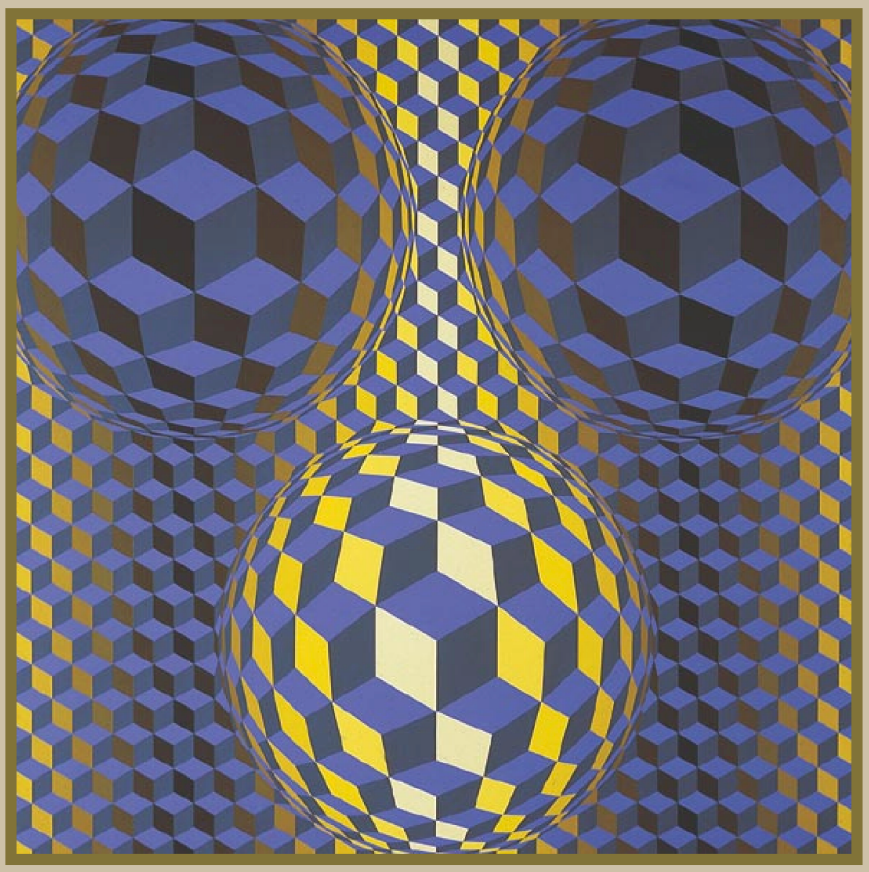
ALGORITHMS of Informatics



volume 3.

ELTE Faculty of Informatics

ALGORITHMS OF INFORMATICS

Volume 3.

APPLICATIONS AND DATA MANAGEMENT

ELTE EÖTVÖS KIADÓ
Budapest, 2006

Editor: Antal Iványi

Authors: Ulrich Tamm (Chapter 13), László Szirmay-Kalos (14), János Demetrovics and Attila Sali (15), Ingo Althöfer and Stefan Schwarz (16), Tibor Gyires and László Lakatos (17), Aurél Galántai and András Jeney (18), Attila Kiss (19)

Validators: Sándor Fridli (Chapter 13), János Vida (14), Attila Kiss (15), Tamás Szántai (16), János Sztrik (18), András Benczúr (19)

ISBN: 963 463 664 0

Published by ELTE EÖTVÖS KIADÓ
Budapest, Szerb utca 21–23.
Hungary
Telephone/facsimile: 411-6740
Internet: http://www.elte.hu/szervezet/eotvos_kiado.html
E-mail: eotvoskiado@ludens.elte.hu

Responsible publisher: András Pándi
Cover design: Antal Iványi

Printed and bound by ???

Introduction

13. Compression and Decompression

Algorithms for data compression usually proceed as follows. They encode a text over some finite alphabet into a sequence of bits, hereby exploiting the fact that the letters of this alphabet occur with different frequencies. For instance, an "e" occurs more frequently than a "q" and will therefore be assigned a shorter codeword. The quality of the compression procedure is then measured in terms of the average codeword length.

So the underlying model is probabilistic, namely we consider a finite alphabet and a probability distribution on this alphabet, where the probability distribution reflects the (relative) frequencies of the letters. Such a pair – an alphabet with a probability distribution – is called a source. We shall first introduce some basic facts from Information Theory. Most important is the notion of entropy, since the source entropy characterizes the achievable lower bounds for compressibility.

The source model to be best understood, is the discrete memoryless source. Here the letters occur independently of each other in the text. The use of prefix codes, in which no codeword is the beginning of another one, allows to compress the text down to the entropy of the source. We shall study this in detail. The lower bound is obtained via Kraft's inequality, the achievability is demonstrated by the use of Huffman codes, which can be shown to be optimal.

There are some assumptions on the discrete memoryless source, which are not fulfilled in most practical situations. Firstly, usually this source model is not realistic, since the letters do not occur independently in the text. Secondly, the probability distribution is not known in advance. So the coding algorithms should be universal for a whole class of probability distributions on the alphabet. The analysis of such universal coding techniques is much more involved than the analysis of the discrete memoryless source, such that we shall only present the algorithms and do not prove the quality of their performance. Universal coding techniques mainly fall into two classes.

Statistical coding techniques estimate the probability of the next letters as accurately as possible. This process is called modelling of the source. Having enough information about the probabilities, the text is encoded, where usually arithmetic coding is applied. Here the probability is represented by an interval and this interval will be encoded.

Dictionary–based algorithms store patterns, which occurred before in the text, in a dictionary and at the next occurrence of a pattern this is encoded via its position in the dictionary. The most prominent procedure of this kind is due to Ziv and Lempel.

We shall also present a third universal coding technique which falls in neither of these

two classes. The algorithm due to Burrows and Wheeler has become quite prominent in recent years, since implementations based on it perform very well in practice.

All algorithms mentioned so far are lossless, i. e., there is no information lost after decoding. So the original text will be recovered without any errors. In contrast, there are lossy data compression techniques, where the text obtained after decoding does not completely coincide with the original text. Lossy compression algorithms are used in applications like image, sound, video, or speech compression. The loss should, of course, only marginally effect the quality. For instance, frequencies not realizable by the human eye or ear can be dropped. However, the understanding of such techniques requires a solid background in image, sound or speech processing, which would be far beyond the scope of this paper, such that we shall illustrate only the basic concepts behind image compression algorithms such as JPEG.

We emphasize here the recent developments such as the Burrows–Wheeler transform and the context–tree weighting method. Rigorous proofs will only be presented for the results on the discrete memoryless source which is best understood but not a very realistic source model in practice. However, it is also the basis for more complicated source models, where the calculations involve conditional probabilities. The asymptotic computational complexity of compression algorithms is often linear in the text length, since the algorithms simply parse through the text. However, the running time relevant for practical implementations is mostly determined by the constants as dictionary size in Ziv-Lempel coding or depth of the context tree, when arithmetic coding is applied. Further, an exact analysis or comparison of compression algorithms often heavily depends on the structure of the source or the type of file to be compressed, such that usually the performance of compression algorithms is tested on benchmark files. The most well-known collections of benchmark files are the Calgary Corpus and the Canterbury Corpus.

13.1. Facts from information theory

13.1.1. The discrete memoryless source

The source model discussed throughout this chapter is the ***Discrete Memoryless Source*** (DMS). Such a source is a pair $(\mathcal{X}, P)$, where $\mathcal{X} = \{1, \ldots, a\}$ is a finite alphabet and $P = (P(1), \ldots, P(a))$ is a probability distribution on $\mathcal{X}$. A discrete memoryless source can also be described by a random variable X, where $\text{Prob}(X = x) = P(x)$ for all $x \in \mathcal{X}$. A word $x^n = (x_1 x_2 \ldots x_n) \in \mathcal{X}^n$ is the realization of the random variable $(X_1 \ldots X_n)$, where the X_i-s are identically distributed and independent of each other. So the probability $P^n(x_1 x_2 \ldots x_n) = P(x_1) \cdot P(x_2) \cdot \cdots \cdot P(x_n)$ is the product of the probabilities of the single letters.

Estimations for letter probabilities in natural languages are obtained by statistical methods. If we consider the English language and choose the latin alphabet with an additional symbol for Space and punctuation marks for $\mathcal{X}$, the probability distribution can be derived from the frequency table in 13.1 which is obtained from the copy–fitting tables used by professional printers. So $P(A) = 0.064$, $P(B) = 0.014$, etc.

Observe that this source model is often not realistic. For instance, in English texts e.g. the combination ‘th’ occurs more often than ‘ht’. This could not be the case if an English text was produced by a discrete memoryless source, since then $P(th) = P(t) \cdot P(h) = P(ht)$.

A	64	H	42	N	56	U	31
B	14	I	63	O	56	V	10
C	27	J	3	P	17	W	10
D	35	K	6	Q	4	X	3
E	100	L	35	R	49	Y	18
F	20	M	20	S	56	Z	2
G	14	T	71				

Space/Punctuation mark 166

Figure 13.1. Frequency of letters in 1000 characters of English

In the discussion of the communication model it was pointed out that the encoder wants to compress the original data into a short sequence of binary digits, hereby using a binary code, i. e., a function $c : \mathcal{X} \longrightarrow \{0,1\}^* = \bigcup_{n=0}^{\infty} \{0,1\}^n$. To each element $x \in \mathcal{X}$ a codeword $c(x)$ is assigned. The aim of the encoder is to minimize the average length of the codewords. It turns out that the best possible data compression can be described in terms of the ***entropy*** $H(P)$ of the probability distribution P. The entropy is given by the formula

$$H(P) = -\sum_{x \in \mathcal{X}} P(x) \cdot \lg P(x)$$

where the logarithm is to the base 2. We shall also use the notation $H(x)$ according to the interpretation of the source as a random variable.

13.1.2. Prefix codes

A ***code*** (of variable length) is a function $c : \mathcal{X} \longrightarrow \{0,1\}^*$, $\mathcal{X} = \{1, \ldots, a\}$. Here $\{c(1), c(2), \ldots, c(a)\}$ is the set of ***codewords***, where for $x = 1, \ldots, a$ the codeword is $c(x) = (c_1(x), c_2(x), \ldots, c_{L(x)}(x))$ where $L(x)$ denotes the ***length*** of $c(x)$, i. e., the number of bits used to present $c(x)$.

In the following example some binary codes for the latin alphabet are presented. (SP=Space/punctuation mark)

$\overline{c} : a \longrightarrow 1, b \longrightarrow 10, c \longrightarrow 100, d \longrightarrow 1000, \ldots, z \longrightarrow \underbrace{10\ldots0}_{26}, SP \longrightarrow \underbrace{10\ldots0}_{27}.$

$\hat{c} : a \longrightarrow 00000, b \longrightarrow 00001, c \longrightarrow 00010, \ldots, z \longrightarrow 11001, SP \longrightarrow 11010$. So $\hat{c}(x)$ is the binary representation of the position of letter x in the alphabetic order.

$\check{c} : a \longrightarrow 0, b \longrightarrow 00, c \longrightarrow 1, \ldots$ (the further codewords are not important for the following discussion).

The last code presented has an undesirable property. Observe that the sequence 00 could either be decoded as b or as aa. Hence the messages encoded using this code are not uniquely decipherable.

A code c is ***uniquely decipherable*** (UDC) , if every word in $\{0,1\}^*$ is representable by at most one sequence of codewords.

Code $\overline{c}$ is uniquely decipherable, since the number of 0s between two 1s determines the next letter in a message encoded using $\overline{c}$. Code $\hat{c}$ is uniquely decipherable, since every

letter is encoded with exactly five bits. Hence the first five bits of a sequence of binary digits are decoded as the first letter of the original text, the bits 6 to 10 as the second letter, etc.

A code c is a ***prefix code,*** if for any two codewords $c(x)$ and $c(y)$, $x \neq y$, with $L(x) \leq L(y)$ holds $(c_1(x), c_2(x), \ldots, c_{L(x)}(x)) \neq (c_1(y), c_2(y), \ldots, c_{L(x)}(y))$. So $c(x)$ and $c(y)$ differ in at least one of the first $L(x)$ components.

Messages encoded using a prefix code are uniquely decipherable. The decoder proceeds by reading the next letter until a codeword $c(x)$ is formed. Since $c(x)$ cannot be the beginning of another codeword, it must correspond to letter $x \in \mathcal{X}$. Now the decoder continues until another codeword is formed. The process may be repeated until the end of the message. So after having found codeword $c(x)$ the decoder instantaneously knows that $x \in \mathcal{X}$ is the next letter of the message. Because of this property a prefix code is also denoted as instantaneous code. Observe that code $\overline{c}$ is not instantaneous, since every codeword is the beginning of the following codewords.

The criterion for data compression is to minimize the average length of the codewords. So if we are given a source $(\mathcal{X}, P)$, where $\mathcal{X} = \{1, \ldots, a\}$ and $P = (P(1), P(2), \ldots, P(a))$ is a probability distribution on $\mathcal{X}$, the ***average length*** $\overline{L}(c)$ is defined by

$$\overline{L}(c) = \sum_{x \in \mathcal{X}} P(x) \cdot L(x) \ .$$

If in English texts all letters (incl. Space/punctuation mark) occured with the same frequency, then code $\overline{c}$ would have an average length of $\frac{1}{27}(1 + 2 + \cdots + 27) = \frac{1}{27} \cdot \frac{27 \cdot 28}{2} = 14$. Hence code $\hat{c}$ with an average length of 5 would be more appropriate in this case. From the frequency table of the previous section we know that the occurrence of the letters in English texts cannot be modelled by the uniform distribution. In this case it is possible to find a better code by assigning short codewords to letters with high probability as demonstrated by the following prefix code c with average length $\overline{L}(c) = 3 \cdot 0.266 + 4 \cdot 0.415 + 5 \cdot 0.190 + 6 \cdot 0.101 + 7 \cdot 0.016 + 8 \cdot 0.012 = 4.222$.

$a \longrightarrow 0110,$	$b \longrightarrow 010111,$	$c \longrightarrow 10001,$	$d \longrightarrow 01001\ ,$
$e \longrightarrow 110,$	$f \longrightarrow 11111,$	$g \longrightarrow 111110,$	$h \longrightarrow 00100\ ,$
$i \longrightarrow 0111,$	$j \longrightarrow 11110110,$	$k \longrightarrow 1111010,$	$l \longrightarrow 01010\ ,$
$m \longrightarrow 001010,$	$n \longrightarrow 1010,$	$o \longrightarrow 1001,$	$p \longrightarrow 010011\ ,$
$q \longrightarrow 01011010,$	$r \longrightarrow 1110,$	$s \longrightarrow 1011,$	$t \longrightarrow 0011\ ,$
$u \longrightarrow 10000,$	$v \longrightarrow 0101100,$	$w \longrightarrow 001011,$	$x \longrightarrow 01011011\ ,$
$y \longrightarrow 010010,$	$z \longrightarrow 11110111,$	$SP \longrightarrow 000\ .$	

We can still do better, if we do not encode single letters, but blocks of n letters for some $n \in N$. In this case we replace the source $(\mathcal{X}, P)$ by $(\mathcal{X}^n, P^n)$ for some $n \in N$. Remember that $P^n(x_1 x_2 \ldots x_n) = P(x_1) \cdot P(x_2) \cdot \cdots \cdot P(x_n)$ for a word $(x_1 x_2 \ldots x_n) \in \mathcal{X}^n$, since the source is memoryless. If e.g. we are given an alphabet with two letters, $\mathcal{X} = \{a, b\}$ and $P(a) = 0.9$, $P(b) = 0.1$, then the code c defined by $c(a) = 0$, $c(b) = 1$ has average length $\overline{L}(c) = 0.9 \cdot 1 + 0.1 \cdot 1 = 1$. Obviously we cannot find a better code. The combinations of two letters now have the following probabilities:

$$P^2(aa) = 0.81, \quad P^2(ab) = 0.09, \quad P^2(ba) = 0.09, \quad P^2(bb) = 0.01\ .$$

The prefix code c^2 defined by

$$c^2(aa) = 0, \quad c^2(ab) = 10, \quad c^2(ba) = 110, \quad c^2(bb) = 111$$

has average length $\overline{L}(c^2) = 1 \cdot 0.81 + 2 \cdot 0.09 + 3 \cdot 0.09 + 3 \cdot 0.01 = 1.29$. So $\frac{1}{2}\overline{L}(c^2) = 0.645$ could be interpreted as the average length the code c^2 requires per letter of the alphabet $\mathcal{X}$. When we encode blocks of n letters we are interested in the behaviour of

$$L(n, P) = \min_{cUDC} \frac{1}{n} \sum_{(x_1 \ldots x_n) \in \mathcal{X}^n} P^n(x_1 \ldots x_n) L(x_1 \ldots x_n) = \min_{cUDC} \overline{L}(c) \; .$$

It follows from the noiseless coding theorem, which is stated in the next section, that $\lim_{n \longrightarrow \infty} L(n, P) = H(P)$, the entropy of the source $(\mathcal{X}, P)$.

In our example for the English language we have $H(P) \approx 4.19$. So the code presented above, where only single letters are encoded, is already nearly optimal in respect of $L(n, P)$. Further compression is possible if we consider the dependencies between the letters.

13.1.3. Kraft's inequality and the noiseless coding theorem

We shall now introduce a necessary and sufficient condition for the existence of a prefix code for $\mathcal{X} = \{1, \ldots, a\}$ with prescribed word lengths $L(1), \ldots, L(a)$.

Theorem 13.1 (Kraft's inequality). *Let $\mathcal{X} = \{1, \ldots, a\}$. A prefix code $c : \mathcal{X} \longrightarrow \{0, 1\}^*$ with word lengths $L(1), \ldots, L(a)$ exists, if and only if*

$$\sum_{x \in \mathcal{X}} 2^{-L(x)} \leq 1.$$

Proof. The central idea is to interpret the codewords as nodes of a rooted binary tree with depth $T = \max_{x \in \mathcal{X}}\{L(x)\}$. The tree is required to be complete (every path from the root to a leaf has length T) and regular (every inner node has outdegree 2). The example in Figure 13.2 for $T = 3$ may serve as an illustration.

So the nodes with distance n from the root are labelled with the words $x^n \in \{0, 1\}^n$. The upper successor of $x_1 x_2 \ldots x_n$ is labelled $x_1 x_2 \ldots x_n 0$, its lower successor is labelled $x_1 x_2 \ldots x_n 1$.

The ***shadow*** of a node labelled by $x_1 x_2 \ldots x_n$ is the set of all the leaves which are labelled by a word (of length T) beginning with $x_1 x_2 \ldots x_n$. In other words, the shadow of $x_1 \ldots x_n$ consists of the leaves labelled by a sequence with prefix $x_1 \ldots x_n$. In our example $\{000, 001, 010, 011\}$ is the shadow of the node labelled by 0.

Now assume that we are given a prefix code with word lengths $L(1), \ldots, L(a)$. Every codeword corresponds to a node in the binary tree of depth T. Observe that the shadows of any two codewords are disjoint. If this was not the case, we could find a word $x_1 x_2 \ldots x_T$, which has as prefix two codewords of length s and t, say (w.l.o.g. $s < t$). But these codewords are $x_1 x_2 \ldots x_t$ and $x_1 x_2 \ldots x_s$ which obviously is a prefix of the first one.

The shadow of codeword $c(x)$ has size $2^{T - L(x)}$. There are 2^T words of length T. For the sum of the shadow sizes follows $\sum_{x \in \mathcal{X}} 2^{T - L(x)} \leq 2^T$, since none of these words can be a

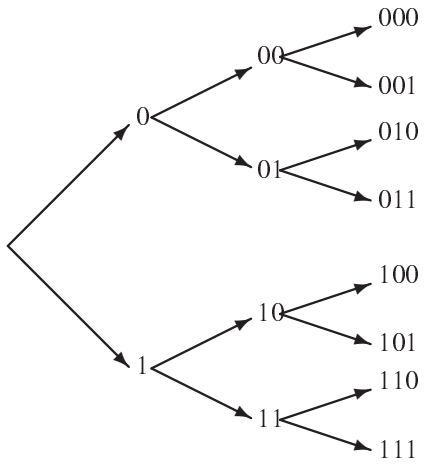


Figure 13.2. Example of a code tree.

member of two shadows. Division by 2^T yields the desired inequality $\sum_{x\in\mathcal{X}} 2^{-L(x)} \le 1$.

Conversely, suppose we are given positive integers $L(1), \dots, L(a)$. We further assume that $L(1) \le L(2) \le \cdots \le L(a)$. As first codeword $c(1) = \underbrace{00\dots0}_{L(1)}$ is chosen. Since $\sum_{x\in\mathcal{X}} 2^{T-L(x)} \le 2^T$, we have $2^{T-L(1)} < 2^T$ (otherwise only one letter has to be encoded). Hence there are some nodes left on the T–th level, which are not in the shadow of $c(1)$. We pick the first of these remaining nodes and go back $T - L(2)$ steps in the direction of the root. Since $L(2) \ge L(1)$, we shall find a node labelled by a sequence of $L(2)$ bits, which is not a prefix of $c(1)$. So we can choose this sequence as $c(2)$. Now again, either $a = 2$, and we are ready, or by the hypothesis $2^{T-L(1)} + 2^{T-L(2)} < 2^T$ and we can find a node on the T–th level, which is not contained in the shadows of $c(1)$ and $c(2)$. We find the next codeword as shown above. The process can be continued until all codewords are assigned. ■

Kraft's inequality gives a necessary and sufficient condition for the existence of a prefix code with codewords of lengths $L(1), \dots, L(a)$. In the following theorem it is shown that this condition is also necessary for the existence of a uniquely decipherable code. This can be interpreted in such a way that it is sufficient to consider only prefix codes, since one cannot expect a better performance by any other uniquely decipherable code.

Theorem 13.2 (Kraft's inequality for uniquely decipherable codes). *A uniquely decipherable code with prescribed word lengths $L(1), \dots, L(a)$ exists, if and only if*

$$\sum_{x\in\mathcal{X}} 2^{-L(x)} \le 1.$$

Proof. Since every prefix code is uniquely decipherable, the sufficiency part of the proof is immediate. Now observe that $\sum_{x\in\mathcal{X}} 2^{-L(x)} = \sum_{j=1}^{T} w_j\, 2^{-j}$, where w_j is the number of codewords with length j in the uniquely decipherable code and T again denotes the maximal word

length. The s–th power of this term can be expanded as

$$\left(\sum_{j=1}^{T} w_j\, 2^{-j}\right)^s = \sum_{k=s}^{T\cdot s} N_k\, 2^{-k}.$$

Here $N_k = \sum_{i_1+\cdots+i_s=k} w_{i_1}\ldots w_{i_s}$ is the total number of messages whose coded representation is of length k. Since the code is uniquely decipherable, to every sequence of k letters corresponds at most one possible message. Hence $N_k \le 2^k$ and $\sum_{k=s}^{T\cdot s} N_k\, 2^{-k} \le \sum_{k=s}^{T\cdot s} 1 = T\cdot s - s + 1 \le T\cdot s$. Taking s–th root this yields $\sum_{j=1}^{T} w_j\, 2^{-j} \le (T\cdot s)^{\frac{1}{s}}$.

Since this inequality holds for any s and $\lim_{s\longrightarrow\infty} (T\cdot s)^{\frac{1}{s}} = 1$, we have the desired result

$$\sum_{j=1}^{T} w_j\, 2^{-j} = \sum_{x\in\mathcal{X}} 2^{-L(x)} \le 1\ .$$

■

Theorem 13.3 (Noiseless coding theorem). *For a source $(\mathcal{X}, P)$, $\mathcal{X} = \{1,\ldots,a\}$, it is always possible to find a uniquely decipherable code $c : \mathcal{X} \longrightarrow \{01,\}^*$ with an average length of*

$$H(P) \le L_{\min}(P) < H(P) + 1.$$

Proof. Let $L(1),\ldots,L(a)$ denote the codeword lengths of an optimal uniquely decipherable code. Now we define a probability distribution Q on $\mathcal{X} = \{1,\ldots,a\}$ by $Q(x) = \frac{2^{-L(x)}}{r}$ for $x \in \mathcal{X}$, where $r = \sum_{x=1}^{a} 2^{-L(x)}$. By Kraft's inequality $r \le 1$.

For two probability distributions P and Q on $\mathcal{X}$ the ***I–divergence*** $D(P\|Q)$ is defined by

$$D(P\|Q) = \sum_{x\in\mathcal{X}} P(x) \lg \frac{P(x)}{Q(x)}$$

I-divergence is a good measure for the distance of two probability distributions. Especially, always the I–divergence $D(P\|Q) \ge 0$. So for any probability distribution P

$$D(P\|Q) = -H(P) - \sum_{x\in\mathcal{X}} P(x)\cdot \lg(2^{-L(x)}\cdot r^{-1}) \ge 0.$$

From this it follows that

$$\begin{aligned} H(P) &\le -\sum_{x\in\mathcal{X}} P(x)\cdot \lg(2^{-L(x)}\cdot r^{-1}) \\ &= \sum_{x\in\mathcal{X}} P(x)\cdot L(x) - \sum_{x\in\mathcal{X}} P(x)\cdot \lg r^{-1} = L_{\min}(P) + \lg r. \end{aligned}$$

Since $r \le 1$, $\lg r \le 0$ and hence $L_{\min}(P) \ge H(P)$.

In order to prove the right-hand side of the noiseless coding theorem for $x = 1,\ldots,a$ we define $L'(x) = \lceil -\lg P(x)\rceil$. Observe that $-\lg P(x) \le L'(x) < -\lg P(x) + 1$ and hence

x	$P(x)$	$Q(x)$	$\overline{Q}(x)$	$\lceil \lg \frac{1}{P(x)} \rceil$	$c_S(x)$	$c_{SFE}(x)$
1	0.25	0	0.125	2	00	001
2	0.2	0.25	0.35	3	010	0101
3	0.11	0.45	0.505	4	0111	10001
4	0.11	0.56	0.615	4	1000	10100
5	0.11	0.67	0.725	4	1010	10111
6	0.11	0.78	0.835	4	1100	11010
7	0.11	0.89	0.945	4	1110	11110
				$\overline{L}$	3.3	4.3

Figure 13.3. Example of Shannon code and Shannon-Fano-Elias code.

$P(x) \geq 2^{-L'(x)}$.

So $1 = \sum_{x \in \mathcal{X}} P(x) \geq \sum_{x \in \mathcal{X}} 2^{-L'(x)}$ and from Kraft's Inequality we know that there exists a uniquely decipherable code with word lengths $L'(1), \ldots, L'(a)$. This code has an average length of

$$\sum_{x \in \mathcal{X}} P(x) \cdot L'(x) < \sum_{x \in \mathcal{X}} P(x)(-\lg P(x) + 1) = H(P) + 1.$$

■

13.1.4. Shannon-Fano-Elias codes and the Shannon-Fano algorithm

In the proof of the noiseless coding theorem it was explicitly shown how to construct a prefix code c to a given probability distribution $P = (P(1), \ldots, P(a))$. The idea was to assign to each $x \in \{1, \ldots, a\}$ a codeword of length $L(x) = \lceil \lg \frac{1}{P(x)} \rceil$ by choosing an appropriate vertex in the tree introduced. However, this procedure does not always yield an optimal code. If e.g. we are given the probability distribution $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, we would encode $1 \longrightarrow 00$, $2 \longrightarrow 01$, $3 \longrightarrow 10$ and thus achieve an average codeword length of 2. But the code with $1 \longrightarrow 00$, $2 \longrightarrow 01, 3 \longrightarrow 1$ has only average length of $\frac{5}{3}$.

Shannon gave an explicit procedure for obtaining codes with codeword lengths $\lceil \lg \frac{1}{P(x)} \rceil$ using the binary representation of cumulative probabilities (Shannon remarked this procedure was originally due to Fano). The elements of the source are ordered according to increasing probabilities $P(1) \geq P(2) \geq \cdots \geq P(a)$. Then codeword $c_S(x)$ consists of the first $\lceil \lg \frac{1}{P(x)} \rceil$ bits of the binary expansion of the sum $Q(x) = \sum_{j<x} P(j)$.

This procedure was further developed by Elias . The elements of the source now may occur in any order. The ***Shannon-Fano-Elias code*** has $c_{SFE}(x)$ the first $\lceil \lg \frac{1}{P(x)} \rceil + 1$ bits of the binary expansion of the sum $\overline{Q}(x) = \sum_{j<x} P(j) + \frac{1}{2}P(x)$ as codewords.

We shall illustrate these procedures by the example in Figure 13.3.

A more efficient procedure is also due to Shannon and Fano. The ***Shannon-Fano algorithm*** will be illustrated by the same example in Figure 13.4:

The messages are first written in order of nonincreasing probabilities. Then the message set is partitioned into two most equiprobable subsets X_0 and X_1. A 0 is assigned to each

x	$P(x)$	$c(x)$	$L(x)$
1	0.25	00	2
2	0.2	01	2
3	0.11	100	3
4	0.11	101	3
5	0.11	110	3
6	0.11	1110	4
7	0.11	1111	4
		$\overline{L}(c)$	2.77

Figure 13.4. Example of the Shannon-Fano algorithm.

message contained in the first subset and a 1 to each of the remaining messages. The same procedure is repeated for subsets of X_0 and X_1; that is, X_0 will be partitioned into two subsets X_{00} and X_{01}. Now the code word corresponding to a message contained in X_{00} will start with 00 and that corresponding to a message in X_{01} will begin with 01. This procedure is continued until each subset contains only one message.

However, this algorithm does not yield an optimal code in general, since the prefix code $1 \longrightarrow 01, 2 \longrightarrow 000, 3 \longrightarrow 001, 4 \longrightarrow 110, 5 \longrightarrow 111, 6 \longrightarrow 100, 7 \longrightarrow 101$ has an average length of 2.75.

13.1.5. The Huffman coding algorithm

The ***Huffman coding algorithm*** is a recursive procedure which we shall illustrate with the same example as for the Shannon-Fano algorithm in Figure 13.5 with $p_x = P(x)$ and $c_x = c(x)$. The source is successively reduced by one element. In each reduction step we add up the two smallest probabilities and insert their sum $P(a) + P(a-1)$ in the increasingly ordered sequence $P(1) \geq \cdots \geq P(a-2)$, thus obtaining a new probability distribution P' with $P'(1) \geq \cdots \geq P'(a-1)$. Finally we arrive at a source with two elements ordered according to their probabilities. The first element is assigned a 0, the second element a 1. Now we again "blow up" the source until the original source is restored. In each step $c(a-1)$ and $c(a)$ are obtained by appending 0 or 1, respectively, to the codeword corresponding to $P(a) + P(a-1)$.

Correctness

The following theorem demonstrates that the Huffman coding algorithm always yields a prefix code optimal with respect to the average codeword length.

Theorem 13.4 *We are given a source $(\mathcal{X}, P)$, where $\mathcal{X} = \{1, \ldots, a\}$ and the probabilities are ordered non–increasingly: $P(1) \geq P(2) \geq \cdots \geq P(a)$. A new probability distribution is defined by*

$$P' = (P(1), \ldots, P(a-2), P(a-1) + P(a)).$$

Let $c' = (c'(1), c'(2), \ldots, c'(a-1))$ be an optimal prefix code for P'. Now we define a code c for the distribution P by

p_1	0.25	p_1	0.25	p_1	0.25	P_{23}	0.31	p_{4567}	0.44	p_{123}	0.56
p_2	0.2	p_{67}	0.22	p_{67}	0.22	p_1	0.25	p_{23}	0.31	p_{4567}	0.44
p_3	0.11	p_2	0.2	p_{45}	0.22	p_{67}	0.22	p_1	0.25		
p_4	0.11	p_3	0.11	p_2	0.2	p_{45}	0.22				
p_5	0.11	p_4	0.11	p_3	0.11						
P_6	0.11	p_5	0.11								
p_7	0.11										

C_{123}	0	c_{4567}	1	c_{23}	00	c_1	01	c_1	01	c_1	01
c_{4567}	1	c_{23}	00	c_1	01	c_{67}	10	c_{67}	10	c_2	000
		c_1	01	c_{67}	10	c_{45}	11	c_2	000	c_3	001
				c_{45}	11	c_2	000	c_3	001	c_4	110
						c_3	001	c_4	110	c_5	111
								c_5	111	c_6	100
										c_7	101

Figure 13.5. Example of a Huffman code.

$$\begin{aligned} c(x) &= c'(x) \text{ for } x = 1, \ldots, a-2, \\ c(a-1) &= c'(a-1)0, \\ c(a) &= c'(a-1)1. \end{aligned}$$

In this case c is an optimal prefix code for P and $L_{\min}(P) - L_{\min}(P') = p(a-1) + p(a)$.

Proof. For a probability distribution P on $\mathcal{X} = \{1, \ldots, a\}$ with $P(1) \geq P(2) \geq \cdots \geq P(a)$ there exists an optimal prefix code c with

i) $L(1) \leq L(2) \leq \cdots \leq L(a)$

ii) $L(a-1) = L(a)$

iii) $c(a-1)$ and $c(a)$ differ exactly in the last position.

This holds, since:

i) Assume that there are $x, y \in \mathcal{X}$ with $P(x) \geq P(y)$ and $L(x) > L(y)$. In this case the code c' obtained by interchanging codewords $c(x)$ and $c(y)$ has average length $\overline{L}(c') \leq \overline{L}(c)$, since

$$\begin{aligned} \overline{L}(c') - \overline{L}(c) &= P(x) \cdot L(y) + P(y) \cdot L(x) - P(x) \cdot L(x) - P(y) \cdot L(y) \\ &= (P(x) - P(y)) \cdot (L(y) - L(x)) \leq 0 \end{aligned}$$

ii) Assume we are given a code c' with $L(1) \leq \cdots \leq L(a-1) < L(a)$. Because of the prefix property we may drop the last $L(a) - L(a-1)$ bits of $c'(a)$ and thus obtain a new code c with $L(a) = L(a-1)$.

iii) If no two codewords of maximal length agree in all places but the last, then we may drop the last digit of all such codewords to obtain a better code.

Now we are ready to prove the statement from the theorem. From the definition of c and c' we have

$$L_{\min}(P) \leq \overline{L}(c) = \overline{L}(c') + p(a-1) + p(a).$$

Now let c'' be an optimal prefix code with the properties ii) and iii) from the preceding lemma. We define a prefix code c''' for

$$P' = (P(1), \ldots, P(a-2), P(a-1) + P(a))$$

by $c'''(x) = c''(x)$ for $x = 1, \ldots, a-2$ and $c'''(a-1)$ is obtained by dropping the last bit of $c''(a-1)$ or $c''(a)$.

Now

$$\begin{aligned} L_{\min}(P) &= \overline{L}(c'') = \overline{L}(c''') + P(a-1) + P(a) \\ &\geq L_{\min}(P') + P(a-1) + P(a) \end{aligned}$$

and hence $L_{\min}(P) - L_{\min}(P') = P(a-1) + P(a)$, since $\overline{L}(c') = L_{\min}(P')$.

■

Analysis

If a denotes the size of the source alphabet, the Huffman coding algorithm needs $a-1$ additions and $a-1$ code modifications (appending 0 or 1). Further we need $a-1$ insertions, such that the total complexity can be roughly estimated to be $O(a \lg a)$. However, observe that with the noiseless coding theorem, the quality of the compression rate can only be improved by jointly encoding blocks of, say, k letters, which would result in a Huffman code of size a^k for the source $\mathcal{X}^k$. So, the price of better compression is a rather drastic increase in complexity. Further, the codewords for all a^k letters have to be stored. Encoding a sequence of n letters can therefore be done in $O(\frac{n}{k} \cdot (a^k \lg a^k))$ steps.

Exercises

13.1-1 Show that the code $c : \{a, b\} \longrightarrow \{0, 1\}^*$ with $c(a) = 0$ and $c(b) = \underbrace{0 \ldots 01}_{n}$ is uniquely decipherable but not instantaneous for any $n > 0$.

13.1-2 Compute the entropy of the source $(\mathcal{X}, P)$, with $\mathcal{X} = \{1, 2\}$ and $P = (0.8, 0.2)$.

13.1-3 Find the Huffman codes and the Shannon-Fano codes for the sources $(\mathcal{X}^n, P^n)$ with $(\mathcal{X}, P)$ as in the previous exercise for $n = 1, 2, 3$ and calculate their average codeword lengths.

13.1-4 Show that $0 \leq H(P) \leq \lg |\mathcal{X}|$ for every source.

13.1-5 Show that the redundancy $\rho(c) = \overline{L}(c) - H(P)$ of a prefix code c for a source with probability distribution P can be expressed as a special I–divergence.

13.1-6 Show that the I-divergence $D(P\|Q) \geq 0$ for all probability distributions P and Q over some alphabet $\mathcal{X}$ with equality exactly if $P = Q$, but that the I–divergence is not a metric.

13.2. Arithmetic coding and modelling

In statistical coding techniques as Shannon-Fano or Huffman coding the probability distribution of the source is modelled as accurately as possible and then the words are encoded such that a higher probability results in a shorter codeword length.

We know that Huffman codes are optimal with respect to the average codeword length. However, the entropy is approached by increasing the block length. On the other hand, for long blocks of source symbols, Huffman coding is a rather complex procedure, since it requires the calculation of the probabilities of all sequences of the given block length and the construction of the corresponding complete code.

For compression techniques based on statistical methods often ***arithmetic coding*** is preferred. Arithmetic coding is a straightforward extension of the Shannon-Fano-Elias code. The idea is to represent a probability by an interval. In order to do so, the probabilities have to be calculated very accurately. This process is denoted as ***modelling*** of the source . So statistical compression techniques consist of two stages: modelling and coding. As just mentioned, coding is usually done by arithmetic coding. The different algorithms like, for instance, DCM (Discrete Markov Coding) and PPM (Prediction by Partial Matching) vary in the way of modelling the source. We are going to present the context–tree weighting method, a transparent algorithm for the estimation of block probabilities due to Willems, Shtarkov, and Tjalkens, which also allows a straightforward analysis of the efficiency.

13.2.1. Arithmetic coding

The idea behind arithmetic coding is to represent a message $x^n = (x_1 \dots x_n)$ by interval $I(x^n) = [Q^n(x^n), Q^n(x^n) + P^n(x^n))$, where $Q^n(x^n) = \sum_{y^n < x^n} P^n(y^n)$ is the sum of the probabilities of those sequences which are smaller than x^n in lexicographic order.

A codeword $c(x^n)$ assigned to message x^n also corresponds to an interval. Namely, we identify codeword $c = c(x^n)$ of length $L = L(x^n)$ with interval $J(c) = [bin(c), bin(c) + 2^{-L})$, where $bin(c)$ is the binary expansion of the nominator in the fraction $\frac{c}{2^L}$. The special choice of codeword $c(x^n)$ will be obtained from $P^n(x^n)$ and $Q^n(x^n)$ as follows:

$$L(x^n) = \lceil \lg \frac{1}{P^n(x^n)} \rceil + 1, \qquad bin(c) = \lceil Q^n(x^n) \cdot 2^{L(x^n)} \rceil.$$

So message x^n is encoded by a codeword $c(x^n)$, whose interval $J(x^n)$ is inside interval $I(x^n)$.

Let us illustrate arithmetic coding by the following example of a discrete memoryless source with $P(1) = 0.1$ and $n = 2$.

x^n	$P^n(x^n)$	$Q^n(x^n)$	$L(x^n)$	$c(x^n)$
00	0.81	0.00	2	00
01	0.09	0.81	5	11010
10	0.09	0.90	5	11101
11	0.01	0.99	8	11111110

At first glance it may seem that this code is much worse than the Huffman code for the same source with codeword lengths $(1, 2, 3, 3)$ we found previously. On the other hand, it can be shown that arithmetic coding always achieves an average codeword length

$\overline{L}(c) < H(P^n) + 2$, which is only two bits apart from the lower bound in the noiseless coding theorem. Huffman coding would usually yield an even better code. However, this "negligible" loss in compression rate is compensated by several advantages. The codeword is directly computed from the source sequence, which means that we do not have to store the code as in the case of Huffman coding. Further, the relevant source models allow to easily compute $P^n(x_1x_2\ldots x_{n-1}x_n)$ and $Q^n(x_1x_2\ldots x_{n-1}x_n)$ from $P^{n-1}(x_1x_2\ldots x_{n-1})$, usually by multiplication by $P(x_n)$. This means that the sequence to be encoded can be parsed sequentially bit by bit, unlike in Huffman coding, where we would have to encode blockwise.

Encoding: The basic algorithm for encoding a sequence $(x_1\ldots x_n)$ by arithmetic coding works as follows. We assume that $P^n(x_1\ldots x_n) = P_1(x_1)\cdot P_2(x_2)\cdots P_n(x_n)$, (in the case $P_i = P$ for all i the discrete memoryless source arises, but in the section on modelling more complicated formulae come into play) and hence $Q_i(x_i) = \sum_{y<x_i} P_i(x_i)$

Starting with $B_0 = 0$ and $A_0 = 1$ the first i letters of the text to be compressed determine the ***current interval*** $[B_i, B_i + A_i)$. These current intervals are successively refined via the recursions

$$B_{i+1} = B_i + A_i \cdot Q_i(x_i), \quad A_{i+1} = A_i \cdot P_i(x_i)$$

$A_i \cdot P_i(x)$ is usually denoted as augend. The final interval $[B_n, B_n + A_n) = [Q^n(x^n), Q^n(x^n) + P^n(x^n))$ will then be encoded by interval $J(x^n)$ as described above. So the algorithm looks as follows.

ARITHMETIC–ENCODER(x)
1 $B \leftarrow 0$
2 $A \leftarrow 1$
3 **for** $i \leftarrow 1$ **to** n
4 $\quad B \leftarrow B + A \cdot Q_i(x[i])$
5 $\quad$ **do** $A \leftarrow A \cdot P_i(x[i])$
6 $L \leftarrow \lceil \lg \frac{1}{A} \rceil + 1$
7 $c \leftarrow \lceil B \cdot 2^L \rceil$
8 **return** c

We shall illustrate the encoding procedure by the following example from the literature. Let the discrete, memoryless source $(\mathcal{X}, P)$ be given with ternary alphabet $\mathcal{X} = \{1, 2, 3\}$ and $P(1) = 0.4$, $P(2) = 0.5$, $P(3) = 0.1$. The sequence $x^4 = (2, 2, 2, 3)$ has to be encoded. Observe that $P_i = P$ and $Q_i = Q$ for all $i = 1, 2, 3, 4$. Further $Q(1) = 0$, $Q(2) = P(1) = 0.4$, and $Q(3) = P(1) + P(2) = 0.9$.

The above algorithm yields

i	B_i	A_i
0	0	1
1	$B_0 + A_0 \cdot Q(2) = 0.4$	$A_0 \cdot P(2) = 0.5$
2	$B_1 + A_1 \cdot Q(2) = 0.6$	$A_1 \cdot P(2) = 0.25$
3	$B_2 + A_2 \cdot Q(2) = 0.7$	$A_2 \cdot P(2) = 0.125$
4	$B_3 + A_3 \cdot Q(3) = 0.8125$	$A_3 \cdot P(3) = 0.0125$

Hence $Q(2, 2, 2, 3) = B_4 = 0.8125$ and $P(2, 2, 2, 3) = A_4 = 0.0125$. From this can be calculated that $L = \lceil \lg \frac{1}{A} \rceil + 1 = 8$ and finally $\lceil B \cdot 2^L \rceil = \lceil 0.8125 \cdot 256 \rceil = 208$ whose binary representation is codeword $c(2, 2, 2, 3) = 11010000$.

Decoding: Decoding is very similar to encoding. The decoder recursively "undoes" the encoder's recursion. We divide the interval $[0,1)$ into subintervals with bounds defined by Q_i. Then we find the interval in which codeword c can be found. This interval determines the next symbol. Then we subtract $Q_i(x_i)$ and rescale by multiplication by $\frac{1}{P_i(x_i)}$.

ARITHMETIC–DECODER(c)
1 **for** $i \leftarrow 1$ **to** n
2 **do** $j \leftarrow 1$
3 **while** $(c < Q_i(j))$ **do** $j \leftarrow j+1$
4 $x[i] \leftarrow j-1$
5 $c \leftarrow (c - Q_i(x[i]))/P_i(x[i])$
6 **return** x

Observe that when the decoder only receives codeword c he does not know when the decoding procedure terminates. For instance $c = 0$ can be the codeword for $x^1 = (1)$, $x^2 = (1,1)$, $x^3 = (1,1,1)$, etc. In the above pseudocode it is implicit that the number n of symbols has also been transmitted to the decoder, in which case it is clear what the last letter to be encoded was. Another possibility would be to provide a special end-of-file (EOF)-symbol with a small probability, which is known to both the encoder and the decoder. When the decoder sees this symbol, he stops decoding. In this case line 1 would be replaced by

1 **while** $(x[i] \neq \text{EOF})$

(and i would have to be increased). In our above example, the decoder would receive the codeword 11010000, the binary expansion of 0.8125 up to $L = 8$ bits. This number falls in the interval $[0.4, 0.9)$ which belongs to the letter 2, hence the first letter $x_1 = 2$. Then he calculates $(0.8075 - Q(2))\frac{1}{P(2)} = (0.815 - 0.4) \cdot 2 = 0.83$. Again this number is in the interval $[0.4, 0.9)$ and the second letter is $x_2 = 2$. In order to determine x_3 the calculation $(0.83 - Q(2))\frac{1}{P(2)} = (0.83 - 0.4) \cdot 2 = 0.86$ must be performed. Again $0.86 \in [0.4, 0.9)$ such that also $x_3 = 2$. Finally $(0.86 - Q(2))\frac{1}{P(2)} = (0.86 - 0.4) \cdot 2 = 0.92$. Since $0.92 \in [0.9, 1)$, the last letter of the sequence must be $x_4 = 3$.

Correctness

Recall that message x^n is encoded by a codeword $c(x^n)$, whose interval $J(x^n)$ is inside interval $I(x^n)$. This follows from $\lceil Q^n(x^n) \cdot 2^{L(x^n)} \rceil 2^{-L(x^n)} + 2^{-L(x^n)} < Q^n(x^n) + 2^{1-L(x^n)} = Q^n(x^n) + 2^{-\lceil \lg \frac{1}{P^n(x^n)} \rceil} \leq Q^n(x^n) + P^n(x^n)$.

Obviously a prefix code is obtained, since a codeword can only be a prefix of another one, if their corresponding intervals overlap – and the intervals $J(x^n) \subset I(x^n)$ are obviously disjoint for different n-s.

Further, we mentioned already that arithmetic coding compresses down to the entropy up to two bits. This is because for every sequence x^n it is $L(x^n) < \lg \frac{1}{P^n(x^n)} + 2$. It can also be shown that the additional transmission of block length n or the introduction of the EOF symbol only results in a negligible loss of compression.

However, the basic algorithms we presented are not useful in order to compress longer files, since with increasing block length n the intervals are getting smaller and smaller, such that rounding errors will be unavoidable. We shall present a technique to overcome this problem in the following.

Analysis

The basic algorithm for arithmetic coding is linear in the length n of the sequence to be

encoded. Usually, arithmetic coding is compared to Huffman coding. In contrast to Huffman coding, we do not have to store the whole code, but can obtain the codeword directly from the corresponding interval. However, for a discrete memoryless source, where the probability distribution $P_i = P$ is the same for all letters, this is not such a big advantage, since the Huffman code will be the same for all letters (or blocks of k letters) and hence has to be computed only once. Huffman coding, on the other hand, does not use any multiplications which slow down arithmetic coding.

For the adaptive case, in which the P_i's may change for different letters x_i to be encoded, a new Huffman code would have to be calculated for each new letter. In this case, usually arithmetic coding is preferred. We shall investigate such situations in the section on modelling.

For implementations in practice floating point arithmetic is avoided. Instead, the subdivision of the interval $[0, 1)$ is represented by a subdivision of the integer range $0, \ldots, M$, say, with proportions according to the source probabilities. Now integer arithmetic can be applied, which is faster and more precise.

Precision problem.

In the basic algorithms for arithmetic encoding and decoding the shrinking of the current interval would require the use of high precision arithmetic for longer sequences. Further, no bit of the codeword is produced until the complete sequence x^n has been read in. This can be overcome by coding ech bit as soon as it is known and then double the length of the current interval $[LO, HI)$, say, so that this expansion represents only the unknown part of the interval. This is the case when the leading bits of the lower and upper bound are the same, i. e. the interval is completely contained either in $[0, \frac{1}{2})$ or in $[\frac{1}{2}, 1)$. The following expansion rules guarantee that the current interval does not become too small.

Case 1 ($[LO, HI) \in [0, \frac{1}{2})$): $LO \leftarrow 2 \cdot Lo$, $HI \leftarrow 2 \cdot HI$.

Case 2 ($[LO, HI) \in [\frac{1}{2}, 1)$): $LO \leftarrow 2 \cdot LO - 1$, $HI \leftarrow 2 \cdot HI - 1$.

Case 3 ($\frac{1}{4} \leq LO < \frac{1}{2} \leq HI < \frac{3}{4}$): $LO \leftarrow 2 \cdot LO - \frac{1}{2}$, $HI \leftarrow 2 \cdot HI - \frac{1}{2}$.

The last case called ***underflow*** (or follow) prevents the interval from shrinking too much when the bounds are close to $\frac{1}{2}$. Observe that if the current interval is contained in $[\frac{1}{4}, \frac{3}{4})$ with $LO < \frac{1}{2} \leq HI$, we do not know the next output bit, but we do know that whatever it is, the following bit will have the opposite value. However, in contrast to the other cases we cannot continue encoding here, but have to wait (remain in the underflow state and adjust a counter *underflowcount* to the number of subsequent underflows, i. e. $underflowcount \leftarrow underflowcount + 1$) until the current interval falls into either $[0, \frac{1}{2})$ or $[\frac{1}{2}, 1)$. In this case we encode the leading bit of this interval – 0 for $[0, \frac{1}{2})$ and 1 for $[\frac{1}{2}, 1)$ – followed by *underflowcount* many inverse bits and then set $underflowcount = 0$. The procedure stops, when all letters are read in and the current interval does not allow any further expansion.

ARITHMETIC-PRECISION-ENCODER(x)
1 $LO \leftarrow 0$
2 $HI \leftarrow 1$
3 $A \leftarrow 1$
4 $underflowcount \leftarrow 0$
5 **for** $i \leftarrow 1$ **to** n
6 **do** $LO \leftarrow LO + Q_i(x[i]) \cdot A$

```
7       A ← P_i(x[i])
8       HI ← LO + A
9       while HI − LO < 1/2 AND NOT (LO < 1/4 AND HI ≥ 1/2)
10          do if HI < 1/2
11                then c ← c||0, underflowcount many 1s
12                     underflowcount ← 0
13                     LO ← 2 · LO
14                     HI ← 2 · HI
15             else if LO ≥ 1/2
16                then c ← c||1, underflowcount many 0s
17                     underflowcount ← 0
18                     LO ← 2 · LO − 1
19                     HI ← 2 · HI − 1
20             else if LO ≥ 1/4 AND HI < 3/4
21                then underflowcount ← underflowcount + 1
22                     LO ← 2 · LO − 1/2
23                     HI ← 2 · HI − 1/2
24 if underflowcount > 0
25    then c ← c||0, underflowcount many 1s)
26 return c
```

We shall illustrate the encoding algorithm in Figure 13.6 by our example – the encoding of the message $(2, 2, 2, 3)$ with alphabet $\mathcal{X} = \{1, 2, 3\}$ and probability distribution $P = (0.4, 0.5, 0.1)$. An underflow occurs in the sixth row: we keep track of the underflow state and later encode the inverse of the next bit, here this inverse bit is the 0 in the ninth row. The encoded string is 1101000.

Precision – decoding involves the consideration of a third variable besides the interval bounds LO and HI.

13.2.2. Modelling

Modelling of memoryless sources with The Krichevsky-Trofimov Estimator

In this section we shall only consider binary sequences $x^n \in \{0, 1\}^n$ to be compressed by an arithmetic coder. Further, we shortly write $P(x^n)$ instead of $P^n(x^n)$ in order to allow further subscripts and superscripts for the description of the special situation. P_e will denote estimated probabilities, P_w weighted probabilities, and P^s probabilities assigned to a special context s.

The application of arithmetic coding is quite appropriate if the probability distribution of the source is such that $P(x_1x_2 \ldots x_{n-1}x_n)$ can easily be calculated from $P(x_1x_2 \ldots x_{n-1})$. Obviously this is the case, when the source is discrete and memoryless, since then $P(x_1x_2 \ldots x_{n-1}x_n) = P(x_1x_2 \ldots x_{n-1}) \cdot P(x_n)$.

Even when the underlying parameter $\theta = P(1)$ of a binary, discrete memoryless source is not known, there is an efficient way due to Krichevsky and Trofimov to estimate the probabilities via

$$P(X_n = 1|x^{n-1}) = \frac{b + \frac{1}{2}}{a + b + 1},$$

Current Interval	Action	Subintervals 1	2	3	Input
$[0.00, 1.00)$	subdivide	$[0.00, 0.40)$	$[0.40, 0.90)$	$[0.90, 1.00)$	2
$[0.40, 0.90)$	subdivide	$[0.40, 0.60)$	$[0.60, 0.85)$	$[0.85, 0.90)$	2
$[0.60, 0.85)$	encode 1				
	expand $[\frac{1}{2}, 1)$				
$[0.20, 0.70)$	subdivide	$[0.20, 0.40)$	$[0.40, 0.65)$	$[0.65, 0.70)$	2
$[0.40, 0.65)$	underflow				
	expand $[\frac{1}{4}, \frac{3}{4})$				
$[0.30, 0.80)$	subdivide	$[0.30, 0.50)$	$[0.50, 0.75)$	$[0.75, 0.80)$	3
$[0.75, 0.80)$	encode 10				
	expand $[\frac{1}{2}, 1)$				
$[0.50, 0.60)$	encode 1				
	expand $[\frac{1}{2}, 1)$				
$[0.00, 0.20)$	encode 0				
	expand $[0, \frac{1}{2})$				
$[0.00, 0.40)$	encode 0				
	expand $[0, \frac{1}{2})$				
$[0.00, 0.80)$	encode 0				

Figure 13.6. Example of Arithmetic encoding with interval expansion.

a	b	0	1	2	3	4	5
0		1	1/2	3/8	5/16	35/128	63/256
1		1/2	1/8	1/16	5/128	7/256	21/1024
2		3/8	1/16	3/128	3/256	7/1024	9/2048
3		5/16	5/128	3/256	5/1024	5/2048	45/32768

Figure 13.7. Table of the first values for the Krichevsky-Trofimov estimator.

where a and b denote the number of 0s and 1s, respectively, in the sequence $x^{n-1} = (x_1 x_2 \dots x_{n-1})$. So given the sequence x^{n-1} with a many 0s and b many 1s, the probability that the next letter x_n will be a 1 is estimated as $\frac{b+\frac{1}{2}}{a+b+1}$. The estimated block probability of a sequence containing a zeros and b ones then is

$$P_e(a,b) = \frac{\frac{1}{2} \cdots (a-\frac{1}{2}) \frac{1}{2} \cdots (b-\frac{1}{2})}{1 \cdot 2 \cdots (a+b)}$$

with initial values $a = 0$ and $b = 0$ as in Figure 13.7, where the values of the ***Krichevsky-Trofimov estimator*** $P_e(a,b)$ for small (a,b) are listed.

