\}$ und $max\{u,v\} < max\{u',v'\})$.

Wählen wir nun für jeden Knoten i eine Kante (i,j) mit minimalem Gewicht so, daß (i,j) die bezüglich der Min-max-Ordnung erste Kante dieser Art ist, so werden Zyklen vermieden. Denn nehmen wir beispielsweise an, es gäbe einen Dreierzyklus. Für drei Knoten i, j und k mit $i < j < k$ seien die Kanten (i,j), (j,k) und (k,i) gewählt worden. Weil zum Knoten i die Kante (i,j) und nicht die Kante (i,k) gewählt wurde, muß für die Gewichte $g(i,j)$ und $g(i,k)$ dieser Kanten gelten:

$$g(i,j) \leq g(i,k) = g(k,i).$$

Aus analogen Gründen muß auch

$$g(j,k) \leq g(j,i) = g(i,j),$$
$$g(k,i) \leq g(k,j) = g(j,k)$$

sein. Daraus erhält man $g(i,j) = g(j,k) = g(k,i)$. Dann kann aber für j nicht die Kante (j,k) gewählt worden sein, weil (j,i) eine Kante mit gleichem Gewicht ist, aber $(j,i) \prec (j,k)$ gilt. Eine ähnliche Argumentation zeigt die Unmöglichkeit von Zyklen beliebiger Länge.

Der folgende Algorithmus zur Berechnung eines minimalen spannenden Baumes stammt von Sollin. Er setzt voraus, daß der Graph G die Knotenmenge $\{1,..,N\}$ besitzt und die Kanten implizit durch die Gewichtsfunktion g gegeben sind mit $g(i,j) = \infty$, falls i und j in G nicht miteinander verbunden sind.

```
procedure Sollin (G : Graph; var F : Wald);
{F ist Wald von Teilbäumen des MST für G, am Ende ist
 F = {T}, T MST für G}
var i : integer; {Laufindex}
begin
{initialisiere F als Menge von N Teilbäumen mit genau einem Knoten
 und keiner Kante}
  for i := 1 to N do T_i = {i};
  F := {T_1,..,T_N};
  while |F| > 1 do
    begin
      for each T ∈ F do {parallel}
        begin
          finde bezüglich „≺“ erstes Paar von Knoten (u,v)
            mit u ∈ T, v ∈ T' ∈ F\{T}, g(u,v) minimal
        end;
      berechne neuen Wald F durch Verschmelzen von Bäumen,
        die durch zuvor gewählte Kanten miteinander verbunden sind
    end {while}
end {Sollin}
```

Wir geben ein Beispiel für Sollins Algorithmus an. Dabei folgen wir der Konvention, beim Verschmelzen von zwei Bäumen T_i und T_j dem neuen Baum den Namen $T_{min\{i,j\}}$ zu geben. Gegeben sei der Graph aus Abbildung 9.10.

Der Initialisierungsschritt liefert den Wald $F = \{T_1,..,T_8\}$ mit $T_i = \{i\}$, $1 \leq i \leq 8$. Nach einmaliger Ausführung der Anweisungen in der **while**-Schleife erhält man den Wald von Abbildung 9.11 mit den Teilbäumen $T_1 = \{1,3,8\}$, $T_2 = \{2,4,5\}$, $T_6 = \{6,7\}$ und den in der Abbildung 9.11 gezeigten Kanten.

Im nächsten Schritt wird nun für T_1 die Kante $(8,2)$ mit Gewicht 3, für T_2 dieselbe Kante und für T_6 die Kante $(6,5)$ gewählt. Durch Verschmelzen der durch Kanten verbundenen Bäume entsteht ein einziger Baum, der MST aus Abbildung 9.12.

Offenbar wird die Anzahl der Bäume im Wald F bei einmaliger Ausführung der Anweisungen der **while**-Schleife wenigstens um die Hälfte reduziert. Daher kann die **while**-Schleife höchstens $\log_2 N$-mal durchlaufen werden.

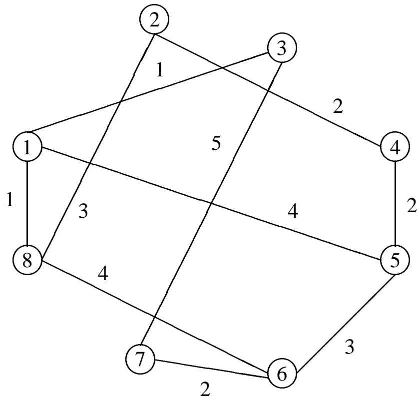


Abbildung 9.10

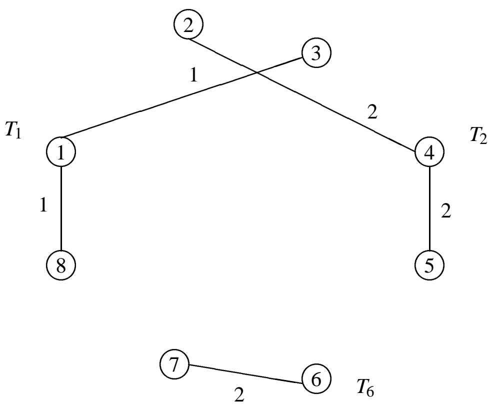


Abbildung 9.11

Nehmen wir jetzt an, wir hätten zur Ausführung des Algorithmus von Sollin N Prozessoren $P_1, \ldots, P_N$ zur Verfügung. Für jedes i, $1 \leq i \leq N$, wird Prozessor P_i dem Knoten i zugeordnet. Wir können annehmen, daß die den Graphen G vollständig charakterisierende Gewichtsfunktion g als Adjazenzmatrix im gemeinsamen Speicher der N Prozessoren abgelegt ist. In einem Bereich $t[1 \,..\, N]$ des gemeinsamen Speichers merkt man sich für jedes i, $1 \leq i \leq N$, den Index j des Baumes T_j, in dem der Knoten i jeweils liegt. Der Initialisierungsschritt besteht also darin, daß für jedes i, $1 \leq i \leq N$, P_i den Wert i nach $t[i]$ schreibt. Das ist parallel in konstanter Zeit ausführbar. Jeder Durchlauf der **while**-Schleife kann jetzt in drei Schritten erledigt werden.

Im ersten Schritt bestimmt jeder Prozessor P_i den nächsten, mit i verbundenen Knoten $j = nn(i)$, der nicht in dem Baum liegt, der i enthält. Diese Suche nach $nn(i)$, d.h. nach dem kleinsten j mit $g(i, j)$ minimal und $j \notin T_{t[i]}$ kann P_i offenbar in Zeit $O(N)$ erledigen. Im zweiten Schritt wird jetzt für jeden Baum eine bezüglich der Min-max-Ordnung

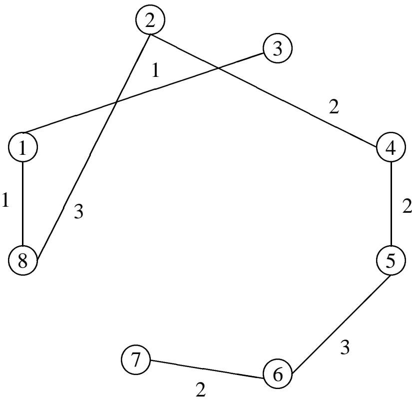


Abbildung 9.12

kleinste Kante minimalen Gewichts bestimmt, die ihn mit einem anderen Baum verbindet. Dazu inspiziert jeder Prozessor P_i noch einmal alle mit i verbundenen Knoten. Trifft P_i dabei auf einen Knoten $k \neq i$ mit $t[i] = t[k]$ und $g(i, nn(i)) = g(i, k)$, weiß P_i, daß es zwei Kanten minimalen Gewichts gibt, die den Baum, in dem der Knoten i liegt, mit einem anderen Baum verbinden können, nämlich die Kanten $(i, nn(i))$ und (i, k). P_i merkt sich dann, daß die Kante $(i, nn(i))$ nicht in Frage kommt (d.h.: P_i scheidet aus), genau dann, wenn $(i, k) \prec (i, nn(i))$ ist. Dieser zweite Schritt ist offenbar ebenfalls parallel in Zeit $O(N)$ ausführbar. Die im zweiten Schritt nicht ausgeschiedenen Prozessoren enthalten jetzt genau die Kanten, die zum Verschmelzen von Bäumen des aktuellen Waldes herangezogen werden müssen. Das geschieht im dritten Schritt. In diesem Schritt werden die Einträge im Array t wie folgt verändert: Der Reihe nach teilt jeder noch aktive Prozessor, der eine Verbindungskante (i, j) gespeichert hat, allen anderen Prozessoren mit, daß der Name $max(t[i], t[j])$ durch $min(t[i], t[j])$ ersetzt werden muß. Jeder Prozessor prüft für sich, ob der Knoten, den er repräsentiert, in einem Baum liegt, der von dieser Namensänderung betroffen ist; die Namensänderung wird dann gleichzeitig in konstanter Zeit ausgeführt. Damit kann das Verschmelzen von Bäumen im dritten Schritt insgesamt in Zeit $O(N)$ ausgeführt werden.