Note that the summand $\frac{1}{2}$ in the nominator guarantees that the probability for the next letter to be a 1 is positive even when the symbol 1 did not occur in the sequence so far. In order to avoid infinite codeword length, this phenomenon has to be carefully taken into account when estimating the probability of the next letter in all approaches to estimate the parameters, when arithmetic coding is applied.

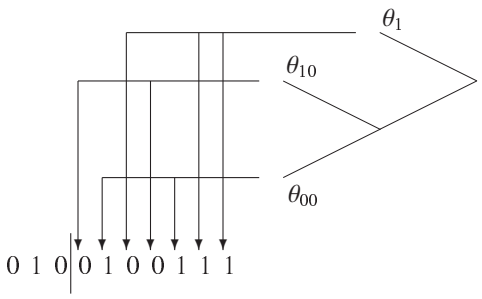


Figure 13.8. An example for a tree source.

Models with known context tree

In most situations the source is not memoryless, i. e., the dependencies between the letters have to be considered. A suitable way to represent such dependencies is the use of a suffix tree, which we denote as ***context tree***. The context of symbol x_t is suffix s preceding x_t. To each context (or leaf in the suffix tree) s there corresponds a parameter $\theta_s = P(X_t = 1|s)$, which is the probability of the occurrence of a 1 when the last sequence of past source symbols is equal to context s (and hence $1 - \theta_s$ is the probability for a 0 in this case). We are distinguishing here between the model (the suffix tree) and the parameters (θ_s).

13.1. Example. Let $\mathcal{S} = \{00, 10, 1\}$ and $\theta_{00} = \frac{1}{2}$, $\theta_{10} = \frac{1}{3}$, and $\theta_1 = \frac{1}{5}$. The corresponding suffix tree jointly with the parsing process for a special sequence can be seen in Figure 13.8.

The actual probability of the sequence '0100111' given the past '...010' is $P^s(0100111|\ldots 010) = (1-\theta_{10})\theta_{00}(1-\theta_1)(1-\theta_{10})\theta_{00}\theta_1\theta_1 = \frac{2}{3}\cdot\frac{1}{2}\cdot\frac{4}{5}\cdot\frac{2}{3}\cdot\frac{1}{2}\cdot\frac{1}{5}\cdot\frac{1}{5} = \frac{4}{1075}$, since the first letter 0 is preceded by suffix 10, the second letter 1 is preceded by suffix 00, etc.

Suppose the model $\mathcal{S}$ is known, but not the parameters θ_s. The problem now is to find a good coding distribution for this case. The tree structure allows to easily determine which context precedes a particular symbol. All symbols having the same context (or suffix) $s \in \mathcal{S}$ form a memoryless source subsequence whose probability is determined by the unknown parameter θ_s. In our example these subsequences are '11' for θ_{00}, '00' for θ_{10} and '011' for θ_1. One uses the Krichevsky-Trofimov-estimator for this case. To each node s in the suffix tree, we count the numbers a_s of zeros and b_s of ones preceded by suffix s. For the children $0s$ and $1s$ of parent node s obviously $a_{0s} + a_{1s} = a_s$ and $b_{0s} + b_{1s} = b_s$ must be satisfied.

In our example $(a_\lambda, b_\lambda) = (3,4)$ for the root λ, $(a_1, b_1) = (1,2)$, $(a_0, b_0) = (2,2)$ and $(a_{10}, b_{10}) = (2,0)$, $(a_{00}, b_{00}) = (0,2)$. Further $(a_{11}, b_{11}) = (0,1)$, $(a_{01}, b_{01}) = (1,1)$, $(a_{111}, b_{111}) = (0,0)$, $(a_{011}, b_{011}) = (0,1)$, $(a_{101}, b_{101}) = (0,0)$,$(a_{001}, b_{001}) = (1,1)$, $(a_{110}, b_{110}) = (0,0)$, $(a_{010}, b_{010}) = (2,0)$, $(a_{100}, b_{100}) = (0,2)$, and $(a_{000}, b_{000}) = (0,0)$. These last numbers are not relevant for our special source $\mathcal{S}$ but will be important later on, when the source model or the corresponding suffix tree, respectively, is not known in advance.

13.2. Example. Let $\mathcal{S} = \{00, 10, 1\}$ as in the previous example. Encoding a subsequence is done by successively updating the corresponding counters for a_s and b_s. For example, when we encode the

sequence '0100111' given the past '...010' using the above suffix tree and Krichevsky–Trofimov–estimator we obtain

$$P_e^s(0100111|\ldots 010) = \frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{3}{4}\cdot\frac{1}{4}\cdot\frac{1}{2} = \frac{3}{8}\cdot\frac{3}{8}\cdot\frac{1}{16} = \frac{9}{1024},$$

where $\frac{3}{8}$, $\frac{3}{8}$ and $\frac{1}{16}$ are the probabilities of the subsequences '11', '00' and '011' in the context of the leaves. These subsequences are assumed to be memoryless.

The context-tree weighting method

Suppose we have a good coding distribution P_1 for source 1 and another one, P_2, for source 2. We are looking for a good coding distribution for both sources. One possibility is to compute P_1 and P_2 and then 1 bit is needed to identify the best model which then will be used to compress the sequence. This method is called selecting. Another possibility is to employ the weighted distribution, which is

$$P_w(x^n) = \frac{P_1(x^n) + P_2(x^n)}{2}.$$

We shall present now the ***context-tree weighting algorithm*** . Under the assumption that a context tree is a full tree of depth D, only a_s and b_s, i. e. the number of zeros and ones in the subsequence of bits preceded by context s, are stored in each node s of the context tree.

Further, to each node s is assigned a weighted probability P_w^s which is recursively defined as

$$P_w^s = \begin{cases} \frac{P_e(a_s,b_s)+P_w^{0s}P_w^{1s}}{2} & \text{for } 0 \le L(s) < D, \\ P_e(a_s,b_s) & \text{for } L(s) = D, \end{cases}$$

where $L(s)$ describes the length of the (binary) string s and $P_e(a_s, b_s)$ is the estimated probability using the Krichevsky - Trofimov estimator.

13.3. Example. After encoding the sequence '0100111' given the past '...010' we obtain the context tree of depth 3 in Figure 13.9. The weighted probability $P_w\lambda = \frac{35}{4096}$ of the root node λ finally yields the coding probability corresponding to the parsed sequence.

Recall that for the application in arithmetic coding it is important that probabilities $P(x_1 \ldots x_{n-1}0)$ and $P(x_1 \ldots x_{n-1}1)$ can be efficiently calculated from $P(x_1 \ldots x_n)$. This is possible with the context–tree weighting method, since the weighted probabilities P_w^s only have to be updated, when s is changing. This just occurs for the contexts along the path from the root to the leaf in the context tree preceding the new symbol x_n — namely the $D+1$ contexts $x_{n-1}, \ldots, x_{n-i}$ for $i = 1, \ldots, D-1$ and the root λ. Along this path, $a_s = a_s + 1$ has to be performed, when $x_n = 0$, and $b_s = b_s + 1$ has to be performed, when $x_n = 1$, and the corresponding probabilities $P_e(a_s, b_s)$ and P_w^s have to be updated.

This suggests the following algorithm for updating the context tree $CT(x_1, \ldots, x_{n-1}|x_{-D+1}, \ldots x_0)$ when reading the next letter x_n. Recall that to each node of the tree we store the parameters (a_s, b_s), $P_e(a_s, b_s)$ and P_w^s. These parameters have to be updated in order to obtain $CT(x_1, \ldots, x_n|x_{-D+1}, \ldots x_0)$. We assume the convention that the ordered pair (x_{n-1}, x_n) denotes the root λ.

UPDATE-CONTEXT-TREE$(x_n, CT(x_1 \ldots x_{n-1}|x_{-D+1} \ldots x_0))$

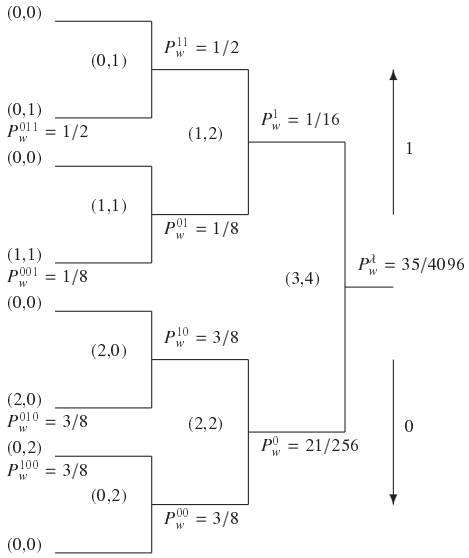


Figure 13.9. Weighted context tree for source sequence '0100111' with past ...010. The pair (a_s, b_s) denotes a_s zeros and b_s ones preceded by the corresponding context s. For the contexts $s = 111, 101, 110, 000$ it is $P_w^s = P_e(0,0) = 1$.

```
1  s ← (x_{n−1} ... x_{n−D})
2  if x_n = 0
3     then P_w^s ← P_w^s · (a_s+1/2)/(a_s+b_s+1)
4          a_s ← a_s + 1
5     else P_w^s ← P_w^s · (b_s+1/2)/(a_s+b_s+1)
6          b_s ← b_s + 1
7  for i ← 1 to D
8      do s ← (x_{n−1}, ..., x_{n−D+i})
9      if x_n = 0
10        then P_e(a_s, b_s) ← P_e(a_s, b_s) · (a_s+1/2)/(a_s+b_s+1)
11             a_s ← a_s + 1
12        else P_e(a_s, b_s) ← P_e(a_s, b_s) · (a_s+1/2)/(a_s+b_s+1)
13             b_s ← b_s + 1
14     P_w^s ← 1/2 · (P_e(a_s, b_s) + P_w^{0s} · P_w^{1s})
15 return P_w^s
```

The probability P_w^λ assigned to the root in the context tree will be used for the successive subdivisions in arithmetic coding. Initially, before reading x_1, the parameters in the context tree are $(a_s, b_s) = (0,0)$, $P_e(a_s, b_s) = 1$, and $P_w^s = 1$ for all contexts s in the tree. In our example the updates given the past $(x_{-2}, x_{-1}, x_0) = (0,1,0)$ would yield the successive probabilities P_w^λ: $\frac{1}{2}$ for $x_1 = 0$, $\frac{9}{32}$ for $(x_1 x_2) = (01)$, $\frac{5}{64}$ for $(x_1 x_2 x_3) = (010)$, $\frac{13}{256}$ for $(x_1 x_2 x_3 x_4) = (0100)$, $\frac{27}{1024}$ for $(x_1 x_2 x_3 x_4) = (01001)$, $\frac{13}{1024}$ for $(x_1 x_2 x_3 x_4 x_5) = (010011)$, $\frac{13}{1024}$ for $(x_1 x_2 x_3 x_4 x_5 x_6) = (010011)$, and finally $\frac{35}{4096}$ for $(x_1 x_2 x_3 x_4 x_5 x_6 x_7) = (0100111)$.

Correctness

Recall that the quality of a code concerning its compression capability is measured with respect to the average codeword length. The average codeword length of the best code comes as close as possible to the entropy of the source. The difference between the average codeword length and the entropy is denoted as the ***redundancy*** $\overline{\rho}(c)$ of code c, hence

$$\overline{\rho}(c) = \overline{L}(c) - H(P),$$

which obviously is the weighted (by $P(x^n)$) sum of the individual redundancies

$$\rho(x^n) = L(x^n) - \lg \frac{1}{P(x^n)}.$$

The individual redundancy $\rho(x^n|\mathcal{S})$ of sequences x^n given the (known) source $\mathcal{S}$ for all $\theta_s \in [0,1]$ for $s \in \mathcal{S}$, $|\mathcal{S}| \leq n$ is bounded by

$$\rho(x^n|\mathcal{S}) \leq \frac{|\mathcal{S}|}{2} \lg \frac{n}{|\mathcal{S}|} + |\mathcal{S}| + 2.$$

The individual redundancy $\rho(x^n|\mathcal{S})$ of sequences x^n using the context–tree weighting algorithm (and hence a complete tree of all possible contexts as model $\mathcal{S}$) is bounded by

$$\rho(x^n|\mathcal{S}) < 2|\mathcal{S}| - 1 + \frac{|\mathcal{S}|}{2} \lg \frac{n}{|\mathcal{S}|} + |\mathcal{S}| + 2.$$

Comparing these two formulae, we see that the difference of the individual redundancies is $2|\mathcal{S}| - 1$ bits. This can be considered as the cost of not knowing the model, i.e. the model redundancy. So, the redundancy splits into the parameter redundancy, i. e. the cost of not knowing the parameter, and the model redundancy. It can be shown that the expected redundancy behaviour of the context–tree weighting method achieves the asymptotic lower bound due to Rissanen who could demonstrate that about $\frac{1}{2} \lg n$ bits per parameter is the minimum possible expected redundancy for $n \longrightarrow \infty$.

Analysis

The computational complexity is proportional to the number of nodes that are visited when updating the tree, which is about $n(D+1)$. Therefore, the number of operations necessary for processing n symbols is linear in n. However, these operations are mainly multiplications with factors requiring high precision.

As for most modelling algorithms, the backlog of implementations in practice is the huge amount of memory. A complete tree of depth D has to be stored and updated. Only with increasing D the estimations of the probabilities are becoming more accurate and hence the average codeword length of an arithmetic code based on these estimations would become shorter. The size of the memory, however, depends exponentially on the depth of the tree.

We presented the context–tree weighting method only for binary sequences. Note that in this case the cumulative probability of a binary sequence $(x_1 \ldots x_n)$ can be calculated as

$$Q^n(x_1 x_2 \ldots x_{n-1} x_n) = \sum_{j=1,\ldots,n;x_j=1} P^j(x_1 x_2 \ldots x_{j-1} 0).$$

For compression of sources with larger alphabets, for instance ASCII-files, we refer to

the literature.

Exercises

13.2-1 Compute the arithmetic codes for the sources $(\mathcal{X}^n, P^n)$, $n = 1, 2, 3$ with $\mathcal{X} = \{1, 2\}$ and $P = (0.8, 0.2)$ and compare these codes with the corresponding Huffman codes derived previously.

13.2-2 For the codes derived in the previous exercise compute the individual redundancies of each codeword and the redundancies of the codes.

13.2-3 Compute the estimated probabilities $P_e(a, b)$ for the sequence 0100110 and all its subsequences using the Krichevsky-Trofimov estimator.

13.2-4 Compute all parameters (a_s, b_s) and the estimated probability P_e^s for the sequence 0100110 given the past 110, when the context tree $\mathcal{S} = \{00, 10, 1\}$ is known. What will be the codeword of an arithmetic code in this case?

13.2-5 Compute all parameters (a_s, b_s) and the estimated probability P_λ for the sequence 0100110 given the past 110, when the context tree is not known, using the context-tree weighting algorithm.

13.2-6 Based on the computations from the previous exercise, update the estimated probability for the sequence 01001101 given the past 110.

Show that for the cumulative probability of a binary sequence $(x_1 \dots x_n)$ it is

$$Q^n(x_1 x_2 \dots x_{n-1} x_n) = \sum_{j=1,\dots,n;x_j=1} P^j(x_1 x_2 \dots x_{j-1} 0).$$

13.3. Ziv-Lempel coding

In 1976–1978 Jacob Ziv and Abraham Lempel introduced two universal coding algorithms, which in contrast to statistical coding techniques, considered so far, do not make explicit use of the underlying probability distribution. The basic idea here is to replace a previously seen string with a pointer into a history buffer (LZ77) or with the index of a dictionary (LZ78). LZ algorithms are widely used – "zip" and its variations use the LZ77 algorithm. So, in contrast to the presentation by several authors, Ziv-Lempel coding is not a single algorithm. Originally, Lempel and Ziv introduced a method to measure the complexity of a string – like in Kolmogorov complexity. This led to two different algorithms, LZ77 and LZ78. Many modifications and variations have been developed since. However, we shall present the original algorithms and refer to the literature for further information.

13.3.1. LZ77

The idea of LZ77 is to pass a ***sliding window*** over the text to be compressed. One looks for the longest substring in this window representing the next letters of the text. The window consists of two parts: a history window of length l_h, say, in which the last l_h bits of the text considered so far are stored, and a lookahead window of length l_f containing the next l_f bits of the text. In the simplest case l_h and l_f are fixed. Usually, l_h is much bigger than l_f. Then one encodes the triple (offset, length, letter). Here the ***offset*** is the number of letters

one has to go back in the text to find the matching substring, the length is just the length of this matching substring, and the letter to be stored is the letter following the matching substring. Let us illustrate this procedure with an example. Assume the text to be compressed is *...abaabbaabbaaabbbaaaabbabbbabbb...*, the window is of size 15 with $l_h = 10$ letters history and $l_f = 5$ letters lookahead buffer. Assume, the sliding window now arrived at

$$...aba||abbaabbaaa|bbbaa||,$$

i. e., the history window contains the 10 letters *abbaabbaaa* and the lookahead window contains the five letters *bbbaa*. The longest substring matching the first letters of the lookahead window is *bb* of length 2, which is found nine letters back from the right end of the history window. So we encode $(9, 2, b)$, since b is the next letter (the string *bb* is also found five letters back, in the original LZ77 algorithm one would select the loargest offset). The window then is moved 3 letters forward

$$...abaabb||aabbaaabbb|aaaab||.$$

The next codeword is $(6, 3, a)$, since the longest matching substring is *aaa* of length 3 found 6 letters backwards and a is the letter following this substring in the lookahead window. We proceed with

$$...abaabbaabb||aaabbbaaaa|bbabb||,$$

and encode $(6, 3, b)$. Further

$$...abaabbaabbaaab||bbaaaabbab|babbb||.$$

Here we encode $(3, 4, b)$. Observe that the match can extend into the lookahead window.

There are many subtleties to be taken into account. If a symbol did not appear yet in the text, offset and length are set to 0. If there are two matching strings of the same length, one has to choose between the first and the second offset. Both variations have advantages. Initially one might start with an empty history window and the first letters of the text to be compressed in the lookahead window - there are also further variations.

A common modification of the original scheme is to output only the pair (offset, length) and not the following letter of the text. Using this coding procedure one has to take into consideration the case in which the next letter does not occur in the history window. In this case, usually the letter itself is stored, such that the decoder has to distinguish between pairs of numbers and single letters. Further variations do not necessarily encode the longest matching substring.

13.3.2. LZ78

LZ78 does not use a sliding window but a dictionary which is represented here as a table with an index and an entry. LZ78 parses the text to be compressed into a collection of strings, where each string is the longest matching string α seen so far plus the symbol s following α in the text to be compressed. The new string αs is added into the dictionary. The new entry is coded as (i, s), where i is the index of the existing table entry α and s is the appended symbol.

As an example, consider the string "*abaabbaabbaaabbbaaaabba*". It is divided by LZ78 into strings as shown below. String 0 is here the empty string.

Input	a	b	aa	bb	aab	ba	$aabb$	baa	$aabba$
String Index	1	2	3	4	5	6	7	8	9
Output	$(0, a)$	$(0, b)$	$(1, a)$	$(2, b)$	$(3, b)$	$(2, a)$	$(5, b)$	$(6, a)$	$(7, a)$

Since we are not using a sliding window, there is no limit for how far back strings can reach. However, in practice the dictionary cannot continue to grow infinitely. There are several ways to manage this problem. For instance, after having reached the maximum number of entries in the dictionary, no further entries can be added to the table and coding becomes static. Another variation would be to replace older entries. The decoder knows how many bits must be reserved for the index of the string in the dictionary, and hence decompression is straightforward.

Correctness

Ziv-Lempel coding asymptotically achieves the best possible compression rate which again is the entropy rate of the source. The source model, however, is much more general than the discrete memoryless source. The stochastic process generating the next letter, is assumed to be stationary (the probability of a sequence does not depend on the instant of time, i. e. $P(X_1 = x_1, \ldots, X_n = x_n) = P(X_{t+1} = x_1, \ldots, X_{t+n} = x_n)$ for all t and all sequences $(x_1 \ldots x_n)$). For stationary processes the limit $\lim_{n\to\infty} \frac{1}{n} H(X_1, \ldots X_n)$ exists and is defined to be the entropy rate.

If s(n) denotes the number of strings in the parsing process of LZ78 for a text generated by a stationary source, then the number of bits required to encode all these strings is $s(n) \cdot (\lg s(n) + 1)$. It can be shown that $\frac{s(n)\cdot(\lg s(n)+1)}{n}$ converges to the entropy rate of the source. However, this would require that all strings can be stored in the dictionary.

Analysis

If we fix the size of the sliding window or the dictionary, the running time of encoding a sequence of n letters will be linear in n. However, as usually in data compression, there is a tradeoff between compression rate and speed. A better compression is only possible with larger memory. Increasing the size of the dictionary or the window will, however, result in a slower performance, since the most time consuming task is the search for the matching substring or the position in the dictionary.

Decoding in both LZ77 and LZ78 is straightforward. Observe that with LZ77 decoding is usually much faster than encoding, since the decoder already obtains the information at which position in the history he can read out the next letters of the text to be recovered, whereas the encoder has to find the longest matching substring in the history window. So algorithms based on LZ77 are useful for files which are compressed once and decompressed more frequently.

Further, the encoded text is not necessarily shorter than the original text. Especially in the beginning of the encoding the coded version may expand a lot. This expansion has to be taken into consideration.

For implementation it is not optimal to represent the text as an array. A suitable data structure will be a circular queue for the lookahead window and a binary search tree for the

history window in LZ77, while for LZ78 a dictionary tree should be used.

Exercises

13.3-1 Apply the algorithms LZ77 and LZ78 to the string "abracadabra".

13.3-2 Which type of files will be well compressed with LZ77 and LZ78, respectively? For which type of files are LZ77 and LZ78 not so advantageous?

13.3-3 Discuss the advantages of encoding the first or the last offset, when several matching substrings are found in LZ77.

13.4. The Burrows-Wheeler transform

The ***Burrows-Wheeler transform*** will best be demonstrated by an example. Assume that our original text is $\vec{X}$ = "WHEELER". This text will be mapped to a second text $\vec{L}$ and an index I according to the following rules.

1) We form a matrix M consisting of all cyclic shifts of the original text $\vec{X}$. In our example

$$M = \begin{pmatrix} W & H & E & E & L & E & R \\ H & E & E & L & E & R & W \\ E & E & L & E & R & W & H \\ E & L & E & R & W & H & E \\ L & E & R & W & H & E & E \\ E & R & W & H & E & E & L \\ R & W & H & E & E & L & E \end{pmatrix}.$$

2) From M we obtain a new matrix M' by simply ordering the rows in M lexicograhically. Here this yields the matrix

$$M' = \begin{pmatrix} E & E & L & E & R & W & H \\ E & L & E & R & W & H & E \\ E & R & W & H & E & E & L \\ H & E & E & L & E & R & W \\ L & E & R & W & H & E & E \\ R & W & H & E & E & L & E \\ W & H & E & E & L & E & R \end{pmatrix}.$$

3) The transformed string $\vec{L}$ then is just the last column of the matrix M' and the index I is the number of the row of M', in which the original text is contained. In our example $\vec{L}$ = "HELWEER" and $I = 6$ – we start counting the the rows with row no. 0.

This gives rise to the following pseudocode. We write here X instead of $\vec{X}$ and L instead of $\vec{L}$, since the purpose of the vector notation is only to distinguish the vectors from the letters in the text.

```
BWT-ENCODER(X)
1 for j ← 0 to n − 1
2     do M[0, j] ← X[j]
3 for i ← 0 to n − 1
```

```
4     do for j ← 0 to n − 1
5        do M[i, j] ← M[i − 1, j + 1 mod n]
6  for i ← 0 to n − 1
7     do row i of M′ ← row i of M in lexicographic order
8  for i ← 0 to n − 1
9     do L[i] ← M′[i, n − 1]
10 i = 0
11 while (row i of M′ ≠ row i of M)
12    do i ← i + 1
13 I ← i
14 return L and I
```

It can be shown that this transformation is invertible, i. e., it is possible to reconstruct the original text $\vec{X}$ from its transform $\vec{L}$ and the index I. This is because these two parameters just yield enough information to find out the underlying permutation of the letters. Let us illustrate this reconstruction using the above example again. From the transformed string $\vec{L}$ we obtain a second string $\vec{E}$ by simply ordering the letters in $\vec{L}$ in ascending order. Actually, $\vec{E}$ is the first column of the matrix M' above. So, in our example

$$\vec{L} = \text{“}H \quad E \quad L \quad W \quad E \quad E \quad R''$$

$$\vec{E} = \text{“}E \quad E \quad E \quad H \quad L \quad R \quad W''.$$

Now obviously the first letter $\vec{X}(0)$ of our original text $\vec{X}$ is the letter in position I of the sorted string $\vec{E}$, so here $\vec{X}(0) = \vec{E}(6) = W$. Then we look at the position of the letter just considered in the string $\vec{L}$ – here there is only one W, which is letter no. 3 in $\vec{L}$. This position gives us the location of the next letter of the original text, namely $\vec{X}(1) = \vec{E}(3) = H$. H is found in position no. 0 in $\vec{L}$, hence $\vec{X}(2) = \vec{E}(0) = E$. Now there are three E–s in the string $\vec{L}$ and we take the first one not used so far, here the one in position no. 1, and hence $\vec{X}(3) = \vec{E}(1) = E$. We iterate this procedure and find $\vec{X}(4) = \vec{E}(4) = L$, $\vec{X}(5) = \vec{E}(2) = E$, $\vec{X}(6) = \vec{E}(5) = R$.

This suggests the following pseudocode.

```
BWT-DECODER(L, I)
1 E[0..n − 1] ← sort L[0..n − 1]
2 pi[−1] ← I
3 for i ← 0 to n − 1
4    do j = 0
5       while (L[j]) ≠ E[pi[i − 1]] OR j is a component of pi)
6             do  j ← j + 1
7       pi[i] ← j
8       X[i] ← L[j]
9 return X
```

This algorithm implies a more formal description. Since the decoder only knows $\vec{L}$, he has to sort this string to find out $\vec{E}$. To each letter $\vec{L}(j)$ from the transformed string $\vec{L}$ record the position $\pi(j)$ in $\vec{E}$ from which it was jumped to by the process described above. So the vector pi in our pseudocode yields a permutation π such that for each $j = 0, \ldots, n-1$ row j it is $\vec{L}(j) = \vec{E}(\pi(j))$ in matrix M. In our example $\pi = (3, 0, 1, 4, 2, 5, 6)$. This permutation

can be used to reconstruct the original text $\vec{X}$ of length n via $\vec{X}(n-1-j) = \vec{L}(\pi^j(I))$, where $\pi^0(x) = x$ and $\pi^j(x) = \pi(\pi^{j-1}(x))$ for $j = 1, \ldots, n-1$.

Observe that so far the original data have only been transformed and are not compressed, since string $\vec{L}$ has exactly the same length as the original string $\vec{L}$. So what is the advantage of the Burrows–Wheeler transformation? The idea is that the transformed string can be much more efficiently encoded than the original string. The dependencies among the letters have the effect that in the transformed string $\vec{L}$ there appear long blocks consisting of the same letter.

In order to exploit such frequent blocks of the same letter, Burrows and Wheeler suggested the following ***move–to–front–code***, which we shall illustrate again with our example above.

We write down a list containing the letters used in our text in alphabetic order indexed by their position in this list.

E	*H*	*L*	*R*	*W*
0	1	2	3	4

Then we parse through the transformed string $\vec{L}$ letter by letter, note the index of the next letter and move this letter to the front of the list. So in the first step we note 1 – the index of the H, move H to the front and obtain the list

H	*E*	*L*	*R*	*W*
0	1	2	3	4

Then we note 1 and move E to the front,

E	*H*	*L*	*R*	*W*
0	1	2	3	4

note 2 and move L to the front,

L	*E*	*H*	*R*	*W*
0	1	2	3	4

note 4 and move W to the front,

W	*L*	*E*	*H*	*R*
0	1	2	3	4

note 2 and move E to the front,

E	*W*	*L*	*H*	*R*
0	1	2	3	4

note 0 and leave E at the front,

E	*W*	*L*	*H*	*R*
0	1	2	3	4

note 4 and move R to the front,

R	*E*	*W*	*L*	*H*
0	1	2	3	4

So we obtain the sequence $(1, 1, 2, 4, 2, 0, 4)$ as our move-to-front-code. The pseudocode may look as follows, where Q is a list of the letters occuring in the string $\vec{L}$.

```
MOVE-TO-FRONT(L)
1 Q[0..n − 1] ← list of m letters occuring in L ordered alphabetically
2 for i ← 0 to n − 1
3     do j = 0
4        while (j ≠ L[i])
5              do j ← j + 1
6     c[i] ← j
7 for l ← 0 to j
8     do Q[l] ← Q[l − 1 mod j + 1]
9 return c
```

The move–to–front code c will finally be compressed, for instance by Huffman coding.

Correctness

The compression is due to the move-to-front code obtained from the transformed string $\vec{L}$. It can easily be seen that this move-to-front coding procedure is invertible, so one can recover the string $\vec{L}$ from the code obtained as above.

Now it can be observed that in the move-to-front-code small numbers occur more frequently. Unfortunately, this will become obvious only with much longer texts than in our example – in long strings it was observed that even about 70 per cent of the numbers are 0. This irregularity in distribution can be exploited by compressing the sequence obtained after move-to-front coding, for instance by Huffman codes or run–length codes.

The algorithm performed very well in practice regarding the compression rate as well as the speed. The asymptotic optimality of compression has been proven for a wide class of sources.

Analysis

The most complex part of the Burrows–Wheeler transform is the sorting of the block yielding the transformed string $\vec{L}$. Due to fast sorting procedures, especially suited for the type of data to be compressed, compression algorithms based on the Burrows–Wheeler transform are usually very fast. On the other hand, compression is done blockwise. The text to be compressed has to be divided into blocks of appropriate size such that the matrices M and M' still fit into the memory. So the decoder has to wait until the whole next block is transmitted and cannot work sequentially bit by bit as in arithmetic coding or Ziv-Lempel coding.

Exercises

13.4-1 Apply the Burrows-Wheeler transform and the move-to-front code to the text "abracadabra".

13.4-2 Verify that the transformed string $\vec{L}$ and the index i of the position in the sorted text $\vec{E}$ (containing the first letter of the original text to be compressed) indeed yield enough information to reconstruct the original text.

13.4-3 Show how in our example the decoder would obtain the string $\vec{L}$ ="HELWEER" from the move-to-front code $(1, 1, 2, 4, 2, 0, 4)$ and the letters E,H,L,W,R occuring in the text. Describe the general procedure for decoding move-to-front codes.

13.4-4 We followed here the encoding procedure presented by Burrows and Wheeler. Can

the encoder obtain the transformed string $\vec{L}$ even without constructing the two matrices M and M'?

13.5. Image compression

The idea of image compression algorithms is similar to the one behind the Burrows-Wheeler transform. The text to be compressed is transformed to a format which is suitable for application of the techniques presented in the previous sections, such as Huffman coding or arithmetic coding. There are several procedures based on the type of image (for instance, black/white, greyscale or colour image) or compression (lossless or lossy). We shall present the basic steps – representation of data, discrete cosine transform, quantization, coding – of lossy image compression procedures like the standard ***JPEG***.

13.5.1. Representation of data

A greyscale image is represented as a two–dimensional array X, where each entry $X(i, j)$ represents the intensity (or brightness) at position (i, j) of the image. Each $X(i, j)$ is either a signed or an unsigned k-bit integers, i. e., $X(i, j) \in \{0, \ldots, 2^k - 1\}$ or $X(i, j) \in \{-2^{k-1}, \ldots, 2^{k-1} - 1\}$.

A position in a colour image is usually represented by three greyscale values $R(i, j)$, $G(i, j)$, and $B(i, j)$ per position corresponding to the intensity of the primary colours red, green and blue.

In order to compress the image, the three arrays (or channels) R, G, B are first converted to the luminance/chrominance space by the YC_bC_r***-transform*** (performed entry–wise)

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & -0.0813 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$Y = 0.299R + 0.587G + 0.114B$ is the luminance or intensity channel, where the coefficients weighting the colours have been found empirically and represent the best possible approximation of the intensity as perceived by the human eye. The chrominance channels $C_b = 0.564(B - Y)$ and $C_r = 0.713(R - Y)$ contain the colour information on red and blue as the differences from Y. The information on green is obtained as big part in the luminance Y.

A first compression for colour images commonly is already obtained after application of the YC_bC_r–transform by removing ***irrelevant information***. Since the human eye is less sensitive to rapid colour changes than to changes in intensity, the resolution of the two chrominance channels C_b and C_r is reduced by a factor of 2 in both vertical and horizontal direction, which results after sub-sampling in arrays of $\frac{1}{4}$ of the original size.

The arrays then are subdivided into 8×8 blocks, on which successively the actual (lossy) data compression procedure is applied.

Let us consider the following example based on a real image, on which the steps of compression will be illustrated. Assume that the 8×8 block of 8-bit unsigned integers below is obtained as a part of an image.

$$f = \begin{pmatrix} 139 & 144 & 149 & 153 & 155 & 155 & 155 & 155 \\ 144 & 151 & 153 & 156 & 159 & 156 & 156 & 155 \\ 150 & 155 & 160 & 163 & 158 & 156 & 156 & 156 \\ 159 & 161 & 162 & 160 & 160 & 159 & 159 & 159 \\ 159 & 160 & 161 & 161 & 160 & 155 & 155 & 155 \\ 161 & 161 & 161 & 161 & 160 & 157 & 157 & 157 \\ 162 & 162 & 161 & 163 & 162 & 157 & 157 & 157 \\ 161 & 162 & 161 & 161 & 163 & 158 & 158 & 158 \end{pmatrix}$$

13.5.2. The discrete cosine transform

Each 8×8 block $(f(i,j))_{i,j=0,\ldots,7}$, say, is transformed into a new block $(F(u,v))_{u,v=0,\ldots,7}$. There are several possible transforms, usually the ***discrete cosine transform*** is applied, which here obeys the formula

$$F(u,v) = \frac{1}{4} c_u c_v \left(\sum_{i=0}^{7} \sum_{j=0}^{7} f(i,j) \cdot \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16} \right)$$

The cosine transform is applied after shifting the unsigned integers to signed integers by subtraction of 2^{k-1}.

DCT(f)
1 **for** $u \leftarrow 0$ **to** 7
2 **do for** $v \leftarrow 0$ **to** 7
3 **do** $F(u,v) \leftarrow$ DCT - coefficient of matrix f
4 **return** F

The coefficients need not be calculated according to the formula above. They can also be obtained via a related Fourier transform (see Exercises) such that a Fast Fourier Transform may be applied. JPEG also supports wavelet transforms, which may replace the discrete cosine transform here.

The discrete cosine transform can be inverted via

$$f(i,j) = \frac{1}{4} \left(\sum_{u=0}^{7} \sum_{v=0}^{7} c_u c_v F(u,v) \cdot \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16} \right),$$

where $c_u = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ 1 & \text{for } u \neq 0 \end{cases}$ and $c_v = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } v = 0 \\ 1 & \text{for } v \neq 0 \end{cases}$ are normalization constants.

In our example, the transformed block F is

$$F = \begin{pmatrix} 235.6 & -1.0 & -12.1 & -5.2 & 2.1 & -1.7 & -2.7 & 1.3 \\ -22.6 & -17.5 & -6.2 & -3.2 & -2.9 & -0.1 & 0.4 & -1.2 \\ -10.9 & -9.3 & -1.6 & 1.5 & 0.2 & -0.9 & -0.6 & -0.1 \\ -7.1 & -1.9 & 0.2 & 1.5 & 0.9 & -0.1 & 0.0 & 0.3 \\ -0.6 & -0.8 & 1.5 & 1.6 & -0.1 & -0.7 & 0.6 & 1.3 \\ 1.8 & -0.2 & 1.6 & -0.3 & -0.8 & 1.5 & 1.0 & -1.0 \\ -1.3 & -0.4 & -0.3 & -1.5 & -0.5 & 1.7 & 1.1 & -0.8 \\ -2.6 & 1.6 & -3.8 & -1.8 & 1.9 & 1.2 & -0.6 & -0.4 \end{pmatrix}$$

where the entries are rounded.

The discrete cosine transform is closely related to the discrete Fourier transform and similarly maps signals to frequencies. Removing higher frequencies results in a less sharp image, an effect that is tolerated, such that higher frequencies are stored with less accuracy.

Of special importance is the entry $F(0,0)$, which can be interpreted as a measure for the intensity of the whole block.

13.5.3. Quantization

The discrete cosine transform maps integers to real numbers, which in each case have to be rounded to be representable. Of course, this rounding already results in a loss of information. However, the transformed block F will now be much easier to manipulate. A ***quantization*** takes place, which maps the entries of F to integers by division by the corresponding entry in a luminance quantization matrix Q. In our example we use

$$Q = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}.$$

The quantization matrix has to be carefully chosen in order to leave the image at highest possible quality. Quantization is the lossy part of the compression procedure. The idea is to remove information which should not be "visually significant". Of course, at this point there is a tradeoff between the compression rate and the quality of the decoded image. So, in JPEG the quantization table is not included into the standard but must be specified (and hence be encoded).

QUANTIZATION(F)
1 **for** $i \leftarrow 0$ **to** 7
2 **do for** $j \leftarrow 0$ **to** 7
3 **do** $T(i,j) \leftarrow \{\frac{F(i,j)}{Q(i,j)}\}$
4 **return** T

This quantization transforms block F to a new block T with $T(i,j) = \{\frac{F(i,j)}{Q(i,j)}\}$, where $\{x\}$ is the closest integer to x. This block will finally be encoded. Observe that in the transformed block F besides the entry $F(0,0)$ all other entries are relatively small numbers, which has

the effect that T mainly consists of 0s .

$$T = \begin{pmatrix} 15 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Coefficient $T(0,0)$, in this case 15, deserves special consideration. It is called DC term (direct current), while the other entries are denoted AC coefficients (alternate current).

13.5.4. Coding

Matrix T will finally be encoded by a Huffman code. We shall only sketch the procedure. First the DC term will be encoded by the difference to the DC term of the previously encoded block. For instance, if the previous DC term was 12, then $T(0,0)$ will be encoded as -3.

After that the AC coefficients are encoded according to the zig–zag order $T(0,1)$, $T(1,0)$, $T(2,0)$, $T(1,1)$, $T(0,2)$, $T(0,3)$, $T(1,2)$, etc.. In our example, this yields the sequence $0,-2,-1,-1,-1,0,0,-1$ followed by 55 zeros. This zig–zag order exploits the fact that there are long runs of successive zeros. These runs will be even more efficiently represented by application of ***run-length coding***, i. e., we encode the number of zeros before the next nonzero element in the sequence followed by this element.

Integers are written in such a way that small numbers have shorter representations. This is achieved by splitting their representation into size (number of bits to be reserved) and amplitude (the actual value). So, 0 has size 0, 1 and -1 have size 1. -3, -2, 2, and 3 have size 2, etc.

In our example this yields the sequence (2)(3) for the DC term followed by $(1,2)(-2)$, $(0,1)(-1)$, $(0,1)(-1)$, $(0,1)(-1)$, $(2,1)(-1)$, and a final $(0,0)$ as an end-of-block symbol indicating that only zeros follow from now on. $(1,2)(-2)$, for instance, means that there is 1 zero followed by an element of size 2 and amplitude -2.

These pairs are then assigned codewords from a Huffman code. There are different Huffman codes for the pairs (run, size) and for the amplitudes. These Huffman codes have to be specified and hence be included into the code of the image.

In the following pseudocode for the encoding of a single 8×8–block T we shall denote the different Huffman codes by encode-1, encode-2, encode-3.

```
RUN-LENGTH-CODE(T)
1 c ← encode-1(size(DC − T[0,0]))
2 c ← c|| encode-3(amplitude(DC − T[00]))
3 DC ← T[0,0]
4 for l ← 1 to 14
5     do for i ← 0 to l
6         do if l = 1 mod 2 then u ← i else u ← l − i
7             if T[u,l-u]=0 then run ← run + 1
8                           else c ← c|| encode-2(run, size(T[u, l − u]))
```

9 $c \leftarrow c \| \text{encode-3}(\text{amplitude}(T[u, l-u]))$
10 $run \leftarrow 0$
11 **if** $run > 0$ **then** encode-2$(0,0)$
12 **return** c

At the decoding end matrix T will be reconstructed. Finally, by multiplication of each entry $T(i,j)$ by the corresponding entry $Q(i,j)$ from the quantization matrix Q we obtain an approximation $\overline{F}$ to the block F, here

$$\overline{F} = \begin{pmatrix} 240 & 0 & -10 & 0 & 0 & 0 & 0 & 0 \\ -24 & -12 & 0 & 0 & 0 & 0 & 0 & 0 \\ -14 & -13 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

To $\overline{F}$ the inverse cosine transform is applied. This allows to decode the original 8×8–block f of the original image – in our example as

$$\overline{f} = \begin{pmatrix} 144 & 146 & 149 & 152 & 154 & 156 & 156 & 156 \\ 148 & 150 & 152 & 154 & 156 & 156 & 156 & 156 \\ 155 & 156 & 157 & 158 & 158 & 157 & 156 & 155 \\ 160 & 161 & 161 & 162 & 161 & 159 & 157 & 155 \\ 163 & 163 & 164 & 163 & 162 & 160 & 158 & 156 \\ 163 & 164 & 164 & 164 & 162 & 160 & 158 & 157 \\ 160 & 161 & 162 & 162 & 162 & 161 & 159 & 158 \\ 158 & 159 & 161 & 161 & 162 & 161 & 159 & 158 \end{pmatrix}.$$

Exercises

13.5-1 Find size and amplitude for the representation of the integers 5, −19, and 32.

13.5-2 Write the entries of the following matrix in zig – zag order.

$$\begin{pmatrix} 5 & 0 & -2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

How would this matrix be encoded if the difference of the DC term to the previous one was −2?

13.5-3 In our example after quantizing the sequence (2)(3), (1, 2)(−2), (0, 1)(−1), (0, 1)(−1), (0, 1)(−1), (2, 1)(−1), (0, 0) has to be encoded. Assume the Huffman codebooks would yield 011 to encode the difference 2 from the preceding block's DC, 0, 01, and

11 for the amplitudes -1, -2, and 3, respectively, and 1010, 00, 11011, and 11100 for the pairs $(0,0)$, $(0,1)$, $(1,2)$, and $(2,1)$, respectively. What would be the bitstream to be encoded for the 8×8 block in our example? How many bits would hence be necessary to compress this block?

13.5-4 What would be matrices T, $\overline{F}$ and $\overline{f}$, if we had used

$$Q = \begin{pmatrix} 8 & 6 & 5 & 8 & 12 & 20 & 26 & 31 \\ 6 & 6 & 7 & 10 & 13 & 29 & 30 & 28 \\ 7 & 7 & 8 & 12 & 20 & 29 & 35 & 28 \\ 7 & 9 & 11 & 15 & 26 & 44 & 40 & 31 \\ 9 & 11 & 19 & 28 & 34 & 55 & 52 & 39 \\ 12 & 18 & 28 & 32 & 41 & 52 & 57 & 46 \\ 25 & 32 & 39 & 44 & 57 & 61 & 60 & 51 \\ 36 & 46 & 48 & 49 & 56 & 50 & 57 & 50 \end{pmatrix}$$

for quantizing after the cosine transform in the block of our example?

13.5-5 What would be the zig–zag code in this case (assuming again that the DC term would have difference -3 from the previous DC term)?

13.5-6 For any sequence $(f(n))_{n=0,\ldots,m-1}$ define a new sequence $(\hat{f}(n))_{n=0,\ldots,2m-1}$ by

$$\hat{f}(n) = \begin{cases} f(n) & \text{for } n = 0, \ldots, m-1 \\ f(2m-1-n) & \text{for } n = m, \ldots, 2m-1 \end{cases}.$$

This sequence can be expanded to a Fourier series via

$$\hat{f}(n) = \frac{1}{\sqrt{2m}} \sum_{n=0}^{2m-1} \hat{g}(u) e^{i\frac{2\pi}{2m}nu} \quad \text{with } \hat{g}(u) = \frac{1}{\sqrt{2m}} \sum_{n=0}^{2m-1} \hat{f}(u) e^{-i\frac{2\pi}{2m}nu}, \quad i = \sqrt{-1}.$$

Show how the coefficients of the discrete cosine transform

$$F(u) = c_u \sum_{n=0}^{m-1} f(n) \cos(\frac{(2n+1)\pi u}{2m}, \quad c_u = \begin{cases} \frac{1}{\sqrt{m}} & \text{for } u = 0 \\ \frac{2}{\sqrt{m}} & \text{for } u \neq 0 \end{cases}$$

arise from this Fourier series.

Chapter notes

The frequency table of the letters in English texts is taken from [120]. The Huffman coding algorithm was introduced by Huffman in [66]. A pseudocode can be found in [33], where the Huffman coding algorithm is presented as a special Greedy algorithm. There are also adaptive or dynamic variants of Huffman coding, which adapt the Huffman code if it is no longer optimal for the actual frequency table, for the case that the probability distribution of the source is not known in advance. The "3/4-conjecture" on Kraft's inequality for fix-free codes is due to Ahlswede, Balkenhol, and Khachatrian [2].

Arithmetic coding has been introduced by Rissanen [94] and Pasco [90]. For a discussion of implementation questions see [78, 78, 124]. In the section on modelling we are

following the presentation of Willems, Shtarkov and Tjalkens in [123]. The exact calculations can be found in their original paper [122] which received the Best Paper Award of the IEEE Information Theory Society in 1996. The Krichevsky-Trofimov estimator had been introduced in [75].

We presented the two original algorithms LZ77 and LZ78 [132, 133] due to Lempel and Ziv. Many variants, modifications and extensions have been developed since that – concerning the handling of the dictionary, the pointers, the behaviour after the dictionary is complete, etc. For a description, see, for instance, [14] or [15]. Most of the prominent tools for data compression are variations of Ziv-Lempel coding. For example "zip" and "gzip" are based on LZ77 and a variant of LZ78 is used by the program "compress".

The Burrows-Wheeler transform was introduced in the technical report [22]. It became popular in the sequel, especially because of the Unix compression tool "bzip" based on the Burrows-Wheeler transform, which outperformed most dictionary – based tools on several benchmark files. Also it avoids arithmetic coding, for which patent rights have to be taken into consideration. Further investigations on the Burrows-Wheeler transform have been carried out, for instance in [6, 45, 76].

We only sketched the basics behind lossy image compression, especially the preparation of the data for application of techniques as Huffman coding. For a detailed discussion we refer to [110], where also the new JPEG2000 standard is described. Our example is taken from [117].

JPEG – short for Joint Photographic Experts Group – is very flexible. For instance, it also supports lossless data compression. All the topics presented in the section on image compression are not unique. There are models involving more basic colours and further transforms besides the YC_bC_r-transform (for which even different scaling factors for the chrominance channels were used, the formula presented here is from [110]). The cosine transform may be replaced by another operation like a wavelet transform. Further, there is freedom to choose the quantization matrix, responsible for the quality of the compressed image, and the Huffman code. On the other hand, this has the effect that these parameters have to be explicitly specified and hence are part of the coded image.

The ideas behind procedures for video and sound compression are rather similar to those for image compression. In principal, they follow the same steps. The amount of data in these cases, however, is much bigger. Again information is lost by removing irrelevant information not realizable by the human eye or ear (for instance by psychoacoustic models) and by quantizing, where the quality should not be reduced significantly. More refined quantizing methods are applied in these cases.

Most information on data compression algorithms can be found in literature on Information Theory, for instance [35, 60], since the analysis of the achievable compression rates requires knowledge of source coding theory. Recently, there have appeared several books on data compression, for instance [15, 61, 86, 97, 99], to which we refer to further reading. The benchmark files of the Calgary Corpus and the Canterbury Corpus are available under [23] or [24].

Problems

13-1. Adaptive Huffman codes

Dynamic and adaptive Huffman coding is based on the following property. A binary code tree has the sibling property if each node has a sibling and if the nodes can be listed in order of nonincreasing probabilities with each node being adjacent to its sibling. Show that a binary prefix code is a Huffman code exactly if the corresponding code tree has the sibling property.

13-2. Generalizations of Kraft's inequality

In the proof of Kraft's inequality it is essential to order the lengths $L(1) \leq \cdots \leq L(a)$. Show that the construction of a prefix code for given lengths $2, 1, 2$ is not possible if we are not allowed to order the lengths. This scenario of unordered lengths occurs with the Shannon-Fano-Elias code and in the theory of alphabetic codes, which are related to special search problems. Show that in this case a prefix code with lengths $L(1) \leq \cdots \leq L(a)$ exists if and only if

$$\sum_{x \in \mathcal{X}} 2^{-L(x)} \leq \frac{1}{2} \, .$$

If we additionally require the prefix codes to be also suffix-free i. e., no codeword is the end of another one, it is an open problem to show that Kraft's inequality holds with the 1 on the right–hand side replaced by 3/4, i. e.,

$$\sum_{x \in \mathcal{X}} 2^{-L(x)} \leq \frac{3}{4} \, .$$

13-3. Redundancy of Krichevsky-Trofimov estimator

Show that using the Krichevsky-Trofimov estimator, when parameter θ of a discrete memoryless source is unknown, the individual redundancy of sequence x^n is at most $\frac{1}{2} \lg n + 3$ for all sequences x^n and all $\theta \in \{0, 1\}$.

13-4. Alternatives to move-to-front codes

Find further procedures which like move-to-front coding prepare the text for compression after application of the Burrows-Wheeler transform.

14. Computer graphics

Computer Graphics algorithms create and render ***virtual worlds*** stored in the computer memory. The virtual world model may contain ***shapes*** (points, line segments, surfaces, solid objects etc.), which are represented by digital numbers. ***Rendering*** computes the displayed ***image*** of the virtual world from a given virtual camera. The image consists of small rectangles, called ***pixels***. A pixel has a unique colour, thus it is sufficient to solve the rendering problem for a single point in each pixel. This point is usually the centre of the pixel. Rendering finds that shape which is visible through this point and writes its visible colour into the pixel. In this chapter we discuss the creation of virtual worlds and the determination of the visible shapes.

14.1. Fundamentals of analytic geometry

The base set of our examination is the Euclidean ***space***. In computer algorithms the elements of this space should be described by numbers. The branch of geometry describing the elements of space by numbers is the ***analytic geometry***. The basic tools of analytic geometry are the vector and the coordinate system.

Definition 14.1 *A **vector** is an oriented line segment or a **translation** that is defined by its direction and length. A vector is denoted by $\vec{v}$.*

The length of the vector is also called its ***absolute value***, and is denoted by $|\vec{v}|$. Vectors can be added, resulting in a new vector that corresponds to subsequent translations. Addition is denoted by $\vec{v}_1 + \vec{v}_2 = \vec{v}$. Vectors can be multiplied by scalar values, resulting also in a vector ($\lambda \cdot \vec{v}_1 = \vec{v}$), which translates at the same direction as $\vec{v}_1$, but the length of translation is scaled by λ.

The ***dot product*** of two vectors is a *scalar* that is equal to the product of the lengths of the two vectors and the cosine of their angle:

$$\vec{v}_1 \cdot \vec{v}_2 = |\vec{v}_1| \cdot |\vec{v}_2| \cdot \cos\alpha, \quad \text{where } \alpha \text{ is the angle between } \vec{v}_1 \text{ and } \vec{v}_2.$$

Two vectors are said to be ***orthogonal*** if their dot product is zero.

On the other hand, the ***cross product*** of two vectors is a *vector* that is orthogonal to the plane of the two vectors and its length is equal to the product of the lengths of the two

vectors and the sine of their angle:

$$\vec{v}_1 \times \vec{v}_2 = \vec{v}, \quad \text{where } \vec{v} \text{ is orthogonal to } \vec{v}_1 \text{ and } \vec{v}_2, \text{ and } |\vec{v}| = |\vec{v}_1| \cdot |\vec{v}_2| \cdot \sin\alpha.$$

There are two possible orthogonal vectors, from which that alternative is selected where our middle finger of the right hand would point if our thumb were pointing to the first and our forefinger to the second vector (***right hand rule***). Two vectors are said to be ***parallel*** if their cross product is zero.

14.1.1. Cartesian coordinate system

Any vector $\vec{v}$ of a plane can be expressed as the linear combination of two, non-parallel vectors $\vec{i}, \vec{j}$ in this plane, that is

$$\vec{v} = x \cdot \vec{i} + y \cdot \vec{j}.$$

Similarly, any vector $\vec{v}$ in the three-dimensional space can be unambiguously defined by the linear combination of three, not coplanar vectors:

$$\vec{v} = x \cdot \vec{i} + y \cdot \vec{j} + z \cdot \vec{k}.$$

Vectors $\vec{i}, \vec{j}, \vec{k}$ are called ***basis vectors***, while scalars x, y, z are referred to as ***coordinates***. We shall assume that the basis vectors have unit length and they are orthogonal to each other. Having defined the basis vectors any other vector can unambiguously be expressed by three scalars, i.e. by its coordinates.

A ***point*** can be defined by that vector which translates the reference point, called ***origin***, to the given point. In this case the translating vector is the ***place vector*** of the given point.

The origin and the basis vectors constitute the ***Cartesian coordinate system***, which is the basic tool to describe the points of the Euclidean plane or space by numbers.

The Cartesian coordinate system is the algebraic basis of the Euclidean geometry, which means that scalar triplets of Cartesian coordinates can be paired with the points of the space, and having made a correspondence between algebraic and geometric concepts, the axioms and the theorems of the Euclidean geometry can be proven by algebraic means.

Exercises

14.1-1 Prove that there is a one-to-one mapping between Cartesian coordinate triplets and points of the three-dimensional space.

14.1-2 Prove that if the basis vectors have unit length and are orthogonal to each other, then $(x_1, y_1, z_1) \cdot (x_2, y_2, z_2) = x_1x_2 + y_1y_2 + z_1z_2$.

14.1-3 Prove that the dot product is distributive with respect to the vector addition.

14.2. Description of point sets with equations

Coordinate systems provide means to define points by numbers. A set of conditions on these numbers, on the other hand, may define point sets. The set of conditions is usually an equation. The points defined by the solutions of these equations form the set.

solid	$f(x,y,z)$ implicit function
sphere of radius R	$R^2 - x^2 - y^2 - z^2$
block of size $2a, 2b, 2c$	$\min\{a - \lvert x\rvert, b - \lvert y\rvert, c - \lvert z\rvert\}$
torus of axis z, radii r (tube) and R (hole)	$r^2 - z^2 - (R - \sqrt{x^2 + y^2})^2$

Figure 14.1. Implicit functions defining the sphere, the block, and the torus.

14.2.1. Solids

A ***solid*** is a subset of the three-dimensional Euclidean space. To define this subset, continuous function f is used which maps the coordinates of points onto the set of real numbers. We say that a point belongs to the solid if the coordinates of the point satisfy the following implicit inequality:

$$f(x,y,z) \geq 0 \,.$$

Points satisfying inequality $f(x,y,z) > 0$ are the ***internal points,*** while points defined by $f(x,y,z) < 0$ are the ***external points.*** Because of the continuity of function f, points satisfying equality $f(x,y,z) = 0$ are between external and internal points and are called the ***boundary surface*** of the solid. Intuitively, function f describes the distance between a point and the boundary surface.

We note that we usually do not consider any subset of the space as a solid, but also require that the point set does not have lower dimensional degeneration (e.g. hanging lines or surfaces), i.e. that arbitrarily small neighbourhoods of each point of the boundary surface contain internal points.

Figure 14.1 defines the implicit functions of the sphere, the box and the torus.

14.2.2. Surfaces

Points having coordinates that satisfy equation $f(x,y,z) = 0$ are the boundary points of the solid, which form a ***surface***. Surfaces can thus be defined by this ***implicit equation.*** Since points can also be given by the place vectors, the implicit equation can be formulated for the place vectors as well:

$$f(\vec{r}) = 0 \,.$$

A surface may have many different equations. For example, equations $f(x,y,z) = 0$, $f^2(x,y,z) = 0$, and $2 \cdot f^3(x,y,z) = 0$ are algebraically different, but they define the same set of points.

A plane of normal $\vec{n}$ and place vector $\vec{r}_0$ contains those points for which vector $\vec{r} - \vec{r}_0$ is perpendicular to the normal, thus their dot product is zero. Based on this, the points of a plane are defined by the following vector and scalar equations:

$$(\vec{r} - \vec{r}_0) \cdot \vec{n} = 0, \qquad n_x \cdot x + n_y \cdot y + n_z \cdot z + d = 0 \,, \tag{14.1}$$

where n_x, n_y, n_z are the coordinates of the normal and $d = -\vec{r}_0 \cdot \vec{n}$. If the normal vector has unit length, then d expresses the signed distance between the plane and the origin of the coordinate system. Two planes are said to be ***parallel*** if their normals are parallel.

Instead of using implicit equations, surfaces can also be defined by ***parametric forms***. In this case, the Cartesian coordinates of surface points are functions of two independent

solid	$x(u,v)$	$y(u,v)$	$z(u,v)$
sphere of radius R	$R \cdot \cos 2\pi u \cdot \sin \pi v$	$R \cdot \sin 2\pi u \cdot \sin \pi v$	$R \cdot \cos \pi v$
cylinder of radius R, axis z, and of height h	$R \cdot \cos 2\pi u$	$R \cdot \sin 2\pi u$	$h \cdot v$
cone of radius R, axis z, and of height h	$R \cdot (1-v) \cdot \cos 2\pi u$	$R \cdot (1-v) \cdot \sin 2\pi u$	$h \cdot v$

Figure 14.2. Parametric forms of the sphere, the cylinder, and the cone, where $u, v \in [0,1]$.

variables. Denoting these free parameters by u and v, the parametric equations of the surface are:

$$x = x(u,v), \qquad y = y(u,v), \qquad z = z(u,v), \qquad u \in [u_{\min}, u_{\max}], \quad v \in [v_{\min}, v_{\max}] \ .$$

The implicit equation of a surface can be obtained from the parametric equations by eliminating free parameters u, v. Figure 14.2 includes the parametric forms of the sphere, the cylinder and the cone.

Parametric forms can also be defined directly for the place vectors:

$$\vec{r} = \vec{r}(u,v) \ .$$

A ***triangle*** is the ***convex combination*** of points $\vec{p}_1, \vec{p}_2$, and $\vec{p}_3$, that is

$$\vec{r}(\alpha,\beta,\gamma) = \alpha \cdot \vec{p}_1 + \beta \cdot \vec{p}_2 + \gamma \cdot \vec{p}_3, \quad \text{where } \alpha,\beta,\gamma \geq 0 \text{ and } \alpha+\beta+\gamma = 1 \ .$$

From this definition we can obtain the usual two-variate parametric form substituting α by u, β by v, and γ by $(1-u-v)$:

$$\vec{r}(u,v) = u \cdot \vec{p}_1 + v \cdot \vec{p}_2 + (1-u-v) \cdot \vec{p}_3, \quad \text{where } u,v \geq 0 \text{ and } u+v \leq 1 \ .$$

14.2.3. Curves

By intersecting two surfaces, we obtain a ***curve*** that may be defined formally by the implicit equations of the two intersecting surfaces

$$f_1(x,y,z) = f_2(x,y,z) = 0,$$

but this is needlessly complicated. Instead, let us consider the parametric forms of the two surfaces, given as $\vec{r}_1(u_1,v_1)$ and $\vec{r}_2(u_2,v_2)$, respectively. The points of the intersection satisfy vector equation $\vec{r}_1(u_1,v_1) = \vec{r}_2(u_2,v_2)$, which corresponds to three scalar equations, one for each coordinate of the three-dimensional space. Thus we can eliminate three from the four unknowns (u_1,v_1,u_2,v_2), and obtain a one-variate parametric equation for the coordinates of the curve points:

$$x = x(t), \qquad y = y(t), \qquad z = z(t), \qquad t \in [t_{\min}, t_{\max}].$$

Similarly, we can use the vector form:

$$\vec{r} = \vec{r}(t), \qquad t \in [t_{\min}, t_{\max}].$$

Figure 14.3 includes the parametric equations of the ellipse, the helix, and the line segment.

Note that we can define curves on a surface by fixing one of free parameters u, v. For

test	$x(u,v)$	$y(u,v)$	$z(u,v)$
ellipse of main axes $2a$, $2b$ on plane $z=0$	$a\cdot\cos 2\pi t$	$b\cdot\sin 2\pi t$	0
helix of radius R, axis z, and elevation h	$R\cdot\cos 2\pi t$	$R\cdot\sin 2\pi t$	$h\cdot t$
line segment between points (x_1,y_1,z_1) and (x_2,y_2,z_2)	$x_1(1-t)+x_2t$	$y_1(1-t)+y_2t$	$z_1(1-t)+z_2t$

Figure 14.3. Parametric forms of the ellipse, the helix, and the line segment, where $t\in[0,1]$.

example, by fixing v the parametric form of the resulting curve is $\vec{r}_v(u)=\vec{r}(u,v)$. These curves are called ***iso-parametric curves***.

Let us select a point of a ***line*** and call the place vector of this point the ***place vector of the line.*** Any other point of the line can be obtained by the translation of this point along the same direction vector. Denoting the place vector by $\vec{r}_0$ and the ***direction vector*** by $\vec{v}$, the equation of the ***line*** is:

$$\vec{r}(t)=r_0+\vec{v}\cdot t,\qquad t\in(-\infty,\infty)\ . \tag{14.2}$$

Two lines are said to be ***parallel*** if their direction vectors are parallel.

Instead of the complete line, we can also specify the points of a line segment if parameter t is restricted to an interval. For example, the ***equation of the line segment*** between points $\vec{r}_1,\vec{r}_2$ is:

$$\vec{r}(t)=\vec{r}_1+(\vec{r}_2-\vec{r}_1)\cdot t=\vec{r}_1\cdot(1-t)+\vec{r}_2\cdot t,\qquad t\in[0,1]\ . \tag{14.3}$$

According to this definition, the points of a line segment are the ***convex-combinations*** of the endpoints.

14.2.4. Normal vectors

In computer graphics we often need the normal vectors of the surfaces (i.e. the normal vector of the tangent plane of the surface). Let us take an example. A mirror reflects light in a way that the incident direction, the normal vector, and the reflection direction are in the same plane, and the angle between the normal and the incident direction equals to the angle between the normal and the reflection direction. To carry out such and similar computations, we need methods to obtain the normal of the surface.

The equation of the tangent plane is obtained as the first order Taylor approximation of the implicit equation around point (x_0,y_0,z_0):

$$f(x,y,z)=f(x_0+(x-x_0),y_0+(y-y_0),z_0+(z-z_0))\approx$$

$$f(x_0,y_0,z_0)+\frac{\partial f}{\partial x}\cdot(x-x_0)+\frac{\partial f}{\partial y}\cdot(y-y_0)+\frac{\partial f}{\partial z}\cdot(z-z_0)\ .$$

Points (x_0,y_0,z_0) and (x,y,z) are on the surface, thus $f(x_0,y_0,z_0)=0$ and $f(x,y,z)=0$, resulting in the following equation of the ***tangent plane:***

$$\frac{\partial f}{\partial x}\cdot(x-x_0)+\frac{\partial f}{\partial y}\cdot(y-y_0)+\frac{\partial f}{\partial z}\cdot(z-z_0)=0\ .$$

Comparing this equation to equation (14.1), we can realize that the normal vector of the

tangent plane is

$$\vec{n} = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right) = \mathrm{grad} f \ . \tag{14.4}$$

The normal vector of parametric surfaces can be obtained by examining the iso-parametric curves. The tangent of curve $\vec{r}_v(u)$ defined by fixing parameter v is obtained by the first-order Taylor approximation:

$$\vec{r}_v(u) = \vec{r}_v(u_0 + (u - u_0)) \approx \vec{r}_v(u_0) + \frac{d\vec{r}_v}{du} \cdot (u - u_0) = \vec{r}_v(u_0) + \frac{\partial \vec{r}}{\partial u} \cdot (u - u_0) \ .$$

Comparing this approximation to equation (14.2) describing a line, we conclude that the direction vector of the tangent line is $\partial\vec{r}/\partial u$. The tangent lines of the curves running on a surface are in the tangent plane of the surface, making the normal vector perpendicular to the direction vectors of these lines. In order to find the normal vector, both the tangent line of curve $\vec{r}_v(u)$ and the tangent line of curve $\vec{r}_u(v)$ are computed, and their cross product is evaluated since the result of the cross product is perpendicular to the multiplied vectors. The normal of surface $\vec{r}(u, v)$ is then

$$\vec{n} = \frac{\partial \vec{r}}{\partial u} \times \frac{\partial \vec{r}}{\partial v} \ . \tag{14.5}$$

14.2.5. Curve modelling

Parametric and implicit equations trace back the geometric design of the virtual world to the solution of these equations. However, these equations are often not intuitive enough, thus they cannot be used directly during design. It would not be reasonable to expect the designer working on a human face or on a car to directly specify the equations of these objects. Clearly, indirect methods are needed which require intuitive data from the designer and define these equations automatically. One category of these indirect approaches apply control points. Another category of methods work with elementary building blocks (box, sphere, cone, etc.) and with set operations.

Let us discuss first how the method based on control points can define curves. Suppose that the designer defined points $\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_m$, and that parametric curve of equation $\vec{r} = \vec{r}(t)$ should be found which „follows" these points. For the time being, the curve is not required to go through these control points.

We use the analogy of the centre of mass of mechanical systems to construct our curve. Assume that we have sand of unit mass, which is distributed at the control points. If a control point has most of the sand, then the centre of mass is close to this point. Controlling the distribution of the sand as a function of parameter t to give the main influence to different control points one after the other, the centre of mass will travel through a curve running close to the control points.

Let us put weights $B_0(t), B_1(t), \ldots, B_m(t)$ at control points at parameter t. These weighting functions are also called the ***basis functions*** of the curve. Since unit weight is distributed, we require that for each t the following identity holds:

$$\sum_{i=0}^{m} B_i(t) = 1 \ .$$

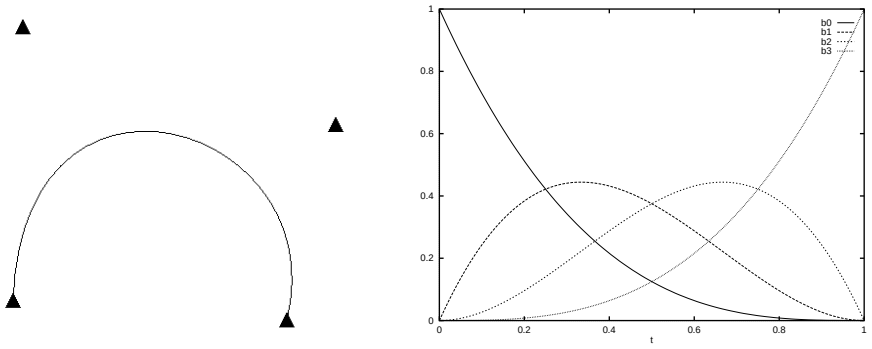


Figure 14.4. A Bézier curve defined by four control points and the respective basis functions ($m = 3$).

For some t, the curve is the centre of mass of this mechanical system:

$$\vec{r}(t) = \frac{\sum_{i=0}^{m} B_i(t) \cdot \vec{r}_i}{\sum_{i=0}^{m} B_i(t)} = \sum_{i=0}^{m} B_i(t) \cdot \vec{r}_i \ .$$

Note that the reason of distributing sand of unit mass is that this decision makes the denominator of the fraction equal to 1. To make the analogy complete, the basis functions cannot be negative since the mass is always non negative. The centre of mass of a point system is always in the ***convex hull***[1] of the participating points, thus if the basis functions are non negative, then the curve remains in the convex hull of the control points.

The properties of the curves are determined by the basis functions. Let us now discuss two popular basis function systems, namely the basis functions of the Bézier curves and the B-spline curves.

Bézier curve

Pierre Bézier, a designer working at Renault, proposed the ***Bernstein polynomials*** as basis functions. Bernstein polynomials can be obtained as the expansion of $1^m = (t + (1 - t))^m$ according to binomial theorem:

$$(t + (1 - t))^m = \sum_{i=0}^{m} \binom{m}{i} \cdot t^i \cdot (1 - t)^{m-i} \ .$$

The basis functions of ***Bézier curves*** are the terms of this sum ($i = 0, 1, \ldots, m$):

$$B_{i,m}^{\text{Bezier}}(t) = \binom{m}{i} \cdot t^i \cdot (1 - t)^{m-i} \ . \tag{14.6}$$

According to the introduction of Bernstein polynomials, it is obvious that they really meet condition $\sum_{i=0}^{m} B_i(t) = 1$ and $B_i(t) \geq 0$ in $t \in [0, 1]$, which guarantees that Bézier curves

[1] The convex hull of a point system is by definition the minimal convex set containing the point system.

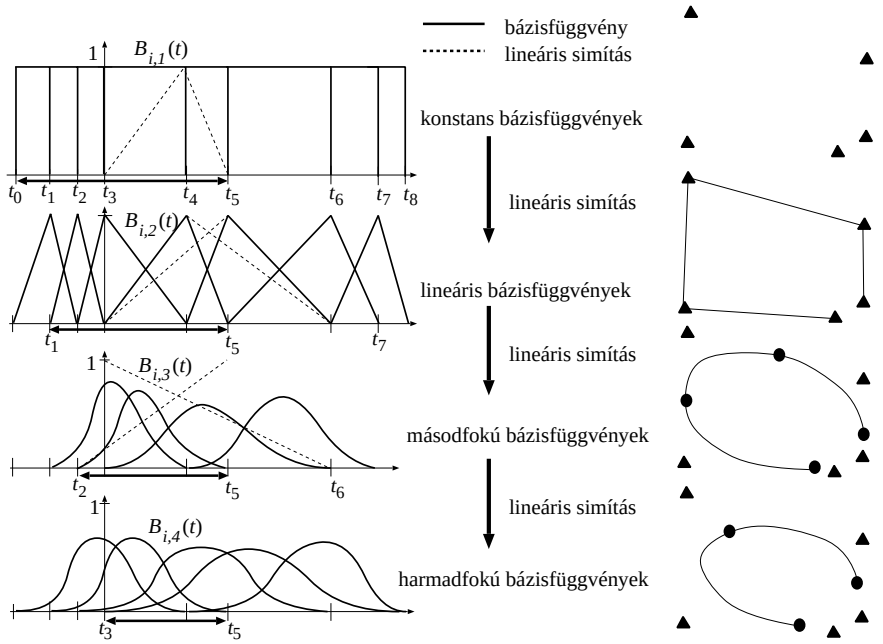


Figure 14.5. Construction of B-spline basis functions. A higher order basis function is obtained by blending two consecutive basis functions on the previous level using a linearly increasing and a linearly decreasing weighting, respectively. Here the number of control points is 5, i.e. $m = 4$. Arrows indicate useful interval $[t_{k-1}, t_{m+1}]$ where we can find $m + 1$ number of basis functions that add up to 1. The right side of the figure depicts control points with triangles and curve points corresponding to the knot values by circles.

are always in the convex hulls of their control points. The basis functions and the shape of the Bézier curve are shown in figure 14.4. At parameter value $t = 0$ the first basis function is 1, while the others are zero, therefore the curve starts at the first control point. Similarly, at parameter value $t = 1$ the curve arrives at the last control point. At other parameter values, all basis functions are positive, thus they simultaneously affect the curve. Consequently, the curve usually does not go through the other control points.

B-spline

The basis functions of the ***B-spline*** can be constructed applying a sequence of linear blending. A B-spline weights the $m + 1$ number of control points by $(k - 1)$-degree polynomials. Value k is called the ***order*** of the curve, which expresses the smoothness of the curve. Let us take a non-decreasing series of $m + k + 1$ parameter values, called the ***knot vector***:

$$\mathbf{t} = [t_0, t_1, \ldots, t_{m+k}], \quad t_0 \le t_1 \le \cdots \le t_{m+k} \ .$$

By definition, the ith first order basis function is 1 in the ith interval, and zero elsewhere (figure 14.5):

$$B_{i,1}^{\mathrm{BS}}(t) = \begin{cases} 1, & \text{if } t_i \le t < t_{i+1} \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

Using this definition, $m + k$ number of first order basis functions are established, which

are non-negative zero-degree polynomials that sum up to 1 for all $t \in [t_0, t_{m+k})$ parameters. These basis functions have too low degree since the centre of mass is not even a curve, but jumps from control point to control point.

The order of basis functions, as well as the smoothness of the curve, can be increased by blending two consecutive basis functions with linear weighting (figure 14.5). The first basis function is weighted by linearly increasing factor $(t - t_i)/(t_{i+1} - t_i)$ in domain $t_i \le t < t_{i+1}$, where the basis function is non-zero. The next basis function, on the other hand, is scaled by linearly decreasing factor $(t_{i+2} - t)/(t_{i+2} - t_{i+1})$ in its domain $t_{i+1} \le t < t_{i+2}$ where it is non zero. The two weighted basis functions are added to obtain the tent-like second order basis functions. Note that while a first order basis function is non-zero in a single interval, the second order basis functions expand to two intervals. Since the construction makes a new basis function from every pair of consecutive lower order basis functions, the number of new basis functions is one less than that of the original ones. We have just $m + k - 1$ second order basis functions. Except for the first and the last first order basis functions, all of them are used once with linearly increasing and once with linearly decreasing weighting, thus with the exception of the first and the last intervals, i.e. in $[t_1, t_{m+k-1}]$, the new basis functions also sum up to 1.

The second order basis functions are first degree polynomials. The degree of basis functions, i.e. the order of the curve, can be arbitrarily increased by the recursive application of the presented blending method. The dependence of the next order basis functions on the previous order ones is as follows:

$$B^{\mathrm{BS}}_{i,k}(t) = \frac{(t - t_i)B^{\mathrm{BS}}_{i,k-1}(t)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - t)B^{\mathrm{BS}}_{i+1,k-1}(t)}{t_{i+k} - t_{i+1}}, \qquad \text{if } k > 1\ .$$

Note that we always take two consecutive basis functions and weight them in their non-zero domain (i.e. in the interval where they are non-zero) with linearly increasing factor $(t - t_i)/(t_{i+k-1} - t_i)$ and with linearly decreasing factor $(t_{i+k} - t)/(t_{i+k} - t_{i+1})$, respectively. The two weighted functions are summed to obtain the higher order, and therefore smoother basis function. Repeating this operation $(k - 1)$ times, k-order basis functions are generated, which sum up to 1 in interval $[t_{k-1}, t_{m+1}]$. The knot vector may have elements that are the same, thus the length of the intervals may be zero. Such intervals result in 0/0 like fractions, which must be replaced by value 1 in the implementation of the construction.

The value of the ith k-order basis function at parameter t can be computed with the following ***Cox-deBoor algorithm***:

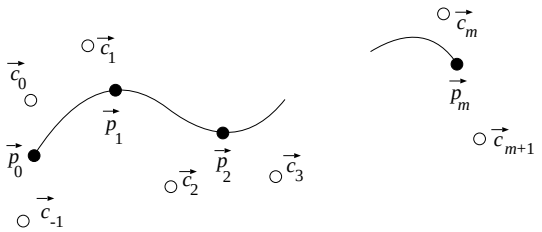


Figure 14.6. A B-spline interpolation. Based on points $\vec{p}_0, \ldots, \vec{p}_m$ to be interpolated, control points $\vec{c}_{-1}, \ldots, \vec{c}_{m+1}$ are computed to make the start and end points of the segments equal to the interpolated points.

B-SPLINE$(i, k, t, \mathbf{t})$

```
 1  if k = 1                                                    ▷ Trivial case.
 2     then if t_i ≤ t < t_{i+1}
 3             then return 1
 4             else return 0
 5  if t_{i+k-1} − t_i > 0
 6     then b_1 ← (t − t_i)/(t_{i+k-1} − t_i)        ▷ Previous with linearly increasing weight.
 7     else  b_1 ← 1                                             ▷ Here: 0/0 = 1.
 8  if t_{i+k} − t_{i+1} > 0
 9     then b_2 ← (t_{i+k} − t)/(t_{i+k} − t_{i+1})     ▷ Next with linearly decreasing weight.
10     else  b_2 ← 1                                             ▷ Here: 0/0 = 1.
11  B ← b_1 · B-SPLINE(i, k − 1, t, t) + b_2 · B-SPLINE(i + 1, k − 1, t, t)   ▷ Recursion.
12  return B
```

In practice, we usually use fourth-order basis functions ($k = 4$), which are third-degree polynomials, and define curves that can be continuously differentiated twice. The reason is that bent rods and motion paths following the Newton laws also have this property.

While the number of control points is greater than the order of the curve, the basis functions are non-zero only in a part of the valid parameter set. This means that a control point affects just a part of the curve. Moving this control point, the change of the curve is ***local.*** Local control is a very important property since the designer can adjust the shape of the curve without destroying its general form.

A fourth-order B-spline usually does not go through its control points. If we wish to use it for interpolation, the control points should be calculated from the points to be interpolated. Suppose that we need a curve which visits points $\vec{p}_0, \vec{p}_1, \ldots, \vec{p}_m$ at parameter values $t_0 = 0, t_1 = 1, \ldots, t_m = m$, respectively (figure 14.6). To find such a curve, control points $[\vec{c}_{-1}, \vec{c}_0, \vec{c}_1, \ldots, \vec{c}_{m+1}]$ should be found to meet the following interpolation criteria:

$$\vec{r}(t_j) = \sum_{i=-1}^{m+1} \vec{c}_i \cdot B_{i,4}^{\mathrm{BS}}(t_j) = \vec{p}_j, \qquad j = 0, 1, \ldots, m\ .$$

These criteria can be formalized as $m + 1$ linear equations with $m + 3$ unknowns, thus the solution is ambiguous. To make the solution unambiguous, two additional conditions should be imposed. For example, we can set the derivatives (for motion paths, the speed) at the start and end points.

B-spline curves can be further generalized by defining the influence of the ith control point as the product of B-spline basis function $B_i(t)$ and additional weight w_i of the control point. The curve obtained this way is called the Non-Uniform Rational B-Spline, abbreviated as ***NURBS***, which is very popular in commercial geometric modelling systems.

Using the mechanical analogy again, the mass put at the ith control point is $w_i B_i(t)$, thus the centre of mass is:

$$\vec{r}(t) = \frac{\sum_{i=0}^{m} w_i B_i^{\mathrm{BS}}(t) \cdot \vec{r}_i}{\sum_{j=0}^{m} w_j B_j^{\mathrm{BS}}(t)} = \sum_{i=0}^{m} B_i^{\mathrm{NURBS}}(t) \cdot \vec{r}_i \ .$$

The correspondence between B-spline and NURBS basis functions is as follows:

$$B_i^{\mathrm{NURBS}}(t) = \frac{w_i B_i^{\mathrm{BS}}(t)}{\sum_{j=0}^{m} w_j B_j^{\mathrm{BS}}(t)} \ .$$

Since B-spline basis functions are polynomials, NURBS basis functions are rational functions. NURBS can describe quadratic curves (e.g. circle, ellipse, etc.) without any approximation error.

14.2.6. Surface modelling

Parametric surfaces are defined by two variate functions $\vec{r}(u, v)$. Instead of specifying this function directly, we can take finite number of control points $\vec{r}_{ij}$ which are weighted with the basis functions to obtain the parametric function:

$$\vec{r}(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \vec{r}_{ij} \cdot B_{ij}(u, v) \ . \tag{14.7}$$

Similarly to curves, basis functions are expected to sum up to 1, i.e. $\sum_{i=0}^{n} \sum_{j=0}^{m} B_{ij}(u, v) = 1$ everywhere. If this requirement is met, we can imagine that the control points have masses $B_{ij}(u, v)$ depending on parameters u, v, and the centre of mass is the surface point corresponding to parameter pair u, v.

Basis functions $B_{ij}(u, v)$ are similar to those of curves. Let us fix parameter v. Changing parameter u, curve $\vec{r}_v(u)$ is obtained on the surface. This curve can be defined by the discussed curve definition methods:

$$\vec{r}_v(u) = \sum_{i=0}^{n} B_i(u) \cdot \vec{r}_i \ , \tag{14.8}$$

where $B_i(u)$ is the basis function of the selected curve type.

Of course, fixing v differently we obtain another curve of the surface. Since a curve of a given type is unambiguously defined by the control points, control points $\vec{r}_i$ must depend on the fixed v value. As parameter v changes, control point $\vec{r}_i = \vec{r}_i(v)$ also runs on a curve, which can be defined by control points $\vec{r}_{i,0}, \vec{r}_{i,2}, \ldots, \vec{r}_{i,m}$:

$$\vec{r}_i(v) = \sum_{j=0}^{m} B_j(v) \cdot \vec{r}_{ij} \ .$$

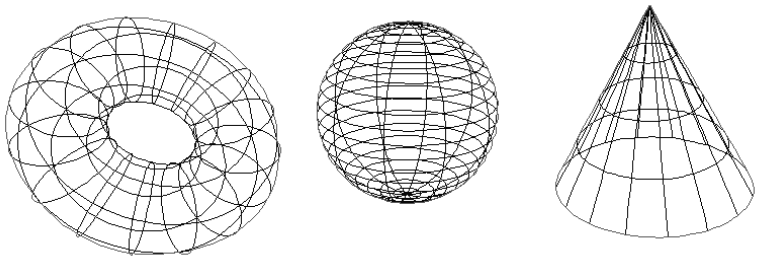


Figure 14.7. Iso-parametric curves of surface.

Substituting this into equation (14.8), the parametric equation of the surface is:

$$\vec{r}(u,v) = \vec{r}_v(u) = \sum_{i=0}^{n} B_i(u) \left(\sum_{j=0}^{m} B_j(v) \cdot \vec{r}_{ij} \right) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i(u) B_j(v) \cdot \vec{r}_{ij} \ .$$

Unlike curves, the control points of a surface form a two-dimensional grid. The two-dimensional basis functions are obtained as the product of one-variate basis functions parameterized by u and v, respectively.

14.2.7. Solid modelling with blobs

Free form solids – similarly to parametric curves and surfaces – can also be specified by finite number of control points. For each control point $\vec{r}_i$, let us assign influence function $h(R_i)$, which expresses the influence of this control point at distance $R_i = |\vec{r} - \vec{r}_i|$. By definition, the solid contains those points where the total influence of the control points is not smaller than threshold T (figure 14.8):

$$f(\vec{r}) = \sum_{i=0}^{m} h_i(R_i) - T \geq 0, \qquad \text{where } R_i = |\vec{r} - \vec{r}_i| \ .$$

With a single control point a sphere can be modelled. Spheres of multiple control points are combined together to result in an object having smooth surface (figure 14.8). The influence of a single point can be defined by an arbitrary decreasing function that converges to zero at infinity. For example, Blinn proposed the

$$h_i(R) = a_i \cdot e^{-b_i R^2}$$

influence functions for his ***blob method.***

14.2.8. Constructive solid geometry

Another type of solid modelling is ***constructive solid geometry*** (***CSG*** for short), which builds complex solids from primitive solids applying set operations (union, intersection, difference) (figures 14.9 and 14.10). Primitives usually include the box, the sphere, the cone, the cylinder, the half-space, etc. whose implicit functions are known.

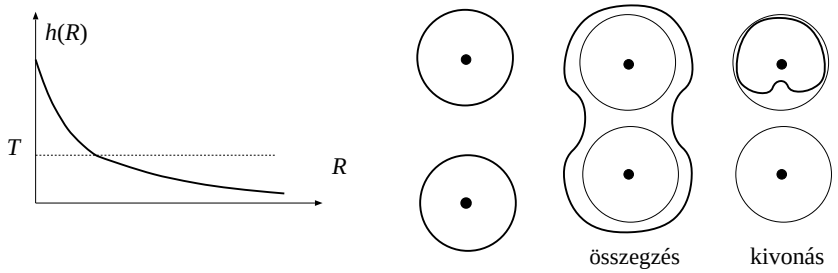


Figure 14.8. The influence decreases with the distance. Spheres of influence of similar signs increase, of different signs decrease each other.

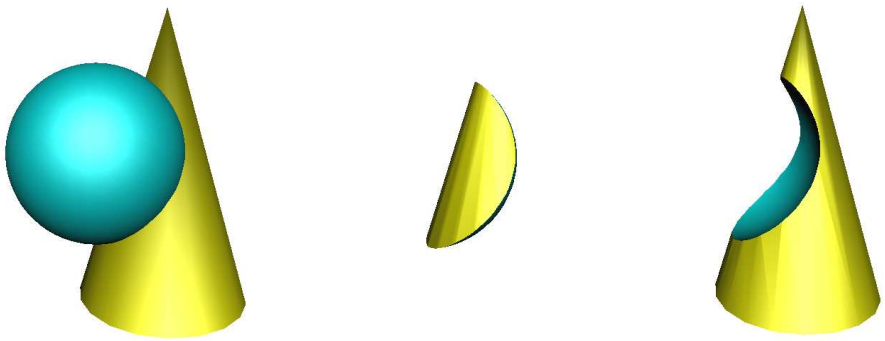


Figure 14.9. The operations of constructive solid geometry for a cone of implicit function f and for a sphere of implicit function g: union ($\max(f,g)$), intersection ($\min(f,g)$), and difference ($\min(f,-g)$).

The results of the set operations can be obtained from the implicit functions of the solids taking part of this operation:

- intersection of f and g: $\min(f,g)$;
- union of f and g: $\max(f,g)$.
- complement of f: $-f$.
- difference of f and g: $\min(f,-g)$.

Implicit functions also allow to ***morph*** between two solids. Suppose that two objects, for example, a box of implicit function f_1 and a sphere of implicit function f_2 need to be morphed. To define a new object, which is similar to the first object with percentage t and to the second object with percentage $(1-t)$, the two implicit equations are weighted appropriately:

$$f^{morph}(x,y,z) = t \cdot f_1(x,y,z) + (1-t) \cdot f_2(x,y,z).$$

Exercises

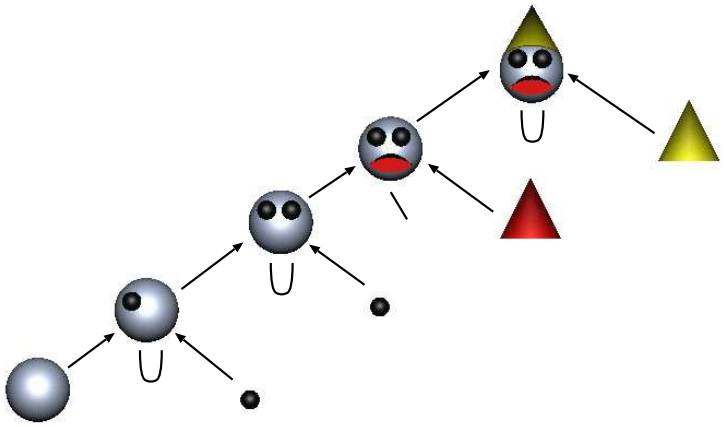


Figure 14.10. Constructing a complex solid by set operations. The root and the leaf of the ***CSG tree*** represents the complex solid, and the primitives, respectively. Other nodes define the set operations (U: union, \: difference).

14.2-1 Find the parametric equation of a torus.
14.2-2 Prove that the fourth-order B-spline with knot-vector [0,0,0,0,1,1,1,1] is a Bézier curve.
14.2-3 Give the equations for the surface points and the normals of the waving flag and waving water disturbed in a single point.
14.2-4 Prove that the tangents of a Bézier curve at the start and the end are the lines connecting the first two and the last two control points, respectively.
14.2-5 Give the algebraic forms of the basis functions of the second, the third, and the fourth-order B-splines.
14.2-6 Develop an algorithm computing the path length of a Bézier curve and a B-spline. Based on the path length computation move a point along the curve with uniform speed.

14.3. Geometry processing and tessellation algorithms

In section 14.2 we met free-form surface and curve definition methods. During image synthesis, however, line segments and triangles play important roles. In this section we present methods that bridge the gap between these two types of representations. These methods convert geometric models to lines and triangles, or further process line and triangle models. Line segments connected to each other in a way that the end point of a line segment is the start point of the next one are called ***polylines.*** Triangles connected at edges, on the other hand, are called ***meshes. Vectorization*** methods approximate free-form curves by polylines. A polyline is defined by its vertices. ***Tessellation*** algorithms, on the other hand, approximate free-form surfaces by meshes. For illumination computation, we often need the normal vector of the original surface, which is usually stored with the vertices. Consequently, a triangle mesh contains a list of triangles, where each triangle is given by three vertices and three normals. Methods processing triangle meshes use other topology information as well,

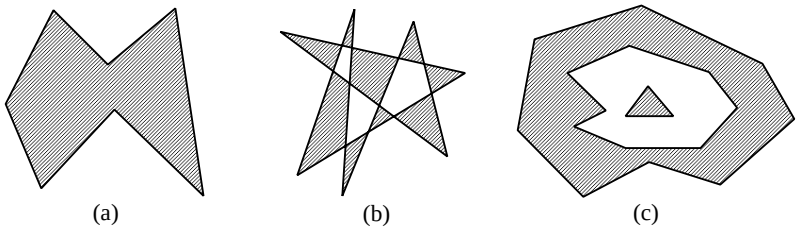


Figure 14.11. Types of polygons. **(a)** simple; **(b)** complex, single connected; **(c)** multiply connected.

for example, which triangles share an edge or a vertex.

14.3.1. Polygon and polyhedron

Definition 14.2 *A **polygon** is a bounded part of the plane, i.e. it does not contain a line, and is bordered by line segments. A polygon is defined by the vertices of the bordering polylines.*

Definition 14.3 *A polygon is **single connected** if its border is a single closed polyline (figure 14.11).*

Definition 14.4 *A polygon is **simple** if it is single connected and the bordering polyline does not intersect itself (figure 14.11(a)).*

For a point of the plane, we can detect whether or not this point is inside the polygon by starting a half-line from this point and counting the number of intersections with the boundary. If the number of intersections is an odd number, then the point is inside, otherwise it is outside.

In the three-dimensional space we can form triangle meshes, where different triangles are in different planes. In this case, two triangles are said to be neighbouring if they share an edge.

Definition 14.5 *A **polyhedron** is a bounded part of the space, which is bordered by polygons.*

Similarly to polygons, a point can be tested for polyhedron inclusion by casting a half line from this point and counting the number of intersections with the face polygons. If the number of intersections is odd, then the point is inside the polyhedron, otherwise it is outside.

14.3.2. Vectorization of parametric curves

Parametric functions map interval $[t_{\min}, t_{\max}]$ onto the points of the curve. During vectorization the parameter interval is discretized. The simplest discretization scheme generates N evenly spaced parameter values $t_i = t_{\min} + (t_{\max} - t_{\min}) \cdot i/N$ $(i = 0, 1, \ldots, N)$, and defines the approximating polyline by the points obtained by substituting these parameter values into parametric equation $\vec{r}(t_i)$.

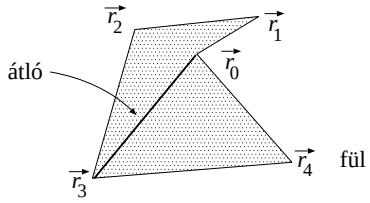


Figure 14.12. Diagonal and ear of a polygon.

14.3.3. Tessellation of simple polygons

Let us first consider the conversion of simple polygons to triangles. This is easy if the polygon is convex since we can select an arbitrary vertex and connect it with all other vertices, which decomposes the polygon to triangles in linear time. Unfortunately, this approach does not work for concave polygons since in this case the line segment connecting two vertices may go outside the polygon, thus cannot be the edge of one decomposing triangle.

Let us start with two definitions:

Definition 14.6 *The **diagonal** of a polygon is a line segment connecting two vertices and is completely contained by the polygon (line segment $\vec{r}_0$ and $\vec{r}_3$ of figure 14.12).*

The diagonal property can be checked for a line segment connecting two vertices by trying to intersect the line segment with all edges and showing that intersection is possible only at the endpoints, and additionally showing that one internal point of the candidate is inside the polygon. For example, this test point can be midpoint of the line segment.

Definition 14.7 *A vertex of the polygon is an **ear** if the line segment of the previous and the next vertices is a diagonal (vertex $\vec{r}_4$ of figure 14.12).*

Clearly, only those vertices may be ears where the inner angle is not greater than 180 degrees. Such vertices are called ***convex vertices***.

For simple polygons the following theorems hold:

Theorem 14.8 *A simple polygon always has a diagonal.*

Proof. Let the vertex standing at the left end (having the minimal x coordinate) be $\vec{r}_i$, and its two neighboring vertices be $\vec{r}_{i-1}$ and $\vec{r}_{i+1}$, respectively (figure 14.13). Since $\vec{r}_i$ is standing at the left end, it is surely a convex vertex. If $\vec{r}_i$ is an ear, then line segment $(\vec{r}_{i-1}, \vec{r}_{i+1})$ is a diagonal (left of figure 14.13), thus the theorem is proven for this case. Since $\vec{r}_i$ is a convex vertex, it is not an ear only if triangle $\vec{r}_{i-1}$, $\vec{r}_i$, $\vec{r}_{i+1}$ contains at least one polygon vertex (right of figure 14.13). Let us select from the contained vertices that vertex $\vec{p}$ which is the farthest from the line defined by points $\vec{r}_{i-1}, \vec{r}_{i+1}$. Since there are no contained points which are farther from line $(\vec{r}_{i-1}, \vec{r}_{i+1})$ than $\vec{p}$, no edge can be between points $\vec{p}$ and $\vec{r}_i$, thus $(\vec{p}, \vec{r}_i)$ must be a diagonal. ■

Theorem 14.9 *A simple polygon can always be decomposed to triangles with its diagonals. If the number of vertices is n, then the number of triangles is $n - 2$.*

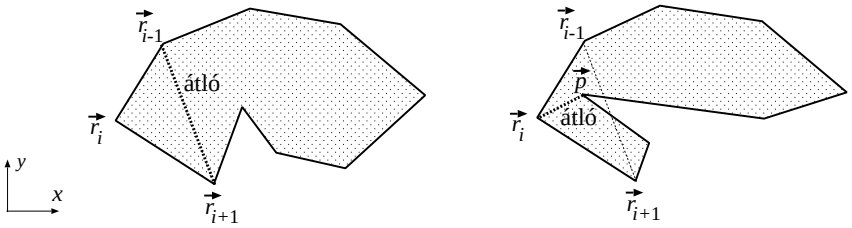


Figure 14.13. The proof of the existence of a diagonal for simple polygons.

Proof. This theorem is proven with induction. The theorem is obviously true when $n = 3$, i.e. when the polygon is a triangle. Let us assume that the statement is also true for polygons having m ($m = 3, \ldots, n-1$) number of vertices, and consider a polygon with n vertices. According to theorem 14.8, this polygon of n vertices has a diagonal, thus we can subdivide this polygon into a polygon of n_1 vertices and a polygon of n_2 vertices, where $n_1, n_2 < n$, and $n_1 + n_2 = n + 2$ since the vertices at the end of the diagonal participate in both polygons. According to the assumption of the induction, these two polygons can be separately decomposed to triangles. Joining the two sets of triangles, we can obtain the triangle decomposition of the original polygon. The number of triangles is $n_1 - 2 + n_2 - 2 = n - 2$. ■

The discussed proof is constructive thus it inspires a subdivision algorithm: let us find a diagonal, subdivide the polygon along this diagonal, and continue the same operation for the two new polygons.

Unfortunately the running time of such an algorithm is in $\Theta(n^3)$ since the number of diagonal candidates is $\Theta(n^2)$, and the time needed by checking whether or not a line segment is a diagonal is in $\Theta(n)$.

We also present a better algorithm, which decomposes a convex or concave polygon defined by vertices $\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_n$. This algorithm is called ***ear cutting***. The algorithm looks for ear triangles and cuts them until the polygon gets simplified to a single triangle. The algorithm starts at vertex $\vec{r}_2$. When a vertex is processed, it is first checked whether or not the previous vertex is an ear. If it is not an ear, then the next vertex is chosen. If the previous vertex is an ear, then the current vertex together with the two previous ones form a triangle that can be cut, and the previous vertex is deleted. If after deletion the new previous vertex has index 0, then the next vertex is selected as the current vertex.

The presented algorithm keeps cutting triangles until no more ears are left. The termination of the algorithm is guaranteed by the following ***two ears theorem***:

Theorem 14.10 *A simple polygon having at least four vertices always has at least two not neighboring ears that can be cut independently.*

Proof. The proof presented here has been given by Joseph O'Rourke. According to theorem 14.9, every simple polygon can be subdivided to triangles such that the edges of these triangles are either the edges or the diagonals of the polygon. Let us make a correspondence between the triangles and the nodes of a graph where two nodes are connected if and only if the two triangles corresponding to these nodes share an edge. The resulting graph is connected and cannot contain circles. Graphs of these properties are trees. The name of this tree graph is the ***dual tree.*** Since the polygon has at least four vertices, the number of nodes

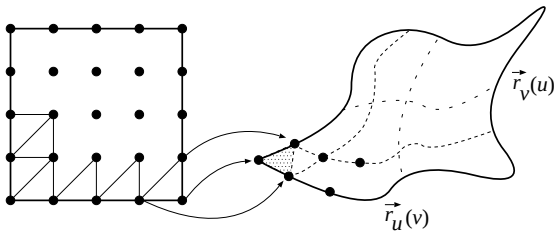


Figure 14.14. Tessellation of parametric surfaces.

in this tree is at least two. Any tree containing at least two nodes has at least two leaves[2]. Leaves of this tree, on the other hand, correspond to triangles having an ear vertex. ■

According to the two ears theorem, the presented algorithm finds an ear in $O(n)$ steps. Cutting an ear the number of vertices is reduced by one, thus the algorithm terminates in $O(n^2)$ steps.

14.3.4. Tessellation of parametric surfaces

Parametric forms of surfaces map parameter rectangle $[u_{\min}, u_{\max}] \times [v_{\min}, v_{\max}]$ onto the points of the surface.

In order to tessellate the surface, first the parameter rectangle is subdivided to triangles. Then applying the parametric equations for the vertices of the parameter triangles, the approximating triangle mesh can be obtained. The simplest subdivision of the parametric rectangle decomposes the domain of parameter u to N parts, and the domain of parameter v to M intervals, resulting in the following parameter pairs:

$$[u_i, v_j] = \left[u_{\min} + (u_{\max} - u_{\min})\frac{i}{N},\ v_{\min} + (v_{\max} - v_{\min})\frac{j}{M}\right].$$

Taking these parameter pairs and substituting them into the parametric equations, point triplets $\vec{r}(u_i, v_j)$, $\vec{r}(u_{i+1}, v_j)$, $\vec{r}(u_i, v_{j+1})$, and point triplets $\vec{r}(u_{i+1}, v_j)$, $\vec{r}(u_{i+1}, v_{j+1})$, $\vec{r}(u_i, v_{j+1})$ are used to define triangles.

The tessellation process can be made ***adaptive*** as well, which uses small triangles only where the high curvature of the surface justifies them. Let us start with the parameter rectangle and subdivide it to two triangles. In order to check the accuracy of the resulting triangle mesh, surface points corresponding to the edge midpoints of the parameter triangles are compared to the edge midpoints of the approximating triangles. Formally the following distance is computed (figure 14.15):

$$\left|\vec{r}\left(\frac{u_1 + u_2}{2}, \frac{v_1 + v_2}{2}\right) - \frac{\vec{r}(u_1, v_1) + \vec{r}(u_2, v_2)}{2}\right|,$$

where (u_1, v_1) and (u_2, v_2) are the parameters of the two endpoints of the edge.

[2] a leaf is a node connected by exactly one edge

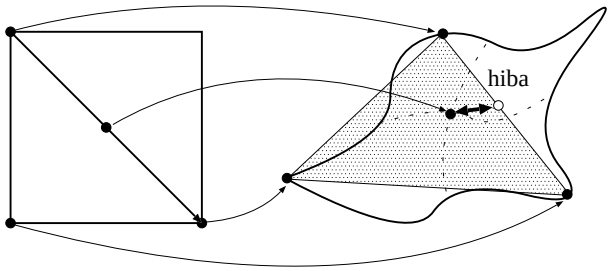


Figure 14.15. Estimation of the tessellation error.

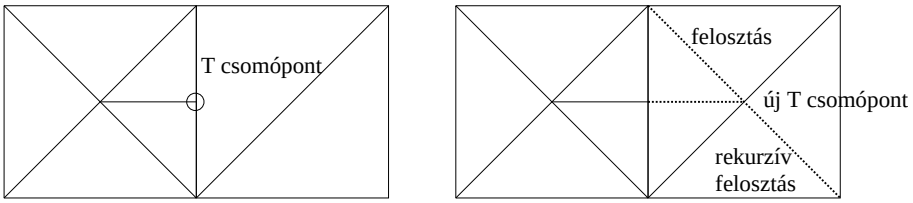


Figure 14.16. T vertices and their elimination with forced subdivision.

A large distance value indicates that the triangle mesh poorly approximates the parametric surface, thus triangles must be subdivided further. This subdivision can be executed by cutting the triangle to two triangles by a line connecting the midpoint of the edge of the largest error and the opposing vertex. Alternatively, a triangle can be subdivided to four triangles with its halving lines. The adaptive tessellation is not necessarily robust since it can happen that the distance at the midpoint is small, but at other points is still quite large.

When the adaptive tessellation is executed, it may happen that one triangle is subdivided while its neighbour is not, which results in a mesh where the previously shared edge is tessellated in one of the triangles, thus has holes. Such problematic midpoints are called ***T vertices*** (figure 14.16).

If the subdivision criterion is based only on edge properties, then T vertices cannot show up. However, if other properties are also taken into account, then T vertices may appear. In such cases, T vertices can be eliminated by recursively forcing the subdivision also for those neighbouring triangles that share subdivided edges.

14.3.5. Subdivision curves and meshes

This section presents algorithms that smooth polyline and mesh models. Smoothing means that a polyline and a mesh are replaced by other polylines and meshes having less faceted look.

Let us consider a polyline of vertices $\vec{r}_0, \ldots, \vec{r}_m$. A smoother polyline is generated by the following vertex doubling approach (figure 14.17). Every line segment of the polyline is halved, and midpoints $\vec{h}_0, \ldots, \vec{h}_{m-1}$ are added to the polyline as new vertices. Then the

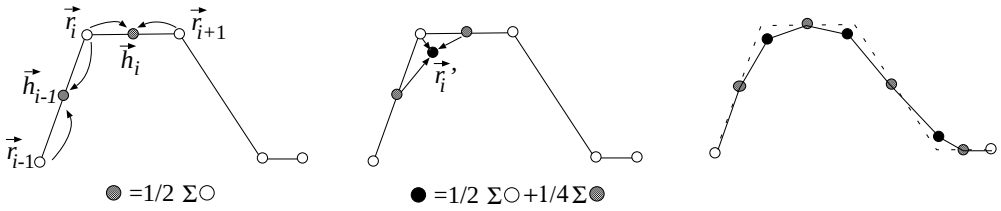


Figure 14.17. Construction of a subdivision curve: at each step midpoints are obtained, then the original vertices are moved to the weighted average of neighbouring midpoints and of the original vertex.

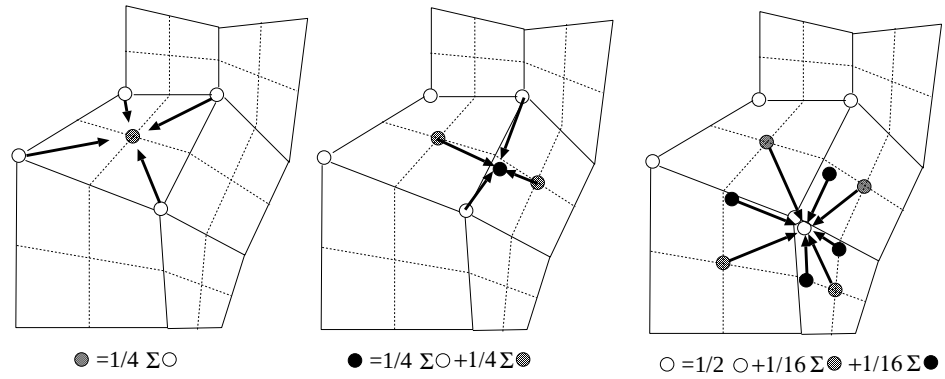


Figure 14.18. One smoothing step of the Catmull-Clark subdivision. First the face points are obtained, then the edge midpoints are moved, and finally the original vertices are refined according to the weighted sum of its neighbouring edge and face points.

old vertices are moved taking into account their old position and the positions of the two enclosing midpoints, applying the following weighting:

$$\vec{r}_i' = \frac{1}{2}\vec{r}_i + \frac{1}{4}\vec{h}_{i-1} + \frac{1}{4}\vec{h}_i = \frac{3}{4}\vec{r}_i + \frac{1}{8}\vec{r}_{i-1} + \frac{1}{8}\vec{r}_{i+1} .$$

The new polyline looks much smoother. If we should not be satisfied with the smoothness yet, the same procedure can be repeated recursively. As can be shown, the result of the recursive process converges to the B-spline curve.

The polyline subdivision approach can also be extended for smoothing three-dimensional meshes. This method is called ***Catmull-Clark subdivision algorithm.*** Let us consider a three-dimensional quadrilateral mesh (figure 14.18). In the first step the midpoints of the edges are obtained, which are called ***edge points***. Then ***face points*** are generated as the average of the vertices of each face polygon. Connecting the edge points with the face points, we still have the original surface, but now defined by four times more quadrilaterals. The smoothing step modifies first the edge points setting them to the average of the vertices at the ends of the edge and of the face points of those quads that share this edge. Then the original vertices are moved to the weighted average of the face points of those faces that share this vertex, and of edge points of those edges that are connected to this vertex. The weight of the original vertex is 1/2, the weights of edge and face are 1/16. Again, this

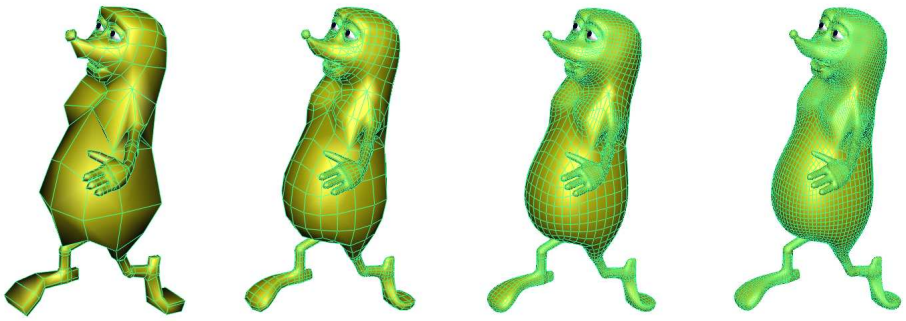


Figure 14.19. Original mesh and its subdivision applying the smoothing step once, twice and three times, respectively.

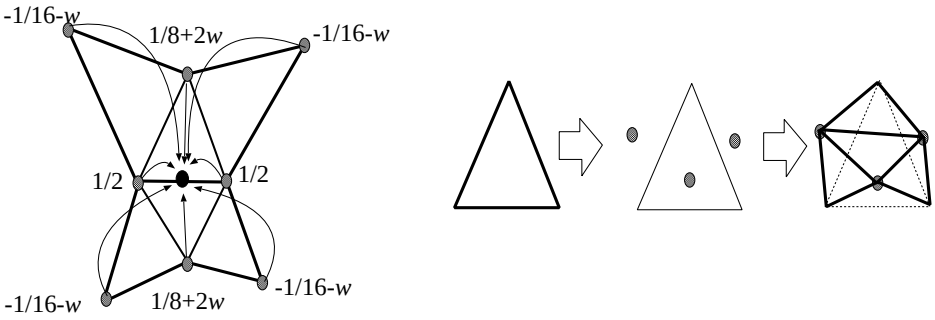


Figure 14.20. Generation of the new edge point with butterfly subdivision.

operation may be repeated until the surface looks smooth enough (figure 14.19).

If we do not want to smooth the mesh at an edge or around a vertex, then the averaging operation ignores the vertices on the other side of the edge to be preserved.