Mit Hilfe von N Prozessoren kann man also jeden Durchlauf der **while**-Schleife in Zeit $O(N)$ ausführen, wobei jedesmal $O(N^2)$ Einzeloperationen durchgeführt werden. Wir fassen unsere Überlegungen in einem Satz zusammen.

Satz 9.3 *Für einen gewichteten Graphen mit N Knoten kann man mit Hilfe von N Prozessoren einen minimalen spannenden Baum in Zeit $O(N \log N)$ berechnen. Dabei werden von den N Prozessoren insgesamt $O(N^2 \log N)$ Operationen ausgeführt.*

Ein wesentlicher Grund für den Zeitbedarf des Sollin'schen Algorithmus bei Verwendung von N Prozessoren liegt darin, daß bei jedem Durchlauf durch die **while**-Schleife alle N Prozessoren Minima bestimmen müssen. Das kostet jeweils $\Theta(N)$ Schritte und führt damit zur Gesamtlaufzeit $O(N \log N)$. Unter Benutzung von N^2 Prozessoren kann

man die Laufzeit des Verfahrens drücken, weil man die Bestimmung des Minimums mit je N^2 Prozessoren in konstanter Zeit erledigen kann.

Schließlich kann man das Verfahren von Sollin ohne Effizienzverlust auch noch auf Parallelrechnern mit stark eingeschränkten Kommunikationsmöglichkeiten implementieren, vgl. [15]. Eine ausführliche Übersicht über parallele Graphenalgorithmen und ihre Implementation auf verschiedenen Parallelrechnern enthält die Arbeit [154].

9.2.2 Paralleles Mischen und Sortieren

Wir untersuchen jetzt die Frage, ob durch den Einsatz von mehreren Prozessoren die zum Sortieren von N Schlüsseln erforderliche Zeit verkürzt werden kann. Es liegt nahe, zunächst die seriellen, also für Rechner des Von-Neumann-Typs mit nur einem Prozessor, entwickelten Sortierverfahren auf ihre Parallelisierbarkeit hin zu untersuchen. Ein typischer Schritt in einem seriellen Sortierverfahren ist, daß der Prozessor zwei Schlüssel miteinander vergleicht. Die restlichen Schlüssel stehen „ungenutzt" im Speicher. Es ist daher naheliegend, jedem Paar von Schlüsseln einen Prozessor zuzuordnen, der eine solche Vergleichsoperation ausführen kann. Wir stellen uns also vor, daß der zum Sortieren benutzte Parallelrechner eine große, von der Zahl N der zu sortierenden Schlüssel abhängige Zahl von sogenannten *Compare-exchange-Moduln* hat, vgl. Abbildung 9.13.

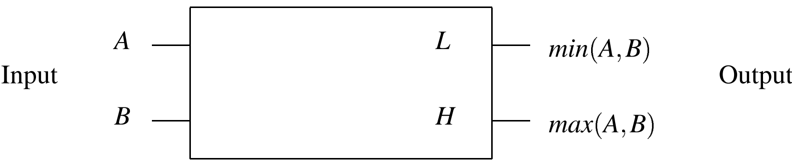


Abbildung 9.13

Ein Compare-exchange-Modul (oder: Vergleichsmodul) kann zwei Werte gleichzeitig lesen, sie miteinander vergleichen und geordnet wieder ausgeben. Der kleinere Schlüssel verläßt den Vergleichsmodul über den mit L (für: Low) und der größere über den mit H (für: High) gekennzeichneten Ausgang. Die N Schlüssel müssen auf die Compare-exchange-Moduln verteilt werden, d.h. es ist die Frage zu beantworten, welche Schlüssel zu welchem Zeitpunkt in welchem Vergleichsmodul zusammentreffen. Wir wollen in diesem Abschnitt nicht voraussetzen, daß die Vergleichsmoduln über einen gemeinsamen Speicher kommunizieren. Wir suchen vielmehr ein festes Verbindungsnetz für die Vergleichsmoduln.

Ein auf einem einzigen Prozessor seriell ablaufendes Sortierprogramm kann seinen Ablauf von Ereignissen abhängig machen, die erst während der Programmausführung auftreten. Ein in Hardware realisiertes Verbindungsschema ist jedoch unveränderlich. Betrachten wir als Beispiel das folgende, zum Sortieren von drei Schlüsseln geeignete, serielle Programmstück.

if $A > B$ **then** $vertausche(A,B)$;
if $B > C$ **then begin**
$vertausche(B,C)$;
if $A > B$ **then** $vertausche(A,B)$
end

Man überprüft leicht,daß für beliebige Anfangswerte von A, B und C die Werte dieser Variablen nach Ausführung des Programmstücks aufsteigend sortiert sind. Es werden aber z.B. für die Eingabe $A = 2, B = 1, C = 3$ Teile des Programms nicht ausgeführt. Der letzte Vergleich ist in diesem Fall unnötig und unterbleibt. Ein aus Vergleichsmoduln aufgebautes Verbindungsnetz kann seine Struktur jedoch nicht von den Eingangsdaten abhängig machen. Dennoch ist Sortieren möglich, wie das in Abbildung 9.14 gezeigte Netz aus drei Vergleichsmoduln zeigt.

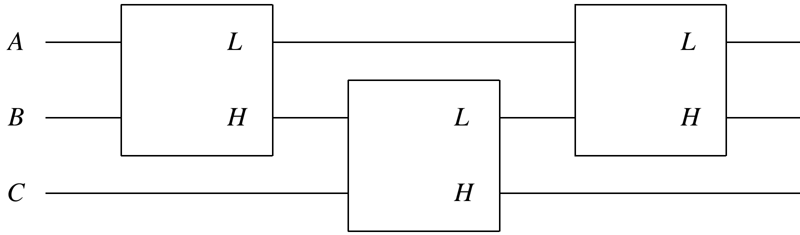


Abbildung 9.14

Den Variablen A, B, C entsprechen die Eingänge des Verbindungsnetzes. Es ist leicht zu überprüfen, daß die bei A, B, C eingegebenen Schlüssel das Netz in aufsteigend sortierter Reihenfolge über die drei rechten Ausgänge verlassen. Wenn wir annehmen, daß ein Paar von Schlüsseln in einer Zeiteinheit verarbeitet werden kann, folgt sofort, daß die am linken Ende des Sortiernetzes eingegebene Folge nach drei Zeiteinheiten am rechten Ende, also am Ausgang des Netzes, in sortierter Reihenfolge vorliegt.

In seriellen Sortierverfahren spielen Merge-Strategien eine wichtige Rolle. Man zerlegt die zu sortierende Folge, sortiert die entstandenen Teilfolgen und verschmilzt die sortierten Teilfolgen zur sortieren Gesamtfolge. Soll diese Technik auch für paralleles Sortieren eingesetzt werden, so benötigt man Verschmelzungsverfahren, die es erlauben, zwei sortierte Schlüsselfolgen mit immer der gleichen Operationsfolge zu einer sortierten Folge zu verschmelzen. Wir erläutern jetzt zwei solcher Verfahren, die unter dem Namen *Odd-even-merge* und *Bitonic-merge* bekannt sind, vgl. [19].