The Catmull-Clark subdivision surface usually does not interpolate the original vertices. This drawback is eliminated by the ***butterfly subdivision***, which works on triangle meshes. First the butterfly algorithm puts new edge points close to the midpoints of the original edges, then the original triangle is replaced by four triangles defined by the original vertices and the new edge points (figure 14.20). The position of the new edge points depend on the vertices of those two triangles incident to this edge, and on those four triangles which share edges with these two. The arrangement of the triangles affecting the edge point resembles a butterfly, hence the name of this algorithm. The edge point coordinates are obtained as a weighted sum of the edge endpoints multiplied by $1/2$, the third vertices of the triangles sharing this edge using weight $1/8 + 2w$, and finally of the other vertices of the additional triangles with weight $-1/16 - w$. Parameter w can control the curvature of the resulting mesh. Setting $w = -1/16$, the mesh keeps its original faceted look, while $w = 0$ results in strong rounding.

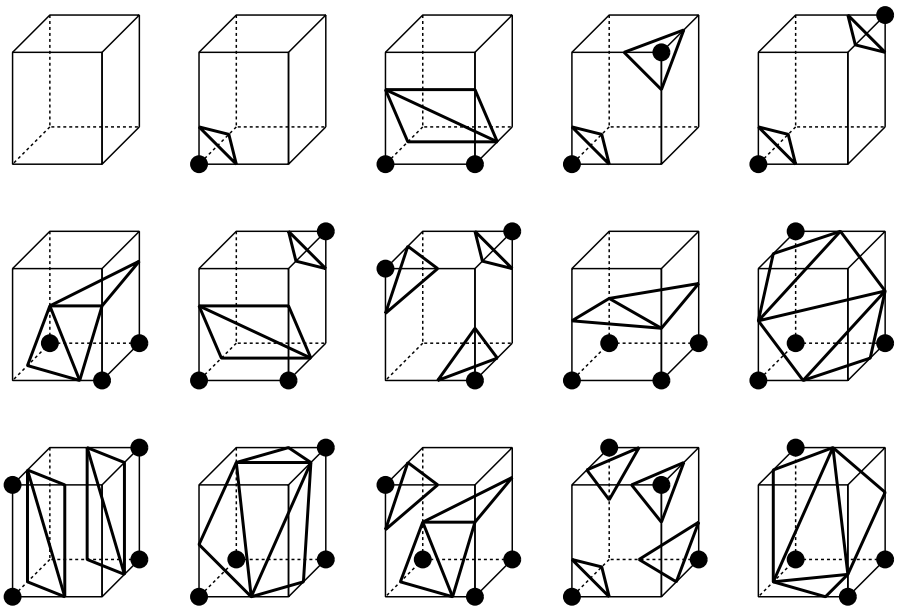


Figure 14.21. Possible intersections of the per-voxel tri-linear implicit surface and the voxel edges. From the possible 256 cases, these 15 topologically equivalent cases can be identified, from which the others can be obtained by rotations. Grid points where the implicit function has the same sign are depicted by circles.

14.3.6. Tessellation of implicit surfaces

An implicit surface can be converted to a triangle mesh by finding points on the surface densely, i.e. generating points satisfying $f(x, y, z) \approx 0$, then assuming the close points to be vertices of the triangles.

First implicit function f is evaluated at the grid points of the Cartesian coordinate system and the results are stored in a three-dimensional array, called ***voxel array.*** Let us call two grid points as neighbours if two of their coordinates are identical and the difference in their third coordinate is 1. The function is evaluated at the grid points and is assumed to be linear between them. The normal vectors needed for shading are obtained as the gradient of function f (equation 14.4), which are also interpolated between the grid points.

When we work with the voxel array, original function f is replaced by its ***tri-linear*** approximation (tri-linear means that fixing any two coordinates the function is linear with respect to the third coordinate). Due to the linear approximation an edge connecting two neighbouring grid points can intersect the surface at most once since linear equations may have at most one root. The density of the grid points should reflect this observation, we have to define them so densely not to miss roots, that is, not to change the topology of the surface.

The method approximating the surface by a triangle mesh is called ***marching cubes algorithm.*** This algorithm first decides whether a grid point is inside or outside of the solid by checking the sign of the function. If two neighbouring grid points are of different type, the surface must go between them. The intersection of the surface and the edge between the neighbouring points, as well as the normal vector at the intersection are determined by linear interpolation. If one grid point is at $\vec{r}_1$, the other is at $\vec{r}_2$, and implicit function f has

different signs at these points, then the intersection of the tri-linear surface and line segment $(\vec{r}_1, \vec{r}_2)$ is:

$$\vec{r}_i = \vec{r}_1 \cdot \frac{f(\vec{r}_2)}{f(\vec{r}_2) - f(\vec{r}_1)} + \vec{r}_2 \cdot \frac{f(\vec{r}_1)}{f(\vec{r}_2) - f(\vec{r}_1)} ,$$

The normal vector here is:

$$\vec{n}_i = \text{grad} f(\vec{r}_1) \cdot \frac{f(\vec{r}_2)}{f(\vec{r}_2) - f(\vec{r}_1)} + \text{grad} f(\vec{r}_2) \cdot \frac{f(\vec{r}_1)}{f(\vec{r}_2) - f(\vec{r}_1)} .$$

Having found the intersection points, triangles are defined using these points as vertices, and the approximation surface becomes the resulting triangle mesh. When defining these triangles, we have to take into account that a tri-linear surface may intersect the voxel edges at most once. Such intersection occurs if the implicit function has different sign at the two grid points. The number of possible variations of positive/negative signs at the 8 vertices of a cube is 256, from which 15 topologically equivalent cases can be identified (figure 14.21).

The algorithm inspects grid points one by one and assigns the sign of the function to them encoding negative sign by 0 and non-negative sign by 1. The resulting 8 bit code is a number in 0–255 which identifies the current case of intersection. If the code is 0, all voxel vertices are outside the solid, thus no voxel surface intersection is possible. Similarly, if the code is 255, the solid is completely inside, making the intersections impossible. To handle other codes, a table can be built which describes where the intersections show up and how they form triangles.

Exercises

14.3-1 Prove the two ears theorem by induction.
14.3-2 Develop an adaptive curve tessellation algorithm.
14.3-3 Prove that the Catmull-Clark subdivision curve and surface converge to a B-spline curve and surface, respectively.
14.3-4 Build a table to control the marching cubes algorithm, which describes where the intersections show up and how they form triangles.
14.3-5 Propose a marching cubes algorithm that does not require the gradients of the implicit function, but estimates these gradients from the values of the implicit function.

14.4. Containment algorithms

When geometric models are processed, we often have to determine whether or not one object contains points belonging to the other object. If only yes/no answer is needed, we have a ***containment test*** problem. However, if the contained part also needs to be obtained, the applicable algorithm is called ***clipping.***

Containment test is often called as discrete time ***collision detection*** since if one object contains points from the other, then the two objects must have been collided before. Of course, checking collisions just at discrete time instances may miss certain collisions. To handle the collision problem robustly, continuous time collision detection is needed which also computes the time of the collision. Continuous time collision detection is based on ray tracing (subsection 14.6). In this section we only deal with the discrete time collision detection and the clipping of simple objects.

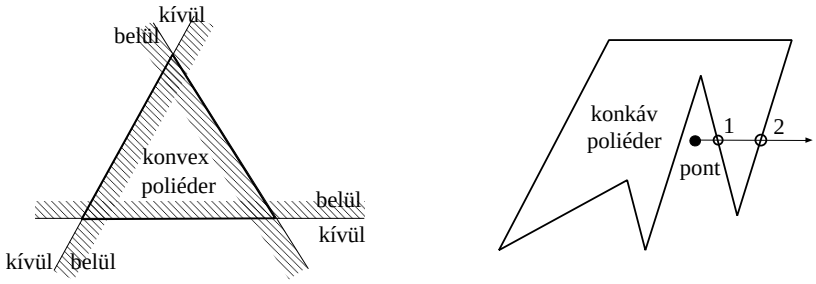


Figure 14.22. Polyhedron-point containment test. A convex polyhedron contains a point if the point is on that side of each face plane where the polyhedron is. To test a concave polyhedron, a half line is cast from the point and the number of intersections is counted. If the result is an odd number, then the point is inside, otherwise it is outside.

14.4.1. Point containment test

A solid defined by implicit function f contains those (x, y, z) points which satisfy inequality $f(x, y, z) \geq 0$. It means that point containment test requires the evaluation of the implicit function and the inspection of the sign of the result.

Half space
Based on equation (14.1), points belonging to a half space are identified by inequality

$$(\vec{r} - \vec{r}_0) \cdot \vec{n} \geq 0, \qquad n_x \cdot x + n_y \cdot y + n_z \cdot z + d \geq 0, \tag{14.9}$$

where the normal vector is supposed to point inward.

Convex polyhedron
Any convex polyhedron can be constructed as the intersection of halfspaces incident to the faces of the polyhedron (left of figure 14.22). The plane of each face subdivides the space into two parts, to an inner part where the polyhedron can be found, and to an outer part. Let us test the point against the planes of the faces. If the points are in the inner part with respect to all planes, then the point is inside the polyhedron. However, if the point is in the outer part with respect to at least one plane, then the point is outside of the polyhedron.

Concave polyhedron
As shown in figure 14.22, let us cast a half line from the tested point and count the number of intersections with the faces of the polyhedron (the calculation of these intersections is discussed in subsection 14.6). If the result is an odd number, then the point is inside, otherwise it is outside. Because of numerical inaccuracies we might have difficulties to count the number of intersections when the half line is close to the edges. In such cases, the simplest solution is to find another half line and carry out the test with that.

Polygon
The methods proposed to test the point in polyhedron can also be used for polygons limiting the space to the two-dimensional plane. For example, a point is in a general polygon if the

half line originating at this point and lying in the plane of the polygon intersects the edges of the polygon odd times.

In addition to those methods, containment in convex polygons can be tested by adding the angles subtended by the edges from the point. If the sum is 360 degrees, then the point is inside, otherwise it is outside. For convex polygons, we can also test whether the point is on the same side of the edges as the polygon itself. This algorithm is examined in details for a particularly important special case, when the polygon is a triangle.

Triangle

Let us consider a triangle of vertices $\vec{a}$, $\vec{b}$ and $\vec{c}$, and point $\vec{p}$ lying in the plane of the triangle. The point is inside the triangle if and only if it is on the same side of the boundary lines as the third vertex. Note that cross product $(\vec{b}-\vec{a})\times(\vec{p}-\vec{a})$ has a different direction for point $\vec{p}$ lying on the different sides of oriented line $\vec{ab}$, thus the direction of this vector can be used to classify points (should point $\vec{p}$ be on line $\vec{ab}$, the result of the cross product is zero). During classification the direction of $(\vec{b}-\vec{a})\times(\vec{p}-\vec{a})$ is compared to the direction of vector $\vec{n} = (\vec{b}-\vec{a})\times(\vec{c}-\vec{a})$ where tested point $\vec{p}$ is replaced by third vertex $\vec{c}$. Note that vector $\vec{n}$ happens to be the normal vector of the triangle plane (figure 14.23).

We can determine whether two vectors have the same direction (their angle is zero) or they have opposite directions (their angle is 180 degrees) by computing their scalar product and looking at the sign of the result. The scalar product of vectors of similar directions is positive. Thus if scalar product $((\vec{b}-\vec{a})\times(\vec{p}-\vec{a}))\cdot\vec{n}$ is positive, then point $\vec{p}$ is on the same side of oriented line $\vec{ab}$ as $\vec{c}$. On the other hand, if this scalar product is negative, then $\vec{p}$ and $\vec{c}$ are on the opposite sides. Finally, if the result is zero, then point $\vec{p}$ is on line $\vec{ab}$. Point $\vec{p}$ is inside the triangle if and only if all the following three conditions are met:

$$\begin{aligned} ((\vec{b}-\vec{a})\times(\vec{p}-\vec{a}))\cdot\vec{n} &\geq 0\,, \\ ((\vec{c}-\vec{b})\times(\vec{p}-\vec{b}))\cdot\vec{n} &\geq 0\,, \\ ((\vec{a}-\vec{c})\times(\vec{p}-\vec{c}))\cdot\vec{n} &\geq 0\,. \end{aligned} \tag{14.10}$$

This test is robust since it gives correct result even if – due to numerical precision problems – point $\vec{p}$ is not exactly in the plane of the triangle as long as point $\vec{p}$ is in the prism obtained by perpendicularly extruding the triangle from the plane.

The evaluation of the test can be speeded up if we work in the two-dimensional projections instead of the three-dimensional space. Let us project point $\vec{p}$ as well as the triangle onto one of the coordinate planes. In order to increase numerical precision, that coordinate plane should be selected on which the area of the projected triangle is maximal. Let us denote the Cartesian coordinates of the normal vector by (n_x, n_y, n_z). If n_z has the maximum absolute value, then the projection of the maximum area is on coordinate plane xy. If n_x or n_y had the maximum absolute value, then planes yz, or xz would be the right choice. Here only the case of maximum n_z is discussed.

First the order of vertices are changed in a way that when traveling from vertex $\vec{a}$ to vertex $\vec{b}$, vertex $\vec{c}$ is on the left side. Let us examine the equation of line $\vec{ab}$:

$$\frac{b_y - a_y}{b_x - a_x}\cdot(x - b_x) + b_y = y\,.$$

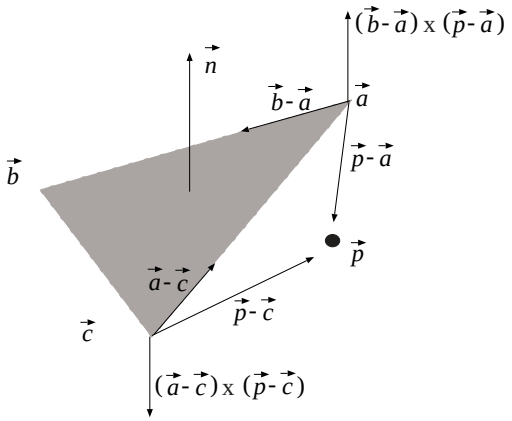


Figure 14.23. Point in triangle containment test. The figure shows that case when point $\vec{p}$ is on the left of oriented lines $\vec{ab}$ and $\vec{bc}$, and on the right of line $\vec{ca}$, that is, when it is not inside the triangle.

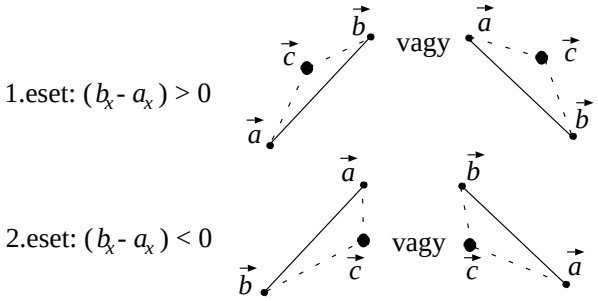


Figure 14.24. Point in triangle containment test on coordinate plane xy. Third vertex $\vec{c}$ can be either on the left or on the right side of oriented line $\vec{ab}$, which can always be traced back to the case of being on the left side by exchanging the vertices.

According to figure 14.24 point $\vec{c}$ is on the left of the line if c_y is above the line at $x = c_x$:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (c_x - b_x) + b_y < c_y \ .$$

Multiplying both sides by $(b_x - a_x)$, we get:

$$(b_y - a_y) \cdot (c_x - b_x) < (c_y - b_y) \cdot (b_x - a_x) \ .$$

In the second case the denominator of the slope of the line is negative. Point $\vec{c}$ is on the left of the line if c_y is below the line at $x = c_x$:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (c_x - b_x) + b_y > c_y \ .$$

When the inequality is multiplied with negative denominator $(b_x - a_x)$, the relation is inverted:

$$(b_y - a_y) \cdot (c_x - b_x) < (c_y - b_y) \cdot (b_x - a_x) \ .$$

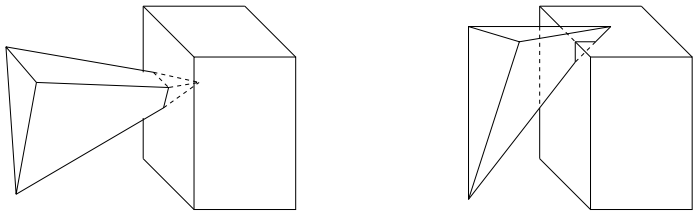


Figure 14.25. Polyhedron-polyhedron collision detection. Only a part of collision cases can be recognized by testing the containment of the vertices of one object with respect to the other object. Collision can also occur when only edges meet, but vertices do not penetrate to the other object.

Note that in both cases we obtained the same condition. If this condition is not met, then point $\vec{c}$ is not on the left of line $\vec{ab}$, but is on the right. Exchanging vertices $\vec{a}$ and $\vec{b}$ in this case, we can guarantee that $\vec{c}$ will be on the left of the new line $\vec{ab}$. It is also important to note that consequently point $\vec{a}$ will be on the left of line $\vec{bc}$ and point $\vec{b}$ will be on the left of line $\vec{ca}$.

In the second step the algorithm tests whether point $\vec{p}$ is on the left with respect to all three boundary lines since this is the necessary and sufficient condition of being inside the triangle:

$$\begin{aligned}(b_y - a_y)\cdot(p_x - b_x) &\le (p_y - b_y)\cdot(b_x - a_x)\ ,\\ (c_y - b_y)\cdot(p_x - c_x) &\le (p_y - c_y)\cdot(c_x - b_x)\ ,\\ (a_y - c_y)\cdot(p_x - a_x) &\le (p_y - a_y)\cdot(a_x - c_x)\ .\end{aligned} \tag{14.11}$$

14.4.2. Polyhedron-polyhedron collision detection

Two polyhedra collide when a vertex of one of them meets a face of the other, and if they are not bounced off, the vertex goes into the internal part of the other object (figure 14.25). This case can be recognized with the discussed containment test. All vertices of one polyhedron is tested for containment against the other polyhedron. Then the roles of the two polyhedra are exchanged.

Apart from the collision between vertices and faces, two edges may also meet without vertex penetration (figure 14.25). In order to recognize this case, all edges of one polyhedron are tested against all faces of the other polyhedron. The test for an edge and a face is started by checking whether or not the two endpoints of the edge are on opposite sides of the plane, using inequality (14.9). If they are, then the intersection of the edge and the plane is calculated, and finally it is decided whether the face contains the intersection point.

Let us realize that the test of edge penetration and the containment of an arbitrary vertex also include the case of vertex penetration. However, edge penetration without vertex penetration happens less frequently, and the vertex penetration is easier to check, thus it is still worth applying the vertex penetration test first.

Polyhedra collision detection tests each edge of one polyhedron against to each face of the other polyhedron, which results in an algorithm of quadratic time complexity with respect to the number of vertices of the polyhedra. Fortunately, the algorithm can be speeded up applying bounding volumes (subsection 14.6.2). Let us assign a simple bounding object

to each polyhedron. Popular choices are the spheres and the boxes. If the two bounding volumes do not collide, then neither can the contained polyhedra collide. If the bounding volumes penetrate each other, then one polyhedra is tested against the other bounding volume. If this test is also positive, then finally the two polyhedra are tested. However, this last test is rarely required, and most of the collision cases can be solved by bounding volumes.

14.4.3. Clipping algorithms

Clipping takes an object defining the clipping region and removes those points from another other object which are outside the clipping region. Clipping may alter the type of the object, which cannot be specified by a similar equation after clipping. To avoid this, we allow only those kinds of clipping regions and objects where the object type does not change. Let us thus assume that the clipping region is a half space or a polyhedron, while the object to be clipped is a point, a line segment or a polygon.

If the object to be clipped is a point, then containment can be tested with the algorithms of the previous subsection. Based on the result of the containment test, the point is either removed or preserved.

Clipping a line segment onto a half space
Let us consider a line segment of endpoints $\vec{r}_1$ and $\vec{r}_2$, and of equation $\vec{r}(t) = \vec{r}_1 \cdot (1-t) + \vec{r}_2 \cdot t$, $(t \in [0,1])$, and a half plane defined by the following equation derived from equation (14.1):

$$(\vec{r} - \vec{r}_0) \cdot \vec{n} \geq 0, \qquad n_x \cdot x + n_y \cdot y + n_z \cdot z + d \geq 0.$$

Three cases need to be distinguished:

1. If both endpoints of the line segment are in the half space, then all points of the line segment are inside, thus the whole segment is preserved.
2. If both endpoints are out of the half space, then all points of the line segment are out, thus the line segment should be completely removed.
3. If one of the endpoints is out, while the other is in, then the endpoint being out should be replaced by the intersection point of the line segment and the boundary plane of the half space. The intersection point can be calculated by substituting the equation of the line segment into the equation of the boundary plane and solving the resulting equation for the unknown parameter:

$$(\vec{r}_1 \cdot (1 - t_i) + \vec{r}_2 \cdot t_i - \vec{r}_0) \cdot \vec{n} = 0, \quad \Longrightarrow \quad t_i = \frac{(\vec{r}_0 - \vec{r}_1) \cdot \vec{n}}{(\vec{r}_2 - \vec{r}_1) \cdot \vec{n}} .$$

Substituting parameter t_i into the equation of the line segment, the coordinates of the intersection point can also be obtained.

Clipping a polygon onto a half space
This clipping algorithm tests first whether a vertex is inside or not. If the vertex is in, then it is also the vertex of the resulting polygon. However, if it is out, it can be ignored. On the other hand, the resulting polygon may have vertices other than the vertices of the original polygon. These new vertices are the intersections of the edges and the boundary plane of

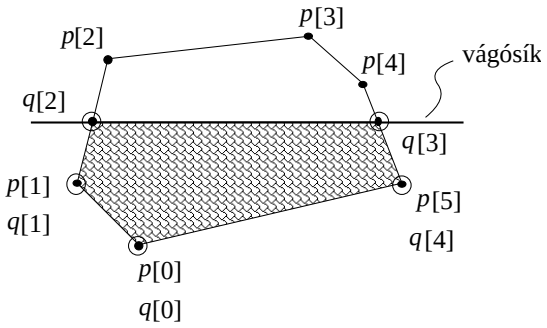


Figure 14.26. Clipping of polygon $\vec{p}[0], \ldots, \vec{p}[5]$ results in polygon $\vec{q}[0], \ldots, \vec{q}[4]$. The vertices of the resulting polygon are the inner vertices of the original polygon and the intersections of the edges and the boundary plane.

the half space. Such intersection occurs when one endpoint is in, but the other is out. While we are testing the vertices one by one, we should also check whether or not the next vertex is on the same side as the current vertex (figure 14.26).

Suppose that the vertices of the polygon to be clipped are given in array $\mathbf{p} = \langle \vec{p}[0], \ldots, \vec{p}[n-1] \rangle$, and the vertices of the clipped polygon is expected in array $\mathbf{q} = \langle \vec{q}[0], \ldots, \vec{q}[m-1] \rangle$. The number of the vertices of the resulting polygon is stored in variable m. Note that the vertex followed by the ith vertex has usually index $(i+1)$, but not in the case of the last, $(n-1)$th vertex, which is followed by vertex 0. Handling the last vertex as a special case is often inconvenient. This can be eliminated by extending input array $\mathbf{p}$ by new element $\vec{p}[n] = \vec{p}[0]$, which holds the element of index 0 once again.

Using these assumptions, the ***Sutherland-Hodgeman polygon clipping algorithm*** is:

SUTHERLAND-HODGEMAN-POLYGON-CLIPPING($\mathbf{p}$)

```
 1  m ← 0
 2  for i ← 0 to n − 1
 3      do if p⃗[i] is inside
 4            then q⃗[m] ← p⃗[i]        ▷ The ith vertex is the vertex of the resulting polygon.
 5                 m ← m + 1
 6                 if p⃗[i + 1] is outside
 7                    then q⃗[m] ← EDGE-PLANE-INTERSECTION(p⃗[i], p⃗[i + 1])
 8                         m ← m + 1
 9            else if p⃗[i + 1] is inside
10                    then q⃗[m] ← EDGE-PLANE-INTERSECTION(p⃗[i], p⃗[i + 1])
11                         m ← m + 1
12  return q
```

Let us apply this algorithm for such a concave polygon which is expected to fall to several pieces during clipping (figure 14.27). The algorithm storing the polygon in a single array is not able to separate the pieces and introduces even number of edges at parts where no edge could show up.

These even number of extra edges, however, pose no problems if the interior of the

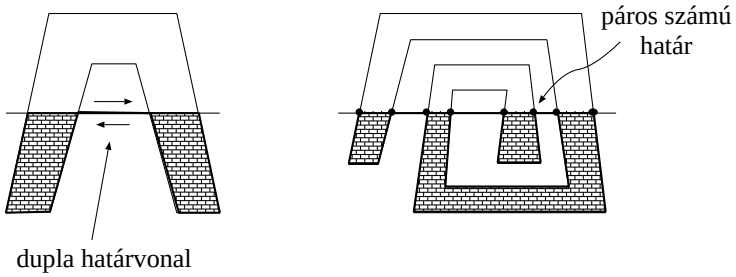


Figure 14.27. When concave polygons are clipped, the parts that should fall apart are connected by even number of edges.

polygon is defined as follows: a point is inside the polygon if and only if starting a half line from here, the boundary polyline is intersected by odd number of times.

The presented algorithm is also suitable for clipping multiple connected polygons if the algorithm is executed separately for each closed polyline of the boundary.

Clipping line segments and polygons on a convex polyhedron

As stated, a convex polyhedron can be obtained as the intersection of the half spaces defined by the planes of the polyhedron faces (left of figure 14.22). It means that the clipping on a convex polyhedron can be traced back to a series of clipping steps on half spaces. The result of one clipping step on a half plane is the input of clipping on the next half space. The final result is the output of the clipping on the last half space.

Clipping a line segment on an AABB

Axis aligned bounding boxes, abbreviated as AABBs, play an important role in image synthesis.

Definition 14.11 *A box aligned parallel to the coordinate axes is called* ***AABB****. An AABB is specified with the minimum and maximum Cartesian coordinates:* $[x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]$.

Although when an object is clipped on an AABB, the general algorithms clipping on a convex polyhedron could also be used, the importance of AABBs is acknowledged by developing algorithms specially tuned for this case.

When a line segment is clipped to a polyhedron, the algorithm would test the line segment with the plane of each face, and calculated intersection points may turn out to be unnecessary later. We should thus find an appropriate order of planes which makes the number of unnecessary intersection calculations minimal. A simple method implementing this idea is the ***Cohen-Sutherland line clipping algorithm.***

Let us assign code bit 1 to a point that is outside with respect to a clipping plane, and code bit 0 if the point is inside with respect to this plane. Since an AABB has 6 sides, we get 6 bits forming a 6-bit code word (figure 14.28). The interpretation of code bits $C[0], \ldots, C[5]$ is the following:

$$C[0] = \begin{cases} 1, & x \le x_{min}, \\ 0 & \text{otherwise.} \end{cases} \qquad C[1] = \begin{cases} 1, & x \ge x_{max}, \\ 0 & \text{otherwise.} \end{cases} \qquad C[2] = \begin{cases} 1, & y \le y_{min}\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

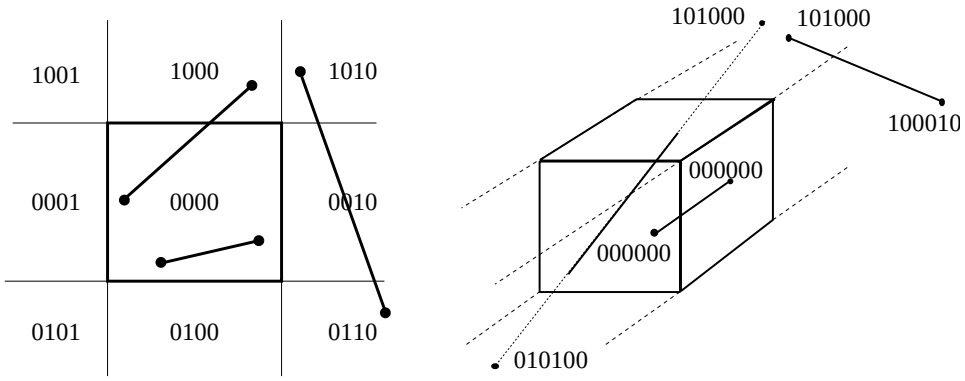


Figure 14.28. The 4-bit codes of the points in a plane and the 6-bit codes of the points in space.

$$C[3] = \begin{cases} 1, & y \geq y_{max}, \\ 0 & \text{otherwise.} \end{cases} \qquad C[4] = \begin{cases} 1, & z \leq z_{min}, \\ 0 & \text{otherwise.} \end{cases} \qquad C[5] = \begin{cases} 1, & z \geq z_{max}, \\ 0 & \text{otherwise.} \end{cases}$$

Points of code word 000000 are obviously inside, points of other code words are outside (figure 14.28). Let the code words of the two endpoints of the line segment be C_1 and C_2, respectively. If both of them are zero, then both endpoints are inside, thus the line segment is completely inside (trivial accept). If the two code words contain bit 1 at the same location, then none of the endpoints are inside with respect to the plane associated with this code bit. This means that the complete line segment is outside with respect to this plane, and can be rejected (trivial reject). This examination can be executed by applying the bitwise AND operation on code words C_1 and C_2 (with the notations of the C programming language C_1 & C_2), and checking whether or not the result is zero. If it is not zero, there is a bit where both code words have value 1.

Finally, if none of the two trivial cases hold, then there must be a bit which is 0 in one code word and 1 in the other. This means that one endpoint is inside and the other is outside with respect to the plane corresponding to this bit. The line segment should be clipped on this plane. Then the same procedure should be repeated starting with the evaluation of the code bits. The procedure is terminated when the conditions of either the trivial accept or the trivial reject are met.

The Cohen-Sutherland line clipping algorithm returns the endpoints of the clipped line by modifying the original vertices and indicates with „true” return value if the line is not completely rejected:

COHEN-SUTHERLAND-LINE-CLIPPING($\vec{r}_1, \vec{r}_2$)

```
1  C_1 ← codeword of r_1                          ▷ Code bits by checking the inequalities.
2  C_2 ← codeword of r_2
3  while TRUE
4        do if C_1 = 0 AND C_2 = 0
5              then return TRUE                   ▷ Trivial accept: inner line segment exists.
6           if C_1 & C_2 ≠ 0
7              then return FALSE                  ▷ Trivial reject: no inner line segment exists.
8           f ← index of the first bit where C_1 and C_2 differ
9           r_i ← intersection of line segment (r_1, r_2) and the plane of index f
10          C_i ← codeword of r_i
11          if C_1[f] = 1
12             then r_1 ← r_i
13                  C_1 ← C_i                     ▷ r_1 is outside w.r.t. plane f.
14             else r_2 ← r_i
15                  C_2 ← C_i                     ▷ r_2 is outside w.r.t. plane f.
```

Exercises

14.4-1 Propose approaches to reduce the quadratic complexity of polyhedron-polyhedron collision detection.

14.4-2 Develop a containment test to check whether a point is in a CSG-tree .

14.4-3 Develop an algorithm clipping one polygon onto a concave polygon.

14.4-4 Find an algorithm computing the bounding sphere and the bounding AABB of a polyhedron.

14.4-5 Develop an algorithm that tests the collision of two triangles in the plane.

14.4-6 Generalize the Cohen-Sutherland line clipping algorithm to convex polyhedron clipping region.

14.4-7 Propose a method for clipping a line segment on a sphere.

14.5. Translation, distortion, geometric transformations

Objects in the virtual world may move, get distorted, grow or shrink, that is, their equations may also depend on time. To describe dynamic geometry, we usually apply two functions. The first function selects those points of space, which belong to the object in its reference state. The second function maps these points onto points defining the object in an arbitrary time instance. Functions mapping the space onto itself are called ***transformations.*** A transformation maps point $\vec{r}$ to point $\vec{r}\,' = \mathcal{T}(\vec{r})$. If the transformation is invertible, we can also find the original for some transformed point $\vec{r}\,'$ using inverse transformation $\mathcal{T}^{-1}(\vec{r}\,')$.

If the object is defined in its reference state by implicit equation $f(\vec{r}) \geq 0$, then the equation of the transformed object is

$$\{\vec{r}\,' : f(\mathcal{T}^{-1}(\vec{r}\,')) \geq 0\}\,, \tag{14.12}$$

since the originals belong to the set of points of the reference state.

Parametric equations define the Cartesian coordinates of the points directly. Thus the transformation of parametric surface $\vec{r} = \vec{r}(u, v)$ requires the transformations of its points

$$\vec{r}\,'(u, v) = \mathcal{T}(\vec{r}(u, v)) \ . \tag{14.13}$$

Similarly, the transformation of curve $\vec{r} = \vec{r}(t)$ is:

$$\vec{r}\,'(t) = \mathcal{T}(\vec{r}(t)) \ . \tag{14.14}$$

Transformation $\mathcal{T}$ may change the type of object in the general case. It can happen, for example, that a simple triangle or a sphere becomes a complicated shape, which are hard to describe and handle. Thus it is worth limiting the set of allowed transformations. Transformations mapping planes onto planes, lines onto lines and points onto points are particularly important. In the next subsection we consider the class of ***homogenous linear transformations***, which meet this requirement.

14.5.1. Projective geometry and homogeneous coordinates

So far the construction of the virtual world has been discussed using the means of the Euclidean geometry, which gave us many important concepts such as distance, parallelism, angle, etc. However, when the transformations are discussed in details, many of these concepts are unimportant, and can cause confusion. For example, parallelism is a relationship of two lines which can lead to singularities when the intersection of two lines is considered. Therefore, transformations are discussed in the context of another framework, the projective geometry.

The axioms of ***projective geometry*** turn around the problem of parallel lines by ignoring the concept of parallelism altogether, and state that two different lines always have an intersection. To cope with this requirement, every line is extended by a „point in infinity" such that two lines have the same extra point if and only if the two lines are parallel. The extra point is called the ***ideal point***. The ***projective space*** contains the points of the Euclidean space (these are the so called ***affine points***) and the ideal points. An ideal point „glues" the „ends" of an Euclidean line, making it topologically similar to a circle. Projective geometry preserves that axiom of the Euclidean geometry which states that two points define a line. In order to make it valid for ideal points as well, the set of lines of the Euclidean space is extended by a new line containing the ideal points. This new line is called the ***ideal line.*** Since the ideal points of two lines are the same if and only if the two lines are parallel, the ideal lines of two planes are the same if and only if the two planes are parallel. Ideal lines are on the ***ideal plane***, which is added to the set of planes of the Euclidean space. Having made these extensions, no distinction is needed between the affine and ideal points. They are equal members of the projective space.

Introducing analytic geometry we noted that everything should be described by numbers in computer graphics. Cartesian coordinates used so far are in one to one relationship with the points of Euclidean space, thus they are inappropriate to describe the points of the projective space. For the projective plane and space, we need a different algebraic base.

Projective plane

Let us consider first the projective plane and find a method to describe its points by numbers. To start, a Cartesian coordinate system x, y is set up in this plane. Simultaneously, another

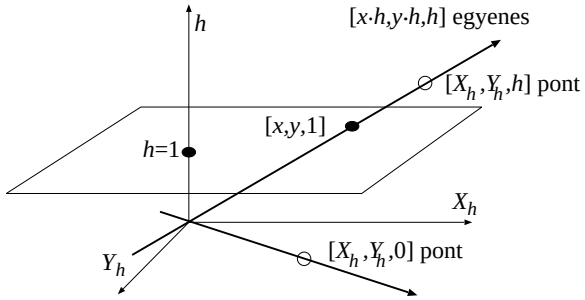


Figure 14.29. The embedded model of the projective plane: the projective plane is embedded into a three-dimensional Euclidean space, and a correspondence is established between points of the projective plane and lines of the embedding three-dimensional Euclidean space by fitting the line to the origin of the three-dimensional space and the given point.

Cartesian system X_h, Y_h, h is established in the three-dimensional space embedding the plane in a way that axes X_h, Y_h are parallel to axes x, y, the plane is perpendicular to axis h, the origin of the Cartesian system of the plane is in point $(0, 0, 1)$ of the three-dimensional space, and the points of the plane satisfy equation $h = 1$. The projective plane is thus embedded into a three-dimensional Euclidean space where points are defined by Descartes-coordinates (figure 14.29). To describe a point of the projective plane by numbers, a correspondence is found between the points of the projective plane and the points of the embedding Euclidean space. An appropriate correspondence assigns that line of the Euclidean space to either affine or ideal point P of the projective plane which is defined by the origin of the coordinate system of the space and point P.

Points of a line in the Euclidean space can be given by parametric equation $[t \cdot X_h, t \cdot Y_h, t \cdot h]$ where t is a free real parameter. If point P is an affine point of the projective plane, then the corresponding line is not parallel with plane $h = 1$ (i.e. h is not constant zero). Such line intersects the plane of equation $h = 1$ at point $[X_h/h, Y_h/h, 1]$, thus the Cartesian coordinates of point P in planar coordinate system x, y are $(X_h/h, Y_h/h)$. On the other hand, if point P is ideal, then the corresponding line is parallel to the plane of equation $h = 1$ (i.e. $h = 0$). The direction of the ideal point is given by vector (X_h, Y_h).

The presented approach assigns three dimensional lines and eventually $[X_h, Y_h, h]$ triplets to both the affine and the ideal points of the projective plane. These triplets are called the ***homogenous coordinates*** of a point in the projective plane. Homogeneous coordinates are enclosed by brackets to distinguish them from Cartesian coordinates.

A three-dimensional line crossing the origin and describing a point of the projective plane can be defined by its arbitrary point except from the origin. Consequently, all three homogeneous coordinates cannot be simultaneously zero, and homogeneous coordinates can be freely multiplied by the same non-zero scalar without changing the described point. This property justifies the name „homogenous”.

It is often convenient to select that triplet from the homogeneous coordinates of an affine point, where the third homogeneous coordinate is 1 since in this case the first two homogeneous coordinates are identical to the Cartesian coordinates:

$$X_h = x, \qquad Y_h = y, \qquad h = 1\ . \tag{14.15}$$

From another point of view, Cartesian coordinates of an affine point can be converted to

homogenous coordinates by extending the pair by a third element of value 1.

The embedded model also provides means to define the equations of the lines and line segments of the projective space. Let us select two different points on the projective plane and specify their homogeneous coordinates. The two points are different if homogeneous coordinates $[X_h^1, Y_h^1, h^1]$ of the first point cannot be obtained as a scalar multiple of homogeneous coordinates $[X_h^2, Y_h^2, h^2]$ of the other point. In the embedding space, triplet $[X_h, Y_h, h]$ can be regarded as Cartesian coordinates, thus the ***equation of the line*** fitted to points $[X_h^1, Y_h^1, h^1]$ and $[X_h^2, Y_h^2, h^2]$ is:

$$\begin{aligned} X_h(t) &= X_h^1 \cdot (1-t) + X_h^2 \cdot t \,, \\ Y_h(t) &= Y_h^1 \cdot (1-t) + Y_h^2 \cdot t \,, \\ h(t) &= h^1 \cdot (1-t) + h^2 \cdot t \,. \end{aligned} \tag{14.16}$$

If $h(t) \neq 0$, then the affine points of the projective plane can be obtained by projecting the three-dimensional space onto the plane of equation $h = 1$. Requiring the two points be different, we excluded the case when the line would be projected to a single point. Hence projection maps lines to lines. Thus the presented equation really identifies the homogeneous coordinates defining the points of the line. If $h(t) = 0$, then the equation expresses the ideal point of the line.

If parameter t has an arbitrary real value, then the points of a line are defined. If parameter t is restricted to interval $[0, 1]$, then we obtain the line segment defined by the two endpoints.

Projective space

We could apply the same method to introduce homogeneous coordinates of the projective space as we used to define the homogeneous coordinates of the projective plane, but this approach would require the embedding of the three-dimensional projective space into a four-dimensional Euclidean space, which is not intuitive. We would rather discuss another construction, which works in arbitrary dimensions. In this construction, a point is described as the centre of mass of a mechanical system. To identify a point, let us place weight X_h at reference point $\vec{p}_1$, weight Y_h at reference point $\vec{p}_2$, weight Z_h at reference point $\vec{p}_3$, and weight w at reference point $\vec{p}_4$. The centre of mass of this mechanical system is:

$$\vec{r} = \frac{X_h \cdot \vec{p}_1 + Y_h \cdot \vec{p}_2 + Z_h \cdot \vec{p}_3 + w \cdot \vec{p}_4}{X_h + Y_h + Z_h + w} \,.$$

Let us denote the total weight by $h = X_h + Y_h + Z_h + w$. By definition, elements of quadruple $[X_h, Y_h, Z_h, h]$ are the ***homogeneous coordinates*** of the centre of mass.

To find the correspondence between homogeneous and Cartesian coordinates, the relationship of the two coordinate systems (the relationship of the basis vectors and the origin of the Cartesian coordinate system and of the reference points of the homogeneous coordinate system) must be established. Let us assume, for example, that the reference points of the homogenous coordinate system are in points (1,0,0), (0,1,0), (0,0,1), and (0,0,0) of the Cartesian coordinate system. The centre of mass (assuming that total weight h is not zero) is expressed in Cartesian coordinates as follows:

$$\vec{r}[X_h, Y_h, Z_h, h] = \frac{1}{h} \cdot (X_h \cdot (1,0,0) + Y_h \cdot (0,1,0) + Z_h \cdot (0,0,1) + w \cdot (0,0,0)) = \left(\frac{X_h}{h}, \frac{Y_h}{h}, \frac{Z_h}{h} \right) .$$

Hence the correspondence between homogeneous coordinates $[X_h, Y_h, Z_h, h]$ and Cartesian coordinates (x, y, z) is $(h \neq 0)$:

$$x = \frac{X_h}{h}, \qquad y = \frac{Y_h}{h}, \qquad z = \frac{Z_h}{h} \ . \tag{14.17}$$

The equations of ***lines in the projective space*** can be obtained either deriving them from the embedding four-dimensional Cartesian space, or using the centre of mass analogy:

$$\begin{aligned} X_h(t) &= X_h^1 \cdot (1-t) + X_h^2 \cdot t \ , \\ Y_h(t) &= Y_h^1 \cdot (1-t) + Y_h^2 \cdot t \ , \\ Z_h(t) &= Z_h^1 \cdot (1-t) + Z_h^2 \cdot t \ , \\ h(t) &= h^1 \cdot (1-t) + h^2 \cdot t \ . \end{aligned} \tag{14.18}$$

If parameter t is restricted to interval $[0, 1]$, then we obtain the equation of the ***projective line segment.***

To find the equation of the ***projective plane***, the equation of the Euclidean plane is considered (equation 14.1). The Cartesian coordinates of the points on a plane satisfy the following implicit equation

$$n_x \cdot x + n_y \cdot y + n_z \cdot z + d = 0 \ .$$

Using the correspondence between the Cartesian and homogenous coordinates (equation 14.17) we still describe the points of the Euclidean plane but now with homogenous coordinates:

$$n_x \cdot \frac{X_h}{h} + n_y \cdot \frac{Y_h}{h} + n_z \cdot \frac{Z_h}{h} + d = 0 \ .$$

Let us multiply both sides of this equation by h, and add those points to the plane which have $h = 0$ coordinate and satisfy this equation. With this step the set of points of the Euclidean plane is extended with the ideal points, that is, we obtained the set of points belonging to the projective plane. Hence the equation of the projective plane is a homogenous linear equation:

$$n_x \cdot X_h + n_y \cdot Y_h + n_z \cdot Z_h + d \cdot h = 0 \ , \tag{14.19}$$

or in matrix form:

$$[X_h, Y_h, Z_h, h] \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} = 0 \ . \tag{14.20}$$

Note that points and planes are described by row and column vectors, respectively. Both the quadruples of points and the quadruples of planes have the homogeneous property, that is, they can be multiplied by non-zero scalars without altering the solutions of the equation.

14.5.2. Homogenous linear transformations

Transformations defined as the multiplication of the homogenous coordinate vector of a point by a constant 4×4 $\mathbf{T}$ matrix are called ***homogeneous linear transformations***:

$$[X'_h, Y'_h, Z'_h, h'] = [X_h, Y_h, Z_h, h] \cdot \mathbf{T} \ . \tag{14.21}$$

Theorem 14.12 *Homogeneous linear transformations map points to points.*

Proof. A point can be defined by homogeneous coordinates in form $\lambda \cdot [X_h, Y_h, Z_h, h]$, where λ is an arbitrary, non-zero constant. The transformation results in $\lambda \cdot [X'_h, Y'_h, Z'_h, h'] = \lambda \cdot [X_h, Y_h, Z_h, h] \cdot \mathbf{T}$ when a point is transformed, which are the λ-multiples of the same vector, thus the result is a single point in homogeneous coordinates. ■

Note that due to the homogeneous property, homogeneous transformation matrix $\mathbf{T}$ is not unambiguous, but can be freely multiplied by non-zero scalars without modifying the realized mapping.

Theorem 14.13 *Invertible homogeneous linear transformations map lines to lines.*

Proof. Let us consider the parametric equation of a line:

$$[X_h(t), Y_h(t), Z_h(t), h(t)] = [X_h^1, Y_h^1, Z_h^1, h^1] \cdot (1-t) + [X_h^2, Y_h^2, Z_h^2, h^2] \cdot t, \qquad t = (-\infty, \infty) ,$$

and transform the points of this line by multiplying the quadruples with the transformation matrix:

$$\begin{aligned}
[X'_h(t), Y'_h(t), Z'_h(t), h'(t)] &= [X_h(t), Y_h(t), Z_h(t), h(t)] \cdot \mathbf{T} \\
&= [X_h^1, Y_h^1, Z_h^1, h^1] \cdot \mathbf{T} \cdot (1-t) + [X_h^2, Y_h^2, Z_h^2, h^2] \cdot \mathbf{T} \cdot t \\
&= [X_h^{1'}, Y_h^{1'}, Z_h^{1'}, h^{1'}] \cdot (1-t) + [X_h^{2'}, Y_h^{2'}, Z_h^{2'}, h^{2'}] \cdot t ,
\end{aligned}$$

where $[X_h^{1'}, Y_h^{1'}, Z_h^{1'}, h^{1'}]$ and $[X_h^{2'}, Y_h^{2'}, Z_h^{2'}, h^{2'}]$ are the transformations of $[X_h^1, Y_h^1, Z_h^1, h^1]$ and $[X_h^2, Y_h^2, Z_h^2, h^2]$, respectively. Since the transformation is invertible, the two points are different. The resulting equation is the equation of a line fitted to the transformed points. ■

We note that if we had not required the invertibility of the the transformation, then it could have happened that the transformation would have mapped the two points to the same point, thus the line would have degenerated to single point.

If parameter t is limited to interval $[0, 1]$, then we obtain the equation of the projective line segment, thus we can also state that a homogeneous linear transformation maps a line segment to a line segment. Even more generally, a homogeneous linear transformation maps convex combinations to convex combinations. For example, triangles of the projective plane are mapped to triangles.

However, we have to be careful when we try to apply this theorem in the Euclidean plane or space. Let us consider a line segment as an example. If coordinate h has different sign at the two endpoints, then the line segment contains an ideal point. Such projective line segment can be intuitively imagined as two half lines and an ideal point sticking the "endpoints" of these half lines at infinity, that is, such line segment is the complement of the line segment we are accustomed to. It may happen that before the transformation coordinates h of the endpoints have similar sign, that is, the line segment meets our intuitive image about Euclidean line segments, but after the transformation, coordinates h of the endpoints will have different sign. Thus the transformation wraps around our line segment.

Theorem 14.14 *Invertible linear transformations map planes to planes.*

Proof. The originals of transformed points $[X'_h, Y'_h, Z'_h, h']$ defined by $[X_h, Y_h, Z_h, h] = [X'_h, Y'_h, Z'_h, h'] \cdot \mathbf{T}^{-1}$ are on a plane, thus satisfy the original equation of the plane:

$$[X_h, Y_h, Z_h, h] \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} = [X'_h, Y'_h, Z'_h, h'] \cdot \mathbf{T}^{-1} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} = 0 \, .$$

Due to the associativity of matrix multiplication, the transformed points also satisfy equation

$$[X'_h, Y'_h, Z'_h, h'] \cdot \begin{bmatrix} n'_x \\ n'_y \\ n'_z \\ d' \end{bmatrix} = 0 \, ,$$

which is also a plane equation, where

$$\begin{bmatrix} n'_x \\ n'_y \\ n'_z \\ d' \end{bmatrix} = \mathbf{T}^{-1} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} \, .$$

This result can be used to obtain the normal vector of a transformed plane. ■

An important subclass of homogeneous linear transformations is the set of ***affine transformations,*** where the Cartesian coordinates of the transformed point are linear functions of the original Cartesian coordinates:

$$[x', y', z'] = [x, y, z] \cdot \mathbf{A} + [p_x, p_y, p_z] \, , \tag{14.22}$$

where vector $\vec{p}$ describes translation, $\mathbf{A}$ is a matrix of size 3×3 and expresses rotation, scaling, mirroring, etc., and their arbitrary combination. For example, the rotation around axis (t_x, t_y, t_z), $(|(t_x, t_y, t_z)| = 1)$ by angle ϕ is given by the following matrix

$$\mathbf{A} = \begin{bmatrix} (1 - t_x^2)\cos\phi + t_x^2 & t_x t_y (1 - \cos\phi) + t_z \sin\phi & t_x t_z (1 - \cos\phi) - t_y \sin\phi \\ t_y t_x (1 - \cos\phi) - t_z \sin\phi & (1 - t_y^2)\cos\phi + t_y^2 & t_x t_z (1 - \cos\phi) + t_x \sin\phi \\ t_z t_x (1 - \cos\phi) + t_y \sin\phi & t_z t_y (1 - \cos\phi) - t_x \sin\phi & (1 - t_z^2)\cos\phi + t_z^2 \end{bmatrix} \, .$$

This expression is known as the ***Rodrigues formula***.

Affine transformations map the Euclidean space onto itself, and transform parallel lines to parallel lines. Affine transformations are also homogeneous linear transformations since equation (14.22) can also be given as a 4×4 matrix operation, having changed the Cartesian coordinates to homogeneous coordinates by adding a fourth coordinate of value 1:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 \\ A_{21} & A_{22} & A_{23} & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ p_x & p_y & p_z & 1 \end{bmatrix} = [x, y, z, 1] \cdot \mathbf{T} \, . \tag{14.23}$$

A further specialization of affine transformations is the set of ***congruence transformations*** (isometries) which are distance and angle preserving.

Theorem 14.15 *In a congruence transformation the rows of matrix* $\mathbf{A}$ *have unit length and are orthogonal to each other.*

Proof. Let us use the property that a congruence is distance and angle preserving for the case when the origin and the basis vectors of the Cartesian system are transformed. The transformation assigns point (p_x, p_y, p_z) to the origin and points $(A_{11} + p_x, A_{12} + p_y, A_{13} + p_z)$, $(A_{21} + p_x, A_{22} + p_y, A_{23} + p_z)$, and $(A_{31} + p_x, A_{32} + p_y, A_{33} + p_z)$ to points $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, respectively. Because the distance is preserved, the distances between the new points and the new origin are still 1, thus $|A_{11}, A_{12}, A_{13}| = 1$, $|A_{21}, A_{22}, A_{23}| = 1$, and $|A_{31}, A_{32}, A_{33}| = 1$. On the other hand, because the angle is also preserved, vectors (A_{11}, A_{12}, A_{13}), (A_{21}, A_{22}, A_{23}), and (A_{31}, A_{32}, A_{33}) are also perpendicular to each other. ■

Exercises

14.5-1 Using the Cartesian coordinate system as an algebraic basis, prove the axioms of the Euclidean geometry, for example, that two points define a line, and that two different lines may intersect each other at most at one point.

14.5-2 Using the homogenous coordinates as an algebraic basis, prove an axiom of the projective geometry stating that two different lines intersect each other in exactly one point.

14.5-3 Prove that homogeneous linear transformations map line segments to line segments using the centre of mass analogy.

14.5-4 How does an affine transformation modify the volume of an object?

14.5-5 Give the matrix of that homogeneous linear transformation which translates by vector $\vec{p}$.

14.5-6 Prove the Rodrigues formula.

14.5-7 A solid defined by equation $f(\vec{r}) \geq 0$ in time $t = 0$ moves with uniform constant velocity $\vec{v}$. Let us find the equation of the solid at an arbitrary time instance t.

14.5-8 Prove that if the rows of matrix $\mathbf{A}$ are of unit length and are perpendicular to each other, then the affine transformation is a congruence. Show that for such matrices $\mathbf{A}^{-1} = \mathbf{A}^T$.

14.5-9 Give that homogeneous linear transformation which projects the space from point $\vec{c}$ onto a plane of normal $\vec{n}$ and place vector $\vec{r}_0$.

14.5-10 Show that five point correspondences unambiguously identify a homogeneous linear transformation if no four points are co-planar.

14.6. Rendering with ray tracing

When a virtual world is rendered, we have to identify the surfaces visible in different directions from the virtual eye. The set of possible directions is defined by a rectangle shaped window which is decomposed to a grid corresponding to the pixels of the screen (figure 14.30). Since a pixel has a unique colour, it is enough to solve the visibility problem in a single point of each pixel, for example, in the points corresponding to pixel centres.

The surface visible at a direction from the eye can be identified by casting a half line, called ***ray***, from the eye at that direction and identifying its intersection closest to the eye position. This operation is called ***ray tracing***.

Ray tracing has many applications. For example, ***shadow*** computation tests whether or

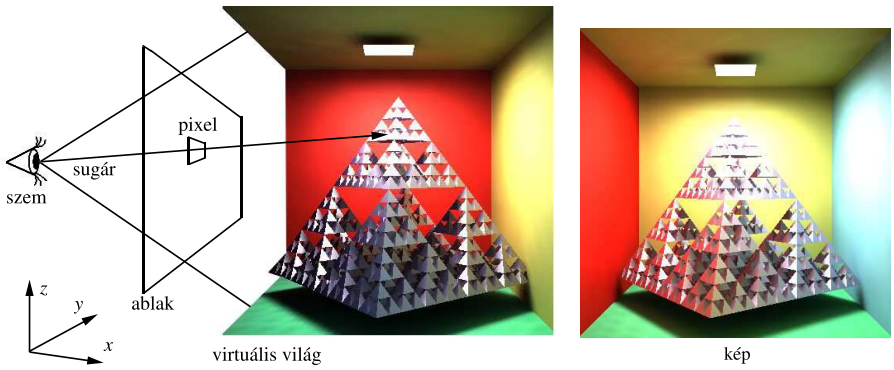


Figure 14.30. Ray tracing

not a point is occluded from the light source, which requires a ray to be sent from the point at the direction of the light source and the determination whether this ray intersects any surface closer than the light source. Ray tracing is also used by ***collision detection*** since a point moving with constant and uniform speed collides that surface which is first intersected by the ray describing the motion of the point.

A ray is defined by the following equation:

$$\vec{r}(t) = \vec{s} + \vec{v} \cdot t, \qquad (t > 0)\ , \tag{14.24}$$

where $\vec{s}$ is the place vector of the ***ray origin***, $\vec{v}$ is the ***direction of the ray***, and ***ray parameter*** t characterizes the distance from the origin. Let us suppose that direction vector $\vec{v}$ has unit length. In this case parameter t is the real distance, otherwise it would only be proportional to the distance[3]. If parameter t is negative, then the point is behind the eye and is obviously not visible. The identification of the closest intersection with the ray means the determination of the intersection point having the smallest, positive ray parameter. In order to find the closest intersection, the intersection calculation is tried with each surface, and the closest is retained. This algorithm is presented in the followings:

[3]In collision detection $\vec{v}$ is not a unit vector, but the velocity of the moving point since this makes ray parameter t express the collision time.

RAY-FIRST-INTERSECTION($\vec{s}$, $\vec{v}$)

```
1  t ← t_max                                  ▷ initialization to the maximum size in the virtual world.
2  for each object o
3      do t_o ←RAY-SURFACE-INTERSECTION(s, v)   ▷ negative if no intersection exists.
4          if 0 ≤ t_o < t                       ▷ is the new intersection closer?
5              then t ← t_o                     ▷ ray parameter of the closest intersection so far.
6                   o_visible ← o               ▷ closest object so far.
7  if t < t_max then                            ▷ has been intersection at all?
8     then x ← s + v · t                        ▷ intersection point using the ray equation.
9          return t, x, o_visible
10    else return „no intersection"             ▷ no intersection.
```

This algorithm inputs the ray defined by origin $\vec{s}$ and direction $\vec{v}$, and outputs the ray parameter of the intersection in variable t, the intersection point in $\vec{x}$, and the visible object in $o_{visible}$. The algorithm calls function RAY-SURFACE-INTERSECTION for each object, which determines the intersection of the ray and the given object, and indicates with a negative return value if no intersection exists. Function RAY-SURFACE-INTERSECTION should be implemented separately for each surface type.

14.6.1. Ray-surface intersection calculation

The identification of the intersection between a ray and a surface requires the solution of an equation. The intersection point is both on the ray and on the surface, thus it can be obtained by inserting the ray equation into the equation of the surface and solving the resulting equation for the unknown ray parameter.

Intersection calculation for implicit surfaces

For implicit surfaces of equation $f(\vec{r}) = 0$, the intersection can be calculated by solving the following scalar equation for t: $f(\vec{s} + \vec{v} \cdot t) = 0$.

Let us take the example of ***quadrics*** that include the sphere, the ellipsoid, the cylinder, the cone, the paraboloid, etc. The implicit equation of a general quadric contains a quadratic form:

$$[x, y, z, 1] \cdot \mathbf{Q} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0 \ ,$$

where $\mathbf{Q}$ is a 4×4 matrix. Substituting the ray equation into the equation of the surface, we obtain

$$[s_x + v_x \cdot t, s_y + v_y \cdot t, s_z + v_z \cdot t, 1] \cdot \mathbf{Q} \cdot \begin{bmatrix} s_x + v_x \cdot t \\ s_y + v_y \cdot t \\ s_z + v_z \cdot t \\ 1 \end{bmatrix} = 0 \ .$$

Rearranging the terms, we get a second order equation for unknown parameter t:

$$t^2 \cdot (\mathbf{v} \cdot \mathbf{Q} \cdot \mathbf{v}^T) + t \cdot (\mathbf{s} \cdot \mathbf{Q} \cdot \mathbf{v}^T + \mathbf{v} \cdot \mathbf{Q} \cdot \mathbf{s}^T) + (\mathbf{s} \cdot \mathbf{Q} \cdot \mathbf{s}^T) = 0 \ ,$$

where $\mathbf{v} = [v_x, v_y, v_z, 0]$ and $\mathbf{s} = [s_x, s_y, s_z, 1]$.

This equation can be solved using the solution formula of second order equations. Now we are interested in only the real and positive roots. If two such roots exist, then the smaller one corresponds to the intersection closer to the origin of the ray.

Intersection calculation for parametric surfaces

The intersection of parametric surface $\vec{r} = \vec{r}(u, v)$ and the ray is calculated by first solving the following equation for unknown parameters u, v, t

$$\vec{r}(u, v) = \vec{s} + t \cdot \vec{v} \,,$$

then checking whether or not t is positive and parameters u, v are inside the allowed parameter range of the surface.

Roots of non-linear equations are usually found by numeric methods. On the other hand, the surface can also be approximated by a triangle mesh, which is intersected by the ray. Having obtained the intersection on the coarse mesh, the mesh around this point is refined, and the intersection calculation is repeated with the refined mesh.

Intersection calculation for a triangle

To compute the ray intersection for a ***triangle*** of vertices $\vec{a}$, $\vec{b}$, and $\vec{c}$, first the ray intersection with the plane of the triangle is found. Then it is decided whether or not the intersection point with the plane is inside the triangle. The normal and a place vector of the triangle plane are $\vec{n} = (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})$, and $\vec{a}$, respectively, thus points $\vec{r}$ of the plane satisfy the following equation:

$$\vec{n} \cdot (\vec{r} - \vec{a}) = 0 \,. \tag{14.25}$$

The intersection of the ray and this plane is obtained by substituting the ray equation (equation (14.24)) into this plane equation, and solving it for unknown parameter t. If root t^* is positive, then it is inserted into the ray equation to get the intersection point with the plane. However, if the root is negative, then the intersection is behind the origin of the ray, thus is invalid. Having a valid intersection with the plane of the triangle, we check whether this point is inside the triangle. This is a containment problem, which is discussed in subsection 14.4.1.

Intersection calculation for an AABB

The surface of an AABB, that is an axis aligned block, can be subdivided to 6 rectangular faces, or alternatively to 12 triangles, thus its intersection can be solved by the algorithms discussed in the previous subsections. However, realizing that in this special case the three coordinates can be handled separately, we can develop more efficient approaches. In fact, an AABB is the intersection of an x-stratum defined by inequality $x_{min} \leq x \leq x_{max}$, a y-stratum defined by $y_{min} \leq y \leq y_{max}$ and a z-stratum of inequality $z_{min} \leq z \leq z_{max}$. For example, the ray parameter of the intersection with the x-stratum is:

$$t_x^1 = \frac{x_{min} - s_x}{v_x}, \qquad t_x^2 = \frac{x_{max} - s_x}{v_x} \,.$$

The smaller of the two parameter values corresponds to the entry at the stratum, while the greater to the exit. Let us denote the ray parameter of the entry by t_{in}, and the ray parameter

of the exit by t_{out}. The ray is inside the x-stratum while the ray parameter is in $[t_{in}, t_{out}]$. Repeating the same calculation for the y and z-strata as well, three ray parameter intervals are obtained. The intersection of these intervals determine when the ray is inside the AABB. If parameter t_{out} obtained as the result of intersecting the strata is negative, then the AABB is behind the eye, thus no ray–AABB intersection is possible. If only t_{in} is negative, then the ray starts at an internal point of the AABB, and the first intersection is at t_{out}. Finally, if t_{in} is positive, then the ray enters the AABB from outside at parameter t_{in}.

The computation of the unnecessary intersection points can be reduced by applying the Cohen – Sutherland line clipping algorithm (subsection 14.4.3). First, the ray is replaced by a line segment where one endpoint is the origin of the ray, and the other endpoint is an arbitrary point on the ray which is farther from the origin than any object of the virtual world.

14.6.2. Speeding up the intersection calculation

A naive ray tracing algorithm tests each object for a ray to find the closest intersection. If there are N objects in the space, the running time of the algorithm is $\Theta(N)$ both in the average and in the worst case. The storage requirement is also linear in terms of the number of objects.

The method would be speeded up if we could exclude certain objects from the intersection test without testing them one by one. The reasons of such exclusion include that these objects are „behind" the ray or „not in the direction of the ray". Additionally, the speed is also expected to improve if we can terminate the search having found an intersection supposing that even if other intersections exist, they are surely farther than the just found intersection point. To make such decisions safely, we need to know the arrangement of objects in the virtual world. This information is gathered during the pre-processing phase. Of course, pre-processing has its own computational cost, which is worth spending if we have to trace a lot of rays.

Bounding volumes

One of the simplest ray tracing acceleration technique uses ***bounding volumes***. The bounding volume is a shape of simple geometry, typically a sphere or an AABB, which completely contains a complex object. When a ray is traced, first the bounding volume is tried to be intersected. If there is no intersection with the bounding volume, then neither can the contained object be intersected, thus the computation time of the ray intersection with the complex object is saved. The bounding volume should be selected in a way that the ray intersection is computationally cheap, and it is a tight container of the complex object.

The application of bounding volumes does not alter the linear time complexity of the naive ray tracing. However, it can increase the speed by a scalar factor.

On the other hand, bounding volumes can also be organized in a hierarchy putting bounding volumes inside bigger bounding volumes recursively. In this case the ray tracing algorithm traverses this hierarchy, which is possible in sub-linear time.

Space subdivision with uniform grids

Let us find the AABB of the complete virtual world and subdivide it by an axis aligned uniform grid of cell sizes (c_x, c_y, c_z) (figure 14.31).

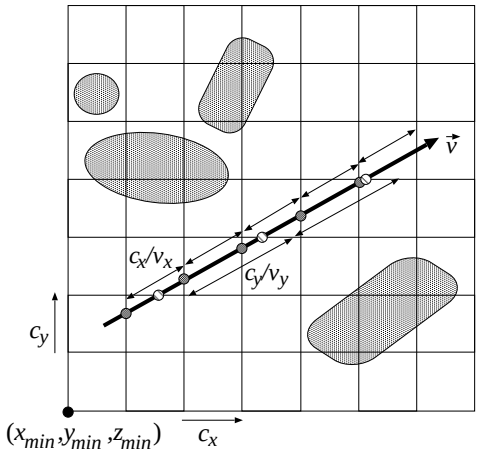


Figure 14.31. Partitioning the virtual world by a uniform grid. The intersections of the ray and the coordinate planes of the grid are at regular distances c_x/v_x,c_y/v_y, and c_z/v_z, respectively.

In the preprocessing phase, for each cell we identify those objects that are at least partially contained by the cell. The test of an object against a cell can be performed using a clipping algorithm (subsection 14.4.3), or simply checking whether the cell and the AABB of the object overlap.

UNIFORM-GRID-CONSTRUCTION()

```
1  Compute the minimum corner of the AABB (x_min, y_min, z_min) and cell sizes (c_x, c_y, c_z)
2  for each cell c
3      do object list of cell c ← empty
4          for each object o                    ▷ register objects overlapping with this cell.
5              do if cell c and the AABB of object o overlap
6                     then add object o to object list of cell c
```

During ray tracing, cells intersected by the ray are visited in the order of their distance from the ray origin. When a cell is processed, only those objects need to be tested for intersection which overlap with this cell, that is, which are registered in this cell. On the other hand, if an intersection is found in the cell, then intersections belonging to other cells cannot be closer to the ray origin than the found intersection. Thus the cell marching can be terminated. Note that when an object registered in a cell is intersected by the ray, we should also check whether the intersection point is also in this cell.

We might meet an object again in other cells. The number of ray–surface intersection can be reduced if the results of ray–surface intersections are stored with the objects and are reused when needed again.

As long as no ray–surface intersection is found, the algorithm traverses those cells which are intersected by the ray. Indices X, Y, Z of the first cell are computed from ray origin $\vec{s}$, minimum corner $(x_{min}, y_{min}, z_{min})$ of the grid, and sizes (c_x, c_y, c_z) of the cells:

UNIFORM-GRID-ENCLOSING-CELL($\vec{s}$)

```
1  X ← INTEGER((s_x − x_min)/c_x)
2  Y ← INTEGER((s_y − y_min)/c_y)
3  Z ← INTEGER((s_z − z_min)/c_z)
4  return X, Y, Z
```

The presented algorithm assumes that the origin of the ray is inside the subspace covered by the grid. Should this condition not be met, then the intersection of the ray and the scene AABB is computed, and the ray origin is moved to this point.

The initial values of ray parameters t_x, t_y, t_z are computed as the intersection of the ray and the coordinate planes by the UNIFORM-GRID-RAY-PARAMETER-INITIALIZATION algorithm:

UNIFORM-GRID-RAY-PARAMETER-INITIALIZATION($\vec{s}, \vec{v}, X, Y, Z$)

```
1  if v_x > 0
2     then t_x ← (x_min + (X + 1) · c_x − s_x)/v_x
3     else if v_x < 0
4             then t_x ← (x_min + X · c_x − s_x)/v_x
5             else  t_x ← t_max                          ▷ The maximum distance.
6  if v_y > 0
7     then  t_y ← (y_min + (Y + 1) · c_y − s_y)/v_y
8     else if v_y < 0
9             then t_y ← (y_min + Y · c_y − s_y)/v_y
10            else  t_y ← t_max
11 if v_z > 0
12    then  t_z ← (z_min + (Z + 1) · c_z − s_z)/v_z
13    else if v_z < 0
14            then t_z ← (z_min + Z · c_z − s_z)/v_z
15            else  t_z ← t_max
16 return t_x, t_y, t_z
```

The next cell of the sequence of the visited cells is determined by the ***3D line drawing algorithm (3DDDA algorithm)***. This algorithm exploits the fact that the ray parameters of the intersection points with planes perpendicular to axis x (and similarly to axes y and z) are regularly placed at distance c_x/v_x (c_y/v_y, and c_z/v_z, respectively), thus the ray parameter of the next intersection can be obtained with a single addition (figure 14.31). Ray parameters t_x, t_y, and t_z are stored in global variables, and are incremented by constant values. The smallest from the three ray parameters of the coordinate planes identifies the next intersection with the cell.

The following algorithm computes indices X, Y, Z of the next intersected cell, and updates ray parameters t_x, t_y, t_z:

UNIFORM-GRID-NEXT-CELL(X, Y, Z, t_x, t_y, t_z)

```
1  if t_x = min(t_x, t_y, t_z)          ▷ Next intersection is on the plane perpendicular to axis x.
2    then X ← X + sgn(v_x)                                  ▷ function sgn(x) returns the sign.
3         t_x ← t_x + c_x/|v_x|
4  else if t_y = min(t_x, t_y, t_z)     ▷ Next intersection is on the plane perpendicular to axis y.
5         then Y ← Y + sgn(v_y)
6              t_y ← t_y + c_y/|v_y|
7  else if t_z = min(t_x, t_y, t_z)     ▷ Next intersection is on the plane perpendicular to axis z.
8         then Z ← Z + sgn(v_z)
9              t_z ← t_z + c_z/|v_z|
```

To summarize, a complete ray tracing algorithm is presented, which exploits the uniform grid generated during preprocessing and computes the ray–surface intersection closest to the ray origin. The minimum of ray parameters assigned to the coordinate planes, variable t_{out}, determines the distance as far as the ray is inside the cell. This parameter is used to decide whether or not a ray-surface intersection is really inside the cell.

RAY-FIRST-INTERSECTION-WITH-UNIFORM-GRID$(\vec{s}, \vec{v})$

```
1  (X, Y, Z) ← UNIFORM-GRID-ENCLOSING-CELL(s⃗)
2  (t_x, t_y, t_z) ← UNIFORM-GRID-RAY-PARAMETER-INITIALIZATION(s⃗, v⃗, X, Y, Z)
3  while X, Y, Z are inside the grid
4        do t_out ← min(t_x, t_y, t_z)                          ▷ Here is the exit from the cell.
5           t ← t_out                                          ▷ Initialization: no intersection yet.
6           for each object o registered in cell (X, Y, Z)
7                 do t_o ←RAY-SURFACE-INTERSECTION(s⃗, v⃗, o)     ▷ Negative: no intersection.
8                    if 0 ≤ t_o < t                            ▷ Is the new intersection closer?
9                    then t ← t_o      ▷ The ray parameter of the closest intersection so far.
10                        o_visible ← o                        ▷ The first intersected object.
11                if t < t_out                                 ▷ Was intersection in the cell?
12                   then x⃗ ← s⃗ + v⃗ · t                       ▷ The position of the intersection.
13                        return t, x⃗, o_visible               ▷ Termination.
14          UNIFORM-GRID-NEXT-CELL(X, Y, Z, t_x, t_y, t_z)     ▷ 3DDDA.
15 return „no intersection"
```

Time and storage complexity of the uniform grid algorithm

The preprocessing phase of the uniform grid algorithm tests each object with each cell, thus runs in $\Theta(N \cdot C)$ time where N and C are the numbers of objects and cells, respectively. In practice, the resolution of the grid is set to make C proportional to N since in this case, the average number of objects per cell becomes independent of the total number of objects. Such resolution makes the preprocessing time quadratic, that is $\Theta(N^2)$. We note that sorting objects before testing them against cells may reduce this complexity, but this optimization is not crucial since not the preprocessing but the ray tracing time is critical. Since in the worst case all objects may overlap with each cell, the storage space is also in $O(N^2)$.

The ray tracing time can be expressed by the following equation:

$$T = T_o + N_I \cdot T_I + N_S \cdot T_S, \tag{14.26}$$

where T_o is the time needed to identify the cell containing the origin of the ray, N_I is the number of ray–surface intersection tests until the first intersection is found, T_I is the time required by a single ray–surface intersection test, N_S is the number of visited cells, and T_S is the time needed to step onto the next cell.

To find the first cell, the coordinates of the ray origin should be divided by the cell sizes, and the cell indices are obtained by rounding the results. This step thus runs in constant time. A single ray–surface intersection test also requires constant time. The next cell is determined by the 3DDDA algorithm in constant time as well. Thus the complexity of the algorithm depends only on the number of intersection tests and the number of the visited cells.

Considering a worst case scenario, a cell may contain all objects, requiring $O(N)$ intersection test with N objects. In the worst case the ray tracing has linear complexity. This means that the uniform grid algorithm needs quadratic preprocessing time and storage, but solves the ray tracing problem still in linear time as the naive algorithm, which is quite disappointing. However, uniform grids are still worth using since worst case scenarios are very unlikely. The fact is that classic complexity measures describing the worst case characteristics are not appropriate to compare the naive algorithm and the uniform grid based ray tracing. For a reasonable comparison, the probabilistic analysis of the algorithms is needed.

Probabilistic model of the virtual world

To carry out the average case analysis, the scene model, i.e. the probability distribution of the possible virtual world models must be known. In practical situations, this probability distribution is not available, therefore it must be estimated. If the model of the virtual world were too complicated, we would not be able to analytically determine the average, i.e. the expected running time of the ray tracing algorithm. A simple, but also justifiable model is the following: *Objects are spheres of the same radius r, and sphere centres are uniformly distributed in space.*

Since we are interested in the asymptotic behavior when the number of objects is really high, uniform distribution in a finite space would not be feasible. On the other hand, the boundary of the space would pose problems. Thus, instead of dealing with a finite object space, the space should also be expanded as the number of objects grows to sustain constant average spatial object density. This is a classical method in probability theory, and its known result is the Poisson point process.

Definition 14.16 *A **Poisson point process** $N(A)$ counts the number of points in subset A of space in a way that*

- *$N(A)$ is a Poisson distribution of parameter $\rho V(A)$, where ρ is a positive constant called "intensity" and $V(A)$ is the volume of A, thus the probability that A contains exactly k points is*

$$\Pr\{N(A) = k\} = \frac{(\rho V(A))^k}{k!} \cdot e^{-\rho V(A)},$$

and the expected number of points in volume $V(A)$ is $\rho V(A)$;

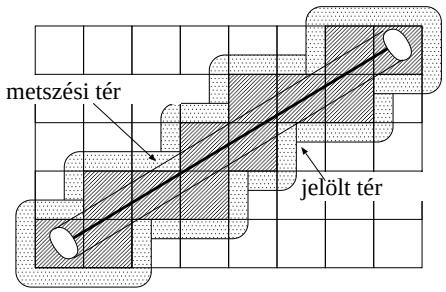


Figure 14.32. Encapsulation of the intersection space by the cells of the data structure in a uniform subdivision scheme. The intersection space is a cylinder of radius r. The candidate space is the union of those spheres that may overlap a cell intersected by the ray.

- *for disjoint $A_1, A_2, \ldots, A_n$ sets random variables $N(A_1), N(A_2), \ldots, N(A_n)$ are independent.*

Using the Poisson point process, the probabilistic model of the virtual world is:

1. The object space consists of spheres of the same radius r.
2. The sphere centres are the realizations of a Poisson point process of intensity ρ.

Having constructed a probabilistic virtual world model, we can start the analysis of the candidate algorithms assuming that the rays are uniformly distributed in space.

Calculation of the expected number of intersections

Looking at figure 14.32 we can see a ray that passes through certain cells of the space partitioning data structure. The collection of those sphere centres where the sphere would have an intersection with a cell is called the ***candidate space*** associated with this cell.

Only those spheres of radius r can have intersection with the ray whose centres are in a cylinder of radius r around the ray. This cylinder is called the ***intersection space*** (figure 14.32). More precisely, the intersection space also includes two half spheres at the bottom and at the top of the cylinder, but these will be ignored.

As the ray tracing algorithm traverses the data structure, it examines each cell that is intersected by the ray. If the cell is empty, then the algorithm does nothing. If the cell is not empty, then it contains, at least partially, a sphere which is tried to be intersected. This intersection succeeds if the centre of the sphere is inside the intersection space and fails if it is outside.

The algorithm should try to intersect objects that are in the candidate space, but this intersection will be successful only if the object is also contained by the intersection space. The probability of the success s is the ratio of the projected areas of the intersection space and the candidate space associated with this cell.

From the probability of the successful intersection in a non-empty cell, the probability that the intersection is found in the first, second, etc. cells can also be computed. Assuming statistical independence, the probabilities that the first, second, third, etc. intersection is the first successful intersection are s, $(1-s)s$, $(1-s)^2 s$, etc., respectively. This is a geometric distribution with expected value $1/s$. Consequently, the expected number of the ray–object

intersection tests is:

$$E[N_I] = \frac{1}{s}. \tag{14.27}$$

If the ray is parallel to one of the sides, then the projected size of the candidate space is $c^2 + 4cr + r^2\pi$ where c is the edge size of a cell and r is the radius of the spheres. The other extreme case happens when the ray is parallel to the diagonal of the cubic cell, where the projection is a rounded hexagon having area $\sqrt{3}c^2 + 6cr + r^2\pi$. The success probability is then:

$$\frac{r^2\pi}{\sqrt{3}c^2 + 6cr + r^2\pi} \leq s \leq \frac{r^2\pi}{c^2 + 4cr + r^2\pi} .$$

According to equation (14.27), the average number of intersection calculations is the reciprocal of this probability:

$$\frac{1}{\pi}\left(\frac{c}{r}\right)^2 + \frac{4}{\pi}\frac{c}{r} + 1 \leq \mathrm{E}\left[N_I\right] \leq \frac{\sqrt{3}}{\pi}\left(\frac{c}{r}\right)^2 + \frac{6}{\pi}\frac{c}{r} + 1 . \tag{14.28}$$

Note that if the size of the cell is equal to the diameter of the sphere ($c = 2r$), then

$$3.54 < \mathrm{E}\left[N_I\right] < 7.03 .$$

This result has been obtained assuming that the number of objects converges to infinity. The expected number of intersection tests, however, remains finite and relatively small.

Calculation of the expected number of cell steps

In the following analysis the conditional expected value theorem will be used. An appropriate condition is the length of the ray segment between its origin and the closest intersection. Using its probability density $p_{t^*}(t)$ as a condition, the expected number of visited cells N_S can be written in the following form:

$$E[N_S] = \int_0^\infty E[N_S | t^* = t] \cdot p_{t^*}(t)\, dt ,$$

where t^* is the length of the ray and p_{t^*} is its probability density.

Since the intersection space is a cylinder if we ignore the half spheres around the beginning and the end, its total volume is $r^2\pi t$. Thus the probability that intersection occurs before t is:

$$\Pr\{t^* < t\} = 1 - e^{-\rho r^2 \pi t} .$$

Note that this function is the cumulative probability distribution function of t^*. The probability density can be computed as its derivative, thus we obtain:

$$p_{t^*}(t) = \rho r^2 \pi \cdot e^{-\rho r^2 \pi t} .$$

The expected length of the ray is then:

$$E[t^*] = \int_0^\infty t \cdot \rho r^2 \pi \cdot e^{-\rho r^2 \pi t}\, dt = \frac{1}{\rho r^2 \pi}. \tag{14.29}$$

In order to simplify the analysis, we shall assume that the ray is parallel to one of the coordinate axes. Since all cells have the same edge size c, the number of cells intersected by a ray of length t can be estimated as $E[N_S|t^* = t] \approx t/c + 1$. This estimation is quite accurate. If the the ray is parallel to one of the coordinate axes, then the error is at most 1. In other cases the real value can be at most $\sqrt{3}$ times the given estimation. The estimated expected number of visited cells is then:

$$\mathrm{E}[N_S] \approx \int_0^\infty \left(\frac{t}{c} + 1\right) \cdot \rho r^2 \pi \cdot e^{-\rho r^2 \pi t}\, dt = \frac{1}{c\rho r^2 \pi} + 1\ . \tag{14.30}$$

For example, if the cell size is similar to the object size ($c = 2r$), and the expected number of sphere centres in a cell is 0.1, then $\mathrm{E}[N_S] \approx 14$. Note that the expected number of visited cells is also constant even for infinite number of objects.

Expected running time and storage space

We concluded that the expected numbers of required intersection tests and visited cells are asymptotically constant, thus the expected time complexity of the uniform grid based ray tracing algorithm is constant after quadratic preprocessing time. The value of the running time can be controlled by cell size c according to equations (14.28) and (14.30). Smaller cell sizes reduce the average number of intersection tests, but increase the number of visited cells.

According to the probabilistic model, the average number of objects overlapping with a cell is also constant, thus the storage is proportional to the number of cells. Since the number of cells is set proportional to the number of objects, the expected storage complexity is also linear unlike the quadratic worst-case complexity.

The expected constant running time means that asymptotically the running time is independent of the number of objects, which explains the popularity of the uniform grid based ray tracing algorithm, and also the popularity of the algorithms presented in the next subsections.

Octree

Uniform grids require many unnecessary cell steps. For example, the empty spaces are not worth partitioning into cells, and two cells are worth separating only if they contain different objects. Adaptive space partitioning schemes are based on these recognitions. The space can be partitioned adaptively following a recursive approach. This results in a hierarchical data structure, which is usually a tree. The type of this tree is the base of the classification of such algorithms.

The adaptive scheme discussed in this subsection uses an octal tree (octree for short), where non-empty nodes have 8 children. An octree is constructed by the following algorithm:

- For each object, an AABB is found, and object AABBs are enclosed by a scene AABB. The scene AABB is the cell corresponding to the root of the octree.
- If the number of objects overlapping with the current cell exceeds a predefined threshold, then the cell is subdivided to 8 cells of the same size by halving the original cell along each coordinate axis. The 8 new cells are the children of the node corresponding to the original cell. The algorithm is recursively repeated for the child cells.

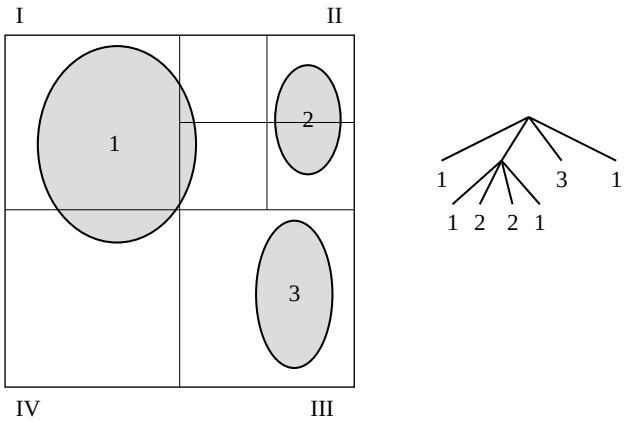


Figure 14.33. A quadtree partitioning the plane, whose three-dimensional version is the octree. The tree is constructed by halving the cells along all coordinate axes until a cell contains „just a few" objects, or the cell sizes gets smaller than a threshold. Objects are registered in the leaves of the tree.

- The recursive tree building procedure terminates if the depth of the tree becomes to big, or when the number of objects overlapping with a cell is smaller than the threshold.

The result of this construction is an ***octree*** (figure 14.33). Overlapping objects are registered in the leaves of this tree.

When a ray is traced, those leaves of the tree should be traversed which are intersected by the ray, and ray–surface intersection test should be executed for objects registered in these leaves:

RAY-FIRST-INTERSECTION-WITH-OCTREE($\vec{s}, \vec{v}$)

```
1  q⃗ ← intersection of the ray and the scene AABB
2  while q⃗ is inside of the scene AABB                    ▷ Traversal of the tree.
3        cell ← OCTREE-CELL-SEARCH(octree root, q⃗)
4        t_out ← ray parameter of the intersection of the cell and the ray
5        t ← t_out                          ▷ Initialization: no ray-surface intersection yet.
6        for each object o registered in cell
7             do t_o ←RAY-SURFACE-INTERSECTION(s⃗, v⃗)   ▷ Negative if no intersection exists.
8                if 0 ≤ t_o < t                          ▷ Is the new intersection closer?
9                   then t ← t_o                ▷ Ray parameter of the closest intersection so far.
10                       o_visible ← o                  ▷ First intersected object so far.
11       if t < t_out                                   ▷ Has been intersection at all ?
12          then x⃗ ← s⃗ + v⃗ · t                          ▷ Position of the intersection.
13               return t, x⃗, o_visible
14       q⃗ ← s⃗ + v⃗ · (t_out + ε)                       ▷ A point in the next cell.
15 return „no intersection"
```

The identification of the next cell intersected by the ray is more complicated for octrees than for uniform grids. The OCTREE-CELL-SEARCH algorithm determines that leaf cell which

contains a given point. At each level of the tree, the coordinates of the point are compared to the coordinates of the centre of the cell. The results of these comparisons determine which child contains the point. Repeating this test recursively, we arrive at a leaf sooner or later.

In order to identify the next cell intersected by the ray, the intersection point of the ray and the current cell is computed. Then, ray parameter t_{out} of this intersection point is increased „a little" (this little value is denoted by ε in algorithm RAY-FIRST-INTERSECTION-WITH-OCTREE). The increased ray parameter is substituted into the ray equation, resulting in point $\vec{q}$ that is already in the next cell. The cell containing this point can be identified with OCTREE-CELL-SEARCH.

Cells of the octree may be larger than the allowed minimal cell, therefore the octree algorithm requires less number of cell steps than the uniform grid algorithm working on the minimal cells. However, larger cells reduce the probability of the successful intersection tests since in a large cell it is less likely that a random ray intersecting the cell also intersects a contained object. Smaller successful intersection probability, on the other hand, results in greater expected number of intersection tests, which affects the performance negatively. It also means that non-empty octree cells are worth subdividing until the minimum cell size is reached even if the cell contains just a single object. Following this strategy, the size of the non-empty cells are similar, thus the results of the complexity analysis made for the uniform grid remain to be applicable to the octree as well. Since the probability of the successful intersection depends on the size of the non-empty cells, the expected number of needed intersection tests is still given by inequality (14.28). It also means that when the minimal cell size of an octree equals to the cell size of a uniform grid, then the expected number of intersection tests is equal in the two algorithms.

The advantage of the ocree is the ability to skip empty spaces, which reduces the number of cell steps. Its disadvantage is, however, that the time of the next cell identification is not constant. This identification requires the traversal of the tree. If the tree construction is terminated when a cell contains small number of objects, then the number of leaf cells is proportional to the number of objects. The depth of the tree is in $O(\lg N)$, so is the time needed to step onto the next cell.

kd-tree

An octree adapts to the distribution of the objects. However, the partitioning strategy of octrees always halves the cells without taking into account where the objects are, thus the adaptation is not perfect. Let us consider a partitioning scheme which splits a cell into two cells to make the tree balanced. Such method builds a binary tree which is called ***binary space partitioning tree,***, abbreviated as ***BSP-tree***. If the separating plane is always perpendicular to one of the coordinate axes, then the tree is called ***kd-tree***.

The separating plane of a kd-tree node can be placed in many different ways:

- the ***spatial median method*** subdivides the cell into two similar cells.
- the ***object median method*** finds the separation plane to have the same number of objects in the two child cells.
- the ***cost driven method*** estimates the average computation time needed when a cell is processed during ray tracing, and minimizes this value by placing the separation plane. An appropriate cost model suggests to separate the cell to make the probabilities of the ray–surface intersection of the two cells similar.

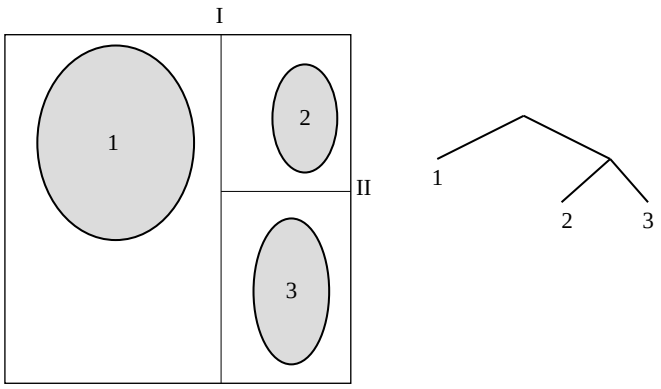


Figure 14.34. A kd-tree. A cell containing „many" objects are recursively subdivided to two cells with a plane that is perpendicular to one of the coordinate axes.

The probability of the ray-surface intersection can be computed using a fundamental theorem of the ***integral geometry***:

Theorem 14.17 *If convex solid A contains another convex solid B, then the probability that a uniformly distributed line intersects solid B provided that the line intersected A equals to the ratio of the surface areas of objects B and A.*

According to this theorem the cost driven method finds the separation plane to equalize the surface areas in the two children.

Let us now present a general kd-tree construction algorithm. Parameter *cell* identifies the current cell, *depth* is the current depth of recursion, and *coordinate* stores the orientation of the current separating plane. A *cell* is associated with its two children (*cell.right* and *cell.left*), and its left-lower-closer and right-upper-farther corners (*cell.min* and *cell.max*). Cells also store the list of those objects which overlap with the cell. The orientation of the separation plane is determined by a round-robin scheme implemented by function ROUND-ROBIN providing a sequence like $(x, y, z, x, y, z, x, \ldots)$. When the following recursive algorithm is called first, it gets the scene AABB in variable *cell* and the value of variable *depth* is zero:

KD-TREE-CONSTRUCTION(*cell*, *depth*, *coordinate*)

```
 1  if the number of objects overlapping with cell is small or depth is large
 2    then return
 3  AABB of cell.left and AABB of cell.right ← AABB of cell
 4  if coordinate = x
 5    then cell.right.min.x ← x perpendicular separating plane of cell
 6         cell.left.max.x ← x perpendicular separating plane of cell
 7    else if coordinate = y
 8           then cell.right.min.y ← y perpendicular separating plane of cell
 9                cell.left.max.y ← y perpendicular separating plane of cell
10    else if coordinate = z
11           then cell.right.min.z ← z perpendicular separating plane of cell
12                cell.left.max.z ← z perpendicular separating plane of cell
13  for each object o of cell
14      do if object o is in the AABB of cell.left
15           then assign object o to the list of cell.left
16         if object o is in the AABB of cell.right
17           then assign object o to the list of cell.right
18  KD-TREE-CONSTRUCTION(cell.left, depth + 1, ROUND-ROBIN(coordinate))
19  KD-TREE-CONSTRUCTION(cell.right, depth + 1, ROUND-ROBIN(coordinate))
```

Now we discuss an algorithm that traverses the constructed kd-tree and finds the visible object. First we have to test whether the origin of the ray is inside the scene AABB. If it is not, the intersection of the ray and the scene AABB is computed, and the origin of the ray is moved there. The identification of the cell containing the ray origin requires the traversal of the tree. During the traversal the coordinates of the point are compared to the coordinates of the separating plane. This comparison determines which child should be processed recursively until a leaf node is reached. If the leaf cell is not empty, then objects overlapping with the cell are intersected with the ray, and the intersection closest the origin is retained. The closest intersection is tested to see whether or not it is inside the cell (since an object may overlap in more than one cells, it can also happen that the intersection is in another cell). If the intersection is in the current cell, then the needed intersection has been found, and the algorithm can be terminated. If the cell is empty, or no intersection is found in the cell, then the algorithm should proceed with the next cell. To identify the next cell, the ray is intersected with the current cell identifying the ray parameter of the exit point. Then the ray parameter is increased „a little” to make sure that the increased ray parameter corresponds to a point in the next cell. The algorithm keeps repeating these steps as it process the cells of the tree.

This method has the disadvantage that the cell search always starts at the root, which results in the repetitive traversals of the same nodes of the tree.

This disadvantage can be eliminated by putting the cells to be visited into a stack, and backtracking only to the point where a new branch should be followed. When the ray arrives at a node having two children, the algorithm decides the order of processing the two child nodes. Child nodes are classified as „near” and „far” depending on whether or not the child cell is on the same side of the separating plane as the origin of the ray. If the ray intersects

only the „near" child, then the algorithm processes only that subtree which originates at this child. If the ray intersects both children, then the algorithm pushes the „far" node onto the stack and starts processing the „near" node. If no intersection exists in the „near" node, then the stack is popped to obtain the next node to be processed.

The notations of the ray tracing algorithm based on kd-tree traversal are shown by figure 14.35. The algorithm is the following:

RAY-FIRST-INTERSECTION-WITH-KD-TREE($root$, $\vec{s}$, $\vec{v}$)

```
1  (t_in, t_out) ← RAY-AABB-INTERSECTION(s, v, root)   ▷ Intersection with the scene AABB.
2  if no intersection
3     then return „no intersection"
4  PUSH(root, t_in, t_out)
5  while the stack is not empty                                   ▷ Visit all nodes.
6        do POP(cell, t_in, t_out)
7           while cell is not a leaf
8                 do coordinate ← orientation of the separating plane of the cell
9                    d ← cell.right.min[coordinate] − s[coordinate]
10                   t ← d/v[coordinate]          ▷ Ray parameter of the separating plane.
11                   if d > 0                  ▷ Is s on the left side of the separating plane?
12                      then (near,far) ← (cell.left, cell.right)                  ▷ Left.
13                      else  (near,far) ← (cell.right, cell.left)                 ▷ Right.
14                   if t > t_out or t < 0
15                      then cell ← near              ▷ The ray intersects only the near cell.
16                      else if t < t_in
17                              then cell ← far       ▷ The ray intersects only the far cell.
18                              else PUSH(far, t, t_out)          ▷ The ray intersects both cells.
19                                   cell ← near                  ▷ First near is intersected.
20                                   t_out ← t          ▷ The ray exists at t from the near cell.
              ▷ If the current cell is a leaf.
21          t ← t_out                                    ▷ Maximum ray parameter in this cell.
22          for each object o of cell
23              do t_o ←RAY-SURFACE-INTERSECTION(s, v) ▷ Negative if no intersection exists.
24                 if t_in ≤ t_o < t          ▷ Is the new intersection closer to the ray origin?
25                 then t ← t_o       ▷ The ray parameter of the closest intersection so far.
26                      o_visible ← o       ▷ The object intersected closest to the ray origin.
27          if t < t_out                                ▷ Has been intersection at all in the cell?
28             then x ← s + v · t                                    ▷ The intersection point.
29                  return t, x, o_visible                          ▷ Intersection has been found.
30 return „no intersection"                                        ▷ No intersection.
```

Similarly to the octree algorithm, the likelihood of successful intersections can be increased by continuing the tree building process until all empty spaces are cut (figure 14.36).

Our probabilistic world model contains spheres of same radius r, thus the non-empty cells are cubes of edge size $c = 2r$. Unlike in uniform grids or octrees, the separating planes of kd-trees are not independent of the objects. Kd-tree splitting planes are rather tangents

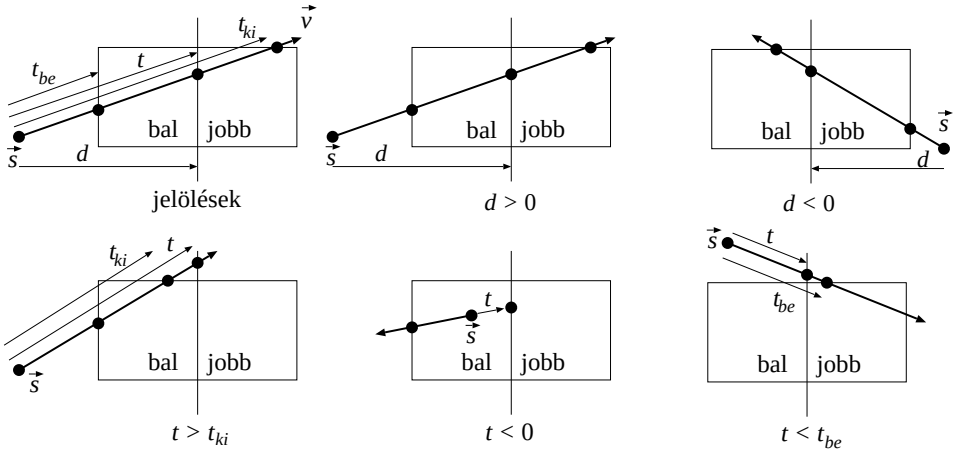


Figure 14.35. Notations and cases of algorithm RAY-FIRST-INTERSECTION-WITH-KD-TREE. t_{in}, t_{out}, and t are the ray parameters of the entry, exit and the separation plane, respectively. d is the signed distance between the ray origin and the separation plane.

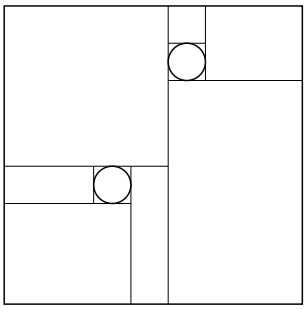


Figure 14.36. Kd-tree based space partitioning with empty space cutting.

of the objects. This means that we do not have to be concerned with partially overlapping spheres since a sphere is completely contained by a cell in a kd-tree. The probability of the successful intersection is obtained applying theorem 14.17. In the current case, the containing convex solid is a cube of edge size $2r$, the contained solid is a sphere of radius r, thus the intersection probability is:

$$s = \frac{4r^2\pi}{6a^2} = \frac{\pi}{6} .$$

The expected number of intersection tests is then:

$$\mathrm{E}[N_I] = \frac{6}{\pi} \approx 1.91 .$$

We can conclude that the kd-tree algorithm requires the smallest number of ray–surface intersection tests according to the probabilistic model.

Exercises

14.6-1 Prove that the expected number of intersection tests is constant in all those ray tracing algorithms which process objects in the order of their distance from the ray origin.
14.6-2 Propose a ray intersection algorithm for subdivision surfaces.
14.6-3 Develop a ray intersection method for B-spline surfaces.
14.6-4 Develop a ray intersection algorithm for CSG models assuming that the ray–primitive intersection tests are already available.
14.6-5 Propose a ray intersection algorithm for transformed objects assuming that the algorithm computing the intersection with the non-transformed objects is available (hints: transform the ray).

14.7. Incremental rendering

Rendering requires the identification of those surface points that are visible through the pixels of the virtual camera. Ray tracing solves this visibility problem for each pixel independently, thus it does not reuse visibility information gathered at other pixels. The algorithms of this section, however, exploit such information using the following simple techniques:

1. They simultaneously attack the visibility problem for all pixels, and handle larger parts of the scene at once.
2. Where feasible, they exploit the ***incremental concept*** which is based on the recognition that the visibility problem becomes simpler to solve if the solution at the neighbouring pixel is taken into account.
3. They solve each task in that coordinate system which makes the solution easier. The scene is transformed from one coordinate system to the other by homogeneous linear transformations.
4. They minimize unnecessary computations, therefore remove those objects by *clipping* in an early stage of rendering which cannot be projected onto the window of the camera.

Homogeneous linear transformations and clipping may change the type of the surface except for points, line segments and polygons [4]. Therefore, before rendering is started, each shape is approximated by points, line segments, and meshes (subsection 14.3).

Steps of incremental rendering are shown in figure 14.37. Objects are defined in their reference state, approximated by meshes, and are transformed to the virtual world. The time dependence of this transformation is responsible for object animation. The image is taken from the camera taken about the virtual world, which requires the identification of those surface points that are visible from the camera, and their projection onto the window plane. The visibility and projection problems could be solved in the virtual world as happens in ray tracing, but this would require the intersection calculations of general lines and polygons. Visibility and projection algorithms can be simplified if the scene is transformed to a coordinate system, where the X, Y coordinates of a point equal to the coordinates of that pixel onto which this point is projected, and the Z coordinate can be used to decide which point is closer if more than one surfaces are projected onto the same pixel. Such coordinate system is called the ***screen coordinate system***. In screen coordinates the units of axes X and

[4] Although Bézier and B-Spline curves and surfaces are invariant to affine transformations, and NURBS is invariant even to homogeneous linear transformations, but clipping changes these object types as well.

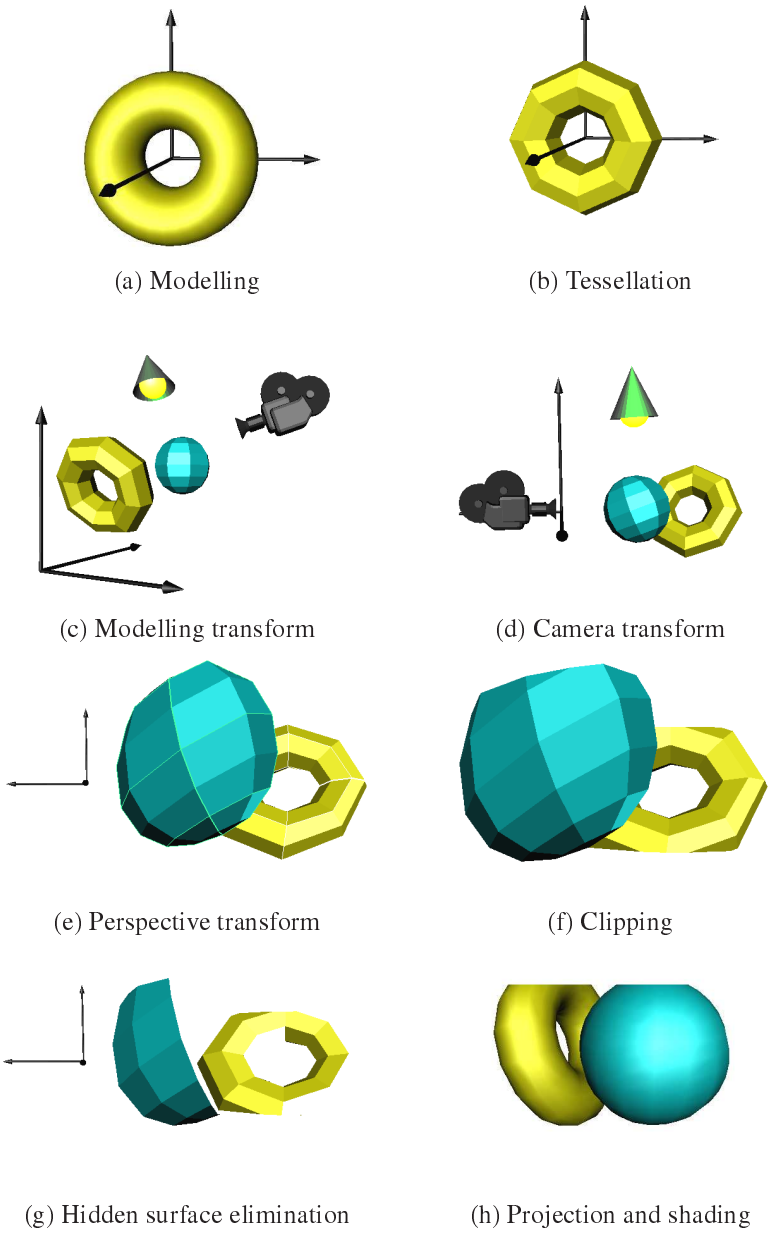


Figure 14.37. Steps of incremental rendering. **(a)** Modelling defines objects in their reference state. **(b)** Shapes are tessellated to prepare for further processing. **(c)** Modelling transform places the object in the world coordinate system. **(d)** Camera transform translates and rotates the scene to get the eye to be at the origin and to look parallel with axis $-z$. **(e)** Perspective transform converts projection lines meeting at the origin to parallel lines, that is, it maps the eye position onto an ideal point. **(f)** Clipping removes those shapes and shape parts, which cannot be projected onto the window. **(g)** Hidden surface elimination removes those surface parts that are occluded by other shapes. **(h)** Finally, the visible polygons are projected and their projections are filled with their visible colours.

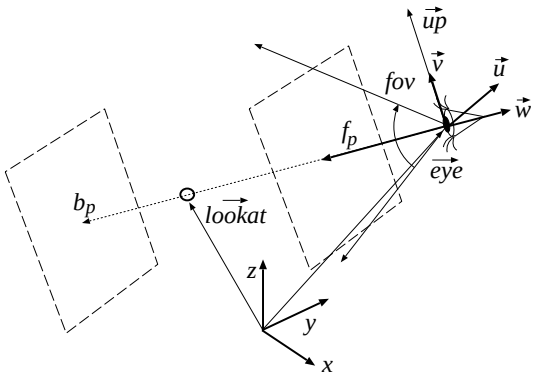


Figure 14.38. Parameters of the virtual camera: eye position $\vec{eye}$, target $\vec{lookat}$, and vertical direction $\vec{up}$, from which camera basis vectors $\vec{u}, \vec{v}, \vec{w}$ are obtained, front f_p and back b_p clipping planes, and vertical field of view *fov* (the horizontal field of view is computed from aspect ratio *aspect*).

Y are equal to the pixel size. Since it is usually not worth computing the image on higher accuracy than the pixel size, coordinates X, Y are integers. Because of performance reasons, coordinate Z is also often integer. Screen coordinates are denoted by capital letters.

The transformation taking to the screen coordinate system is defined by a sequence of transformations, and the elements of this sequence are discussed separately. However, this transformation is executed as a single multiplication with a 4×4 transformation matrix obtained as the product of elementary transformation matrices.

14.7.1. Camera transform

Rendering is expected to generate an image from a camera defined by ***eye position*** ($\vec{eye}$) (the focal point of the camera), looking target ($\vec{lookat}$) where the camera looks at, and by vertical direction $\vec{up}$ (figure 14.38).

Camera parameter *fov* defines the vertical field of view, *aspect* is the ratio of the width and the height of the window, f_p and b_p are the distances of the front and back clipping planes from the eye, respectively. These clipping planes allow to remove those objects that are behind, too close to, or too far from the eye.

We assign a coordinate system, i.e. three orthogonal unit basis vectors to the camera. Horizontal basis vector $\vec{u} = (u_x, u_y, u_z)$, vertical basis vector $\vec{v} = (v_x, v_y, v_z)$, and basis vector $\vec{w} = (w_x, w_y, w_z)$ pointing to the looking direction are obtained as follows:

$$\vec{w} = \frac{\vec{eye} - \vec{lookat}}{|\vec{eye} - \vec{lookat}|}, \qquad \vec{u} = \frac{\vec{up} \times \vec{w}}{|\vec{up} \times \vec{w}|}, \qquad \vec{v} = \vec{w} \times \vec{u} \; .$$

The ***camera transform*** translates and rotates the space of the virtual world in order to get the camera to move to the origin, to look at direction axis $-z$, and to have vertical direction parallel to axis y, that is, this transformation maps unit vectors $\vec{u}, \vec{v}, \vec{w}$ to the basis vectors of the coordinate system. Transformation matrix $\mathbf{T}_{camera}$ can be expressed as the product of a matrix translating the eye to the origin and a matrix rotating basis vectors

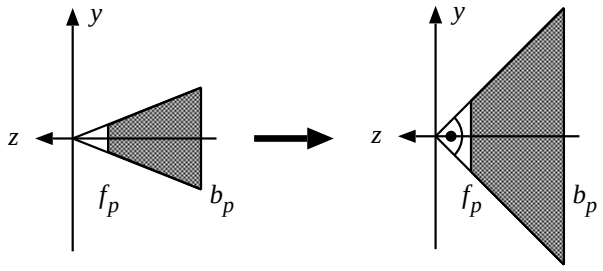


Figure 14.39. The normalizing transform sets the field of view to 90 degrees.

$\vec{u}, \vec{v}, \vec{w}$ of the camera to the basis vectors of the coordinate system:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \mathbf{T}_{camera} = [x, y, z, 1] \cdot \mathbf{T}_{translation} \cdot \mathbf{T}_{rotation} , \tag{14.31}$$

where

$$\mathbf{T}_{translation} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -eye_x & -eye_y & -eye_z & 1 \end{bmatrix}, \quad \mathbf{T}_{rotation} = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

Let us note that the columns of the rotation matrix are vectors $\vec{u}, \vec{v}, \vec{w}$. Since these vectors are orthogonal, it is easy to see that this rotation maps them to coordinate axes x, y, z. For example, the rotation of vector $\vec{u}$ is:

$$[u_x, u_y, u_z, 1] \cdot \mathbf{T}_{rotation} = [\vec{u} \cdot \vec{u}, \vec{u} \cdot \vec{v}, \vec{u} \cdot \vec{w}, 1] = [1, 0, 0, 1] .$$

14.7.2. Normalizing transform

In the next step the viewing pyramid containing those points which can be projected onto the window is normalized making the field of view equal to 90 degrees (figure 14.39).

Normalization is a simple scaling transform:

$$\mathbf{T}_{norm} = \begin{bmatrix} 1/(\tan(fov/2) \cdot aspect) & 0 & 0 & 0 \\ 0 & 1/\tan(fov/2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

14.7.3. Perspective transform

The perspective transform distorts the virtual world to allow the replacement of the perspective projection by parallel projection during rendering.

After the normalizing transform, the points potentially participating in rendering are inside a symmetrical finite frustum of pyramid (figure 14.39). The perspective transform maps this frustum onto a cube, converting projection lines crossing the origin to lines parallel to axis z (figure 14.40).

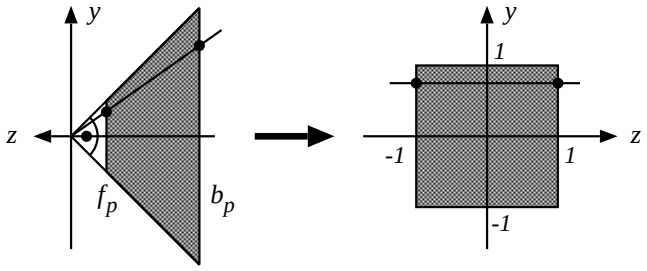


Figure 14.40. The perspective transform maps the finite frustum of pyramid defined by the front and back clipping planes, and the edges of the window onto an axis aligned, origin centred cube of edge size 2.