Wir erläutern zunächst das *Odd-even-merge-Verfahren*. Gegeben seien zwei Folgen $a_1,\dots,a_n$ und $b_1,\dots,b_n$ von jeweils aufsteigend sortierten Zahlen gleicher Länge, d.h. es gilt für alle i, $1 \le i < n$, $a_i \le a_{i+1}$ und $b_i \le b_{i+1}$. Wir wollen diese zwei Folgen zu einer einzigen, aufsteigend sortierten Folge der Länge $2n$ verschmelzen. Wir lösen diese Aufgabe rekursiv und nehmen der Einfachheit halber an, daß $n = 2^k$ für ein $k \ge 0$ ist. Ist $n = 1$, werden a_1 und b_1 miteinander verglichen und in die richtige Reihenfolge gebracht. Ist $n > 1$, so betrachten wir zunächst die Folgen halber Länge mit *ungeradzahligem* Index $a_1,a_3,\dots,a_{n-1}$ und $b_1,b_3,\dots,b_{n-1}$ und verschmelzen sie auf dieselbe Weise zu einer aufsteigend sortierten Folge $c_1,\dots,c_n$. Dann betrachten wir die Folgen halber Länge mit *geradzahligem* Index $a_2,a_4,\dots,a_n$ und $b_2,b_4,\dots,b_n$ und verschmelzen sie zu einer aufsteigend sortierten Folge $d_1,\dots,d_n$. Nun kann man zeigen, daß für jedes i, $1 \le i < n$, das Element c_{i+1} unmittelbar vor oder unmittelbar nach dem Element d_i der Größe nach eingeordnet werden muß. (Einen Beweis findet man in [89] oder in [5].) Wir können aus $c_1,\dots,c_n$ und $d_1,\dots,d_n$ also eine sortierte Folge $e_1,\dots,e_{2n}$ herstellen, indem wir setzen:

$$\begin{aligned} e_1 &= c_1 \\ e_{2i} &= min(c_{i+1},d_i), \text{ für } 1 \le i < n \\ e_{2i+1} &= max(c_{i+1},d_i), \text{ für } 1 \le i < n \\ e_{2n} &= d_n. \end{aligned}$$

Beispiel: Gegeben seien die aufsteigend sortierten Folgen

$a:$	2	15	19	43
$b:$	4	8	17	47

Verschmelzen der Teilfolgen mit geradzahligem bzw. ungeradzahligem Index ergibt

$c:$	2	4	17	19
$d:$	8	15	43	47

Vergleichen und gegebenenfalls Vertauschen der Paare (c_{i+1},d_i), also (4,8), (17,15), (19,43), ergibt die sortierte Folge

$$e: \quad 2 \quad 4 \quad 8 \quad 15 \quad 17 \quad 19 \quad 43 \quad 47.$$

Es ist offensichtlich, daß das Odd-even-merge-Verfahren als Netzwerk von Vergleichsmoduln realisiert werden kann. Für $n = 1$ besteht das Netzwerk genau aus einem Vergleichsmodul. Für $n > 1$, wobei der Einfachheit halber $n = 2^k$ für ein $k > 0$ gelte, hat das Netzwerk genau $2n$ Eingabeleitungen, die linear angeordnet sind, und zwar für die Folgen der Eingabewerte $a_1,b_1,a_3,b_3,\dots,a_{n-1},b_{n-1}$ und $a_2,b_2,a_4,b_4,\dots,a_n,b_n$, in dieser Reihenfolge, und $2n$ Ausgabeleitungen $e_1,e_2,\dots,e_{2n}$. Nehmen wir an, wir hätten bereits ein Netzwerk zum Verschmelzen zweier Folgen der Länge $n/2$, so erhält man ein Netzwerk zum Verschmelzen von zwei Folgen mit Länge n, wenn man es aus gegebenen Netzen und Vergleichsmoduln wie in Abbildung 9.15 gezeigt zusammensetzt. Dabei gehört der links gezeigte Teil sich kreuzender Leitungen nicht zum Netzwerk; er sorgt lediglich dafür, daß beim Zusammensetzen von Netzen die zu verschmelzenden Eingabefolgen korrekt verzahnt an die Teil-Netzwerke weitergeleitet werden.

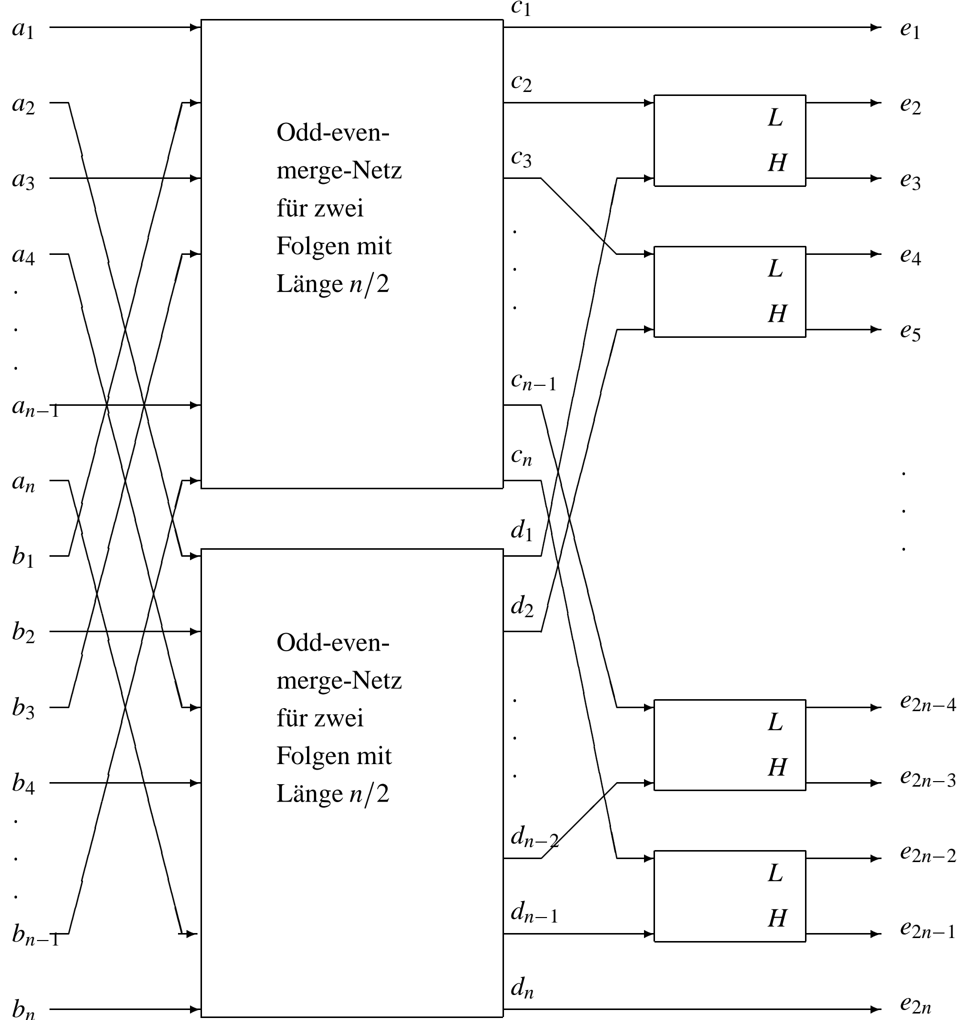


Abbildung 9.15

Wir nennen ein Netzwerk zum Verschmelzen von zwei sortierten Folgen mit Längen $n/2$ nach dem Odd-even-merge-Verfahren ein *OEM-Netz der Größe n*. Das in Abbildung 9.15 gezeigte Verfahren zur Konstruktion von OEM-Netzen der Größe n zeigt unmittelbar, daß eine in ein OEM-Netz der Größe $n = 2^k$ eingegebene Zahl höchstens k Vergleichsmoduln durchläuft bis sie das Netz verläßt.

Analog zum reinen 2-Wege-Mergesort, vgl. Abschnitt 2.4.2, kann man jetzt $n = 2^k$ Zahlen wie folgt sortieren: Man beginnt mit n Folgen der Länge 1 und verschmilzt sie gleichzeitig mit 2^{k-1} OEM-Netzen der Größe 2^1 zu $n/2$ Folgen der Länge 2. Dann verschmilzt man $n/2$ Folgen der Länge 2 mit 2^{k-2} OEM-Netzen der Größe 2^2 zu Folgen

der Länge 2^2 usw. Daraus kann man unmittelbar ein Konstruktionsprinzip für ein Sortiernetz zum parallelen Sortieren von $n = 2^k$ Zahlen ablesen: Ein Sortiernetz für zwei Zahlen ist ein Vergleichsmodul. Ein Sortiernetz für $n > 2$ Zahlen erhält man aus zwei Sortiernetzen für $n/2$ Zahlen und einem OEM-Netz der Größe n wie in Abbildung 9.16 dargestellt. Wir nennen ein nach diesem Prinzip aufgebautes Sortiernetz ein OES-Netz der Größe n.

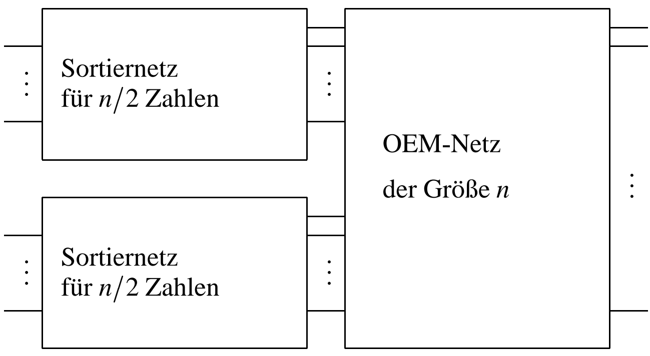


Abbildung 9.16

Abbildung 9.17 zeigt explizit ein OES-Netz der Größe 8. Offenbar können alle in einer Spalte untereinander stehenden Vergleichsmoduln des Netzes parallel arbeiten.

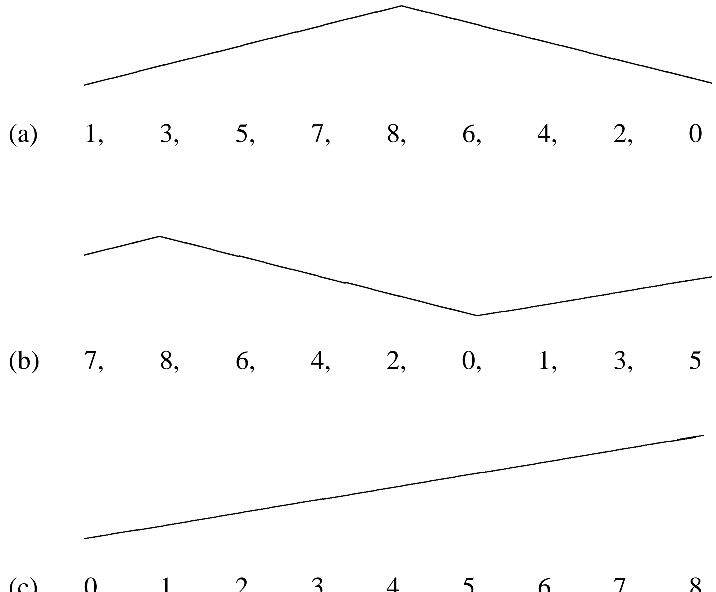
Man kann aus dem Verfahren zur Konstruktion von OES-Netzen der Größe $n = 2^k$ unmittelbar ablesen, daß ein in das Netz eingegebener Schlüssel höchstens $1+2+\ldots+k = k(k+1)/2$ Vergleichsmoduln durchläuft, bevor er das Netz (an der richtigen Stelle) wieder verläßt. Ferner enthält ein OES-Netz der Größe n offenbar höchstens $(1+2+\ldots+k)\cdot n/2$ Vergleichsmoduln insgesamt, für größere n sogar weit weniger. Wegen $k = \log_2 n$ folgt damit sofort:

Satz 9.4 *n Zahlen können in Zeit $O(\log^2 n)$ mit Hilfe eines aus $O(n \log^2 n)$ Vergleichsmoduln bestehenden Netzes sortiert werden.*

Nicht alle in ein OES-Netz eingegebenen Schlüssel durchlaufen dieselbe Anzahl von Vergleichsmoduln, bevor sie das Netz verlassen. Wir geben jetzt ein Verfahren zum Verschmelzen zweier sogenannter *bitonischer* Folgen an, das schließlich zu einem sehr regelmäßig aufgebauten Sortiernetz führt. Eine Zahlenfolge heißt *bitonisch*, wenn sie durch Aneinanderhängen einer absteigend an eine aufsteigend sortierte Zahlenfolge oder durch zyklische Vertauschung aus einer solchen Zahlenfolge entsteht. Hier sind einige Beispiele bitonischer Folgen, die wir auf naheliegende Weise zugleich graphisch veranschaulicht haben.

2^1 2^2 2^3

Abbildung 9.17



Das *Bitonic-merge-Verfahren* überführt zwei bitonische Zahlenfolgen in sortierte Folgen. Es basiert auf der Beobachtung, daß eine bitonische Folge in zwei bitonische Folgen zerlegt werden kann, indem man je zwei $n/2$ Positionen voneinander entfernte Elemente miteinander vergleicht und gegebenenfalls vertauscht, wobei n die Länge der bitonischen Folge ist. Genauer gilt:

Lemma 9.1 *Sei $a = a_0, \ldots, a_{n-1}$ eine bitonische Folge. Sei $b_i = min(a_i, a_{i+n/2})$ und $c_i = max(a_i, a_{i+n/2})$ für $0 \leq i < n/2$. Dann sind die Folgen $b = b_0, \ldots, b_{n/2-1}$ und $c = c_0, c_1, \ldots, c_{n/2-1}$ ebenfalls bitonisch. Darüberhinaus gilt $b_i \leq c_j$ für alle i und j.*

Zum Beweis nehmen wir zunächst an, daß die gegebene Folge aus zwei gleichlangen Teilfolgen besteht, von denen die erste $a_0, \ldots, a_{n/2-1}$ aufsteigend und die zweite $a_{n/2}, \ldots, a_{n-1}$ absteigend sortiert ist. Die Bildung der Folgen b und c aus a kann durch Abbildung 9.18 veranschaulicht werden.

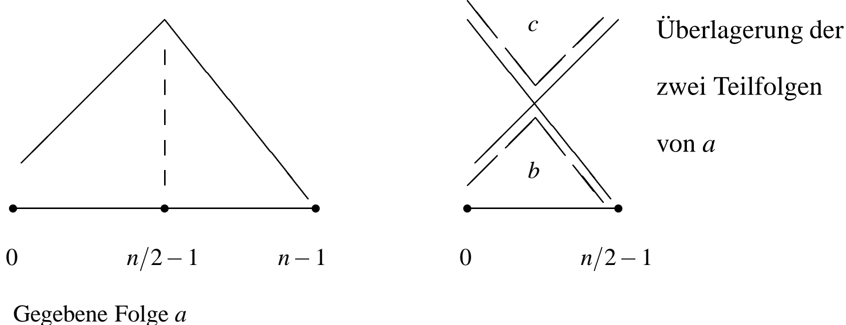


Abbildung 9.18

Es ist klar, daß die so gebildeten Folgen b und c bitonisch sind und alle Elemente von c größer als alle Elemente von b sein müssen. Man sieht leicht, daß die Behauptung auch dann noch gilt, wenn die beiden Teilfolgen von a unterschiedliche Länge haben oder a durch zyklische Vertauschung aus einer zunächst auf- und dann absteigend sortierten Folge entsteht. Abbildung 9.19 zeigt ein weiteres Beispiel für die Bildung der Folgen b und c. □

Aus dem Lemma kann man ein rekursives Konstruktionsprinzip zur Konstruktion von Netzen zum Sortieren von bitonischen Folgen ablesen. Wir nennen ein Netzwerk zum Sortieren einer bitonischen Folge mit Länge n nach dem Bitonic-merge-Verfahren ein *BM-Netz der Größe n*. Ein Vergleichsmodul ist ein BM-Netz der Größe 2. Nehmen wir an, wir haben bereits zwei BM-Netze der Größe $n/2$. Dann ist das in Abbildung 9.20 gezeigte Netz ein BM-Netz der Größe n.