Perspective transform is expected to map point to point, line to line, but to map the eye position to infinity. It means that perspective transform cannot be a linear transform of Cartesian coordinates. Fortunately, homogenous linear transforms also map point to point, line to line, and are able to handle points at infinity with finite coordinates. Let us thus try to find the perspective transform in the form of a homogeneous linear transform defined by a 4×4 matrix:

$$\mathbf{T}_{persp} = \begin{bmatrix} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \\ t_{41} & t_{42} & t_{43} & t_{44} \end{bmatrix} .$$

Figure 14.40 shows a line (projection ray) and its transform. Let m_x and m_y be the x/z and the y/z slopes of the line, respectively. This line is defined by equation $[-m_x \cdot z, -m_y \cdot z, z]$ in the normalized camera space. The perspective transform maps this line to a „horizontal" line crossing point $[m_x, m_y, 0]$ and being parallel to axis z. Let us examine the intersection points of this line with the front and back clipping planes, that is, let us substitute $(-f_p)$ and $(-b_p)$ into parameter z of the line equation. The transformation should map these points to $[m_x, m_y, -1]$ and $[m_x, m_y, 1]$, respectively.

The perspective transformation of the point on the first clipping plane is:

$$\left[m_x \cdot f_p, m_y \cdot f_p, -f_p, 1\right] \cdot \mathbf{T}_{persp} = \left[m_x, m_y, -1, 1\right] \cdot \lambda \ ,$$

where λ is an arbitrary, non-zero scalar since the point defined by homogeneous coordinates does not change if the homogenous coordinates are simultaneously multiplied by a non-zero scalar. Setting λ to f_p, we get:

$$\left[m_x \cdot f_p, m_y \cdot f_p, -f_p, 1\right] \cdot \mathbf{T}_{persp} = \left[m_x \cdot f_p, m_y \cdot f_p, -f_p, f_p\right] \ . \tag{14.32}$$

Note that the first coordinate of the transformed point equals to the first coordinate of the original point on the clipping plane for arbitrary m_x, m_y, and f_p values. This is possible only if the first column of matrix $\mathbf{T}_{persp}$ is $[1, 0, 0, 0]^T$. Using the same argument for the second coordinate, we can conclude that the second column of the matrix is $[0, 1, 0, 0]^T$. Furthermore, in equation (14.32) the third and the fourth homogeneous coordinates of the transformed point are not affected by the first and the second coordinates of the original point, requiring $t_{13} = t_{14} = t_{23} = t_{24} = 0$. The conditions on the third and the fourth

homogeneous coordinates can be formalized by the following equations:

$$-f_p \cdot t_{33} + t_{43} = -f_p, \qquad -f_p \cdot t_{34} + t_{44} = f_p \ .$$

Applying the same procedure for the intersection point of the projection line and the back clipping plane, we can obtain other two equations:

$$-b_p \cdot t_{33} + t_{43} = b_p, \qquad -b_p \cdot t_{34} + t_{44} = b_p \ .$$

Solving this system of linear equations, the matrix of the perspective transform can be expressed as:

$$\mathbf{T}_{persp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -(f_p + b_p)/(b_p - f_p) & -1 \\ 0 & 0 & -2 \cdot f_p \cdot b_p/(b_p - f_p) & 0 \end{bmatrix} .$$

Since perspective transform is not affine, the fourth homogeneous coordinate of the transformed point is usually not 1. If we wish to express the coordinates of the transformed point in Cartesian coordinates, the first three homogeneous coordinates should be divided by the fourth coordinate. Homogeneous linear transforms map line segment to line segment and triangle to triangle, but it may happen that the resulting line segment or triangle contains ideal points (subsection 14.5.2). The intuition behind the homogeneous division is a traveling from the projective space to the Euclidean space, which converts a line segment containing an ideal point to two half lines. If just the two endpoints of the line segment is transformed, then it is not unambiguous whether the two transformed points need to be connected by a line segment or the complement of this line segment should be considered as the result of the transformation. This ambiguity is called the ***wrap around problem***.

The wrap around problem does not occur if we can somehow make sure that the original shape does not contain points that might be mapped onto ideal points. Examining the matrix of the perspective transform we can conclude that the fourth homogeneous coordinate of the transformed point will be equal to the $-z$ coordinate of the original point. Ideal points having zero fourth homogeneous coordinate ($h = 0$) may thus be obtained transforming the points of plane $z = 0$, i.e. the plane crossing the origin and parallel to the window. However, if the shapes are clipped onto a first clipping plane being in front of the eye, then these points are removed. Thus the solution of the wrap around problem is the execution of the clipping step before the homogeneous division.

14.7.4. Clipping in homogeneous coordinates

The purpose of ***clipping*** is to remove all shapes that either cannot be projected onto the window or are not between the front and back clipping planes. To solve the ***wrap around problem***, clipping should be executed before the homogeneous division. The clipping boundaries in homogeneous coordinates can be obtained by transforming the screen coordinate AABB back to homogeneous coordinates. In screen coordinates, i.e. after homogeneous division, the points to be preserved by clipping meet the following inequalities:

$$-1 \le X = X_h/h \le 1, \qquad -1 \le Y = Y_h/h \le 1, \qquad -1 \le Z = Z_h/h \le 1 \ . \tag{14.33}$$

On the other hand, points that are in front of the eye after camera transform have negative z coordinates, and the perspective transform makes the fourth homogeneous coordinate

h equal to $-z$ in normalized camera space. Thus the fourth homogeneous coordinate of points in front of the eye is always positive. Let us thus add condition $h > 0$ to the set of conditions of inequalities (14.33). If h is positive, then inequalities (14.33) can be multiplied by h, resulting in the definition of the clipping region in homogeneous coordinates:

$$-h \le X_h \le h, \qquad -h \le Y_h \le h, \qquad -h \le Z_h \le h\ . \tag{14.34}$$

Points can be clipped easily, since we should only test whether or not the conditions of inequalities (14.34) are met. Clipping line segments and polygons, on the other hand, requires the computation of the intersection points with the faces of the clipping boundary, and only those parts should be preserved which meet inequalities (14.34).

Clipping algorithms using Cartesian coordinates were discussed in subsection 14.4.3. Those methods can also be applied in homogeneous coordinates with two exceptions. Firstly, for homogeneous coordinates, inequalities (14.34) define whether a point is in or out. Secondly, intersections should be computed using the homogeneous coordinate equations of the line segments and the planes.

Let us consider a line segment with endpoints $[X_h^1, Y_h^1, Z_h^1, h^1]$ and $[X_h^2, Y_h^2, Z_h^2, h^2]$. This line segment can be an independent shape or an edge of a polygon. Here we discuss the clipping on half space of equation $X_h \le h$ (clipping methods on other half spaces are very similar). Three cases need to be distinguished:

1. If both endpoints of the line segment are inside, that is $X_h^1 \le h^1$ and $X_h^2 \le h^2$, then the complete line segment is in, thus is preserved.
2. If both endpoints are outside, that is $X_h^1 > h^1$ and $X_h^2 > h^2$, then all points of the line segment are out, thus it is completely eliminated by clipping.
3. If one endpoint is outside, while the other is in, then the intersection of the line segment and the clipping plane should be obtained. Then the endpoint being out is replaced by the intersection point. Since the points of a line segment satisfy equation (14.19), while the points of the clipping plane satisfy equation $X_h = h$, parameter t_i of the intersection point is computed as:

$$X_h(t_i) = h(t_i) \implies X_h^1 \cdot (1-t_i) + X_h^2 \cdot t_i = h^1 \cdot (1-t_i) + h^2 \cdot t_i \implies t_i = \frac{X_h^1 - h^1}{X_h^1 - X_h^2 + h^2 - h^1}\ .$$

Substituting parameter t_i into the equation of the line segment, homogeneous coordinates $[X_h^i, Y_h^i, Z_h^i, h^i]$ of the intersection point are obtained.

Clipping may introduce new vertices. When the vertices have some additional features, for example, the surface colour or normal vector at these vertices, then these additional features should be calculated for the new vertices as well. We can use linear interpolation. If the values of a feature at the two endpoints are I^1 and I^2, then the feature value at new vertex $[X_h(t_i), Y_h(t_i), Z_h(t_i), h(t_i)]$ generated by clipping is $I^1 \cdot (1 - t_i) + I^2 \cdot t_i$.

14.7.5. Viewport transform

Having executed the perspective transform, the Cartesian coordinates of the visible points are in $[-1, 1]$. These normalized device coordinates should be further scaled and translated according to the resolution of the screen and the position of the viewport where the image is

expected. Denoting the left-bottom corner pixel of the screen viewport by (X_{min}, Y_{min}), the right-top corner by (X_{max}, Y_{max}), and Z coordinates expressing the distance from the eye are expected in (Z_{min}, Z_{max}), the matrix of the viewport transform is:

$$\mathbf{T}_{viewport} = \begin{bmatrix} (X_{max} - X_{min})/2 & 0 & 0 & 0 \\ 0 & (Y_{max} - Y_{min})/2 & 0 & 0 \\ 0 & 0 & (Z_{max} - Z_{min})/2 & 0 \\ (X_{max} + X_{min})/2 & (Y_{max} + Y_{min})/2 & (Z_{max} + Z_{min})/2 & 1 \end{bmatrix}.$$

Coordinate systems after the perspective transform are ***left handed***, unlike the coordinate systems of the virtual world and the camera, which are ***right handed***. Left handed coordinate systems seem to be unusual, but they meet our natural expectation that the screen X coordinates grow from left to right, the Y coordinates from bottom to top and, the Z coordinates grow with the distance from the virtual observer.

14.7.6. Rasterization algorithms

After clipping, homogeneous division, and viewport transform, shapes are in the screen coordinate system where a point of coordinates (X, Y, Z) can be assigned to a pixel by extracting the first two Cartesian coordinates (X, Y).

Rasterization works in the screen coordinate system and identifies those pixels which have to be coloured to approximate the projected shape. Since even simple shapes can cover many pixels, rasterization algorithms should be very fast, and should be appropriate for hardware implementation.

Line drawing

Let the endpoints of a line segment be (X_1, Y_1) and (X_2, Y_2) in screen coordinates. Let us further assume that while we are going from the first endpoint toward the second, both coordinates are growing, and X is the faster changing coordinate, that is,

$$\Delta X = X_2 - X_1 \geq \Delta Y = Y_2 - Y_1 \geq 0\,.$$

In this case the line segment is moderately ascending. We discuss only this case, other cases can be handled by exchanging the X, Y coordinates and replacing additions by substractions.

Line drawing algorithms are expected to find pixels that approximate a line in a way that there are no holes and the approximation is not fatter than necessary. In case of moderately ascending line segments this means that in each pixel column exactly one pixel should be filled with the colour of the line. This coloured pixel is the one closest to the line in this column. Using the following equation of the line

$$y = m \cdot X + b, \qquad \text{where} \quad m = \frac{Y_2 - Y_1}{X_2 - X_1}, \quad \text{and} \quad b = Y_1 - X_1 \cdot \frac{Y_2 - Y_1}{X_2 - X_1}\,, \tag{14.35}$$

in pixel column of coordinate X the pixel closest to the line has Y coordinate that is equal to the rounding of $m \cdot x + b$. Unfortunately, the determination of Y requires a floating point multiplication, addition, and a rounding operation, which are too slow.

In order to speed up line drawing, we apply a fundamental trick of computer graphics, the ***incremental concept***. The incremental concept is based on the recognition that it is

usually simpler to evaluate a function $y(X+1)$ using value $y(X)$ than computing it from X. Since during line drawing the columns are visited one by one, when column $(X+1)$ is processed, value $y(X)$ is already available. In case of a line segment we can write:

$$y(X+1) = m \cdot (X+1) + b = m \cdot X + b + m = y(X) + m \ .$$

Note that the evaluation of this formula requires just a single floating point addition (m is less than 1). This fact is exploited in ***digital differential analyzator algorithms*** (DDA-algorithms). The ***DDA line drawing algorithm*** is then:

DDA-Line-Drawing$(X_1, Y_1, X_2, Y_2, colour)$

```
1  m ← (Y2 − Y1)/(X2 − X1)
2  y ← Y1
3  for X ← X1 to X2
4      do Y ← Round(y)
5         Pixel-Write(X, Y, colour)
6         y ← y + m
```

Further speedups can be obtained using ***fixed point number representation***. This means that the product of the number and 2^T is stored in an integer variable, where T is the number of fractional bits. The number of fractional bits should be set to exclude cases when the rounding errors accumulate to an incorrect result during long iteration sequences. If the longest line segment covers L columns, then the minimum number of fractional bits guaranteeing that the accumulated error is less than 1 is $\log_2 L$. Thanks to clipping only lines fitting to the screen are rasterized, thus L is equal to the maximum screen resolution.

The performance and simplicity of the DDA line drawing algorithm can still be improved. On the one hand, the software implementation of the DDA algorithm requires shift operations to realize truncation and rounding operations. On the other hand—once for every line segment—the computation of slope m involves a division which is computationally expensive. Both problems are solved in the ***Bresenham line drawing algorithm.***

Let us denote the vertical, signed distance of the line segment and the closest pixel centre by s, and the vertical distance of the line segment and the pixel centre just above the closest pixel by t (figure 14.41). As the algorithm steps onto the next pixel column, values s and t change and should be recomputed. While the new s and t values satisfy inequality $s < t$, that is, while the lower pixel is still closer to the line segment, the shaded pixel of the next column is in the same row as in the previous column. Introducing error variable $e = s - t$, the row of the shaded pixel remains the same until this error variable is negative ($e < 0$). As the pixel column is incremented, variables s, t, e are updated using the incremental formulae ($\Delta X = X_2 - X_1$, $\Delta Y = Y_2 - Y_1$):

$$s(X+1) = s(X) + \frac{\Delta Y}{\Delta X}, \quad t(X+1) = t(X) - \frac{\Delta Y}{\Delta X} \quad \Longrightarrow \quad e(X+1) = e(X) + 2\frac{\Delta Y}{\Delta X} \ .$$

These formulae are valid if the closest pixel in column $(X+1)$ is in the same row as in column X. If stepping to the next column, the upper pixel gets closer to the line segment (error variable e becomes positive), then variables s, t, e should be recomputed for the new

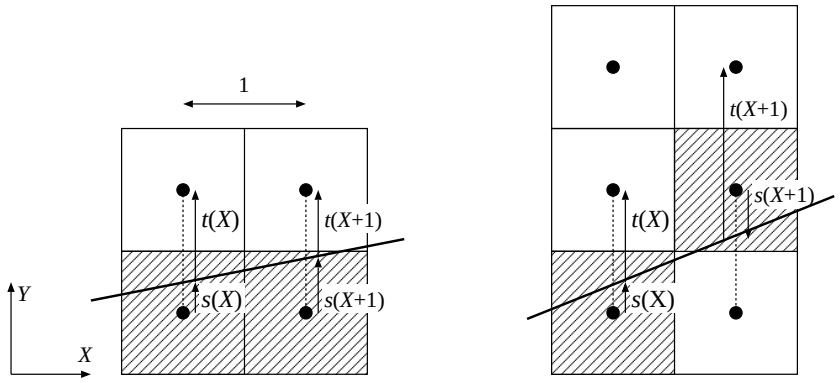


Figure 14.41. Notations of the Bresenham algorithm: s is the signed distance between the closest pixel centre and the line segment along axis Y, which is positive if the line segment is above the pixel centre. t is the distance along axis Y between the pixel centre just above the closest pixel and the line segment.

closest row and for the pixel just above it. The formulae describing this case are as follows:

$$s(X+1) = s(X) + \frac{\Delta Y}{\Delta X} - 1, \quad t(X+1) = t(X) - \frac{\Delta Y}{\Delta X} + 1 \Longrightarrow e(X+1) = e(X) + 2\left(\frac{\Delta Y}{\Delta X} - 1\right) .$$

Note that s is a signed distance which is negative if the line segment is below the closest pixel centre, and positive otherwise. We can assume that the line starts at a pixel centre, thus the initial values of the control variables are:

$$s(X_1) = 0, \qquad t(X_1) = 1 \implies e(X_1) = s(X_1) - t(X_1) = -1 .$$

This algorithm keeps updating error variable e and steps onto the next pixel row when the error variable becomes positive. In this case, the error variable is decreased to have a negative value again. The update of the error variable requires a non-integer addition and the computation of its increment involves a division, similarly to the DDA algorithm. It seems that this approach is not better than the DDA.

Let us note, however, that the sign changes of the error variable can also be recognized if we examine the product of the error variable and a positive number. Multiplying the error variable by ΔX we obtain ***decision variable*** $E = e \cdot \Delta X$. In case of moderately ascending lines the decision and error variables change their sign simultaneously. The incremental update formulae of the decision variable can be obtained by multiplying the update formulae of error variable by ΔX:

$$E(X+1) = \begin{cases} E(X) + 2\Delta Y, \text{ if } Y \text{ is not incremented} , \\ \\ E(X) + 2(\Delta Y - \Delta X), \text{ if } Y \text{ needs to be incremented} . \end{cases}$$

The initial value of the decision variable is $E(X_1) = e(X_1) \cdot \Delta X = -\Delta X$.

The decision variable starts at an integer value and is incremented by integers in each step, thus it remains to be an integer and does not require fractional numbers at all. The computation of the increments need only integer additions or subtractions and multiplications

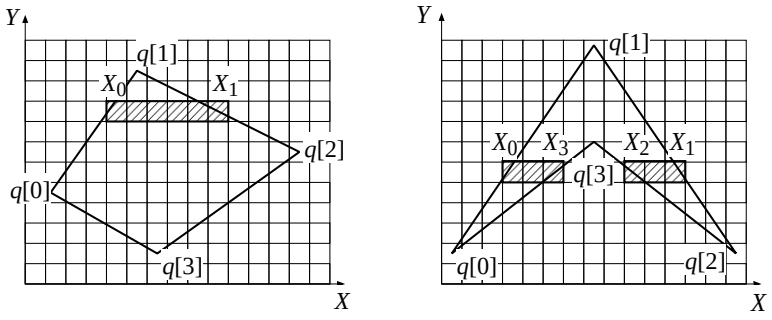


Figure 14.42. Polygon fill. Pixels inside the polygon are identified scan line by scan line.

by 2.

The complete ***Bresenham line drawing algorithm*** is:

BRESENHAM-LINE-DRAWING($X_1, Y_1, X_2, Y_2, colour$)

```
 1  ΔX ← X2 − X1
 2  ΔY ← Y2 − Y1
 3  (dE+, dE−) ← (2(ΔY − ΔX), 2ΔY)
 4  E ← −ΔX
 5  Y ← Y1
 6  for X ← X1 to X2
 7      do if E ≤ 0
 8           then E ← E + dE−          ▷ The line stays in the current pixel row.
 9           else E ← E + dE+          ▷ The line steps onto the next pixel row.
10                Y ← Y + 1
11         PIXEL-WRITE(X, Y, colour)
```

The fundamental idea of the Bresenham algorithm was the replacement of the fractional error variable by an integer decision variable in a way that the conditions used by the algorithm remained equivalent. This approach is also called the ***method of invariants***, which is useful in many rasterization algorithms.

Polygon fill

The input of an algorithm filling single connected polygons is the array of vertices $\vec{q}[0], \ldots, \vec{q}[m-1]$ (this array is usually the output of the polygon clipping algorithm). Edge e of the polygon connects vertices $\vec{q}[e]$ and $\vec{q}[e+1]$. The last vertex needs not be treated in a special way if the first vertex is put again after the last vertex in the array. Multiply connected polygons are defined by more than one closed polylines, thus are specified by more than one vertex arrays.

The filling is executed by processing a horizontal pixel row called ***scan line*** at a time. For a single scan line, the pixels belonging to the interior of the polygon can be found by the following steps. First the intersections of the polygon edges and the scan line are calculated.

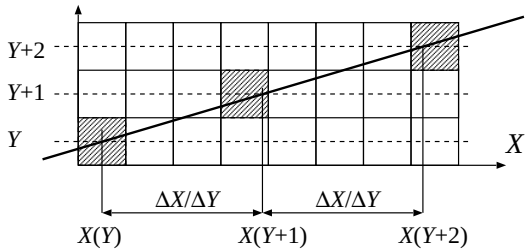


Figure 14.43. Incremental computation of the intersections between the scan lines and the edges. Coordinate X always increases with the reciprocal of the slope of the line.

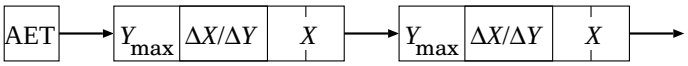


Figure 14.44. The structure of the active edge table.

Then the intersection points are sorted in the ascending order of their X coordinates. Finally, pixels between the first and the second intersection points, and between the third and the fourth intersection points, or generally between the $(2i + 1)$th and the $(2i + 2)$th intersection points are set to the colour of the polygon (figure 14.42). This algorithm fills those pixels which can be reached from infinity by crossing the polygon boundary odd number of times.

The computation of the intersections between scan lines and polygon edges can be speeded up using the following observations:

1. An edge and a scan line can have intersection only if coordinate Y of the scan line is between the minimum and maximum Y coordinates of the edge. Such edges are the ***active edges***. When implementing this idea, an ***active edge table*** (***AET*** for short) is needed which stores the currently active edges.

2. The computation of the intersection point of a line segment and the scan line requires floating point multiplication, division, and addition, thus it is time consuming. Applying the ***incremental concept***, however, we can also obtain the intersection point of the edge and a scan line from the intersection point with the previous scan line using a single, fixed-point addition (figure 14.43).

When the incremental concept is exploited, we realize that coordinate X of the intersection with an edge always increases by the same amount when scan line Y is incremented. If the edge endpoint having the larger Y coordinate is (X_{max}, Y_{max}) and the endpoint having the smaller Y coordinate is (X_{min}, Y_{min}), then the increment of the X coordinate of the intersection is $\Delta X/\Delta Y$, where $\Delta X = X_{max} - X_{min}$ and $\Delta Y = Y_{max} - Y_{min}$. This increment is usually not an integer, hence increment $\Delta X/\Delta Y$ and intersection coordinate X should be stored in non-integer, preferably fixed-point variables. An active edge is thus represented by a fixed-point increment $\Delta X/\Delta Y$, the fixed-point coordinate value of intersection X, and the maximum vertical coordinate of the edge (Y_{max}). The maximum vertical coordinate is needed to recognize when the edge becomes inactive.

Scan lines are processed one after the other. First, the algorithm determines which edges become active for this scan line, that is, which edges have minimum Y coordinate being

equal to the scan line coordinate. These edges are inserted into the active edge table. The active edge table is also traversed and those edges whose maximum Y coordinate equals to the scan line coordinate are removed (note that this way the lower end of an edge is supposed to belong to the edge, but the upper edge is not). Then the active edge table is sorted according to the X coordinates of the edges, and the pixels between each pair of edges are filled. Finally, the X coordinates of the intersections in the edges of the active edge table are prepared for the next scan line by incrementing them by the reciprocal of the slope $\Delta X/\Delta Y$.

POLYGON-FILL(*polygon*, *colour*)

```
1  for Y ← 0 to Y_max
2     do for each edge of polygon                  ▷ Put activated edges into the AET.
3           do if edge.ymin = Y
4                 then PUT-AET(edge)
5        for each edge of the AET                  ▷ Remove deactivated edges from the AET.
6           do if edge.ymax ≤ Y
7                 then DELETE-FROM-AET(edge)
8        SORT-AET                                  ▷ Sort according to X.
9        for each pair of edges (edge1, edge2) of the AET
10          do for X ← ROUND(edge1.x) to ROUND(edge2.x)
11                do PIXEL-WRITE(X, Y, colour)
11       for each edge in the AET                  ▷ Incremental concept.
12          do edge.x ← edge.x + edge.ΔX/ΔY
```

The algorithm works scan line by scan line and first puts the activated edges ($edge.ymin = Y$) to the active edge table. The active edge table is maintained by three operations. Operation PUT-AET(*edge*) computes variables ($Y_{max}, \Delta X/\Delta Y, X$) of an edge and inserts this structure into the table. Operation DELETE-FROM-AET removes an item from the table when the edge is not active any more ($edge.ymax \leq Y$). Operation SORT-AET sorts the table in the ascending order of the X value of the items. Having sorted the lists, every two consecutive items form a pair, and the pixels between the endpoints of each of these pairs are filled. Finally, the X coordinates of the items are updated according to the incremental concept.

14.7.7. Incremental visibility algorithms

The three-dimensional ***visibility problem*** is solved in the screen coordinate system. We can assume that the surfaces are given as triangle meshes.

Z-buffer algorithm

The ***z-buffer algorithm*** finds that surface for each pixel, where the Z coordinate of the visible point is minimal. For each pixel we allocate a memory to store the minimum Z coordinate of those surfaces which have been processed so far. This memory is called the ***z-buffer*** or the ***depth-buffer.***

When a triangle of the surface is rendered, all those pixels are identified which fall into

the interior of the projection of the triangle by a triangle filling algorithm. As the filling algorithm process a pixel, the Z coordinate of the triangle point visible in this pixel is obtained. If this Z value is larger than the value already stored in the z-buffer, then there exists an already processed triangle that is closer than the current triangle in this given pixel. Thus the current triangle is occluded in this pixel and its colour should not be written into the raster memory. However, if the new Z value is smaller than the value stored in the z-buffer, then the current triangle is the closest so far, and its colour and Z coordinate should be written into the pixel and the z-buffer, respectively.

The z-buffer algorithm is then:

```
Z-BUFFER()
1  for each pixel p                                          ▷ Clear screen.
2      do PIXEL-WRITE(p, background-colour)
3          z-buffer[p] ← maximum value after clipping
4  for each triangle o                                       ▷ Rendering.
5      do for each pixel p of triangle o
6          do Z ← coordinate Z of that point o which projects onto pixel p
7              if Z < z-buffer[p]
8                  then PIXEL-WRITE(p, colour of triangle o in this point)
9                       z-buffer[p] ← Z
```

When the triangle is filled, the general polygon filling algorithm of the previous section could be used. However, it is worth exploiting the special features of the triangle. Let us sort the triangle vertices according to their Y coordinates and assign index 1 to the vertex of the smallest Y coordinate and index 3 to the vertex of the largest Y coordinate. The third vertex gets index 2. Then let us cut the triangle into two pieces with scan line Y_2. After cutting we obtain a „lower" triangle and an „upper" triangle. Let us realize that in such triangles the first (left) and the second (right) intersections of the scan lines are always on the same edges, thus the administration of the polygon filling algorithm can be significantly simplified. In fact, the active edge table management is not needed anymore, only the incremental intersection calculation should be implemented. The classification of left and right intersections depend on whether (X_2, Y_2) is on the right or on the left side of the oriented line segment from (X_1, Y_1) to (X_3, Y_3). If (X_2, Y_2) is on the left side, the projected triangle is called ***left oriented,*** and ***right oriented*** otherwise.

When the details of the algorithm is introduced, we assume that the already re-indexed triangle vertices are

$$\vec{r}_1 = [X_1, Y_1, Z_1], \qquad \vec{r}_2 = [X_2, Y_2, Z_2], \qquad \vec{r}_3 = [X_3, Y_3, Z_3].$$

The rasterization algorithm is expected to fill the projection of this triangle and also to compute the Z coordinate of the triangle in every pixel (figure 14.45).

The Z coordinate of the triangle point visible in pixel X, Y is computed using the equation of the plane of the triangle (equation (14.1)):

$$n_X \cdot X + n_Y \cdot Y + n_Z \cdot Z + d = 0, \qquad \text{where } \vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1) \text{ and } d = -\vec{n} \cdot \vec{r}_1 \ . \quad (14.36)$$

Whether the triangle is left oriented or right oriented depends on the sign of the Z coordinate

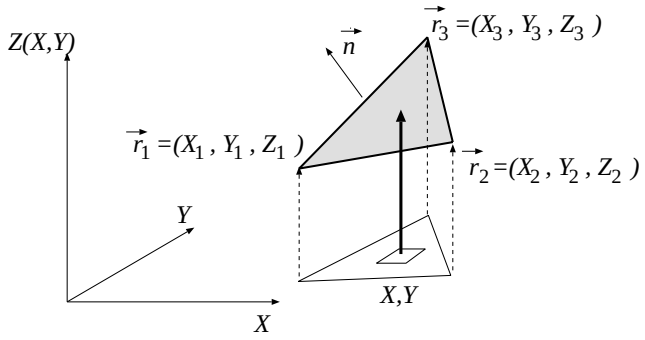


Figure 14.45. A triangle in the screen coordinate system. Pixels inside the projection of the triangle on plane XY need to be found. The Z coordinates of the triangle in these pixels are computed using the equation of the plane of the triangle.

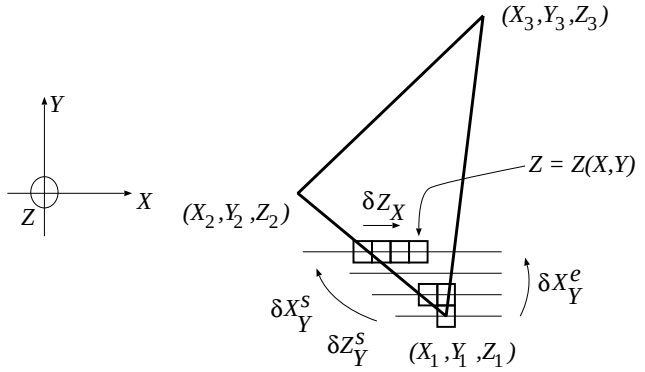


Figure 14.46. Incremental Z coordinate computation for a left oriented triangle.

of the normal vector of the plane. If n_Z is negative, then the triangle is left oriented. If it is negative, then the triangle is right oriented. Finally, when n_Z is zero, then the projections maps the triangle onto a line segment, which can be ignored during filling.

Using the equation of the plane, function $Z(X, Y)$ expressing the Z coordinate corresponding to pixel X, Y is:

$$Z(X,Y) = -\frac{n_X \cdot X + n_Y \cdot Y + d}{n_Z} \ . \tag{14.37}$$

According to the incremental concept, the evaluation the Z coordinate can take advantage of the value of the previous pixel:

$$Z(X+1,Y) = Z(X,Y) - \frac{n_X}{n_Z} = Z(X,Y) + \delta Z_X \ . \tag{14.38}$$

Since increment δZ_X is constant for the whole triangle, it needs to be computed only once. Thus the calculation of the Z coordinate in a scan line requires just a single addition per pixel. The Z coordinate values along the edges can also be obtained incrementally from the respective values at the previous scan line (figure 14.46). The complete incremental

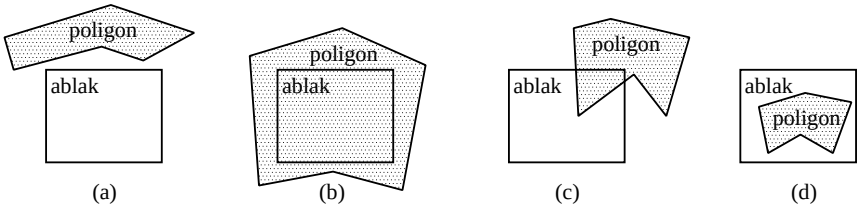


Figure 14.47. Polygon–window relations:: **(a)** distinct; **(b)** surrounding ; **(c)** intersecting; **(d)** contained.

algorithm which renders a lower left oriented triangle is (the other cases are very similar):

Z-BUFFER-LOWER-TRIANGLE$(X_1, Y_1, Z_1, X_2, Y_2, Z_2, X_3, Y_3, Z_3, colour)$

1 $\vec{n} \leftarrow ((X_2, Y_2, Z_2) - (X_1, Y_1, Z_1)) \times ((X_3, Y_3, Z_3) - (X_1, Y_1, Z_1))$ ▷ Normal vector.
2 $\delta Z_X \leftarrow -n_X/n_Z$ ▷ Z increment.
3 $(\delta X_Y^s, \delta Z_Y^s, \delta X_Y^e) \leftarrow ((X_2 - X_1)/(Y_2 - Y_1), (Z_2 - Z_1)/(Y_2 - Y_1), (X_3 - X_1)/(Y_3 - Y_1))$
4 $(X_{left}, X_{right}, Z_{left}) \leftarrow (X_1, X_1, Z_1)$
5 **for** $Y \leftarrow Y_1$ **to** Y_2
6 **do** $Z \leftarrow Z_{left}$
7 **for** $X \leftarrow$ ROUND(X_{left}) **to** ROUND(X_{right}) ▷ One scan line.
8 **if** $Z <$ *z-buffer*$[X, Y]$ ▷ Visibility test.
9 **then** PIXEL-WRITE$(X, Y, colour)$
10 *z-buffer*$[X, Y] \leftarrow Z$
11 $Z \leftarrow Z + \delta Z_X$
12 $(X_{left}, X_{right}, Z_{left}) \leftarrow (X_{left} + \delta X_Y^s, X_{right} + \delta X_Y^e, Z_{left} + \delta Z_Y^s)$ ▷ next scan line.

This algorithm simultaneously identifies the pixels to be filled and computes the Z coordinates with linear interpolation. Linear interpolation requires just a single addition when a pixel is processed. This idea can also be used for other features as well. For example, if the colour of the triangle vertices are available, the colour of the internal points can be set to provide smooth transitions applying linear interpolation. Note also that the addition to compute the feature value can also be implemented by a special purpose hardware. Graphics cards have a great number of such interpolation units.

The z-buffer algorithm fills triangles one by one, thus requires $\Theta(N \cdot P)$ time, where N is the number of triangles and P is the number of pixels on screen. In practice, however, the algorithm is since if there are more triangles in the virtual world due to higher tessellation levels, then their projected sizes are smaller, making the running time $\Theta(P)$.

Warnock algorithm

If a pixel of the image corresponds to a given object, then its neighbours usually correspond to the same object, that is, visible parts of objects appear as connected territories on the screen. This is a consequence of object coherence and is called ***image coherence***.

If the situation is so fortunate – from a labor saving point of view – that a polygon in the object scene obscures all the others and its projection onto the image plane covers the image window completely, then we have to do no more than simply fill the image with the colour

of the polygon. If no polygon edge falls into the window, then either there is no visible polygon, or some polygon covers it completely. The window is filled with the background colour in the first case, and with the colour of the closest polygon in the second case. If at least one polygon edge falls into the window, then the solution is not so simple. In this case, using a divide-and-conquer approach, the window is subdivided into four quarters, and each subwindow is searched recursively for a simple solution.

The basic form of the algorithm called ***Warnock algorithm*** rendering a rectangular window with screen coordinates X_1, Y_1 (lower left corner) and X_2, Y_2 (upper right corner) is this:

```
WARNOCK(X1, Y1, X2, Y2)
 1  if X1 ≠ X2 or Y1 ≠ Y2                          ▷ Is the window larger than a pixel?
 2    then if at least one edge projects onto the window      ▷ Subdivision and recursion.
 3           then WARNOCK(X1, Y1, (X1 + X2)/2, (Y1 + Y2)/2)
 4                WARNOCK(X1, (Y1 + Y2)/2, (X1 + X2)/2, Y2)
 5                WARNOCK((X1 + X2)/2, Y1, X2, (Y1 + Y2)/2)
 6                WARNOCK((X1 + X2)/2, (Y1 + Y2)/2, X2, Y2)
 7                return
    ▷ Trivial case: window (X1, Y1, X2, Y2) is homogeneous.
 8  polygon ← the polygon visible in pixel ((X1 + X2)/2, (Y1 + Y2)/2)
 9  if no visible polygon
10    then fill rectangle (X1, Y1, X2, Y2) with the background colour
11    else fill rectangle (X1, Y1, X2, Y2) with the colour of polygon
```

Note that the algorithm can handle non-intersecting polygons only. The algorithm can be accelerated by filtering out those distinct polygons which can definitely not be seen in a given subwindow at a given step. Furthermore, if a surrounding polygon appears at a given stage, then all the others behind it can be discarded, that is all those which fall onto the opposite side of it from the eye. Finally, if there is only one contained or intersecting polygon, then the window does not have to be subdivided further, but the polygon (or rather the clipped part of it) is simply drawn. The price of saving further recurrence is the use of a scan-conversion algorithm to fill the polygon.

Painter's algorithm

If we simply scan convert polygons into pixels and draw the pixels onto the screen without any examination of distances from the eye, then each pixel will contain the colour of the last polygon falling onto that pixel. If the polygons were ordered by their distance from the eye, and we took the farthest one first and the closest one last, then the final picture would be correct. Closer polygons would obscure farther ones — just as if they were painted an opaque colour. This method, is really known as the ***painter's algorithm***.

The only problem is that the order of the polygons necessary for performing the painter's algorithm is not always simple to compute. We say that a polygon P *does not obscure* another polygon Q if none of the points of Q is obscured by P. To have this relation, one of the following conditions should hold

1. Polygons P and Q do not overlap in Z range, and the minimum Z coordinate of polygon

P is greater than the maximum Z coordinate of polygon Q.

2. The bounding rectangle of P on the XY plane does not overlap with that of Q.

3. Each vertex of P is farther from the viewpoint than the plane containing Q.

4. Each vertex of Q is closer to the viewpoint than the plane containing P.

5. The projections of P and Q do not overlap on the XY plane.

All these conditions are sufficient. The difficulty of their test increases, thus it is worth testing the conditions in the above order until one of them proves to be true. The first step is the calculation of an ***initial depth order***. This is done by sorting the polygons according to their maximal Z value into a list. Let us first take the polygon P which is the last item on the resulting list. If the Z range of P does not overlap with any of the preceding polygons, then P is correctly positioned, and the polygon preceding P can be taken instead of P for a similar examination. Otherwise P overlaps a set $\{Q_1, \ldots, Q_m\}$ of polygons. The next step is to try to check whether P *does not* obscure any of the polygons in $\{Q_1, \ldots, Q_m\}$, that is, that P is at its right position despite the overlapping. If it turns out that P obscures Q for a polygon in the set $\{Q_1, \ldots, Q_m\}$, then Q has to be moved behind P in the list, and the algorithm continues stepping back to Q. Unfortunately, this algorithm can run into an infinite loop in case of cyclic overlapping. Cycles can also be resolved by cutting. In order to accomplish this, whenever a polygon is moved to another position in the list, we mark it. If a marked polygon Q is about to be moved again, then — assuming that Q is a part of a cycle — Q is cut into two pieces Q_1, Q_2, so that Q_1 does not obscure P and P does not obscure Q_2, and only Q_1 is moved behind P.

BSP-tree

Binary space partitioning divides first the space into two halfspaces, the second plane divides the first halfspace, the third plane divides the second halfspace, further planes split the resulting volumes, etc. The subdivision can well be represented by a binary tree, the so-called ***BSP-tree*** illustrated in figure 14.48. The kd-tree discussed in subsection 14.6.2 is also a special version of BSP-trees where the splitting planes are parallel with the coordinate planes. The BSP-tree of this subsection, however, uses general planes.

The first splitting plane is associated with the root node of the BSP-tree, the second and third planes are associated with the two children of the root, etc. For our application, not so much the planes, but rather the polygons defining them, will be assigned to the nodes of the tree, and the set of polygons contained by the volume is also necessarily associated with each node. Each leaf node will then contain either no polygon or one polygon in the associated set.

The BSP-TREE-CONSTRUCTION algorithm for creating the BSP-tree for a set S of polygons uses the following notations. A node of the binary tree is denoted by *node*, the polygon associated with the node by *node.polygon*, and the two child nodes by *node.left* and *node.right*, respectively. Let us consider a splitting plane of normal $\vec{n}$ and place vector $\vec{r}_0$. Point $\vec{r}$ belongs to the positive (right) subspace of this plane if the sign of scalar product $\vec{n} \cdot (\vec{r} - \vec{r}_0)$ is positive, otherwise it is in the negative (left) subspace. The BSP construction algorithm is:

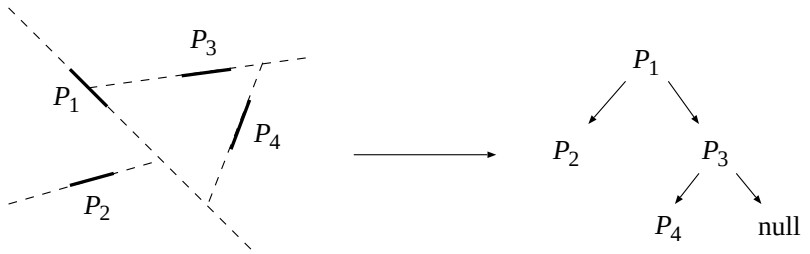


Figure 14.48. A BSP-tree. The space is subdivided by the planes of the contained polygons.

BSP-TREE-CONSTRUCTION(S)

```
1  Create a new node
2  if S is empty or contains just a single polygon
3    then node.polygon ← S
4         node.left ← null
5         node.right ← null
6    else node.polygon ← one polygon from list S
7         Remove polygon node.polygon from list S
8         S⁺ ← polygons of S which overlap with the positive subspace of node.polygon
9         S⁻ ← polygons of S which overlap with the negative subspace of node.polygon
10        node.right ← BSP-Tree-Construction(S⁺)
11        node.left ← BSP-Tree-Construction(S⁻)
12 return node
```

The size of the BSP-tree, i.e. the number of polygons stored in it, is on the one hand highly dependent on the nature of the object scene, and on the other hand on the "choice strategy" used when one polygon from list S is selected.

Having constructed the BSP-tree the visibility problem can be solved by traversing the tree in the order that if a polygon obscures another than it is processed later. During such a traversal, we determine whether the eye is at the left or right subspace at each node, and continue the traversal in the child *not* containing the eye. Having processed the child not containing the eye, the polygon of the node is drawn and finally the child containing the eye is traversed recursively.

Exercises

14.7-1 Implement the complete Bresenham algorithm that can handle not only moderately ascending but arbitrary line segments.

14.7-2 The presented polygon filling algorithm tests each edges at a scan line whether it becomes active here. Modify the algorithm in a way that such tests are not executed at each scan line, but only once.

14.7-3 Implement the complete z-buffer algorithm that renders left/righ oriented, upper/lower triangles.

14.7-4 Improve the presented Warnock algorithm and eliminate further recursions when only one edge is projected onto the subwindow.

14.7-5 Apply the BSP-tree for discrete time collision detection.
14.7-6 Apply the BSP-tree as a space partitioning structure for ray tracing.

Problems

14-1. Ray tracing renderer
Implement a rendering system applying the ray tracing algorithm. Objects are defined by triangle meshes and quadratic surfaces, and are associated with diffuse reflectivities. The virtual world also contains point light sources. The visible colour of a point is proportional to the diffuse reflectivity, the intensity of the light source, the cosine of the angle between the surface normal and the illumination direction (Lambert's law), and inversely proportional with the distance of the point and the light source. To detect whether or not a light source is not occluded from a point, use the ray tracing algorithm as well.

14-2. Continuous time collision detection with ray tracing
Using ray tracing develop a continuous time collision detection algorithm which computes the time of collision between a moving and rotating polyhedron and a still half space. Approximate the motion of a polygon vertex by a uniform, constant velocity motion in small intervals dt.

14-3. Incremental rendering system
Implement a three-dimensional renderer based on incremental rendering. The modelling and camera transforms can be set by the user. The objects are given as triangle meshes, where each vertex has colour information as well. Having transformed and clipped the objects, the z-buffer algorithm should be used for hidden surface removal. The colour at the internal points is obtained by linear interpolation from the vertex colors.

Chapter notes

The elements of Euclidean, analytic and projective geometry are discussed in the books of Maxwell [82, 83] and Coxeter [36]. The application of projective geometry in computer graphics is presented in Herman's dissertation [65] and Krammer's paper [74]. Curve and surface modelling is the main focus of computer aided geometric design (CAD, CAGD), which is discussed by Gerald Farin [49], and Rogers and Adams [95]. Geometric models can also be obtained measuring real objects, as proposed by *reverse engineering methods* [116]. Implicit surfaces can be studied by reading Bloomenthal's work [18]. Solid modelling with implicit equations is also booming thanks to the emergence of *functional representation methods (F-Rep)*, which are surveyed at http://cis.k.hosei.ac.jp/~F-rep. Blobs have been first proposed by Blinn [17]. Later the exponential influence function has been replaced by polynomials [125], which are more appropriate when roots have to be found in ray tracing.

Geometric algorithms give solutions to geometric problems such as the creation of convex hulls, clipping, containment test, tessellation, point location, etc. This field is discussed in the books of Preparata and Shamos [93] and of Marc de Berg [37, 38]. The triangulation of general polygons is still a difficult topic despite to a lot of research efforts. Practical triangulation algorithms run in $O(n \lg n)$ [38, 100, 129], but Chazelle [26] proposed an optimal

algorithm having linear time complexity. The presented proof of the *two ears theorem* has originally been given by Joseph O'Rourke [88]. Subdivision surfaces have been proposed and discussed by Catmull and Clark [25], Warren and Heimer [118], and by Brian Sharp [102, 101]. The butterfly subdivision approach has been published by Dyn et al. [44]. The *Sutherland-Hodgeman polygon clipping* algorithm is taken from [104].

Collision detection is one of the most critical problems in computer games [107] since it prevents objects to fly through walls and it is used to decide whether a bullet hits an enemy or not. Collision detection algorithms are reviewed by Jiménez, Thomas and Torras [67].

Glassner's book [56] presents many aspects of ray tracing algorithms. The *3D DDA algorithm* has been proposed by Fujimoto et al. [54]. Many papers examined the complexity of ray tracing algorithms. It has been proven that for N objects, ray tracing can be solved in $O(\lg N)$ time [37, 108], but this is theoretical rather than practical result since it requires $\Omega(N^4)$ memory and preprocessing time, which is practically unacceptable. In practice, the discussed heuristic schemes are preferred, which are better than the naive approach only in the average case. Heuristic methods have been analyzed by probabilistic tools by Márton [108] , who also proposed the probabilistic scene model used in this chapter as well. We can read about heuristic algorithms, especially about the efficient implementation of the kd-tree based ray tracing in Havran's dissertation [62]. A particularly efficient solution is given in Szécsi's paper [106].

The probabilistic tools, such as the Poisson point process can be found in the books of Karlin and Taylor [68] and Lamperti [77]. The cited fundamental law of integral geometry can be found in the book of Santaló [98].

The geoinformatics application of quadtrees and octrees are also discussed in chapter 16 of this book.

The algorithms of incremental image synthesis are discussed in many computer graphics textbooks [51]. Visibility algorithms have been compared in [105, 109]. The *painter's algorithm* has been proposed by Newell et al. [87]. Fuchs examined the construction of minimal depth BSP-trees [53]. The source of the Bresenham algorithm is [20].

Graphics cards implement the algorithms of incremental image synthesis, including transformations, clipping, z-buffer algorithm, which are accessible through graphics libraries *(OpenGL, DirectX)*. Current graphics hardware includes two programmable processors, which enables the user to modify the basic rendering pipeline. Furthermore, this flexibility allows non graphics problems to be solved on the graphics hardware. The reason of using the graphics hardware for non graphics problems is that graphics cards have much higher computational power than CPUs. We can read about such algorithms in the ShaderX or in the GPU Gems [50] series or visiting the http://www.gpgpu.org web page.

15. Relational Database Design

16. Relational Database Design

16.1. Introduction

The relational datamodel was introduced by Codd in 1970. It is the most widely used datamodel—extended with the possibilities of the World Wide Web—because of its simplicity and flexibility. The main idea of the relational model is that data is organised in relational tables, where rows correspond to individual ***records*** and columns to ***attributes***. A ***relational schema*** consists of one or more relations and their attribute sets. In the present chapter only schemata consisting of one relation are considered for the sake of simplicity. In contrast to the mathematical concept of relations, in the relational schema the order of the attributes is not important, always ***sets*** of attributes are considered instead of ***lists***. Every attribute has an associated ***domain*** that is a set of elementary values that the attribute can take values from. As an example, consider the following schema.

Employee(Name,Mother's name,Social Security Number,Post,Salary)

The domain of attributes ***Name*** and ***Mother's name*** is the set of finite character strings (more precisely its subset containing all possible names). The domain of ***Social Security Number*** is the set of integers satisfying certain formal and parity check requirements. The attribute ***Post*** can take values from the set {Director,Section chief,System integrator,Programmer,Receptionist,Janitor,Handyman}. An ***instance*** of a schema R is a relation r if its columns correspond to the attributes of R and its rows contain values from the domains of attributes at the attributes' positions. A typical row of a relation of the Employee schema could be

(John Brown,Camille Parker,184-83-2010,Programmer,$172,000)

There can be dependencies between different data of a relation. For example, in an instance of the Employee schema the value of Social Security Number determines all other values of a row. Similarly, the pair (Name,Mother's name) is a unique identifier. Naturally, it may occur that some set of attributes do not determine all attributes of a record uniquely, just some of its subsets.

A relational schema has several ***integrity constraints*** attached. The most important kind of these is ***functional dependency***. Let U and V be two sets of attributes. V ***functionally depends*** on U, $U \to V$ in notation, means that whenever two records are identical in the

attributes belonging to U, then they must agree in the attribute belonging to V, as well. Throughout this chapter the attribute set $\{A_1, A_2, \ldots, A_k\}$ is denoted by $A_1A_2 \ldots A_k$ for the sake of convenience.

16.1. Example. *Functional dependencies* Consider the schema

R(**P**rofessor,**Su**bject,**R**oom,**St**udent,**G**rade,**T**ime).

The meaning of an individual record is that a given student got a given grade of a given subject that was taught by a given professor at a given time slot. The following functional dependencies are satisfied.

Su→P: One subject is taught by one professor.
PT→R: A professor teaches in one room at a time.
StT→R: A student attends a lecture in one room at a time.
StT→Su: A student attends a lecture of one subject at a time.
SuSt→G: A student receives a unique final grade of a subject.

In Example 16.1. the attribute set **StT** uniquely determines the values of all other attributes, furthermore it is minimal such set with respect to containment. This kind attribute sets are called ***keys***. If all attributes are functionally dependent on a set of attributes X, then X is called a ***superkey***. It is clear that every superkey contains a key and that any set of attributes containing a superkey is also a superkey.

16.2. Functional dependencies

Some functional dependencies valid for a given relational schema are known already in the design phase, others are consequences of these. The **StT→P** dependency is implied by the **StT→Su** and **Su→P** dependencies in Example 16.1.. Indeed, if two records agree on attributes **St** and **T**, then they must have the same value in attribute **Su**. Agreeing in **Su** and **Su→P** implies that the two records agree in **P**, as well, thus **StT→P** holds.

Definition 16.1 *Let R be a relational schema, F be a set of functional dependencies over R. The functional dependency $U \to V$ is **logically implied** by F, in notation $F \models U \to V$, if each instance of R that satisfies all dependencies of F also satisfies $U \to V$. The **closure** of a set F of functional dependencies is the set F^+ given by*

$$F^+ = \{U \to V \colon F \models U \to V\}.$$

16.2.1. Armstrong-axioms

In order to determine keys, or to understand logical implication between functional dependencies, it is necessary to know the closure F^+ of a set F of functional dependencies, or for a given $X \to Z$ dependency the question whether it belongs to F^+ must be decidable. For this, ***inference rules*** are needed that tell that from a set of functional dependencies what others follow. The ***Armstrong-axioms*** form a system of ***sound*** and ***complete*** inference rules. A system of rules is sound if only valid functional dependencies can be derived using it. It is complete, if every dependency $X \to Z$ that is logically implied by the set F is derivable

from F using the inference rules.

ARMSTRONG-AXIOMS

(A1) ***Reflexivity*** $Y \subseteq X \subseteq R$ implies $X \to Y$.

(A2) ***Augmentation*** If $X \to Y$, then for arbitrary $Z \subseteq R$, $XZ \to YZ$ holds.

(A3) ***Transitivity*** If $X \to Y$ and $Y \to Z$ hold, then $X \to Z$ holds, as well.

16.2. Example. *Derivation by the Armstrong-axioms* Let $R = ABCD$ and $F = \{A \to C, B \to D\}$, then AB is a key:

1. $A \to C$ is given.
2. $AB \to ABC$ 1. is augmented by (A2) with AB.
3. $B \to D$ is given.
4. $ABC \to ABCD$ 3. is augmented by (A2) with ABC.
5. $AB \to ABCD$ transitivity (A3) is applied to 2. and 4..

Thus it is shown that AB is superkey. That it is really a key, follows from algorithm CLOSURE(R, F, X).

There are other valid inference rules besides (A1)–(A3). The next lemma lists some, the proof is left to the Reader (Exercise 16.2-5.).

Lemma 16.2

1. ***Union rule*** $\{X \to Y, X \to Z\} \models X \to YZ$.
2. ***Pseudo transitivity*** $\{X \to Y, WY \to Z\} \models XW \to YZ$.
3. ***Decomposition*** *If* $X \to Y$ *holds and* $Z \subseteq Y$, *then* $X \to Z$ *holds, as well.*

The soundness of system (A1)–(A3) can be proven by easy induction on the length of the derivation. The completeness will follow from the proof of correctness of algorithm CLOSURE(R, F, X) by the following lemma. Let X^+ denote the ***closure*** of the set of attributes $X \subseteq R$ with respect to the family of functional dependencies F, that is $X^+ = \{A \in R\colon X \to A$ follows from F by the Armstrong-axioms$\}$.

Lemma 16.3 *The functional dependency* $X \to Y$ *follows from the family of functional dependencies* F *by the Armstrong-axioms iff* $Y \subseteq X^+$.

Proof. Let $Y = A_1A_2 \ldots A_n$ where A_i's are attributes, and assume that $Y \subseteq X^+$. $X \to A_i$ follows by the Armstrong-axioms for all i by the definition of X^+. Applying the union rule of Lemma 16.2 $X \to Y$ follows. On the other hand, assume that $X \to Y$ can be derived by the Armstrong-axioms. By the decomposition rule of Lemma 16.2 $X \to A_i$ follows by (A1)–(A3) for all i. Thus, $Y \subseteq X^+$. ■

16.2.2. Closures

Calculation of closures is important in testing equivalence or logical implication between systems of functional dependencies. The first idea could be that for a given family F of functional dependencies in order to decide whether $F \models \{X \to Y\}$, it is enough to calculate F^+ and check whether $\{X \to Y\} \in F^+$ holds. However, the size of F^+ could be exponential

in the size of input. Consider the family F of functional dependencies given by

$$F = \{A \to B_1, A \to B_2, \ldots, A \to B_n\}.$$

F^+ consists of all functional dependencies of the form $A \to Y$, where $Y \subseteq \{B_1, B_2, \ldots, B_n\}$, thus $|F^+| = 2^n$. Nevertheless, the closure X^+ of an attribute set X with respect to F can be determined in linear time of the total length of functional dependencies in F. The following is an algorithm that calculates the closure X^+ of an attribute set X with respect to F. The input consists of the schema R, that is a finite set of attributes, a set F of functional dependencies defined over R, and an attribute set $X \subseteq R$.