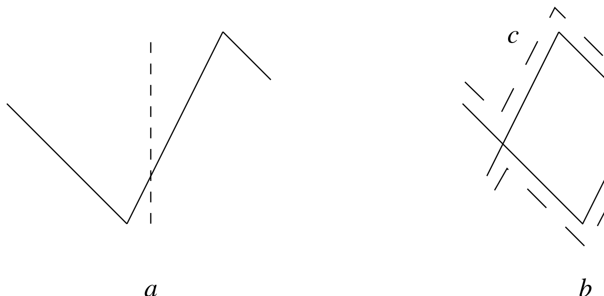


Abbildung 9.19

Für die spätere Realisierung eines Sortiernetzes weisen wir bereits hier auf eine wichtige Eigenschaft von BM-Netzen hin. Nehmen wir an, daß $n = 2^k$ ist und die Folgenindizes der in das BM-Netz der Größe n eingegebenen Schlüssel als Dualzahlen der Länge k dargestellt werden. Dann kann man aus Abbildung 9.20 sofort ablesen, daß die Schlüssel, die in einem Vergleichsmodul in der ersten Spalte des Netzes miteinander verglichen werden, Indizes haben, deren Dualdarstellung sich genau an der höchstwertigen, also k-ten Position von rechts unterscheidet.

Beispiel: Ist $n = 8$, so werden in der ersten Spalte von Vergleichsmoduln die Schlüsselpaare mit folgenden Indizes in Dualdarstellung mit Länge 3 miteinander verglichen:

$$(000, 100)\ (001, 101)\ (010, 110)\ (011, 111)$$

Wegen des rekursiven Aufbaus von BM-Netzen gilt eine entsprechende Aussage natürlich auch für die in BM-Netzen mit Größe $n/2$ in Abbildung 9.20 auftretenden Vergleichsmoduln.

Eine in ein BM-Netz der Größe $n = 2^k$ eingegebene bitonische Zahlenfolge verläßt das Netz aufsteigend sortiert, nachdem jede Zahl genau k Vergleichsmoduln durchlaufen hat. Natürlich kann man auf dieselbe Weise ein Netz konstruieren, das eine bitonische Folge absteigend sortiert. Dazu genügt es, die Ausgänge L und H der Vergleichsmoduln in Abbildung 9.20 zu vertauschen und anzunehmen, daß die zwei in der rekursiven Konstruktion eines BM-Netzes der Größe n benutzten BM-Netze der Größe $n/2$ jeweils eine von oben nach unten absteigend sortierte Folge liefern. Wir kennzeichnen ein BM-Netz, das eine aufsteigend bzw. absteigend sortierte Folge liefert durch ein „+“ bzw. „−“. Mit Hilfe solcher Netze kann man jetzt rekursiv Netze zum Sortieren von Folgen der Länge n konstruieren. Wir nennen ein Netz dieser Art ein *BS-Netz der Größe* n und nehmen der Einfachheit halber wieder an, daß $n = 2^k$ für ein $k \geq 0$ ist. Falls $n = 2$ ist, definieren wir als auf- bzw. absteigend sortierendes BS-Netz der Größe 2 die aus je einem Vergleichsmodul bestehenden Netze, vgl. Abbildung 9.21.

Nehmen wir an, wir haben bereits zwei BS-Netze der Größe $n/2$, die zwei Folgen der Länge $n/2$ auf- bzw. absteigend sortieren. Dann erhalten wir ein BS-Netz der Größe n, das aufsteigend sortiert, indem wir es wie in Abbildung 9.22 gezeigt mit einem BM-Netz der Größe n verbinden. Ein BS-Netz der Größe n, das absteigend sortiert, erhält man analog.

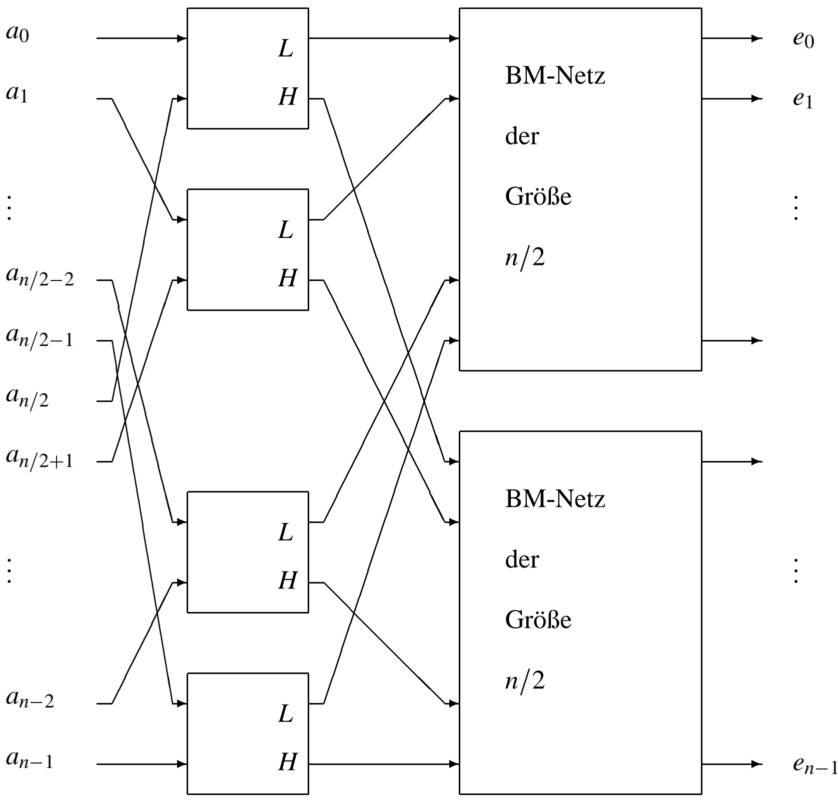


Abbildung 9.20

Abbildung 9.23 zeigt explizit ein nach diesem Prinzip konstruiertes BS-Netz für Zahlenfolgen der Länge 8. Die von links her erste Spalte von Vergleichsmoduln sortiert vier Paare von Zahlen zu auf- bzw. absteigenden Folgen der Länge 2; nach der ersten Spalte hat man also zwei bitonische Folgen mit Länge 4. Die nächsten zwei Spalten von Vergleichsmoduln stellen daraus eine bitonische Folge mit Länge 8 her und die letzten drei Spalten von Vergleichsmoduln stellen daraus schließlich eine aufsteigend sortierte Folge her.

+ ≡ L H , − ≡ H L

Abbildung 9.21

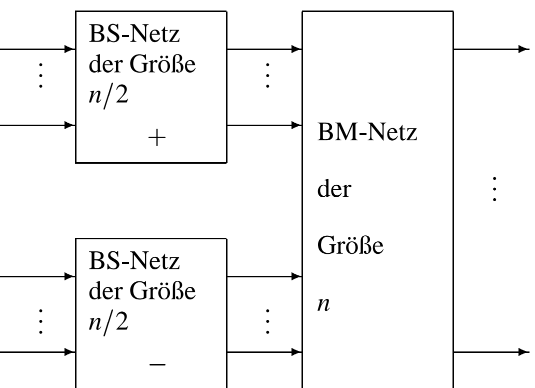


Abbildung 9.22

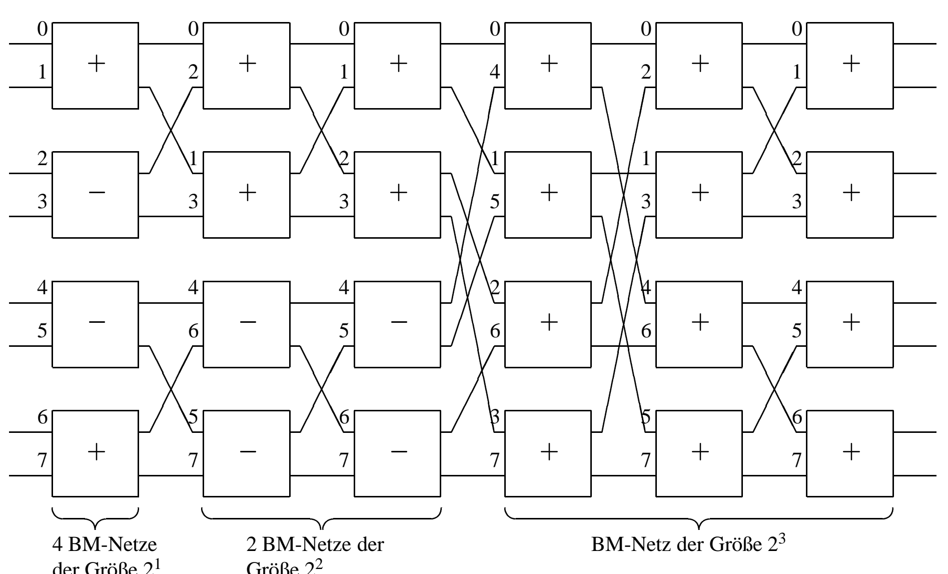


Abbildung 9.23

Wie im Falle des Odd-even-mergesort folgt auch hier, daß $n = 2^k$ Zahlen in $k(k+1)/2$ Schritten mit Hilfe eines BS-Netzes der Größe n sortiert werden können. Dabei besteht ein BS-Netz der Größe n aus $n/2$ BS-Netzen der Größe 2, $n/4$ BS-Netzen der Größe 4 usw. Jedes BS-Netz der Größe 2^j besteht wiederum aus j Spalten von Vergleichsmoduln, die nach dem in Abbildung 9.20 angegebenen Prinzip miteinander verbunden sind. Ein BS-Netz der Größe n besteht also aus $O(n \log^2 n)$ Vergleichsmoduln. Damit gilt der oben für OES-Netze formulierte Satz auch für BS-Netze.

Von H.S. Stone [176] wurde gezeigt, daß man mit nur $n/2$ Vergleichsmoduln insgesamt auskommen kann. Die Vergleichsmoduln werden allerdings mehrfach benutzt und die Eingänge zuvor geeignet permutiert. Betrachten wir ein BS-Netz für $n = 2^k$ Zahlen, also z.B. das Netz aus Abbildung 9.23 für acht Zahlen. Es ist aus $1+2+3+\ldots+k$ Spalten von je $n/2$ Vergleichsmoduln aufgebaut. Stellt man die Indizes aller n Schlüssel als Dualzahlen gleicher Länge k dar, so werden in der ersten Spalte Schlüssel in einen Vergleichsmodul zusammengeführt, deren Index sich genau an der 0-ten Position unterscheidet. Die Indizes von Schlüsseln, die in Vergleichsmoduln der nächsten zwei Spalten zusammentreffen, unterscheiden sich durch die Bits an den Positionen 1 (in Spalte 2) und 0 (in Spalte 3) usw. Wir zählen dabei Bitpositionen wie üblich von rechts nach links, beginnend mit Position 0. D.h. die Bitpositionen, an denen sich die Indizes von miteinander verglichenen Schlüsseln unterscheiden, sind der Reihe nach die folgenden Positionen:

$$0;\ 1,\ 0;\ 2,\ 1,\ 0;\ \ldots\ ;\ k-1,\ \ldots\ ,\ 1,\ 0.$$

Ein *Shuffle-exchange-Netz* der Größe $n = 2^k$ ist ein Netz, das die Eingänge so vertauscht, daß sich die Indizes je zweier aufeinanderfolgender Ausgänge genau im höchstwertigen Bit unterscheiden, also im Bit an Position $k-1$. Wird dasselbe Netz zweimal hintereinander durchlaufen, unterscheiden sich die Indizes der Eingänge von je zwei aufeinanderfolgenden Ausgängen an der zweithöchsten Bitposition, also an Bitposition $k-2$ usw. Abbildung 9.24 zeigt ein Shuffle-exchange-Netz der Größe 8.

Damit liegt es nahe, ein Sortiernetz aus einer einzigen Spalte von Vergleichsmoduln zu konstruieren und die Eingänge mit Hilfe eines Shuffle-exchange-Netzes zunächst so lange zu permutieren, bis die Schlüssel mit den richtigen Indizes in Vergleichsmoduln zusammentreffen. Abbildung 9.25 zeigt ein solches Netz für $n = 8$.

Bevor die $n = 2^k$ Schlüssel miteinander verglichen werden, deren Indizes sich an den Bitpositionen $j-1,\ldots,0$ unterscheiden, muß man die Vergleichsmoduln zunächst „abschalten“ und die Eingänge $k-j$-mal das Shuffle-exchange-Netz durchlaufen lassen, für j von 1 bis k. Ein nach dem in Abbildung 9.25 angegebenen Prinzip aus $n/2$ (abschaltbaren) Vergleichsmoduln aufgebautes Sortiernetz muß also $k^2 = \log^2 n$-mal durchlaufen werden, um n Schlüssel zu sortieren. Man erhält also:

Satz 9.5 *Mit Hilfe eines aus $n/2$ Vergleichsmoduln aufgebauten, nach dem Shuffle-exchange-Prinzip verbundenen Netzes können n Schlüssel in Zeit $O(\log^2 n)$ sortiert werden.*

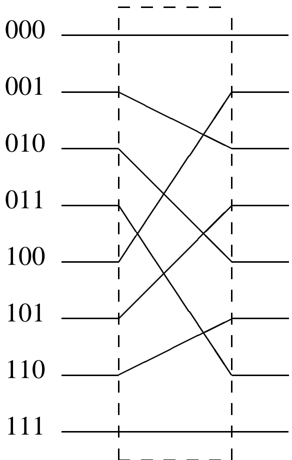


Abbildung 9.24

Weil das Sortieren von n Zahlen mit Hilfe eines einzigen Prozessors $\Omega(n \log n)$ Vergleichsoperationen von Schlüsseln erfordert, wird man nicht erwarten können, daß das Produkt der Zahl der Vergleichsmoduln eines Sortiernetzes und der zum parallelen Sortieren erforderlichen Zeit unter $\Omega(n \log n)$ liegt. Das schließt aber nicht aus, daß es Sortiernetze geben kann, die n Zahlen in logarithmischer Zeit mit $O(n)$ Prozessoren sortieren können. Ein wichtiges, neues Ergebnis in dieser Richtung stammt von Ajtai, Komlós und Szemerédi [4]. Sie zeigen, daß ein aus $O(n \log n)$ Vergleichsmoduln bestehendes Netz n Zahlen in Zeit $O(\log n)$ sortieren kann.

9.2.3 Systolische Algorithmen

Der Begriff *systolische Algorithmen* stammt von Kung und Leiserson [97]. Damit sollen Algorithmen mit folgenden Eigenschaften charaktersiert werden: Sie können mit Hilfe weniger Typen einfacher Prozessoren implementiert werden. Der Daten- und Kontrollfluß ist einfach und regulär. D.h. die einzelnen Prozessoren lassen sich in einem regelmäßigen Netz mit nur lokalen Verbindungen anordnen. Es wird extensiv Parallelverarbeitung und das Fließbandprinzip (Pipelining) zur Verarbeitung der Daten benutzt. Typischerweise bewegen sich mehrere Datenströme mit konstanter Geschwindigkeit über vorgegebene Wege im Netz und werden an Stellen, an denen sie sich treffen, parallel verarbeitet. Man stellt sich vor, daß die Rechnung nach einem globalen Takt abläuft. Alle beteiligten Prozessoren arbeiten schrittweise simultan. Zu jedem Zeitpunkt, d.h. in jedem Takt kann ein Prozessor nur mit seinen durch die vorgegebene Geometrie verbundenen Nachbarn kommunizieren. Die beteiligten Prozessoren verarbeiten also einen oder mehrere Datenströme, indem sie rhythmisch pulsierend operieren und Daten aufnehmen, verarbeiten und weiterleiten ähnlich wie das Blut durch die Arterien gepumpt

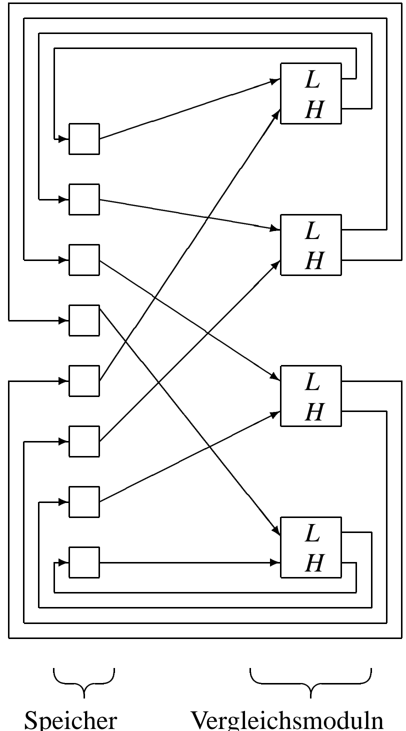


Abbildung 9.25

wird. Diese Analogie hat den Algorithmen und Arrays von Prozessoren den Namen systolisch eingebracht.