CLOSURE(R,F,X)

```
 1  X^(0) ← X
 2  i ← 0
 3  G ← F                                   ▷ Functional dependencies not used yet.
 4  repeat
 5      X^(i+1) ← X^(i)
 6      for all Y → Z in G
 7          do if Y ⊆ X^(i)
 8              then X^(i+1) ← X^(i+1) ∪ Z
 9                   G ← G \ {Y → Z}
10      i ← i + 1
11  until X^(i−1) = X^(i)
```

It is easy to see that the attributes that are put into any of the $X^{(j)}$'s by CLOSURE(R,F,X) really belong to X^+. The harder part of the correctness proof of this algorithm is to show that each attribute belonging to X^+ will be put into some of the $X^{(j)}$'s.

Theorem 16.4 CLOSURE(R,F,X) *correctly calculates* X^+.

Proof. First we prove by induction that if an attribute A is put into an $X^{(j)}$ during CLOSURE(R,F,X), then A really belongs to X^+.
Base case: $j = 0$. I this case $A \in X$ and by reflexivity (A1) $A \in X^+$.
Induction step: Let $j > 0$ and assume that $X^{(j-1)} \subseteq X^+$. A is put into $X^{(j)}$, because there is a functional dependency $Y \to Z$ in F, where $Y \subseteq X^{(j-1)}$ and $A \in Z$. By induction, $Y \subseteq X^+$ holds, which implies using Lemma 16.3 that $X \to Y$ holds, as well. By transitivity (A3) $X \to Y$ and $Y \to Z$ implies $X \to Z$. By reflexivity (A1) and $A \in Z$, $Z \to A$ holds. Applying transitivity again, $X \to A$ is obtained, that is $A \in X^+$.

On the other hand, we show that if $A \in X^+$, then A is contained in the result of CLOSURE(R,F,X). Suppose in contrary that $A \in X^+$, but $A \notin X^{(i)}$, where $X^{(i)}$ is the result of CLOSURE(R,F,X). By the stop condition in line 9 this means $X^{(i)} = X^{(i+1)}$. An instance r of the schema R is constructed that satisfies every functional dependency of F, but $X \to A$ does not hold in r if $A \notin X^{(i)}$. Let r be the following two-rowed relation:

Attributes of $X^{(i)}$				Other attributes			
1	1	…	1	1	1	…	1
1	1	…	1	0	0	…	0

Let us suppose that the above r violates a $U \to V$ functional dependency of F, that is $U \subseteq X^{(i)}$, but V is not a subset of $X^{(i)}$. However, in this case CLOSURE(R,F,X) could not have stopped yet, since $X^{(i)} \neq X^{(i+1)}$.

$A \in X^+$ implies using Lemma 16.3 that $X \to A$ follows from F by the Armstrong-axioms. (A1)–(A3) is a sound system of inference rules, hence in every instance that satisfies F, $X \to A$ must hold. However, the only way this could happen in instance r is if $A \in X^{(i)}$. ■

Let us observe that the relation instance r given in the proof above provides the completeness proof for the Armstrong-axioms, as well. Indeed, the closure X^+ calculated by CLOSURE(R,F,X) is the set of those attributes for which $X \to A$ follows from F by the Armstrong-axioms. Meanwhile, for every other attribute B, there exist two rows of r that agree on X, but differ in B, that is $F \models X \to B$ ***does not*** hold.

The running tome of CLOSURE(R,F,X) is $O(n^2)$, where n is the length of he input. Indeed, in the **repeat – until** loop of lines 4–11 every not yet used dependency is checked, and the body of the loop is executed at most $|R \setminus X| + 1$ times, since it is started again only if $X^{(i-1)} \neq X^{(i)}$, that is a new attribute is added to the closure of X. However, the running time can be reduced to linear with appropriate bookkeeping.

1. For every yet unused $W \to Z$ dependency of F it is kept track of how many attributes of W are not yet included in the closure ($i[W,Z]$).

2. For every attribute A those yet unused dependencies are kept in a doubly linked list L_A whose left side contains A.

3. Those not yet used dependencies $W \to Z$ are kept in a linked list J, whose left side W's every attribute is contained in the closure already, that is for which $i[W,Z] = 0$.

It is assumed that the family of functional dependencies F is given as a set of attribute pairs (W,Z), representing $W \to Z$. The LINEAR-CLOSURE(R,F,X) algorithm is a modification of CLOSURE(R,F,X) using the above bookkeeping, whose running time is linear. R is the schema, F is the given family of functional dependencies, and we are to determine the closure of attribute set X.

Algorithm LINEAR-CLOSURE(R,F,X) consists of two parts. In the initialisation phase (lines 1–13) the lists are initialised. The loops of lines 2–5 and 6–8, respectively, take $O(\sum_{(W,Z)\in F} |W|)$ time. The loop in lines 9–11 means $O(|F|)$ steps. If the length of the input is denoted by n, then this is $O(n)$ steps altogether.

During the execution of lines 14–23, every functional dependency (W,Z) is examined at most once, when it is taken off from list J. Thus, lines 15–16 and 23 take at most $|F|$ steps. The running time of the loops in line 17–22 can be estimated by observing that the sum $\sum i[W,Z]$ is decreased by one in each execution, hence it takes $O(\sum i_0[W,Z])$ steps, where $i_0[W,Z]$ is the $i[W,Z]$ value obtained in the initialisation phase. However, $\sum i_0[W,Z] \leq \sum_{(W,Z)\in F} |W|$, thus lines 14–23 also take $O(n)$ time in total.

```
LINEAR-CLOSURE(R,F,X)
 1  ▷ Initialisation phase.
 2  for all (W,Z) ∈ F
 3      do for all A ∈ W
 4             do add (W,Z) to list L_A
 5      i[W,Z] ← 0
 6  for all A ∈ R \ X
 7      do for all (W,Z) of list L_A
 8             do i[W,Z] ← i[W,Z] + 1
 9  for all (W,Z) ∈ F
10      do if i[W,Z] = 0
11            then add (W,Z) to list J
12  X⁺ ← X
13  ▷ End of initialisation phase.
14  while J is nonempty
15        do (W,Z) ← head(J)
16           delete (W,Z) from list J
17           for all A ∈ Z \ X⁺
18                do for all (W,Z) of list L_A
19                       do i[W,Z] ← i[W,Z] − 1
20                          if i[W,Z] = 0
21                             then add (W,Z) to list J
22                                  delete (W,Z)from list L_A
23        X⁺ ← X⁺ ∪ Z
24  return X⁺
```

16.2.3. Minimal cover

Algorithm LINEAR-CLOSURE(R,F,X) can be used to test equivalence of systems of dependencies. Let F and G be two families of functional dependencies. F and G are said to be ***equivalent,*** if exactly the same functional dependencies follow from both, that is $F^+ = G^+$. It is clear that it is enough to check for all functional dependencies $X \to Y$ in F whether it belongs to G^+, and vice versa, for all $W \to Z$ in G, whether it is in F^+. Indeed, if some of these is not satisfied, say $X \to Y$ is not in G^+, then surely $F^+ \neq G^+$. On the other hand, if all $X \to Y$ are in G^+, then a proof of a functional dependency $U \to V$ from F^+ can be obtained from dependencies in G in such a way that to the derivation of the dependencies $X \to Y$ of F from G, the derivation of $U \to V$ from F is concatenated. In order to decide that a dependency $X \to Y$ from F is in G^+, it is enough to construct the closure $X^+(G)$ of attribute set X with respect to G using LINEAR-CLOSURE(R,G,X), then check whether $Y \subseteq X^+(G)$ holds. The following special functional dependency system equivalent with F is useful.

Definition 16.5 *The system of functional dependencies G is a **minimal cover** of the family of functional dependencies F iff G is equivalent with F, and*

1. *functional dependencies of G are in the form $X \to A$, where A is an attribute and $A \notin X$,*
2. *no functional dependency can be dropped from G, i.e., $(G - \{X \to A\})^+ \subsetneqq G^+$,*
3. *the left sides of dependencies in G are minimal, that is $X \to A \in G$, $Y \subsetneqq X \Longrightarrow ((G - \{X \to A\}) \cup \{Y \to A\})^+ \neq G^+$.*

Every set of functional dependencies have a minimal cover, namely algorithm MINIMAL-COVER(R, F) constructs one.

MINIMAL-COVER(R, F)

```
 1  G ← ∅
 2  for all X → Y ∈ F
 3      do for all A ∈ Y − X
 4              do G ← G ∪ X → A
 5  ▷ Each right hand side consists of a single attribute.
 6  for all X → A ∈ G
 7      do while there exists B ∈ X : A ∈ (X − B)⁺(G)
 8              X ← X − B
 9  ▷ Each left hand side is minimal.
10  for all X → A ∈ G
11      do if A ∈ X⁺(G − {X → A})
12          then G ← G − {X → A}
13  ▷ No redundant dependency exists.
```

After executing the loop of lines 2–4, the right hand side of each dependency in G consists of a single attribute. The equality $G^+ = F^+$ follows from the union rule of Lemma 16.2 and the reflexivity axiom. Lines 6–8 minimise the left hand sides. In line 11 it is checked whether a given functional dependency of G can be removed without changing the closure. $X^+(G - \{X \to A\})$ is the closure of attribute set X with respect to the family of functional dependencies $G - \{X \to A\}$.

Proposition 16.6 MINIMAL-COVER*(R, F) calculates a minimal cover of F.*

Proof. It is enough to show that during execution of the loop in lines 10–12, no functional dependency $X \to A$ is generated whose left hand side could be decreased. Indeed, if a $X \to A$ dependency would exist, such that for some $Y \subsetneqq X$ $Y \to A \in G^+$ held, then $Y \to A \in G'^+$ would also hold, where G' is the set of dependencies considered when $X \to A$ is checked in lines 6–8. $G \subseteq G'$, which implies $G^+ \subseteq G'^+$ (see Exercise 16.2-1.). Thus, X should have been decreased already during execution of the loop in lines 6–8. ■

16.2.4. Keys

In database design it is important to identify those attribute sets that uniquely determine the data in individual records.

Definition 16.7 *Let (R, F) be a relational schema. The set of attributes $X \subseteq R$ is called a* ***superkey,*** *if $X \to R \in F^+$. A superkey X is called a* ***key,*** *if it is minimal with respect to containment, that is no proper subset $Y \subsetneqq X$ is key.*

The question is how the keys can be determined from (R, F)? What makes this problem hard is that the number of keys could be super exponential function of the size of (R, F). In particular, Yu and Johnson constructed such relational schema, where $|F| = k$, but the number of keys is $k!$. Békéssy and Demetrovics gave a beautiful and simple proof of the fact that starting from k functional dependencies, at most $k!$ key can be obtained. (This was independently proved by Osborne and Tompa.)

The proof of Békéssy and Demetrovics is based on the operation $*$ they introduced, which is defined for functional dependencies.

Definition 16.8 *Let $e_1 = U \to V$ and $e_2 = X \to Y$ be two functional dependencies. The binary operation $*$ is defined by*

$$e_1 * e_2 = U \cup ((R - V) \cap X) \to V \cup Y.$$

Some properties of operation $*$ is listed, the proof is left to the Reader (Exercise 16.2-3.). Operation $*$ is associative, furthermore it is idempotent in the sense that if $e = e_1 * e_2 * \cdots * e_k$ and $e' = e * e_i$ for some $1 \leq i \leq k$, then $e' = e$.

Proposition 16.9 (Békéssy and Demetrovics). *Let (R, F) be a relational schema and let $F = \{e_1, e_2, \ldots, e_k\}$ be a listing of the functional dependencies. If X is a key, then $X \to R = e_{\pi_1} * e_{\pi_2} * \ldots * e_{\pi_s} * d$, where $(\pi_1, \pi_2, \ldots, \pi_s)$ is an ordered subset of the index set $\{1, 2, \ldots, k\}$, and d is a trivial dependency in the form $D \to D$.*

Proposition 16.9 bounds in some sense the possible sets of attributes in the search for keys. The next proposition gives lower and upper bounds for the keys.

Proposition 16.10 *Let (R, F) be a relational schema and let $F = \{U_i \to V_i \colon 1 \leq i \leq k\}$. Let us assume without loss of generality that $U_i \cap V_i = \emptyset$. Let $\mathcal{U} = \bigcup_{i=1}^{k} U_i$ and $\mathcal{V} = \bigcup_{i=1}^{k} V_i$. If K is a key in the schema (R, F), then*

$$\mathcal{H}_L = R - \mathcal{V} \subseteq K \subseteq (R - \mathcal{V}) \cup \mathcal{U} = \mathcal{H}_U.$$

The proof is not too hard, it is left as an exercise for the Reader (Exercise 16.2-4.). The algorithm List-keys(R, F) that lists the keys of the schema (R, F) is based on the bounds of Proposition 16.10. The running time can be bounded by $O(n!)$, but one cannot expect any better, since to list the output needs that much time in worst case.

LIST-KEYS(R, F)

```
1  ▷ Let 𝒰 and 𝒱 be as defined in Proposition 16.10
2  if 𝒰 ∩ 𝒱 = ∅
3     then return R − 𝒱
4  ▷ R − 𝒱 is the only key.
5  if (R − 𝒱)⁺ = R
6     then return R − 𝒱
7     ▷ R − 𝒱 is the only key.
8  𝒦 ← ∅
9  for all permutations A_1, A_2, ... A_h of the attributes of 𝒰 ∩ 𝒱
10    do K ← (R − 𝒱) ∪ 𝒰
11       for i ← 1 to h
12          do Z ← K − A_i
13             if Z⁺ = R
14                then K ← Z
15       𝒦 ← 𝒦 ∪ {K}
16 return 𝒦
```

Exercises

16.2-1 Let R be a relational schema and let F and G be families of functional dependencies over R.Show that

a. $F \subseteq F^+$.

b. $(F^+)^+ = F^+$.

c. If $F \subseteq G$, then $F^+ \subseteq G^+$.

Formulate and prove similar properties of the closure X^+ – with respect to F – of an attribute set X.

16.2-2 Derive the functional dependency $AB \to F$ from the set of dependencies $G = \{AB \to C, A \to D, CD \to EF\}$ using Armstrong-axioms (A1)–(A3).

16.2-3 Show that operation $*$ is associative, furthermore if for functional dependencies $e_1, e_2, \ldots, e_k$ we have $e = e_1 * e_2 * \cdots * e_k$ and $e' = e * e_i$ for some $1 \leq i \leq k$, then $e' = e$.

16.2-4 Prove Proposition 16.10.

16.2-5 Prove the union, pseudo transitivity and decomposition rules of Lemma 16.2.

16.3. Decomposition of relational schemata

A ***decomposition*** of a relational schema $R = \{A_1, A_2, \ldots, A_n\}$ is a collection $\rho = \{R_1, R_2, \ldots, R_k\}$ of subsets of R such that

$$R = R_1 \cup R_2 \cup \cdots \cup R_k.$$

The R_i's need not be disjoint, in fact in most application they must not be. One important motivation of decompositions is to avoid ***anomalies***.

16.3. Example. *Anomalies* Consider the following schema

SUPPLIER-INFO(SNAME,ADDRESS,ITEM,PRICE)

This schema encompasses the following problems:

1. ***Redundancy.*** The address of a supplier is recorded with every item it supplies.
2. ***Possible inconsistency (update anomaly).*** As a consequence of redundancy, the address of a supplier might be updated in some records and might not be in some others, hence the supplier would not have a unique address, even though it is expected to have.
3. ***Insertion anomaly.*** The address of a supplier cannot be recorded if it does not supply anything at the moment. One could try to use `NULL` values in attributes ITEM and PRICE, but would it be remembered that it must be deleted, when a supplied item is entered for that supplier? More serious problem that SNAME and ITEM together form a key of the schema, and the `NULL` values could make it impossible to search by an index based on that key.
4. ***Deletion anomaly*** This is the opposite of the above. If all items supplied by a supplier are deleted, then as a side effect the address of the supplier is also lost.

All problems mentioned above are eliminated if schema SUPPLIER-INFO is replaced by two subschemata:

SUPPLIER(SNAME,ADDRESS),
SUPPLIES(SNAME,ITEM,PRICE).

In this case each suppliers address is recorded only once, and it is not necessary that the supplier supplies a item in order its address to be recorded. For the sake of convenience the attributes are denoted by single characters S (SNAME), A (ADDRESS), I (ITEM), P (PRICE).

Question is that is it correct to replace the schema $SAIP$ by SA and SIP? Let r be and instance of schema $SAIP$. It is natural to require that if SA and SIP is used, then the relations belonging to them are obtained projecting r to SA and SIP, respectively, that is $r_{SA} = \pi_{SA}(r)$ and $r_{SIP} = \pi_{SIP}(r)$. r_{SA} and r_{SIP} contains the same information as r, if r can be reconstructed using only r_{SA} and r_{SIP}. The calculation of r from r_{SA} and r_{SIP} can bone by the ***natural join*** operator.

Definition 16.11 *The **natural join** of relations r_i of schemata R_i $(i = 1, 2, \ldots n)$ is the relation s belonging to the schema $\cup_{i=1}^{n} R_i$, which consists of all rows μ that for all i there exists a row ν_i of relation r_i such that $\mu[R_i] = \nu_i[R_i]$. In notation $s = \bowtie_{i=1}^{n} r_i$.*

16.4. Example. Let $R_1 = AB$, $R_2 = BC$, $r_1 = \{ab, a'b', ab''\}$ and $r_2\{bc, bc', b'c''\}$. The natural join of r_1 and r_2 belongs to the schema $R = ABC$, and it is the relation $r_1 \bowtie r_2 = \{abc, abc', a'b'c''\}$.

If s is the natural join of r_{SA} and r_{SIP}, that is $s = r_{SA} \bowtie r_{SIP}$, then $\pi_{SA}(s) = r_{SA}$ és $\pi_{SIP}(s) = r_{SIP}$ by Lemma 16.12. If $r \neq s$, then the original relation could not be reconstructed knowing only r_{SA} and r_{SIP}.

16.3.1. Lossless join

Let $\rho = \{R_1, R_2, \ldots, R_k\}$ be a decomposition of schema R, furthermore let F be a family of functional dependencies over R. The decomposition ρ is said to have ***lossless join property*** (with respect to F), if every instance r of R that satisfies F also satisfies

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \cdots \bowtie \pi_{R_k}(r).$$

That is, relation r is the natural join of its projections to attribute sets R_i, $i = 1, 2, \ldots, k$. For a decomposition $\rho = \{R_1, R_2, \ldots, R_k\}$, let m_ρ denote the the mapping which assigns to relation r the relation $m_\rho(r) = \bowtie_{i=1}^{k} \pi_{R_i}(r)$. Thus, the lossless join property with respect to a family of functional dependencies means that $r = m_\rho(r)$ for all instances r that satisfy F.

Lemma 16.12 *Let $\rho = \{R_1, R_2, \ldots, R_k\}$ be a decomposition of schema R, and let r be an arbitrary instance of R. Furthermore, let $r_i = \pi_{R_i}(r)$. Then*

1. $r \subseteq m_\rho(r)$.

2. *If $s = m_\rho(r)$, then $\pi_{R_i}(s) = r_i$.*

3. $m_\rho(m_\rho(r)) = m_\rho(r)$.

The proof is left to the Reader (Exercise 16.3-7.).

16.3.2. Checking the lossless join property

It is relatively not hard to check that a decomposition $\rho = \{R_1, R_2, \ldots, R_k\}$ of schema R has the lossless join property. The essence of algorithm JOIN-TEST(R,F,ρ) is the following.
A $k \times n$ array T is constructed, whose column j corresponds to attribute A_j, while row i corresponds to schema R_i. $T[i, j] = 0$ if $A_j \in R_i$, otherwise $T[i, j] = i$.

The following step is repeated until there is no more possible change in the array. Consider a functional dependency $X \to Y$ from F. If a pair of rows i and j agree in all attributes of X, then their values in attributes of Y are made equal. More precisely, if one of the values in an attribute of Y is 0, then the other one is set to 0, as well, otherwise it is arbitrary which of the two values is set to be equal to the other one. If a symbol is changed, then ***each*** of its occurrences in that column must be changed accordingly. If at the end of this process there is an all 0 row in T, then the decomposition has the lossless join property, otherwise, it is lossy.

JOIN-TEST(R,F,ρ)

```
 1 ▷ Initialisation phase.
 2 for i ← 1 to |ρ|
 3     do for j ← 1 to |R|
 4           do if A_j ∈ R_i
 5                 then T[i, j] ← 0
 6                 else T[i, j] ← i
 7 ▷ End of initialisation phase.
 8 S ← T
 9 repeat
10       T ← S
11       for all {X → Y} ∈ F
12            do for i ← 1 to |ρ| − 1
13                  do for j ← i + 1 to |R|
14                        do if for all A_h in X (S[i, h] = S[j, h])
15                              then EQUATE(i, j, S, Y)
16 until S = T
17 if there exist an all 0 row in S
18   then return "Lossless join"
19   else return "Lossy join"
```

Procedure EQUATE(i, j, S, Y) makes the appropriate symbols equal.

EQUATE(i, j, S, Y)

```
 1 for A_l ∈ Y
 2     do if S[i, l] · S[j, l] = 0
 3           then
 4                for d ← 1 to k
 5                    do if S[d, l] = S[i, l] ∨ S[d, l] = S[j, l]
 6                          then S[d, l] ← 0
 7           else
 8                for d ← 1 to k
 9                    do if S[d, l] = S[j, l]
10                          then S[d, l] ← S[i, l]
```

16.5. Example. *Checking lossless join property* Let $R = ABCDE$, $R_1 = AD$, $R_2 = AB$, $R_3 = BE$, $R_4 = CDE$, $R_5 = AE$, furthermore let the functional dependencies be $\{A \to C, B \to C, C \to D, DE \to C, CE \to A\}$. The initial array is shown on Figure 16.1(a). Using $A \to C$ values 1,2,5 in column C can be equated to 1. Then applying $B \to C$ value 3 of column C again can be changed to 1. The result is shown on Figure 16.1(b). Now $C \to D$ can be used to change values 2,3,5 of column D to 0. Then applying $DE \to C$ (the only nonzero) value 1 of column C can be set to 0. Finally, $CE \to A$ makes it possible to change values 3 and 4 in column A to be changed to 0. The final result is shown on Figure 16.1(c). The third row consists of only zeroes, thus the decomposition has the lossless join property.

It is clear that the running time of algorithm JOIN-TEST(R,F,ρ) is polynomial in the length

A	B	C	D	E
0	1	1	0	1
0	0	2	2	2
3	0	3	3	0
4	4	0	0	0
0	5	5	5	0

(a)

A	B	C	D	E
0	1	1	0	1
0	0	1	2	2
3	0	1	3	0
4	4	0	0	0
0	5	1	5	0

(b)

A	B	C	D	E
0	1	0	0	1
0	0	0	2	2
0	0	0	0	0
0	4	0	0	0
0	5	0	0	0

(c)

Figure 16.1. Application of JOIN-TEST(R,F,ρ).

of the input. The important thing is that it uses only the schema, not the instance r belonging to the schema. Since the size of an instance is larger than the size of the schema by many orders of magnitude, the running time of an algorithm using the schema only is negligible with respect to the time required by an algorithm processing the data stored.

Theorem 16.13 *Procedure* JOIN-TEST(R,F,ρ) *correctly determines whether a given decomposition has the lossless join property.*

Proof. Let us assume first that the resulting array T contains no all zero row. T itself can be considered as a relational instance over the schema R. This relation satisfies all functional dependencies from F, because the algorithm finished since there was no more change in the table during checking the functional dependencies. It is true for the starting table that its projections to every R_i's contain an all zero row, and this property does not change during the running of the algorithm, since a 0 is never changed to another symbol. It follows, that the natural join $m_\rho(T)$ contains the all zero row, that is $T \neq m_\rho(T)$. Thus the decomposition is lossy. The proof of the other direction is only sketched.

Logic, domain calculus is used. The necessary definitions can be found in the books of Abiteboul, Hull and Vianu, or Ullman, respectively. Imagine that variable a_j is written in place of zeroes, and b_{ij} is written in place of i's in column j, and JOIN-TEST(R,F,ρ) is run in this setting. The resulting table contains row $a_1 a_2 \dots a_n$, which corresponds to the all zero row. Every table can be viewed as a shorthand notation for the following domain calculus expression

$$\{a_1 a_2 \dots a_n \mid (\exists b_{11}) \dots (\exists b_{kn})\, (R(w_1) \wedge \dots \wedge R(w_k))\}, \tag{16.1}$$

where w_i is the ith row of T. If T is the starting table, then formula (16.1) defines m_ρ exactly. As a justification note that for a relation r, $m_\rho(r)$ contains the row $a_1 a_2 \dots a_n$ iff r contains for all i a row whose jth coordinate is a_j if A_j is an attribute of R_i, and arbitrary values represented by variables b_{il} in the other attributes.

Consider an arbitrary relation r belonging to schema R that satisfies the dependencies of F. The modifications (equating symbols) of the table done by JOIN-TEST(R,F,ρ) do not change the set of rows obtained from r by (16.1), if the modifications are done in the formula, as well. Intuitively it can be seen from the fact that only such symbols are equated in (16.1), that can only take equal values in a relation satisfying functional dependencies of F. The exact proof is omitted, since it is quiet tedious.

Since in the result table of JOIN-TEST(R,F,ρ) the all a's row occurs, the domain calculus formula that belongs to this table is of the following form:

$$\{a_1 a_2 \dots a_n \mid (\exists b_{11}) \dots (\exists b_{kn})\,(R(a_1 a_2 \dots a_n) \wedge \cdots)\}\,. \tag{16.2}$$

It is obvious that if (16.2) is applied to relation r belonging to schema R, then the result will be a subset of r. However, if r satisfies the dependencies of F, then (16.2) calculates $m_\rho(r)$. According to Lemma 16.12, $r \subseteq m_\rho(r)$ holds, thus if r satisfies F, then (16.2) gives back r exactly, so $r = m_\rho(r)$, that is the decomposition has the lossless join property. ■

Procedure JOIN-TEST(R,F,ρ) can be used independently of the number of parts occurring in the decomposition. The price of this generality is paid in the running time requirement. However, if R is to be decomposed only into ***two*** parts, then CLOSURE(R,F,X) or LINEAR-CLOSURE(R,F,X) can be used to obtain the same result faster, according to the next theorem.

Theorem 16.14 *Let $\rho = (R_1, R_2)$ be a decomposition of R, furthermore let F be a set of functional dependencies. Decomposition ρ has the lossless join property with respect to F iff*

$$(R_1 \cap R_2) \to (R_1 - R_2) \;\; \textit{or} \;\; (R_1 \cap R_2) \to (R_2 - R_1)\,.$$

These dependencies need not be in F, it is enough if they are in F^+.

Proof. The starting table in procedure JOIN-TEST(R,F,ρ) is the following:

	$R_1 \cap R_2$	$R_1 - R_2$	$R_2 - R_1$
row of R_1	$00\dots0$	$00\dots0$	$11\dots1$
row of R_2	$00\dots0$	$22\dots2$	$00\dots0$

(16.3)

It is not hard to see using induction on the number of steps done by JOIN-TEST(R,F,ρ) that if the algorithm changes both values of the column of an attribute A to 0, then $A \in (R_1 \cap R_2)^+$. This is obviously true at the start. If at some time values of column A must be equated, then by lines 11–14 of the algorithm, there exists $\{X \to Y\} \in F$, such that the two rows of the table agree on X, and $A \in Y$. By the induction assumption $X \subseteq (R_1 \cap R_2)^+$ holds. Applying Armstrong-axioms (transitivity and reflexivity), $A \in (R_1 \cap R_2)^+$ follows.

On the other hand, let us assume that $A \in (R_1 \cap R_2)^+$, that is $(R_1 \cap R_2) \to A$. Then this functional dependency can be derived from F using Armstrong-axioms. By induction on the length of this derivation it can be seen that procedure JOIN-TEST(R,F,ρ) will equate the two values of column A, that is set them to 0. Thus, the row of R_1 will be all 0 iff $(R_1 \cap R_2) \to (R_2 - R_1)$, similarly, the row of R_2 will be all 0 iff $(R_1 \cap R_2) \to (R_1 - R_2)$. ■

16.3.3. Dependency preserving decompositions

The lossless join property is important so that a relation can be recovered from its projections. In practice, usually not the relation r belonging to the underlying schema R is stored, but relations $r_i = r[R_i]$ for an appropriate decomposition $\rho = (R_1, R_2, \ldots, R_k)$, in order to avoid anomalies. The functional dependencies F of schema R are ***integrity constraints*** of the database, relation r is consistent if it satisfies all prescribed functional dependencies. When during the life time of the database updates are executed, that is rows are inserted into or deleted from the projection relations, then it may happen that the natural join of the new projections does not satisfy the functional dependencies of F. It would be too costly to join the projected relations – and then project them again – after each update to check the integrity constraints. However, the ***projection*** of the family of functional dependencies F to an attribute set Z can be defined: $\pi_Z(F)$ consists of those functional dependencies $\{X \to Y\} \in F^+$, where $XY \subseteq Z$. After an update, if relation r_i is changed, then it is relatively easy to check whether $\pi_{R_i}(F)$ still holds. Thus, it would be desired if family F would be logical implication of the families of functional dependencies $\pi_{R_i}(F)$ $i = 1, 2, \ldots, k$. Let $\pi_\rho(F) = \bigcup_{i=1}^{k} \pi_{R_i}(F)$.

Definition 16.15 *The decomposition ρ is said to be **dependency preserving**, if*

$$\pi_\rho(F)^+ = F^+.$$

Note that $\pi_\rho(F) \subseteq F^+$, hence $\pi_\rho(F)^+ \subseteq F^+$ always holds. Consider the following example.

16.6. Example. Let $R =$ (**C**ity,**S**treet,**Z**ip code) be the underlying schema, furthermore let $F = \{CS \to Z, Z \to C\}$ be the functional dependencies. Let the decomposition ρ be $\rho = (CZ, SZ)$. This has the lossless join property by Theorem 16.14. $\pi_\rho(F)$ consists of $Z \to C$ besides the trivial dependencies. Let $R_1 = CZ$ and $R_2 = SZ$. Two rows are inserted into each of the projections belonging to schemata R_1 and R_2, respectively, so that functional dependencies of the projections are satisfied:

R_1	C	Z
	Fort Wayne	46805
	Fort Wayne	46815

R_2	S	Z
	Coliseum Blvd	46805
	Coliseum Blvd	46815

In this case R_1 and R_2 satisfy the dependencies prescribed for them separately, however in $R_1 \bowtie R_2$ the dependency $CS \to Z$ does not hold.

It is true as well, that none of the decompositions of this schema preserves the dependency $CS \to Z$. Indeed, this is the only dependency that contains Z on the right hand side, thus if it is to be preserved, then there has to be a subschema that contains C, S, Z, but then the decomposition would not be proper. This will be considered again when decomposition into normal forms is treated.

Note that it may happen that decomposition ρ preserves functional dependencies, but does not have the lossless join property. Indeed, let $R = ABCD$, $F = \{A \to B, C \to D\}$, and let the decomposition be $\rho = (AB, CD)$.

Theoretically it is very simple to check whether a decomposition $\rho = (R_1, R_2, \ldots R_k)$ is dependency preserving. Just F^+ needs to be calculated, then projections need to be taken, finally one should check whether the union of the projections is equivalent with F. The main problem with this approach is that even calculating F^+ may need exponential time.

Nevertheless, the problem can be solved without explicitly determining F^+. Let $G = \pi_\rho(F)$. G will not be calculated, only its equivalence with F will be checked. For this end,

it needs to be decidable for all functional dependencies $\{X \to Y\} \in F$ that if X^+ is taken with respect to G, whether it contains Y. The trick is that X^+ is determined ***without*** full knowledge of G by repeatedly taking the effect to the closure of the projections of F onto the individual R_i's. That is, the concept of S-operation on an attribute set Z is introduced, where S is another set of attributes: Z is replaced by $Z \cup ((Z \cap S)^+ \cap S)$, where the closure is taken with respect to F. Thus, the closure of the part of Z that lies in S is taken with respect to F, then from the resulting attributes those are added to Z, which also belong to S.

It is clear that the running time of algorithm PRESERVE(ρ, F) is polynomial in the length of the input. More precisely, the outermost **for** loop is executed at most once for each dependency in F (it may happen that it turns out earlier that some dependency is not preserved). The body of the **repeat–until** loop in lines 3–7. requires linear number of steps, it is executed at most $|R|$ times. Thus, the body of the **for** loop needs quadratic time, so the total running time can be bounded by the cube of the input length.

PRESERVE(ρ, F)

```
 1 for all (X → Y) ∈ F
 2     do Z ← X
 3        repeat
 4                W ← Z
 5                for i ← 1 to k
 6                    do Z ← Z ∪ (LINEAR-CLOSURE(R, F, Z ∩ R_i) ∩ R_i)
 7        until Z = W
 8     if Y ⊈ Z
 9        then return "Not dependency preserving"
10 return "Dependency preserving"
```

16.7. Example. Consider the schema $R = ABCD$, let the decomposition be $\rho = \{AB, BC, CD\}$, and dependencies be $F = \{A \to B, B \to C, C \to D, D \to A\}$. That is, by the visible cycle of the dependencies, every attribute determines all others. Since D and A do not occur together in the decomposition one might think that the dependency $D \to A$ is not preserved, however this intuition is wrong. The reason is that during the projection to AB, not only the dependency $A \to B$ is obtained, but $B \to A$, as well, since not F, but F^+ is projected. Similarly, $C \to B$ and $D \to C$ are obtained, as well, but $D \to A$ is a logical implication of these by the transitivity of the Armstrong axioms. Thus it is expected that PRESERVE(ρ, F) claims that $D \to A$ is preserved.

Start from the attribute set $Y = \{D\}$. There are three possible operations, the AB-operation, the BC-operation and the CD-operation. The first two obviously does not add anything to $\{D\}^+$, since $\{D\} \cap \{A, B\} = \{D\} \cap \{B, C\} = \emptyset$, that is the closure of the empty set should be taken, which is empty (in the present example). However, using the CD-operation:

$$\begin{aligned} Z &= \{D\} \cup ((\{D\} \cap \{C, D\})^+ \cap \{C, D\}) \\ &= \{D\} \cup (\{D\}^+ \cap \{C, D\}) \\ &= \{D\} \cup (\{A, B, C, D\} \cap \{C, D\}) \\ &= \{C, D\}. \end{aligned}$$

In the next round using the BC-operation the actual $Z = \{C, D\}$ is changed to $Z = \{B, C, D\}$, finally applying the AB-operation on this, $Z = \{A, B, C, D\}$ is obtained. This cannot change, so procedure

PRESERVE(ρ, F) stops. Thus, with respect to the family of functional dependencies

$$G = \pi_{AB}(F) \cup \pi_{BC}(F) \cup \pi_{CD}(F),$$

$\{D\}^+ = \{A, B, C, D\}$ holds, that is $G \models D \to A$. It can be checked similarly that the other dependencies of F are in G^+ (as a fact in G).

Theorem 16.16 *The procedure* PRESERVE(ρ, F) *determines correctly whether the decomposition* ρ *is dependency preserving.*

Proof. It is enough to check for a single functional dependency $X \to Y$ whether whether the procedure decides correctly if it is in G^+. When an attribute is added to Z in lines 3–7, then Functional dependencies from G are used, thus by the soundness of the Armstrong-axioms if PRESERVE(ρ, F) claims that $X \to Y \in G^+$, then it is indeed so.

On the other hand, if $X \to Y \in G^+$, then LINEAR-CLOSURE(R, F, X) (run by G as input) adds the attributes of Y one-by-one to X. In every step when an attribute is added, some functional dependency $U \to V$ of G is used. This dependency is in one of $\pi_{R_i}(F)$'s, since G is the union of these. An easy induction on the number of functional dependencies used in procedure LINEAR-CLOSURE(R, F, X) shows that sooner or later Z becomes a subset of U, then applying the R_i-operation all attributes of V are added to Z. ■

16.3.4. Normal forms

The goal of transforming (decomposing) relational schemata into ***normal forms*** is to avoid the anomalies described in the previous section. Normal forms of many different strengths were introduced in the course of evolution of database theory, here only the ***Boyce-Codd*** normal formát (BCNF) and the ***third,*** furthermore ***fourth*** normal form (3NF and 4NF) are treated in detail, since these are the most important ones from practical point of view.

Boyce-Codd normal form

Definition 16.17 *Let R be relational schema, F be a family of functional dependencies over R. (R, F) is said to be in* ***Boyce-Codd normal form*** *if $X \to A \in F^+$ and $A \notin X$ implies that A is a superkey.*

The most important property of BCNF is that it eliminates redundancy. This is based on the following theorem whose proof is left to the Reader as an exercise (Exercise 16.3-8.).

Theorem 16.18 *Schema (R, F) is in BCNF iff for arbitrary attribute $A \in R$ and key $X \subset R$ there exists no $Y \subseteq R$, for which $X \to Y \in F^+$; $Y \to X \notin F^+$; $Y \to A \in F^+$ and $A \notin Y$.*

In other words, Theorem 16.18 states that "BCNF $\Longleftrightarrow$ There is no transitive dependence on keys". Let us assume that a given schema is not in BCNF, for example $C \to B$ and $B \to A$ hold, but $B \to C$ does not, then the same B value could occur besides many different C values, but at each occasion the same A value would be stored with it, which is redundant. Formulating somewhat differently, the meaning of BCNF is that (only) using functional dependencies an attribute value in a row cannot be predicted from other attribute

values. Indeed, assume that there exists a schema R, in which the value of an attribute can be determined using a functional dependency by comparison of two rows. That is, there exists two rows that agree on an attribute set X, differ on the set Y and the value of the remaining (unique) attribute A can be determined in one of the rows from the value taken in the other row.

X	Y	A
x	y_1	a
x	y_2	?

If the value ? can be determined by a functional dependency, then this value can only be a, the dependency is $Z \to A$, where Z is an appropriate subset of X. However, Z cannot be a superkey, since the two rows are distinct, thus R is not in BCNF.

3NF

Although BCNF helps eliminating anomalies, it is not true that every schema can be decomposed into subschemata in BCNF so that the decomposition is dependency preserving. As it was shown in Example 16.6., no proper decomposition of schema CSZ preserves the $CS \to Z$ dependency. At the same time, the schema is clearly not in BCNF, because of the $Z \to C$ dependency.

Since dependency preserving is important because of consistency checking of a database, it is practical to introduce a normal form that every schema has dependency preserving decomposition into that form, and it allows minimum possible redundancy. An attribute is called ***prime attribute,*** if it occurs in a key.

Definition 16.19 *The schema (R, F) is in **third normal form,** if whenever $X \to A \in F^+$, then either X is a superkey, or A is a prime attribute.*

The schema $SAIP$ of Example 16.3. with the dependencies $SI \to P$ and $S \to A$ is not in 3NF, since SI is the only key and so A is not a prime attribute. Thus, functional dependency $S \to A$ violates the 3NF property.

3NF is clearly weaker condition than BCNF, since "or A is a prime attribute" occurs in the definition. The schema CSZ in Example 16.6. is trivially in 3NF, because every attribute is prime, but it was already shown that it is not in BCNF.

Testing normal forms

Theoretically every functional dependency in F^+ should be checked whether it violates the conditions of BCNF or 3NF, and it is known that F^+ can be exponentially large in the size of F. Nevertheless, it can be shown that if the functional dependencies in F are of the form that the right hand side is a single attribute always, then it is enough to check violation of BCNF, or 3NF respectively, for dependencies of F. Indeed, let $X \to A \in F^+$ be a dependency that violates the appropriate condition, that is X is not a superkey and in case of 3NF, A is not prime. $X \to A \in F^+ \iff A \in X^+$. In the step when Closure(R,F,X) puts A into $X+$ (line 8) it uses a functional dependency $Y \to A$ from F that $Y \subset X^+$ and $A \notin Y$. This dependency is non-trivial and A is (still) not prime. Furthermore, if Y were a superkey, than by $R = Y^+ \subseteq (X^+)^+ = X^+$, X would also be a superkey. Thus, the functional dependency $Y \to A$ from F violates the condition of the normal form. The functional dependencies easily can be checked in polynomial time, since it is enough to calculate the closure of the left hand side of each dependency. This finishes checking for BCNF, because if the closure

of each left hand side is R, then the schema is in BCNF, otherwise a dependency is found that violates the condition. In order to test 3NF it may be necessary to decide about an attribute whether it is prime or not. However this problem is NP-complete, see Problem *16-4.*.

Lossless join decomposition into BCNF
Let (R, F) be a relational schema (where F is the set of functional dependencies). The schema is to be decomposed into union of subschemata $R_1, R_2, \ldots, R_k$, such that the decomposition has the lossless join property, furthermore each R_i endowed with the set of functional dependencies $\pi_{R_i}(F)$ is in BCNF. The basic idea of the decomposition is simple:

- If (R, F) is in BCNF, then ready.
- If not, it is decomposed into two proper parts (R_1, R_2), whose join is lossless.
- Repeat the above for R_1 and R_2.

In order to see that this works one has to show two things:

- If (R, F) is not in BCNF, then it has a lossless join decomposition into smaller parts.
- If a part of a lossless join decomposition is further decomposed, then the new decomposition has the lossless join property, as well.

Lemma 16.20 *Let (R, F) be a relational schema (where F is the set of functional dependencies), $\rho = (R_1, R_2, \ldots, R_k)$ be a lossless join decomposition of R. Furthermore, let $\sigma = (S_1, S_2)$ be a lossless join decomposition of R_1 with respect to $\pi_{R_1}(F)$. Then $(S_1, S_2, R_2, \ldots, R_k)$ is a lossless join decomposition of R.*

The proof of Lemma 16.20 is based on the associativity of natural join. The details are left to the Reader (Exercise 16.3-9.).

This can be applied for a simple, but unfortunately exponential time algorithm that decomposes a schema into subschemata of BCNF property. The projections in lines 4–5 of Naïv-BCNF(S, G) may be of exponential size in the length of the input. In order to decompose schema (R, F), the procedure must be called with parameters R, F. Procedure Naïv-BCNF(S, G) is recursive, S is the actual schema with set of functional dependencies G. It is assumed that the dependencies in G are of the form $X \to A$, where A is a single attribute.

Naïv-BCNF(S, G)

```
1 while there exists {X → A} ∈ G, that violates BCNF
2       do S1 ← {XA}
3          S2 ← S − A
4          G1 ← π_{S1}(G)
5          G2 ← π_{S2}(G)
6          return (Naïv-BCNF(S1, G1), Naïv-BCNF(S2, G2))
7 return S
```

However, if the algorithm is allowed overdoing things, that is to decompose a schema even if it is already in BCNF, then there is no need for projecting the dependencies. The procedure is based on the following two lemmae.

Lemma 16.21

1. *A schema of only two attributes is in BCNF.*

2. *If R is not in BCNF, then there exists two attributes A and B in R, such that $(R-AB) \to A$ holds.*

Proof. If the schema consists of two attributes, $R = AB$, then there are at most two possible non-trivial dependencies, $A \to B$ and $B \to A$. It is clear, that if some of them holds, then the left hand side of the dependency is a key, so the dependency does not violate the BCNF property. However, if none of the two holds, then BCNF is trivially satisfied.

On the other hand, let us assume that the dependency $X \to A$ violates the BCNF property. Then there must exists an attribute $B \in R - (XA)$, since otherwise X would be a superkey. For this B, $(R - AB) \to A$ holds. ■

Let us note, that the converse of the second statement of Lemma 16.21 is not true. It may happen that a schema R is in BCNF, but there are still two attributes $\{A, B\}$ that satisfy $(R - AB) \to A$. Indeed, let $R = ABC$, $F = \{C \to A, C \to B\}$. This schema is obviously in BCNF, nevertheless $(R - AB) = C \to A$.

The main contribution of Lemma 16.21 is that the projections of functional dependencies need not be calculated in order to check whether a schema obtained during the procedure is in BCNF. It is enough to calculate $(R - AB)^+$ for pairs $\{A, B\}$ of attributes, which can be done by LINEAR-CLOSURE(R, F, X) in linear time, so the whole checking is polynomial (cubic) time. However, this requires a way of calculating $(R - AB)^+$ without actually projecting down the dependencies. The next lemma is useful for this task.

Lemma 16.22 *Let $R_2 \subset R_1 \subset R$ and let F be the set of functional dependencies of scheme R. Then*

$$\pi_{R_2}(\pi_{R_1}(F)) = \pi_{R_2}(F).$$

The proof is left for the Reader (Exercise 16.3-10.). The method of lossless join BCNF decomposition is as follows. Schema R is decomposed into two subschemata. One is XA that is in BCNF, satisfying $X \to A$. The other subschema is $R - A$, hence by Theorem 16.14 the decomposition has the lossless join property. This is applied recursively to $R - A$, until such a schema is obtained that satisfies property 2 of Lemma 16.21. The lossless join property of this recursively generated decomposition is guaranteed by Lemma 16.20.

POLYNOMIAL-BCNF(R, F)

```
 1  Z ← R
 2  ▷ Z is the schema that is not known to be in BCNF during the procedure.
 3  ρ ← ∅
 4  while there exist A, B in Z, such that A ∈ (Z − AB)^+ and |Z| > 2
 5        do Let A and B be such a pair
 6           E ← A
 7           Y ← Z − B
 8           while there exist C, D in Y, such that C ∈ (Z − CD)^+
 9                 do Y ← Y − D
10                    E ← C
11           ρ ← ρ ∪ {Y}
12           Z ← Z − E
13  ρ ← ρ ∪ {Z}
14  return ρ
```

The running time of POLYNOMIAL-BCNF(R, F) is polynomial, in fact it can be bounded by $O(n^5)$, as follows. During each execution of the loop in lines 4–12 the size of Z is decreased by at least one, so the loop body is executed at most n times. $(Z - AB)^+$ is calculated in line 4 for at most $O(n^2)$ pairs that can be done in linear time using LINEAR-CLOSURE that results in $O(n^3)$ steps for each execution of the loop body. In lines 8–10 the size of Y is decreased in each iteration, so during each execution of lines 3–12, they give at most n iteration. The condition of the command **while** of line 8 is checked for $O(n^2)$ pairs of attributes, each checking is done in linear time. The running time of the algorithm is dominated by the time required by lines 8–10 that take $n \cdot n \cdot O(n^2) \cdot O(n) = O(n^5)$ steps altogether.

Dependency preserving decomposition into 3NF

We have seen already that its is not always possible to decompose a schema into subschemata in BCNF so that the decomposition is dependency preserving. Nevertheless, if only 3NF is required then a decomposition can be given using MINIMAL-COVER(R, F). Let R be a relational schema and F be the set of functional dependencies. Using MINIMAL-COVER(R, F) a minimal cover G of F is constructed. Let $G = \{X_1 \to A_1, X_2 \to A_2, \ldots, X_k \to A_k\}$.

Theorem 16.23 *The decomposition $\rho = (X_1A_1, X_2A_2, \ldots, X_kA_k)$ is dependency preserving decomposition of R into subschemata in 3NF.*

Proof. Since $G^+ = F^+$ and the functional dependency $X_i \to A_i$ is in $\pi_{R_i}(F)$, the decomposition preserves every dependency of F. Let us suppose indirectly, that the schema $R_i = X_iA_i$ is not in 3NF, that is there exists a dependency $U \to B$ that violates the conditions of 3NF. This means that the dependency is non-trivial and U is not a superkey in R_i and B is not a prime attribute of R_i. There are two cases possible. If $B = A_i$, then using that U is not a superkey $U \subsetneq X_i$ follows. In this case the functional dependency $U \to A_i$ contradicts to that $X_i \to A_i$ was a member of minimal cover, since its left hand side could be decreased. In the case when $B \neq A_i$, $B \in X_i$ holds. B is not prime in R_i, thus X_i is not a key, only a superkey. However, then X_i would contain a key Y such that $Y \subsetneq X_i$. Furthermore, $Y \to A_i$ would hold, as well, that contradicts to the minimality of G since the left hand side of $X_i \to A_i$ could be decreased. ■ If the decomposition needs to have the

lossless join property besides being dependency preserving, then ρ given in Theorem 16.23 is to be extended by a key X of R. Although it was seen before that it is not possible to list ***all*** keys in polynomial time, ***one*** can be obtained in a simple greedy way, the details are left to the Reader (Exercise 16.3-11.).

Theorem 16.24 *Let (R, F) be a relational schema, and let $G = \{X_1 \to A_1, X_2 \to A_2, \ldots, X_k \to A_k\}$ be a minimal cover of F. Furthermore, let X be a key in (R, F). Then the decomposition $\tau = (X, X_1A_1, X_2A_2, \ldots, X_kA_k)$ is a lossless join and dependency preserving decomposition of R into subschemata in 3NF.*

Proof. It was shown during the proof of Theorem 16.23 that the subschemata $R_i = X_iA_i$ are in 3NF for $i = 1, 2, \ldots, k$. There cannot be a non-trivial dependency in the subschema $R_0 = X$, because if it were, then X would not be a key, only a superkey.

The lossless join property of τ is shown by the use of Join-test(R, G, ρ) procedure. Note that it is enough to consider the minimal cover G of F. More precisely, we show that the row corresponding to X in the table will be all 0 after running Join-test(R, G, ρ). Let $A_1, A_2, \ldots, A_m$ be the order of the attributes of $R - X$ as Closure(R,G,X) inserts them into X^+. Since X is a key, every attribute of $R - X$ is taken during Closure(R,G,X). It will be shown by induction on i that the element in row of X and column of A_i is 0 after running Join-test(R, G, ρ).

The base case of $i = 0$ is obvious. Let us suppose that the statement is true for $i-$ and consider when and why A_i is inserted into X^+. In lines 6–8 of Closure(R,G,X) such a functional dependency $Y \to A_i$ is used where $Y \subseteq X \cup \{A_1, A_2, \ldots, A_{i-1}\}$. Then $Y \to A_i \in G$, $YA_i = R_j$ for some j. The rows corresponding to X and $YA_i = R_j$ agree in columns of X (all 0 by the induction hypothesis), thus the entries in column of A_i are equated by Join-test(R, G, ρ). This value is 0 in the row corresponding to $YA_i = R_j$, thus it becomes 0 in the row of X, as well. ■

It is interesting to note that although an arbitrary schema can be decomposed into subschemata in 3NF in polynomial time, nevertheless it is NP-complete to decide whether a given schema (R, F) is in 3NF, see Problem *16-4.*. However, the BCNF property can be decided in polynomial time. This difference is caused by that in order to decide 3NF property one needs to decide about an attribute whether it is prime. This latter problem requires the listing of all keys of a schema.

16.3.5. Multivalued dependencies

16.8. Example. Besides functional dependencies, some other dependencies hold in Example 16.1., as well. There can be several lectures of a subject in different times and rooms. Part of an instance of the schema could be the following.

Professor	Subject	Room	Student	Grade	Time
Caroline Doubtfire	Analysis	MA223	John Smith	A^-	Monday 8–10
Caroline Doubtfire	Analysis	CS456	John Smith	A^-	Wednesday 12–2
Caroline Doubtfire	Analysis	MA223	Ching Lee	A^+	Monday 8–10
Caroline Doubtfire	Analysis	CS456	Ching Lee	A^+	Wednesday 12–2

A set of values of Time and Room attributes, respectively, belong to each given value of Subject, and all other attribute values are repeated with these. Sets of attributes SR and StG are independent, that

is their values occur in each combination.

The set of attributes Y is said to be ***multivalued dependent*** on set of attributes X, in notation $X \twoheadrightarrow Y$, if for every value on X, there exists a set of values on Y that is not dependent in any way on the values taken in $R - X - Y$. The precise definition is as follows.

Definition 16.25 *The relational schema R satisfies the **multivalued dependency** $X \twoheadrightarrow Y$, if for every relation r of schema R and arbitrary tuples t_1, t_2 of r that satisfy $t_1[X] = t_2[X]$, there exists tuples $t_3, t_4 \in r$ such that*

- $t_3[XY] = t_1[XY]$
- $t_3[R - XY] = t_2[R - XY]$
- $t_4[XY] = t_2[XY]$
- $t_4[R - XY] = t_1[R - XY]$

holds.[1]

In Example 16.8. S↠TR holds.

Remark 16.26 *Functional dependency is **equality generating** dependency, that is from the equality of two objects it deduces the equality of other other two objects. On the other hand, multivalued dependency is **tuple generating dependency**, that is the existence of two rows that agree somewhere implies the existence of some other rows.*

There exists a sound and complete axiomatisation of multivalued dependencies similar to the Armstrong-axioms of functional dependencies. Logical implication and inference can be defined analogously. The multivalued dependency $X \twoheadrightarrow Y$ is ***logically implied*** by the set M of multivalued dependencies, in notation $M \models X \twoheadrightarrow Y$, if every relation that satisfies all dependencies of M also satisfies $X \twoheadrightarrow Y$.

Note, that $X \rightarrow Y$ implies $X \twoheadrightarrow Y$. The rows t_3 and t_4 of Definition 16.25 can be chosen as $t_3 = t_2$ and $t_4 = t_1$, respectively. Thus, functional dependencies and multivalued dependencies admit a common axiomatisation. Besides Armstrong-axioms (A1)–(A3), five other are needed. Let R be a relational schema.

(A4) ***Complementation***: $\{X \twoheadrightarrow Y\} \models X \twoheadrightarrow (R - X - Y)$.

(A5) ***Extension***: If $X \twoheadrightarrow Y$ holds, and $V \subseteq W$, then $WX \twoheadrightarrow VY$.

(A6) ***Transitivity***: $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \models X \twoheadrightarrow (Z - Y)$.

(A7) $\{X \rightarrow Y\} \models X \twoheadrightarrow Y$.

(A8) If $X \twoheadrightarrow Y$ holds, $Z \subseteq Y$, furthermore for some W disjoint from Y $W \rightarrow Z$ holds, then $X \rightarrow Z$ is true, as well.

Beeri, Fagin and Howard proved that (A1)–(A8) is sound and complete system of axioms for functional and Multivalued dependencies together. Proof of soundness is left for the Reader (Exercise 16.3-12.), the proof of the completeness exceeds the level of this book. The rules of Lemma 16.2 are valid in exactly the same way as when only functional dependencies were considered. Some further rules are listed in the next Proposition.