Kung und Leiserson zeigen unter anderem, wie man zwei Bandmatrizen mit Bandweite q in einem hexagonalen Array von q^2 Prozessoren miteinander multiplizieren kann. Dabei werden die Datenströme zur Berechnung der Produktmatrix $C = A \cdot B$ so aufeinander abgestimmt, daß die Ergebnismatrix C parallel zur Eingabe von A und B berechnet werden kann.

Wir beschränken uns auf einfachere Geometrien systolischer Netze und zeigen als repräsentatives Beispiel für diese Klasse von Algorithmen, wie eine Matrix-Vektor-Multiplikation auf einem linearen systolischen Array durchgeführt werden kann. Gegeben seien eine Matrix $A = (a_{ij})$ und ein Vektor $x = (x_1, \ldots, x_n)^T$. Die Elemente des Produkts

$$(y_1, \ldots, y_n)^T = A \cdot (x_1, \ldots, x_n)^T$$

lassen sich wie folgt berechnen:

$$\begin{aligned} {y_i}^{(1)} &= 0 \\ {y_i}^{(k+1)} &= {y_i}^{(k)} + a_{ik} x_k \end{aligned}$$

$$y_i \quad = \quad {y_i}^{(n+1)}$$

Denn durch Induktion über k zeigt man leicht, daß ${y_i}^{(k)} = \sum_{j=1}^{k-1} a_{ij} \cdot x_j$, für alle k mit $2 \leq k \leq n+1$, ist.

Häufig ist A eine $n \times n$ Band-Matrix mit Bandweite $w = p+q-1$ und x ein Vektor mit Länge n wie in folgendem Beispiel für $p = 2$ und $q = 3$ (vgl. Abbildung 9.26).

$$\begin{matrix} \\ q=3\left\{\begin{matrix}\\ \\ \\ \end{matrix}\right. \end{matrix} \begin{bmatrix} \overbrace{a_{11} \quad a_{12}}^{p=2} & & & & 0 \\ a_{21} \quad a_{22} & a_{23} & & & \\ a_{31} \quad a_{32} & a_{33} & a_{34} & & \\ \quad\quad a_{42} & a_{43} & a_{44} & a_{45} & \\ & a_{53} \; w & \cdots & & \\ & & \ddots & & \\ \quad 0 & & & & \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \\ \vdots \\ \\ \end{bmatrix}$$

Abbildung 9.26

In diesem Beispiel ist

$$y_i = a_{i(i-2)}x_{i-2} + a_{i(i-1)}x_{i-1} + a_{ii}x_i + a_{i(i+1)}x_{i+1}.$$

Das Matrix-Vektor-Produkt kann nun dadurch berechnet werden, daß man die Elemente von A und x durch ein systolisches Array hindurchschiebt, das aus w linear miteinander verbundenen Prozessoren besteht, die jeweils einen Schritt zur Berechnung des Produkts $A \cdot x$ ausführen. Genauer läßt sich die Rechnung wie folgt beschreiben. Die y_i sind anfangs Null und wandern von rechts nach links, die x_i wandern von links nach rechts und die a_{ij} von oben nach unten wie in Abbildung 9.27.

In jedem geraden Takt wird das nächste y_i von rechts und in jedem ungeraden Takt das nächste x_j von links eingegeben. Die a_{ij} werden abwechselnd auf die geraden und ungeraden Prozessoren eingegeben. Die Datenströme während der ersten vier Takte veranschaulicht die folgende Tabelle 9.2.

Darin sind die von den Prozessoren durchgeführten Rechnungen nicht angegeben. Jedes y_i summiert auf seinem Weg durch das Array von Prozessoren der Reihe nach alle seine Produktterme

$$a_{i(i-2)}x_{i-2}, a_{i(i-1)}x_{i-1}, a_{ii}x_i, a_{i(i+1)}x_{i+1}$$

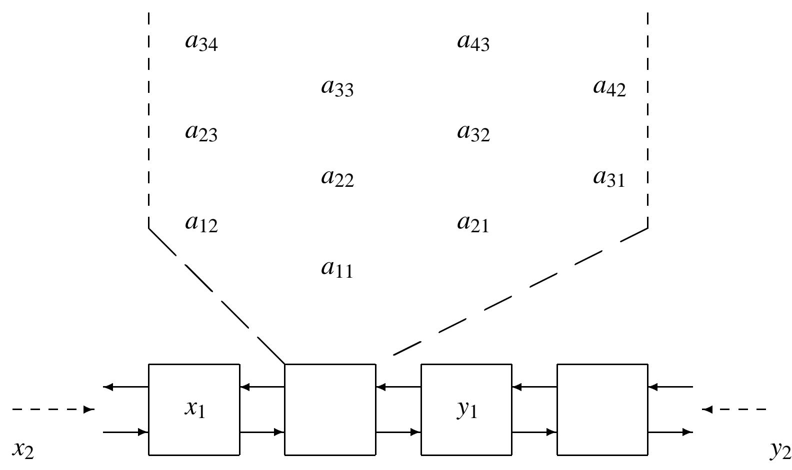


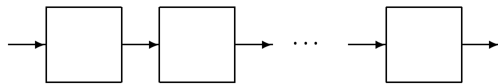
Abbildung 9.27

Zeit/ Takt				
1	x_1	—	y_1	—
2	—	x_1, y_1 a_{11}	—	y_2
3	x_2, y_1 a_{12}	—	x_1, y_2 a_{21}	—
4	—	x_2, y_2 a_{22}	—	x_1, y_3 a_{32}

Tabelle 9.2

auf, bevor es das Array am linken Ende verläßt. Beispielsweise verläßt y_1 das Array im vierten Takt mit Wert $y_1 = a_{11}x_1 + a_{12}x_2$, nachdem der Produktterm $a_{11}x_1$ im zweiten und $a_{12}x_2$ im dritten Takt berechnet wurde. Benachbarte Prozessoren sind jeweils abwechselnd aktiv. Ist $w = p + q - 1$ die Bandweite von A (und ohne Einschränkung w gerade), so werden nach w Takten die Komponenten des Produkts $y = Ax$ am linken Endprozessor ausgegeben, und zwar bei jedem zweiten Takt die nächste Komponente von y. Damit berechnet dieses systolische Array alle n Komponenten des Produkts $y = Ax$ in Zeit $2n + w$.

Die in diesem Beispiel benutzten Prozessoren sind „gedächtnislos“. Denn die jeweils nach links oder rechts weitergegebenen Daten hängen nur von den Eingaben, aber nicht von lokal zwischengespeicherten Werten ab. Im allgemeinen läßt man zu, daß die Prozessoren ein (beschränktes) Speichervermögen haben. Hat man beispielsweise eine lineare Folge von N Prozessoren und hat jeder Prozessor einen lokalen Speicher, der



zwei Schlüssel aufnehmen kann, so kann man mit Hilfe eines solchen N-Prozessor-Vektors $2N$ Schlüssel in Zeit $O(N)$ sortieren. Die $2N$ Schlüssel werden am linken Ende der Reihe nach eingegeben. Ein Prozessor wartet stets, bis er (erstmals) zwei Schlüssel erhalten hat. Im nächsten Takt werden dann gleichzeitig und parallel ausgeführt: Weitergeben des Minimums der gespeicherten zwei Schlüssel an den rechten Nachbarn und Aufnahme des nächsten Schlüssels von links. Schließt man die zu sortierende Folge von Schlüsseln dadurch ab, daß man schließlich nur noch den „fiktiven“ Schlüssel ∞ von links her eingibt, so hat nach insgesamt $4N$ Schritten eine sortierte Schlüsselfolge den N-Prozessor-Vektor verlassen.

9.3 Aufgaben

Aufgabe 9.1

Verändern Sie die Funktion *kmp_search* aus Abschnitt 9.1.2 so, daß sie nicht nur die Position des ersten Vorkommens eines Musters von links in einem gegebenen Text, sondern alle Positionen, an denen das Muster im Text auftritt, liefert.

Aufgabe 9.2

Gegeben sei das Muster `abrakadabra` mit Länge 11. Berechnen Sie für dieses Mu-

ster die Werte $next[j]$ für alle j mit $1 \leq j \leq 11$. Geben Sie ferner die Anzahl der Vergleichsoperationen zwischen den Zeichen (des deutschen Alphabets einschließlich des Leerzeichens und der Satzzeichen) an, die das Verfahren von Knuth-Morris-Pratt ausführt, bis das Muster im Text

```
er sprach abrakadabra, aber ...
```

erstmals gefunden wird.

Aufgabe 9.3

Die Linearität des Verfahrens von Knuth-Morris-Pratt kann man sich anschaulich folgendermaßen klarmachen (vgl. *kmp_search*): Jeder Schritt (jeder Durchgang durch die **until**-Schleife) bewegt entweder den Textzeiger nach rechts oder das Muster. Beides kann jedoch höchstens N-mal geschehen, d.h. die Laufzeit ist linear in N. Um dieses Argument mathematisch umzusetzen, definieren wir eine *Potentialfunktion* $p(i,j) := 2i - j$, die sich aus dem Textzeiger und der Position des Musters ergibt. Zeigen Sie, daß jeder Durchlauf durch die **until**-Schleife das Potential erhöht, und folgern Sie daraus, daß das Verfahren von Knuth-Morris-Pratt lineare Laufzeit hat.

Aufgabe 9.4

Die in diesem Text dargestellte Variante des Verfahrens von Knuth-Morris-Pratt beruht auf einem Array *next*, das folgendermaßen definiert wurde:

$$next[j] := \begin{cases} 1 + \max\{0 \leq k \leq j-1 \mid b_1 \dots b_k = b_{j-k} \dots b_{j-1}\} & \text{falls } j > 1 \\ 0 & \text{falls } j = 1 \end{cases}$$

Dabei wird im Falle eines Mismatches an Stelle j die Information, welches Zeichen an Stelle j gelesen wurde, nicht ausgenutzt. Die folgende Definition des Arrays *next*1 stellt dagegen sicher, daß das nach einem Mismatch mit b_j verglichene Zeichen von diesem verschieden ist. Die Verschiebungen des Musters sind also im allgemeinen größer als bei *next*. Der lineare Platzbedarf bleibt jedoch erhalten.

$$next1[j] := \begin{cases} 1 + \max\{0 \leq k \leq j-1 \mid \\ b_1 \dots b_k = b_{j-k} \dots b_{j-1} \text{ und } b_{k+1} \neq b_j\} & \text{falls } j > 1 \text{ und ein solches} \\ & k \text{ existiert} \\ 0 & \text{sonst} \end{cases}$$

Lösen Sie die folgenden Aufgaben:

a) Berechnen Sie *next1* für das Wort `abrakadabra`.

b) Zeigen Sie, daß sich *next1* in Zeit $O(M)$ berechnen läßt (Hinweis: Benutzen Sie *next*).

Aufgabe 9.5

Berechnen Sie die möglichen Verschiebungen *delta*$-1(a)$ für jedes Zeichen a des deutschen Alphabets einschließlich des Leerzeichens und der Satzzeichen und *delta*$-2(j)$ nach der Vorkommens- und Match-Heuristik für das Muster `abrakadabra` und alle j mit $1 \leq j \leq 10$. Geben Sie ferner die genaue Zahl der Vergleichsoperationen zwischen Zeichen an, die das Verfahren von Boyer-Moore benötigt, um das Muster in dem in Aufgabe 9.2 genannten Text zu finden. Ändern Sie anschließend das Verfahren von Boyer-Moore so ab, daß alle Vorkommen eines Musters im Text gefunden werden.

Aufgabe 9.6

Unter einer *shared-memory* Prozessorarchitektur mit *CRCW* (Concurrent Read Concurrent Write) versteht man eine parallele Anordnung von Prozessoren $P_1, \ldots, P_n$, die sich einen gemeinsamen Speicher teilen und bei der eine beliebige Anzahl von Prozessoren *gleichzeitig* von einer Speicherzelle lesen oder in eine Speicherzelle schreiben können. Ein Algorithmus für diese Architektur ist zulässig, falls zu jedem Zeitpunkt sichergestellt ist, daß

- niemals gleichzeitig ein Prozessor eine Speicherzelle lesen und ein anderer in sie schreiben möchte und,
- falls zwei Prozessoren gleichzeitig in eine Speicherzelle schreiben, so schreiben sie denselben Wert.

a) Entwerfen Sie zunächst einen sequentiellen Algorithmus, der in linearer Zeit für einen gegebenen Punkt p und ein Polygon P mit den Kanten $e_1, \ldots, e_n$ feststellt, ob p innerhalb oder außerhalb von P liegt. (Hinweis: Betrachten Sie die Anzahl der Schnittpunkte eines (horizontalen) Strahls, der in p beginnt, mit den Kanten von P. Sie können davon ausgehen, daß alle Ecken von P eine von p verschiedene y-Koordinate haben.)

b) Entwerfen Sie einen parallelen Algorithmus für das obige Problem, wobei ihnen eine CRCW-Architektur zur Verfügung stehe. Für diesen und den folgenden Aufgabenteil gelte, daß die Anzahl der Prozessoren gleich der Anzahl der Kanten von P sei. Ihr Algorithmus sollte nicht mehr als $O(\log n)$ Schritte benötigen. (Sie können davon ausgehen, daß eine Speicherzelle in der Lage ist, die Beschreibung einer Kante oder eine beliebige ganze Zahl aufzunehmen.)

c) Entwerfen Sie einen parallelen Algorithmus für das obige Problem in einer CRCW-Umgebung, falls P konvex ist. Können Sie eine Laufzeit von $O(1)$ erreichen? Wie lange benötigt man, wenn es nicht erlaubt ist, gleichzeitig in eine Speicherzelle zu schreiben?

Aufgabe 9.7

Entwerfen Sie ein Netzwerk aus n Vergleichsmoduln, das für beliebige Zahlenfolgen der Länge n das Maximum der Zahlen in einer Zeit von $O(\log n)$ bestimmt. Sie können davon ausgehen, daß die Zahlen über n Eingabeleitungen simultan an dem Netz anliegen.

Aufgabe 9.8

Gegeben seien zwei aufsteigend sortierte Folgen $a_1, \ldots, a_n$ und $b_1, \ldots, b_n$, d.h. es gilt für alle $1 \leq i < n$, daß $a_i \leq a_{i+1}$ und $b_i \leq b_{i+1}$. Sei $c_1, \ldots, c_n$ die Folge von Zahlen, die sich durch Verschmelzen der Folgen $a_1, a_3, a_5, \ldots$ und $b_1, b_3, b_5, \ldots$ ergibt, und $d_1, \ldots, d_n$ die resultierende Folge bei Verschmelzung von $a_2, a_4, a_6, \ldots$ und $b_2, b_4, b_6, \ldots$ Zeigen Sie, daß für

$$
\begin{aligned}
e_1 &:= c_1 \\
e_{2i} &:= \min\{c_{i+1}, d_i\} && \text{für } 1 \leq i \leq n-1 \text{ und} \\
e_{2i+1} &:= \max\{c_{i+1}, d_i\} && \text{für } 1 \leq i \leq n-1 \text{ und} \\
e_{2n} &:= d_n
\end{aligned}
$$

gilt: $e_i \leq e_{i+1}$ für $1 \leq i \leq 2n-1$.