[1] It would be enough to require the existence of t_3, since the existence of t_4 would follow. However, the symmetry of multivalued dependency is more apparent in this way.

Proposition 16.27 *The followings are true for multivalued dependencies.*

1. ***Union rule:*** $\{X \twoheadrightarrow Y, X \twoheadrightarrow Z\} \models X \twoheadrightarrow YZ$.
2. ***Pseudotransitivity:*** $\{X \twoheadrightarrow Y, WY \twoheadrightarrow Z\} \models WX \twoheadrightarrow (Z - WY)$.
3. ***Mixed pseudotransitivity:*** $\{X \twoheadrightarrow Y, XY \rightarrow Z\} \models X \rightarrow (Z - Y)$.
4. ***Decomposition rule*** *for multivalued dependencies: ha* $X \twoheadrightarrow Y$ *and* $X \twoheadrightarrow Z$ *holds, then* $X \twoheadrightarrow (Y \cap Z)$, $X \twoheadrightarrow (Y - Z)$ *and* $X \twoheadrightarrow (Z - Y)$ *holds, as well.*

Th proof of Proposition 16.27 is left for the Reader (Exercise 16.3-13.).

Dependency basis

Important difference between functional dependencies and multivalued dependencies is that $X \rightarrow Y$ immediately implies $X \rightarrow A$ for all A in Y, however $X \twoheadrightarrow A$ is deduced by the decomposition rule for multivalued dependencies from $X \twoheadrightarrow Y$ only if there exists a set of attributes Z such that $X \twoheadrightarrow Z$ and $Z \cap Y = A$, or $Y - Z = A$. Nevertheless, the following theorem is true.

Theorem 16.28 *Let R be a relational schema, $X \subset R$ be a set of attributes. Then there exists a partition $Y_1, Y_2, \ldots, Y_k$ of the set of attributes $R - X$ such that for $Z \subseteq R - X$ the multivalued dependency $X \twoheadrightarrow Z$ holds if and only if Z is the union of some Y_i's.*

Proof. We start from the one-element partition $W_1 = R - X$. This will be refined successively, while the property that $X \twoheadrightarrow W_i$ holds for all W_i in the actual decomposition, is kept. If $X \twoheadrightarrow Z$ and Z is not a union of some of the W_i's, then replace every W_i such that neither $W_i \cap Z$ nor $W_i - Z$ is empty by $W_i \cap Z$ and $W_i - Z$. According to the decomposition rule of Proposition 16.27, both $X \twoheadrightarrow (W_i \cap Z)$ and $X \twoheadrightarrow (W_i - Z)$ hold. Since $R - X$ is finite, the refinement process terminates after a finite number of steps, that is for all Z such that $X \twoheadrightarrow Z$ holds, Z is the union of some blocks of the partition. In order to complete the proof one needs to observe only that by the union rule of Proposition 16.27, the union of some blocks of the partition depends on X in multivalued way. ■

Definition 16.29 *The partition $Y_1, Y_2, \ldots, Y_k$ constructed in Theorem 16.28 from a set D of functional and multivalued dependencies is called the* ***dependency basis*** *of X (with respect to D).*

16.9. Example. Consider the familiar schema

R(**P**rofessor,**Su**bject,**R**oom,**St**udent,**G**rade,**T**ime)

of Examples 16.1. and 16.8.. **Su→→RT** was shown in Example 16.8.. By the complementation rule **Su→→PStG** follows. **Su→P** is also known. This implies by axiom (A7) that **Su→→P**. By the decomposition rule **Su→→Stg** follows. It is easy to see that no other one-element attribute set is determined by **Su** via multivalued dependency. Thus, the dependency basis of **Su** is the partition{**P**,**RT**,**StG**}.

We would like to compute the set D^+ of logical consequences of a given set D of functional and multivalued dependencies. One possibility is to apply axioms (A1)–(A8) to extend the set of dependencies repeatedly, until no more extension is possible. However, this could be an exponential time process in the size of D. One cannot expect any better, since it

was shown before that even D^+ can be exponentially larger than D. Nevertheless, in many applications it is not needed to compute the whole set D^+, one only needs to decide whether a given functional dependency $X \to Y$ or multivalued dependency $X \twoheadrightarrow Y$ belongs to D^+ or not. In order to decide about a multivalued dependency $X \twoheadrightarrow Y$, it is enough to compute the dependency basis of X, then to check whether $Z - X$ can be written as a union of some blocks of the partition. The following is true.

Theorem 16.30 (Beeri). *In order to compute the dependency basis of a set of attributes X with respect to a set of dependencies D, it is enough to consider the following set M of multivalued dependencies:*

1. *All multivalued dependencies of D and*
2. *for every $X \to Y$ in D the set of multivalued dependencies $X \twoheadrightarrow A_1, X \twoheadrightarrow A_2, \ldots, X \twoheadrightarrow A_k$, where $Y = A_1 A_2 \ldots A_k$, and the A_i's are single attributes.*

The only thing left is to decide about functional dependencies based on the dependency basis. Closure(R,F,X) works correctly only if multivalued dependencies are not considered. The next theorem helps in this case.

Theorem 16.31 (Beeri). *Let us assume that $A \notin X$ and the dependency basis of X with respect to the set M of multivalued dependencies obtained in Theorem 16.30 is known. $X \to A$ holds if and only if*

1. *A forms a single element block in the partition of the dependency basis, and*
2. *There exists a set Y of attributes that does not contain A, $Y \to Z$ is an element of the originally given set of dependencies D, furthermore $A \in Z$.*

Based on the observations above, the following polynomial time algorithm can be given to compute the dependency basis of a set of attributes X.

Dependency-basis(R, M, X)

```
1 S ← {R − X}                          ▷ The collection of sets in the dependency basis is S.
2 repeat
3       for all V ↠ W ∈ M
4           do if there exists Y ∈ S such that Y ∩ W ≠ ∅ ∧ Y ∩ V = ∅
5                 then S ← S − {{Y}} ∪ {{Y ∩ W}, {Y − W}}
6 until S does not change
7 return S
```

It is immediate that if $\mathcal{S}$ changes in lines 3–5. of Dependency-basis(R, M, X), then some block of the partition is cut by the algorithm. This implies that the running time is a polynomial function of the sizes of M and R. In particular, by careful implementation one can make this polynomial to $O(|M| \cdot |R|^3)$, see Problem 16-5..

Fourth normal form 4NF

The Boyce-Codd normal form can be generalised to the case where multivalued dependencies are also considered besides functional dependencies, and one needs to get rid of the redundancy caused by them.

Definition 16.32 *Let R be a relational schema, D be a set of functional and multivalued dependencies over R. R is in **fourth normal form (4NF)**, if for arbitrary multivalued dependency $X \twoheadrightarrow Y \in D^+$ for which $Y \not\subseteq X$ and $R \neq XY$, holds that X is superkey in R.*

Observe that 4NF$\Longrightarrow$BCNF. Indeed, if $X \to A$ violated the BCNF condition, then $A \notin X$, furthermore XA could not contain all attributes of R, because that would imply that X is a superkey. However, $X \to A$ implies $X \twoheadrightarrow A$ by (A8), which in turn would violate the 4NF condition.

Schema R together with set of functional and multivalued dependencies D can be decomposed into $\rho = (R_1, R_2, \ldots, R_k)$, where each R_i is in 4NF and the decomposition has the lossless join property. The method follows the same idea as the decomposition into BCNF subschemata. If schema S is not in 4NF, then there exists a multivalued dependency $X \twoheadrightarrow Y$ in the projection of D onto S that violates the 4NF condition. That is, X is not a superkey in S, Y neither is empty, nor is a subset of X, furthermore the union of X and Y is not S. It can be assumed without loss of generality that X and Y are disjoint, since $X \twoheadrightarrow (Y - X)$ is implied by $X \twoheadrightarrow Y$ using (A1), (A7) and the decomposition rule. In this case S can be replaced by subschemata $S_1 = XY$ and $S_2 = S - Y$, each having a smaller number of attributes than S itself, thus the process terminates in finite time.

Two things has to be dealt with in order to see that the process above is correct.

- Decomposition S_1, S_2 has the lossless join property.
- How can the projected dependency set $\pi_S(D)$ be computed?

The first problem is answered by the following theorem.

Theorem 16.33 *The decomposition $\rho = (R_1, R_2)$ of schema R has the lossless join property with respect to a set of functional and multivalued dependencies D iff*

$$(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2).$$

Proof. The decomposition $\rho = (R_1, R_2)$ of schema R has the lossless join property iff for any relation r over the schema R that satisfies all dependencies from D holds that if μ and ν are two tuples of r, then there exists a tuple φ satisfying $\varphi[R_1] = \mu[R_1]$ and $\varphi[R_2] = \nu[R_2]$, then it is contained in r. More precisely, φ is the natural join of the projections of μ on R_1 and of ν on R_2, respectively, which exist iff $\mu[R_1 \cap R_2] = \nu[R_1 \cap R_2]$. Thus the fact that φ is always contained in r is equivalent with that $(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$. ■

To compute the projection $\pi_S(D)$ of the dependency set D one can use the following theorem of Aho, Beeri and Ullman. $\pi_S(D)$ is the set of multivalued dependencies that are logical implications of D and use attributes of S only.

Theorem 16.34 (Aho, Beeri és Ullman). *$\pi_S(D)$ consists of the following dependencies:*

- *For all $X \to Y \in D^+$, if $X \subseteq S$, then $X \to (Y \cap S) \in \pi_S(D)$.*
- *For all $X \twoheadrightarrow Y \in D^+$, if $X \subseteq S$, then $X \twoheadrightarrow (Y \cap S) \in \pi_S(D)$.*

Other dependencies cannot be derived from the fact that D holds in R.

Unfortunately this theorem does not help in computing the projected dependencies in polynomial time, since even computing D^+ could take exponential time. Thus, the algorithm of 4NF decomposition is not polynomial either, because the 4NF condition must be checked with respect to the projected dependencies in the subschemata. This is in deep contrast with

the case of BCNF decomposition. The reason is, that to check BCNF condition one does not need to compute the projected dependencies, only closures of attribute sets need to be considered according to Lemma 16.21.

Exercises

16.3-1 Are the following inference rules sound?

a. If $XW \to Y$ and $XY \to Z$, then $X \to (Z - W)$.

b. If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow Z$.

c. If $X \twoheadrightarrow Y$ and $XY \to Z$, then $X \to Z$.

16.3-2 Prove Theorem 16.30, that is show the following. Let D be a set of functional and multivalued dependencies, and let $m(D) = \{X \twoheadrightarrow Y\colon X \twoheadrightarrow Y \in D\} \cup \{X \twoheadrightarrow A\colon A \in Y \text{ for some } X \to Y \in D\}$. Then

a. $D \models X \to Y \Longrightarrow m(D) \models X \twoheadrightarrow Y$, and

b. $D \models X \twoheadrightarrow Y \iff m(D) \models X \twoheadrightarrow Y$.

Hint. Use induction on the inference rules to prove b.

16.3-3 Consider the database of an investment firm, whose attributes are as follows: B (stockbroker), O (office of stockbroker), I (investor), S (stock), A (amount of stocks of the investor), D (dividend of the stock). The following functional dependencies are valid: $S \to D$, $I \to B$, $IS \to A$, $B \to O$.

a. Determine a key of schema $R = BOISAD$.

b. How many keys are in schema R?

c. Give a lossless join decomposition of R into subschemata in BCNF.

d. Give a dependency preserving and lossless join decomposition of R into subschemata in 3NF.

16.3-4 The schema R of Exercise 16.3-3. is decomposed into subschemata SD, IB, ISA and BO. Does this decomposition have the lossless join property?

16.3-5 Assume that schema R of Exercise 16.3-3. is represented by ISA, IB, SD and ISO subschemata. Give a minimal cover of the projections of dependencies given in Exercise 16.3-3.. Exhibit a minimal cover for the union of the sets of projected dependencies. Is this decomposition dependency preserving?

16.3-6 Let the functional dependency $S \to D$ of Exercise 16.3-3. be replaced by the multivalued dependency $S \twoheadrightarrow D$. That is , D represents the stock's dividend "history".

a. Compute the dependency basis of I.

b. Compute the dependency basis of BS.

c. Give a decomposition of R into subschemata in 4NF.

16.3-7 Consider the decomposition $\rho = \{R_1, R_2, \ldots, R_k\}$ of schema R. Let $r_i = \pi_{R_i}(r)$, furthermore $m_\rho(r) = \bowtie_{i=1}^{k} \pi_{R_i}(r)$. Prove:

a. $r \subseteq m_\rho(r)$.

b. If $s = m_\rho(r)$, then $\pi_{R_i}(s) = r_i$.

c. $m_\rho(m_\rho(r)) = m_\rho(r)$.

16.3-8 Prove that schema (R, F) is in BCNF iff for arbitrary $A \in R$ and key $X \subset R$, it holds that there exists no $Y \subseteq R$, for which $X \to Y \in F^+$; $Y \to X \notin F^+$; $Y \to A \in F^+$ and $A \notin Y$.
16.3-9 Prove Lemma 16.20.
16.3-10 Let us assume that $R_2 \subset R_1 \subset R$ and the set of functional dependencies of schema R is F. Prove that $\pi_{R_2}(\pi_{R_1}(F)) = \pi_{R_2}(F)$.
16.3-11 Give a $O(n^2)$ running time algorithm to find a key of the relational schema (R, F). *Hint.* Use that R is superkey and each superkey contains a key. Try to drop attributes from R one-by-one and check whether the remaining set is still a key.
16.3-12 Prove that axioms (A1)–(A8) are sound for functional and multivalued dependencies.
16.3-13 Derive the four inference rules of Proposition 16.27 from axioms (A1)–(A8).

16.4. Generalised dependencies

Two such dependencies will be discussed in this section that are generalizations of the previous ones, however cannot be axiomatised with axioms similar to (A1)–(A8).

16.4.1. Join dependencies

Theorem 16.33 states that multivalued dependency is equivalent with that some decomposition the schema into two parts has the lossless join property. Its generalisation is the ***join dependency***.

Definition 16.35 *Let R be a relational schema and let $R = \bigcup_{i=1}^{k} X_i$. The relation r belonging to R is said to satisfy the* ***join dependency***

$$\bowtie[X_1, X_2, \ldots, X_k]$$

if

$$r = \bowtie_{i=1}^{k} \pi_{X_i}(r).$$

In this setting r satisfies multivalued dependency $X \twoheadrightarrow Y$ iff it satisfies the join dependency $\bowtie[XY, X(R-Y)]$. The join dependency $\bowtie[X_1, X_2, \ldots, X_k]$ expresses that the decomposition $\rho = (X_1, X_2, \ldots, X_k)$ has the lossless join property. One can define the ***fifth normal form, 5NF***.

Definition 16.36 *The relational schema R is in* ***fifth normal form,*** *if it is in 4NF and has no non-trivial join dependency.*

The fifth normal form has theoretical significance primarily. The schemata used in practice usually have ***primary keys***. Using that the schema could be decomposed into subschemata of two attributes each, where one of the attributes is a superkey in every subschema.

16.10. Example. Consider the database of clients of a bank (Client-number,Name,Address,accountBalance). Here C is unique identifier, thus the schema could be decomposed into (CN,CA,CB), which has the lossless join property. However, it is not worth doing so, since no storage place can be saved, furthermore no anomalies are avoided with it.

There exists an ***axiomatisation*** of a dependency system if there is a finite set of inference rules that is sound and complete, i.e. logical implication coincides with being derivable by using the inference rules. For example, the Armstrong-axioms give an axiomatisation of functional dependencies, while the set of rules (A1)–(A8) is the same for functional and multivalued dependencies considered together. Unfortunately, the following negative result is true.

Theorem 16.37 *The family of join dependencies has no finite axiomatisation.*

In contrary to the above, Abiteboul, Hull and Vianu show in their book that the logical implication problem can be decided by an algorithm for the family of functional and join dependencies taken together. The complexity of the problem is as follows.

Theorem 16.38

- *It is NP-complete to decide whether a given join dependency is implied by another given join dependency and a functional dependency.*
- *It is NP-hard to decide whether a given join dependency is implied by given set of multivalued dependencies.*

16.4.2. Branching dependencies

A generalisation of functional dependencies is the family of ***branching dependencies***. Let us assume that $A, B \subset R$ and there exists no $q+1$ rows in relation r over schema R, such that they contain at most p distinct values in columns of A, but all $q+1$ values are pairwise distinct in some column of B. Then B is said to be (p,q)***-dependent*** on A, in notation $A \overset{p,q}{\longrightarrow} B$. In particular, $A \overset{1,1}{\longrightarrow} B$ holds if and only if functional dependency $A \to B$ holds.

16.11. Example.
Consider the database of the trips of an international transport truck.

- One trip: four distinct countries.
- One country has at most five neighbours.
- There are 30 countries to be considered.

Let x_1, x_2, x_3, x_4 be the attributes of the countries reached in a trip. In this case $x_i \overset{1,1}{\longrightarrow} x_{i+1}$ does not hold, however another dependency is valid:

$$x_i \overset{1,5}{\longrightarrow} x_{i+1}.$$

The storage space requirement of the database can be significantly reduced using these dependencies. The range of each element of the original table consists of 30 values, names of countries or some codes of them (5 bits each, at least). Let us store a little table ($30 \times 5 \times 5 = 750$ bits) that contains a numbering of the neighbours of each country, which assigns to them the numbers 0,1,2,3,4 in some order. Now we can replace attribute x_2 by these numbers (x_2^*), because the value of x_1 gives the starting

country and the value of x_2^* determines the second country with the help of the little table. The same holds for the attribute x_3, but we can decrease the number of possible values even further, if we give a table of numbering the possible third countries for each x_1, x_2 pair. In this case, the attribute x_3^* can take only 4 different values. The same holds for x_4, too. That is, while each element of the original table could be encoded by 5 bits, now for the cost of two little auxiliary tables we could decrease the length of the elements in the second column to 3 bits, and that of the elements in the third and fourth columns to 2 bits.

The (p, q)-closure of an attribute set $X \subset R$ can be defined:

$$C_{p,q}(X) = \{A \in R \colon X \xrightarrow{p,q} A\}.$$

In particular, $C_{1,1}(X) = X^+$. In case of branching dependencies even such basic questions are hard as whether there exists an ***Armstrong-relation*** for a given family of dependencies.

Definition 16.39 *Let R be a relational schema, F be a set of dependencies of some dependency family $\mathcal{F}$ defined on R. A relation r over schema R is **Armstrong-relation** for F, if the set of dependencies from $\mathcal{F}$ that r satisfies is exactly F, that is $F = \{\sigma \in \mathcal{F} \colon r \models \sigma\}$.*

Armstrong proved that for an arbitrary set of functional dependencies F there exists Armstrong-relation for F^+. The proof is based on the three properties of closures of attributes sets with respect to F, listed in Exercise 16.2-1. For branching dependencies only the first two holds in general.

Lemma 16.40 *Let $0 < p \leq q$, furthermore let R be a relational schema. For $X, Y \subseteq R$ one has*

1. $X \subseteq C_{p,q}(X)$ *and*
2. $X \subseteq Y \Longrightarrow C_{p,q}(X) \subseteq C_{p,q}(Y)$.

There exists such $C \colon 2^R \to 2^R$ mapping and natural numbers p, q that there exists no Armstrong-relation for C in the family if (p, q)-dependencies.

Grant Minker investigated ***numerical dependencies*** that are similar to branching dependencies. For attribute sets $X, Y \subseteq R$ the dependency $X \xrightarrow{k} Y$ holds in a relation r over schema R if for every tuple value taken on the set of attributes X, there exists at most k distinct tuple values taken on Y. This condition is stronger than that of $X \xrightarrow{1,k} Y$, since the latter only requires that in each column of Y there are at most k values, independently of each other. That allows $k^{|Y-X|}$ different Y projections. Numerical dependencies were axiomatised in some special cases, based on that Katona showed that branching dependencies have no finite axiomatisation. It is still an open problem whether logical implication is algorithmically decidable amongst branching dependencies.

Exercises

16.4-1 Prove Theorem 16.38.

16.4-2 Prove Lemma 16.40.

16.4-3 Prove that if $p = q$, then $C_{p,p}\left(C_{p,p}(X)\right) = C_{p,p}(X)$ holds besides the two properties of Lemma 16.40.

16.4-4 A $C \colon 2^R \to 2^R$ mapping is called a ***closure***, if it satisfies the two properties of

Lemma 16.40 and and the third one of Exercise 16.4-3.. Prove that if $C\colon 2^R \to 2^R$ is a closure, and F is the family of dependencies defined by $X \to Y \iff Y \subseteq C(X)$, then there exists an Armstrong-relation for F in the family of $(1,1)$-dependencies (functional dependencies) and in the family of $(2,2)$-dependencies, respectively.

16.4-5 Let C be the closure defined by

$$C(X) = \begin{cases} X, \text{ if } |X| < 2 \\ R \text{ otherwise.} \end{cases}$$

Prove that there exists no Armstrong-relation for C in the family of (n,n)-dependencies, if $n > 2$.

Problems

16-1. External attributes

Maier calls attribute A an ***external attribute*** in the functional dependency $X \to Y$ with respect to the family of dependencies F over schema R, if the following two conditions hold:

1. $(F - \{X \to Y\}) \cup \{X \to (Y - A)\} \models X \to Y$, or
2. $(F - \{X \to Y\}) \cup \{(X - A) \to Y\} \models X \to Y$.

Design an $O(n^2)$ running time algorithm, whose input is schema (R, F) and output is a set of dependencies G equivalent with F that has no external attributes.

16-2. The order of the elimination steps in the construction of minimal cover is important

In the procedure Minimal-cover(R, F) the set of functional dependencies was changed in two ways: either by dropping redundant dependencies, or by dropping redundant attributes from the left hand sides of the dependencies. If the latter method is used first, until there is no more attribute that can be dropped from some left hand side, then the first method, this way a minimal cover is obtained really, according to Proposition 16.6. Prove that if the first method applied first and then the second, until there is no more possible applications, respectively, then the obtained set of dependencies is not necessarily a minimal cover of F.

16-3. BCNF subschema

Prove that the following problem is coNP-complete: Given a relational schema R with set of functional dependencies F, furthermore $S \subset R$, decide whether $(S, \pi_S(F))$ is in BCNF.

16-4. 3NF is hard to recognise

Let (R, F) be a relational schema, where F is the system of functional dependencies.

The k ***size key problem*** is the following: given a natural number k, determine whether there exists a key of size at most k.

The ***prime attribute problem*** is the following: for a given $A \in R$, determine whether it is a prime attribute.

a. Prove that the k size key problem is NP-complete. *Hint.* Reduce the ***vertex cover*** problem to the prime attribute problem.

b. Prove that the prime attribute problem is NP-complete by reducing the k size key problem to it.

c. Prove that determining whether the relational schema (R, F) is in 3NF is NP-complete. *Hint.* Reduce the prime attribute problem to it.

***16-5. Running time of* DEPENDENCY-BASIS**
Give an implementation of procedure DEPENDENCY-BASIS, whose running time is $O(|M| \cdot |R|^3)$.

Chapter notes

The relational data model was introduced by Codd [28] in 1970. Functional dependencies were treated in his paper of 1972 [32], their axiomatisation was completed by Armstrong [5]. The logical implication problem for functional dependencies were investigated by Beeri and Bernstein [11], furthermore Maier [80]. Maier also treats the possible definitions of minimal covers, their connections and the complexity of their computations in that paper. Maier, Mendelzon and Sagiv found method to decide logical implications among general dependencies [81]. Beeri, Fagin and Howard proved that axiom system (A1)–(A8) is sound and complete for functional and multivalued dependencies taken together [13]. Yu and Johnson [128] constructed such relational schema, where $|F| = k$ and the number of keys is $k!$. Békéssy and Demetrovics [16] gave a simple and beautiful proof for the statement, that from k functional dependencies at most $k!$ keys can be obtained, thus Yu and Johnson's construction is extremal.

Armstrong-relations were introduced and studied by Fagin [47, 48], furthermore by Beeri, Fagin, Dowd and Statman [12].

Multivalued dependencies were independently discovered by Zaniolo [130], Fagin [46] and Delobel [39].

The necessity of normal forms was recognised by Codd while studying update anomalies [30, 31]. The Boyce-Codd normal form was introduced in [29]. The definition of the third normal form used in this chapter was given by Zaniolo [131]. Complexity of decomposition into subschemata in certain normal forms was studied by Lucchesi and Osborne [79], Beeri and Bernstein [11], furthermore Tsou and Fischer [112].

Theorems 16.30 and 16.31 are results of Beeri [10]. Theorem 16.34 is from a paper of Aho, Beeri és Ullman [3].

Theorems 16.37 and 16.38 are from the book of Abiteboul, Hull and Vianu [1], the non-existence of finite axiomatisation of join dependencies is Petrov's result [91].

Branching dependencies were introduced by Demetrovics, Katona and Sali, they studied existence of Armstrong-relations and the size of minimal Armstrong-relations [40, 41, 42, 96]. Katona showed that there exists no finite axiomatisation of branching dependencies in (ICDT'92 Berlin, invited talk) but never published.

Possibilities of axiomatisation of numerical dependencies were investigated by Grant and Minker [58, 59].

Good introduction of the concepts of this chapter can be found in the books of Abiteboul, Hull and Vianu [1], Ullman [114] furthermore Thalheim [111], respectively.

17. Human-Computer Interaction

18. Memory Management

19. Scientific computing

This title refers to a fast developing interdisciplinary area between mathematics, computers and applications. The subject is also often called as Computational Science and Engineering. Its aim is the efficient use of computer algorithms to solve engineering and scientific problems. One can say with a certain simplification that our subject is related to numerical mathematics, software engineering, computer graphics and applications. Here we can deal only with some basic elements of the subject such as the fundamentals of the *floating point computer arithmetic*, *error analysis*, the basic numerical methods of *linear algebra* and related mathematical software.

19.1. Floating point arithmetic and error analysis

19.1.1. Classical error analysis

Let x be the exact value and let a be an approximation of x ($a \approx x$). The error of the approximation a is defined by the formula $\Delta a = x - a$ (sometimes with opposite sign). The quantity $\delta a \geq 0$ is called an ***(absolute) error (bound)*** of approximation a, if $|x - a| = |\Delta a| \leq \delta a$. For example, the error of the approximation $\sqrt{2} \approx 1.41$ is at most 0.01. In other words, the error bound of the approximation is 0.01. The quantities x and a (and accordingly Δa and δa) may be vectors or matrices. In such cases the absolute value and relation operators must be understood componentwise. We also measure the error by using matrix and vector norms. In such cases, the quantity $\delta a \in \mathbb{R}$ is an error bound, if the inequality $\|\Delta a\| \leq \delta a$ holds.

The absolute error bound can be irrelevant in many cases. For example, an approximation with error bound 0.05 has no value in estimating a quantity of order 0.001. The goodness of an approximation is measured by the ***relative error*** $\delta a/|x|$ ($\delta a/\|x\|$ for vectors and matrices), which compares the error bound to the approximated quantity. Since the exact value is generally unknown, we use the approximate relative error $\delta a/|a|$ ($\delta a/\|a\|$). The committed error is proportional to the quantity $(\delta a)^2$, which can be neglected, if the absolute value (norm) of x and a is much greater than $(\delta a)^2$. The relative error is often expressed in percentages.

In practice, the (absolute) error bound is used as a substitute for the generally unknown true error.

In the *classical error analysis* we assume input data with given error bounds, exact computations (operations) and seek for the error bound of the final result. Let x and y be exact values with approximations a and b, respectively. Assume that the absolute error bounds of approximations a and b are δa and δb, respectively. Using the classical error analysis approach we obtain the following error bounds for the four basic arithmetic operations:

$$\begin{aligned}
\delta(a+b) &= \delta a + \delta b, & \frac{\delta(a+b)}{|a+b|} &= \max\left\{\frac{\delta a}{|a|}, \frac{\delta b}{|b|}\right\} \quad (ab>0),\\
\delta(a-b) &= \delta a + \delta b, & \frac{\delta(a-b)}{|a-b|} &= \frac{\delta a+\delta b}{|a-b|} \quad (ab>0),\\
\delta(ab) &\approx |a|\,\delta b + |b|\,\delta a & \frac{\delta(ab)}{|ab|} &\approx \frac{\delta a}{|a|} + \frac{\delta b}{|b|} \quad (ab\neq 0),\\
\delta(a/b) &\approx \frac{|a|\,\delta b + |b|\,\delta a}{|b|^2} & \frac{\delta(a/b)}{|a/b|} &\approx \frac{\delta a}{|a|} + \frac{\delta b}{|b|} \quad (ab\neq 0).
\end{aligned}$$

We can see that the division with a number near to 0 can make the absolute error arbitrarily big. Similarly, if the result of subtraction is near to 0, then its relative error can become arbitrarily big. One has to avoid these cases. Especially the subtraction operation can be quite dangerous.

19.1. Example. Calculate the quantity $\sqrt{1996}-\sqrt{1995}$ with approximations $\sqrt{1996}\approx 44.67$ and $\sqrt{1995}\approx 44.66$ whose common absolute and relative error bounds are 0.01 and 0.022%, respectively. One obtains the approximate value $\sqrt{1996}-\sqrt{1995}\approx 0.01$, whose relative error bound is

$$\frac{0.01+0.01}{0.01}=2,$$

that is 200%. The true relative error is about 10.66%. Yet it is too big, since it is approximately 5×10^2 times bigger than the relative error of the initial data. We can avoid the subtraction operation by using the following trick

$$\sqrt{1996}-\sqrt{1995}=\frac{1996-1995}{\sqrt{1996}+\sqrt{1995}}=\frac{1}{\sqrt{1996}+\sqrt{1995}}\approx\frac{1}{89.33}\approx 0.01119.$$

Here the nominator is exact, while the absolute error of the denominator is 0.02. Hence the relative error (bound) of the quotient is about $0.02/89.33\approx 0.00022=0.022\%$. The latter result is in agreement with the relative error of the initial data and it is substantially smaller than the one obtained with direct subtraction operation.

The first order error terms of twice differentiable functions can be obtained by their first order *Taylor polynomial*:

$$\begin{aligned}
\delta(f(a)) &\approx \left|f'(a)\right|\delta a, \quad f:\mathbb{R}\to\mathbb{R},\\
\delta(f(a)) &\approx \sum_{i=1}^{n}\left|\frac{\partial f(a)}{\partial x_i}\right|\delta a_i, \quad f:\mathbb{R}^n\to\mathbb{R}.
\end{aligned}$$

The numerical sensitivity of functions at a given point is characterized by the ***condition number***, which is the ratio of the relative errors of approximate function value and the input

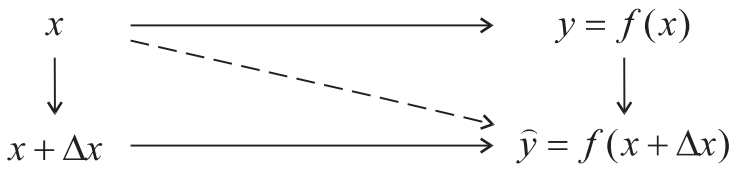


Figure 19.1. Forward and backward error

data (the *Jacobian matrix* of functions $F : \mathbb{R}^n \to \mathbb{R}^m$ is denoted by $F'(a)$ at the point $a \in \mathbb{R}^n$):

$$c(f,a) = \frac{|f'(a)|\,|a|}{|f(a)|}, \quad f : \mathbb{R} \to \mathbb{R},$$
$$c(F,a) = \frac{\|a\|\,\|F'(a)\|}{\|F(a)\|}, \quad F : \mathbb{R}^n \to \mathbb{R}^m.$$

We can consider the condition number as the magnification number of the input relative error. Therefore the functions is considered ***numerically stable*** (or ***well-conditioned***) at the point a, if $c(f,a)$ is „small". Otherwise f is considered as ***numerically unstable*** (***ill-conditioned***). The condition number depends on the point a. A function can be well-conditioned at point a, while it is ill-conditioned at point b. The term „small" is relative. It depends on the problem, the computer and the required precision.

The condition number of matrices can be defined as the upper bound of a function condition number. Let us define the mapping $F : \mathbb{R}^n \to \mathbb{R}^n$ by the solution of the equation $Ay = x$ ($A \in \mathbb{R}^{n\times n}$, $\det(A) \neq 0$), that is, let $F(x) = A^{-1}x$. Then $F' \equiv A^{-1}$ and

$$c(F,a) = \frac{\|a\|\left\|A^{-1}\right\|}{\left\|A^{-1}a\right\|} = \frac{\|Ay\|\left\|A^{-1}\right\|}{\|y\|} \leq \|A\|\left\|A^{-1}\right\| \quad (Ay = a).$$

The upper bound of the right side is called the ***condition number of the matrix*** A. This bound is sharp, since there exists a vector $a \in \mathbb{R}^n$ such that $c(F,a) = \|A\|\left\|A^{-1}\right\|$.

19.1.2. Forward and backward errors

Let us investigate the calculation of the function value $f(x)$. If we calculate the approximation $\hat{y}$ instead of the exact value $y = f(x)$, then the forward error $\Delta y = \hat{y} - y$. If for a value $x + \Delta x$ the equality $\hat{y} = f(x + \Delta x)$ holds, that is, $\hat{y}$ is the exact function value of the perturbed input data $\hat{x} = x + \Delta x$, then Δx is called the ***backward error***. The connection of the two concepts is shown on the Figure 19.1.

The continuous line shows exact value, while the dashed one indicates computed value. The analysis of the backward error is called the ***backward error analysis*** . If there exist more than one backward error, then the estimation of the smallest one is the most important.

An algorithm for computing the value $y = f(x)$ is called ***backward stable***, if for any x it gives a computed value $\hat{y}$ with small backward error Δx. Again, the term „small" is relative to the problem environment.

The connection of the forward and backward errors is described by the approximate

thumb rule

$$\frac{\delta\hat{y}}{|y|} \lessapprox c\,(f,x)\,\frac{\delta\hat{x}}{|x|}, \tag{19.1}$$

which means that

relatíve forward error ≤ condition number × relative backward error.

This inequality indicates that the computed solution of an ill-conditioned problem may have a big relative forward error. An algorithm is said to be ***forward stable*** if the forward error is small. A forward stable method is not necessarily backward stable. If the forward error and the condition number are small, then the algorithm is forward stable.

19.2. Example. Consider the function $f\,(x) = \log x$ the condition number of which is $c\,(f,x) = c\,(x) = 1/\left|\log x\right|$. For $x \approx 1$ the condition number $c\,(f,x)$ is big. Therefore the relative forward error is big for $x \approx 1$.

19.1.3. Rounding errors and floating point arithmetic

The classical error analysis investigates only the effects of the input data errors and assumes exact arithmetic operations. The digital computers however are representing the numbers with a finite number of digits, the arithmetic computations are carried out on the elements of a finite set F of such numbers and the results of operations belong to F. Hence the computer representation of the numbers may add further errors to the input data and the results of arithmetic operations may also be subject to further rounding. If the result of operation belongs to F, then we have the exact result. Otherwise we have three cases:

(i) rounding to representable (nonzero) number;

(ii) underflow (rounding to 0);

(iii) overflow (in case of results whose moduli too large).

The most of the scientific-engineering calculations are done in *floating point arithmetic* whose generally accepted model is the following:

Definition 19.1 *The set of floating point numbers is given by*

$$F(\beta,t,L,U) =$$
$$= \left\{\pm m\times\beta^{e} \mid \tfrac{1}{\beta} \le m < 1,\ m = 0.d_1d_2\ldots d_t,\ L \le e \le U\right\} \cup \{0\},$$

where
- β is the base (or radix) of the number system,
- m is the mantissa in the number system with base β,
- e is the exponent,
- t is the length of mantissa (the precision of arithmetic),
- L is the smallest exponent (underflow exponent),
- U is the biggest exponent (overflow exponent).

The parameters of the three most often used number systems are indicated in the following table

Name	β	Machines
binary	2	most computer
decimal	10	most calculators
hexadecimal	16	IBM mainframe computers

The mantissa can be written in the form

$$m = 0.d_1 d_2 \dots d_t = \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t}. \tag{19.2}$$

We can observe that condition $1/\beta \le m < 1$ implies the inequality $1 \le d_1 \le \beta - 1$ for the first digit d_1. The remaining digits must satisfy $0 \le d_i \le \beta - 1$ $(i = 2, \dots, t)$. Such arithmetic systems are called ***normalized***. The zero digit and the dot is not represented. If $\beta = 2$, then the first digit is 1, which is also unrepresented. Using the representation (19.2) we can give the set $F = F(\beta, t, L, U)$ in the form

$$F = \left\{ \pm \left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) \beta^e \mid \ L \le e \le U \right\} \cup \{0\}, \tag{19.3}$$

where $0 \le d_i \le \beta - 1$ $(i = 1, \dots, t)$ and $1 \le d_1$.

19.3. Example. The set $F\,(2, 3, -1, 2)$ contains 33 elements and its positive elements are given by

$$\left\{ \frac{1}{4}, \frac{5}{16}, \frac{6}{16}, \frac{7}{16}, \frac{1}{2}, \frac{5}{8}, \frac{6}{8}, \frac{7}{8}, 1, \frac{10}{8}, \frac{12}{8}, \frac{14}{8}, 2, \frac{20}{8}, 3, \frac{28}{8} \right\}.$$

The elements of F are not equally distributed on the real line. The distance of two consecutive numbers in $[1/\beta, 1] \cap F$ is β^{-t}. Since the elements of F are of the form $\pm m \times \beta^e$, the distance of two consecutive numbers in F is changing with the exponent. The maximum distance of two consecutive floating point numbers is β^{U-t}, while the minimum distance is β^{L-t}.

For the mantissa we have $m \in [1/\beta, 1 - 1/\beta^t]$, since

$$\frac{1}{\beta} \le m = \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \le \frac{\beta - 1}{\beta} + \frac{\beta - 1}{\beta^2} + \cdots + \frac{\beta - 1}{\beta^t} = 1 - \frac{1}{\beta^t}.$$

Using this observation we can easily prove the following result on the range of floating point numbers.

Theorem 19.2 *If $a \in F$, $a \neq 0$, then $M_L \le |a| \le M_U$, where*

$$M_L = \beta^{L-1}, \quad M_U = \beta^U (1 - \beta^{-t}).$$

Let $a, b \in F$ and denote $\square$ any of the four arithmetic operations $(+, -, *, /)$. The following cases are possible:

(1) $a \square b \in F$ (exact result),

(2) $|a \square b| > M_U$ (arithmetic overflow),

(3) $0 < |a \square b| < M_L$ (arithmetic underflow),

(4) $a\square b \notin F$, $M_L < |a\square b| < M_U$ (not representable result).

In the last two cases the *floating point arithmetic* is rounding the result $a\square b$ to the nearest floating point number in F. If two consecutive floating point numbers are equally distant from $a\square b$, then we generally round to the greater number. For example, in a five digit decimal arithmetic, the number 2.6457513 is rounded to the number 2.6458.

Let $G = [-M_U, M_U]$. It is clear that $F \subset G$. Let $x \in G$. The $fl(x)$ denotes an element of F nearest to x. The mapping $x \to fl(x)$ is called rounding. The quantity $|x - fl(x)|$ is called the rounding error. If $fl(x) = 1$, then the rounding error is at most $\beta^{1-t}/2$. The quantity $u = \beta^{1-t}/2$ is called the ***unit roundoff***. The quantity u is the relative error bound of $fl(x)$.

Theorem 19.3 *If $x \in G$, then*

$$fl(x) = x(1+\varepsilon), \quad |\varepsilon| \le u.$$

Proof. Without loss of generality we can assume that $x > 0$. Let $m_1\beta^e$, $m_2\beta^e \in F$ be two consecutive numbers such that

$$m_1\beta^e \le x \le m_2\beta^e.$$

Either $1/\beta \le m_1 < m_2 \le 1-\beta^{-t}$ or $1-\beta^{-t} = m_1 < m_2 = 1$ holds. Since $m_2 - m_1 = \beta^{-t}$ holds in both cases, we have

$$|fl(x) - x| \le \frac{|m_2 - m_1|}{2}\beta^e = \frac{\beta^{e-t}}{2}$$

either $fl(x) = m_1\beta^e$ or $fl(x) = m_2\beta^e$. It follows that

$$\frac{|fl(x) - x|}{|x|} \le \frac{|fl(x) - x|}{m_1\beta^e} \le \frac{\beta^{e-t}}{2m_1\beta^e} = \frac{\beta^{-t}}{2m_1} \le \frac{1}{2}\beta^{1-t} = u.$$

Hence $fl(x) - x = \lambda x u$, where $|\lambda| \le 1$. A simple arrangement yields

$$fl(x) = x(1+\varepsilon) \quad (\varepsilon = \lambda u)$$

Since $|\varepsilon| \le u$, we proved the claim. ∎

Thus we proved that the relative error of the rounding is bounded in floating point arithmetic and the bound is the unit roundoff u.

Another quantity used to measure the rounding errors is the so called the ***machine epsilon*** $\epsilon_M = 2u = \beta^{1-t}$ ($\epsilon_M = 2u$). The number ϵ_M is the distance of 1 and its nearest neighbor greater than 1. The following algorithm determines ϵ_M in the case of binary base.

MACHINE-EPSILON

```
1  x ← 1
2  while 1 + x > 1
3        do x ← x/2
4  ε_M ← 2x
5  return ε_M
```

In the MATLAB system $\epsilon_M \approx 2.2204 \times 10^{-16}$.

For the results of floating point arithmetic operations we assume the following (standard model):

$$fl(a\square b) = (a\square b)(1+\varepsilon), \quad |\varepsilon| \le u \quad (a, b \in F). \tag{19.4}$$

The IEEE arithmetic standard satisfies this assumption. It is an important consequence of the assumption that for $a\square b \neq 0$ the relative error of arithmetic operations satisfies

$$\frac{|fl(a\square b) - (a\square b)|}{|a\square b|} \leq u.$$

Hence the relative error of the floating point arithmetic operations is small.

There exist computer floating point arithmetics that do not comply with the standard model (19.4). The usual reason for this is that the arithmetic lacks a guard digit in subtraction. For simplicity we investigate the subtraction $1-0.111$ in a three digit binary arithmetic. In the first step we equate the exponents:

$$\begin{array}{rcccccc} & 2 & \times & 0 & . & 1 & 0 & 0 \\ - & 2 & \times & 0 & . & 0 & 1 & 1 & 1 \end{array}.$$

If the computation is done with four digits, the result is the following

$$\begin{array}{rcccccc} & 2^1 & \times & 0 & . & 1 & 0 & 0 \\ - & 2^1 & \times & 0 & . & 0 & 1 & 1 & 1 \\ \hline & 2^1 & \times & 0 & . & 0 & 0 & 0 & 1 \end{array},$$

from which the normalized result is $2^{-2} \times 0.100$. Observe that the subtracted number is unnormalized. The temporary fourth digit of the mantissa is called a guard digit. Without a guard digit the computations are the following:

$$\begin{array}{rcccccc} & 2^1 & \times & 0 & . & 1 & 0 & 0 \\ - & 2^1 & \times & 0 & . & 0 & 1 & 1 \\ \hline & 2^1 & \times & 0 & . & 0 & 0 & 1 \end{array}.$$

Hence the normalized result is $2^{-1} \times 0.100$ with a relative error of 100%. Several CRAY computers and pocket calculators lack guard digits.

Without the guard digit the floating point arithmetic operations satisfy only the weaker conditions

$$fl\,(x \pm y) = x\,(1+\alpha) \pm y\,(1+\beta), \quad |\alpha|, |\beta| \leq u, \tag{19.5}$$

$$fl\,(x\square y) = (x\square y)\,(1+\delta), \quad |\delta| \leq u, \quad \square = *, /\,. \tag{19.6}$$

Assume that we have a guard digit and the arithmetic complies with standard model (19.4). Introduce the following notations:

$$|z| = [|z_1|, \ldots, |z_n|]^T \quad (z \in \mathbb{R}^n), \tag{19.7}$$

$$|A| = \left[\left|a_{ij}\right|\right]_{i,j=1}^{m,n} \quad (A \in \mathbb{R}^{m\times n}), \tag{19.8}$$

$$A \leq B \Leftrightarrow a_{ij} \leq b_{ij} \quad (A, B \in \mathbb{R}^{m\times n}). \tag{19.9}$$

The following results hold:

$$\left|fl\left(x^T y\right) - x^T y\right| \leq 1.01nu\,|x|^T\,|y| \quad (nu \leq 0.01), \tag{19.10}$$

$$fl(\alpha A) = \alpha A + E \quad (|E| \le u|\alpha A|), \tag{19.11}$$

$$fl(A + B) = (A + B) + E \quad (|E| \le u|A + B|), \tag{19.12}$$

$$fl(AB) = AB + E \quad \left(|E| \le nu|A||B| + O\left(u^2\right)\right), \tag{19.13}$$

where E denotes the error (matrix) of the actual operation.

The standard *floating point arithmetics* have many special properties. It is an important property that the addition is not associative because of the rounding.

19.4. Example. If $a = 1$, $b = c = 3 \times 10^{-16}$, then using MATLAB and AT386 type PC we obtain

$$1.000000000000000e + 000 = (a + b) + c \neq a + (b + c) = 1.000000000000001e + 000.$$

We can have a similar result on Pentium1 machine with the choice $b = c = 1.15 \times 10^{-16}$.

The example also indicates that for different (numerical) processors may produce different computational results for the same calculations. The commutativity can also be lost in addition. Consider the computation of the sum $\sum_{i=1}^{n} x_i$. The usual algorithm is the recursive summation.

RECURSIVE-SUMMATION(n, x)

```
1  s ← 0
2  for i ← 1 to n
3       do s ← s + x_i
4  return s
```

19.5. Example. Compute the sum

$$s_n = 1 + \sum_{i=1}^{n} \frac{1}{i^2 + i}$$

for $n = 4999$. The recursive summation algorithm (and MATLAB) gives the result

$$1.999800000000002e + 000.$$

If the summation is done in the reverse (increasing) order, then the result is

$$1.999800000000000e + 000.$$

If the two values are compared with the exact result $s_n = 2 - 1/(n + 1)$, then we can see that the second summation gives better result. In this case the sum of smaller numbers gives significant digits to the final result unlike in the first case.

The last example indicates that the summation of a large number of data varying in modulus and sign is a complicated task. The following algorithm of W. Kahan is one of the most interesting procedures to solve the problem.

COMPENSATED-SUMMATION(n, x)

```
1  s ← 0
2  e ← 0
3  for i ← 1 to n
4       do t ← s
5          y ← x_i + e
6          s ← t + y
7          e ← (t − s) + y
8  return s
```

19.1.4. The floating point arithmetic standard

The ANSI/IEEE Standard 754-1985 of a binary ($\beta = 2$) *floating point arithmetic system* was published in 1985. The standard specifies the basic arithmetic operations, comparisons, rounding modes, the arithmetic exceptions and their handling, and conversion between the different arithmetic formats. The square root is included as a basic operation. The standard does not deal with the exponential and transcendent functions. The standard defines two main floating point formats:

Type	Size	Mantissa	e	u	$[M_L, M_U] \approx$
Single	32 bits	23 + 1 bits	8 bits	$2^{-24} \approx 5.96 \times 10^{-8}$	$10^{\pm 38}$
Double	64 bits	52 + 1 bits	11 bits	$2^{-53} \approx 1.11 \times 10^{-16}$	$10^{\pm 308}$

In both formats one bit is reserved as a sign bit. Since the floating point numbers are normalized and the first digit is always 1, this bit is not stored. This hidden bit is denoted by the „+1" in the table.

The arithmetic standard contains the handling of arithmetic exceptions.

Exception type	Example	Default result
Invalid operation	$0/0$, $0 \times \infty$, $\sqrt{-1}$	NaN (Not a Number)
Overflow	$\lvert x \square y \rvert > M_U$	$\pm\infty$
Divide by zero	Finite nonzero/0	$\pm\infty$
Underflow	$0 < \lvert x \square y \rvert < M_L$	Subnormal numbers
Inexact	$fl\,(x \square y) \neq x \square y$	Correctly rounded result

(The numbers of the form $\pm m \times \beta^{L-t}$, $0 < m < \beta^{t-1}$ are called ***subnormal numbers***.) The IEEE arithmetic is a closed system. Every arithmetic operations has a result, whether it is expected mathematically or not. The exceptional operations raise a signal and continue. The arithmetic standard conforms with the standard model (19.4).

The first hardware implementation of the IEEE standard was the Intel 8087 mathematical coprocessor. Since then it is generally accepted and used.

Remark. In the single precision we have about 7 significant digit precision in the decimal system. For double precision we have approximately 16 digit precision in decimals. There also exists an extended precision format of 80 bits, where $t = 63$ and the exponential has 15 bits.

Exercises

19.1-1 The measured values of two resistors are $R_1 = 110.2 \pm 0.3\Omega$ and $R_2 = 65.6 \pm 0.2\Omega$. We connect the two resistors parallel and obtain the circuit resistance $R_e = R_1R_2/(R_1 + R_2)$. Calculate the relative error bounds of the initial data and the approximate value of the resistance R_e. Evaluate the absolute and relative error bounds δR_e and $\delta R_e/R_e$, respectively in the following three ways:
(i) Estimate first δR_e using only the absolute error bounds of the input data, then estimate the relative error bound $\delta R_e/R_e$.
(ii) Estimate first the relative error bound $\delta R_e/R_e$ using only the relative error bounds of the input data, then estimate the absolute error bound δR_e.
(iii) Consider the circuit resistance as a two variable function $R_e = F(R_1, R_2)$.

19.1-2 Assume that $\sqrt{2}$ is calculated with the absolute error bound 10^{-8}. The following two expressions are theoretically equal:
(i) $1/\left(1 + \sqrt{2}\right)^6$;
(ii) $99 - 70\sqrt{2}$.
Which expression can be calculated with less relative error and why?

19.1-3 Consider the arithmetic operations as two variable functions of the form $f(x, y) = x \square y$, where $\square \in \{+, -, *, /\}$.
(i) Derive the *error bounds* of the arithmetic operations from the error formula of two variable functions.
(ii) Derive the *condition numbers* of these functions. When are they ill-conditioned?
(iii) Derive *error bounds* for the power function assuming that both the base and the exponent have errors. What is the result if the exponent is exact?
(iv) Let $y = 16x^2$, $x \approx a$ and $y \approx b = 16a^2$. Determine the smallest and the greatest value of a as a function of x such that the *relative error bound* of b should be at most 0.01.

19.1-4 Assume that the number $C = \mathrm{EXP}(4\pi^2/\sqrt{83})$ $(= 76.1967868\ldots)$ is calculated in a 24 bit long mantissa and the exponential function is also calculated with 24 significant bits. Estimate the *absolute error* of the result. Estimate the *relative error* without using the actual value of C.

19.1-5 Consider the emphfloating point number set $F(\beta, t, L, U)$ and show that
(i) Every arithmetic operation can result *arithmetic overflow*;
(ii) Every arithmetic operation can result *arithmetic underflow*.

19.1-6 Show that the following expressions are *numerically unstable* for $x \approx 0$:
(i) $(1 - \cos x)/\sin^2 x$;
(ii) $\sin(100\pi + x) - \sin(x)$;
(iii) $2 - \sin x - \cos x - e^{-x}$.
Calculate the values of the above expressions for $x = 10^{-3}, 10^{-5}, 10^{-7}$ and estimate the error. Manipulate the expressions into *numerically stable* ones and estimate the error as well.

19.1-7 How many elements does the set $F = F(\beta, t, L, U)$ have? How many *subnormal numbers* can we find?

19.1-8 If $x, y \geq 0$, then $(x+y)/2 \geq \sqrt{xy}$ and equality holds if and only if $x = y$. Is it true numerically? Check the inequality experimentally for various data (small and large numbers, numbers close to each other or different in magnitude).

19.2. Linear systems of equations

The general form of linear algebraic systems with n unknowns and m equations is given by

$$\begin{array}{rcl} a_{11}x_1 + \cdots + a_{1j}x_j + \cdots + a_{1n}x_n & = & b_1 \\ & \vdots & \\ a_{i1}x_1 + \cdots + a_{ij}x_j + \cdots + a_{in}x_n & = & b_i \\ & \vdots & \\ a_{m1}x_1 + \cdots + a_{mj}x_j + \cdots + a_{mn}x_n & = & b_m \end{array} \tag{19.14}$$

This system can be written in the more compact form

$$Ax = b, \tag{19.15}$$

where

$$A = \left[a_{ij}\right]_{i,j=1}^{m,n} \in \mathbb{R}^{m\times n},\ x \in \mathbb{R}^n,\ b \in \mathbb{R}^m.$$

The systems is called *underdetermined* if $m < n$. For $m > n$, the systems is called *overdetermined*. Here we investigate only the case $m = n$, when the coefficient matrix A is square. We also assume that the inverse matrix A^{-1} exists (or equivalently $\det(A) \neq 0$). Under this assumption the linear system $Ax = b$ has exactly one solution: $x = A^{-1}b$.

19.2.1. Direct methods for solving linear systems

Triangular linear systems

Definition 19.4 *The matrix $A = [a_{ij}]_{i,j=1}^n$ is **upper triangular** if $a_{ij} = 0$ for all $i > j$. The matrix A is **lower triangular** if $a_{ij} = 0$ for all $i < j$.*

For example the general form of the upper triangular matrices is the following:

$$\begin{bmatrix} * & * & \cdots & \cdots & * \\ 0 & * & & & \vdots \\ \vdots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & * & * \\ 0 & \cdots & \cdots & 0 & * \end{bmatrix}.$$

We note that the diagonal matrices are both lower and upper triangular. It is easy to show that $\det(A) = a_{11}a_{22}\ldots a_{nn}$ holds for the upper or lower triangular matrices. It is easy to solve linear systems with triangular coefficient matrices. Consider the following upper triangular

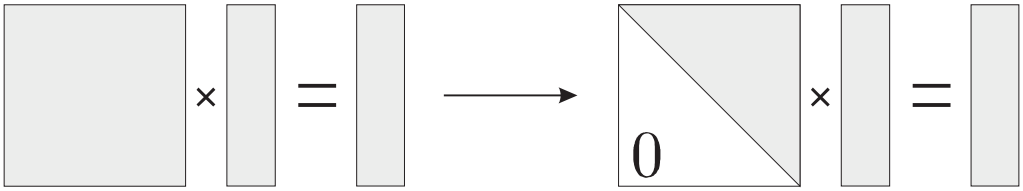


Figure 19.2. Gaussian elimination.

linear system:

$$\begin{array}{ccccccccc}
a_{11}x_1+ & \cdots & +a_{1i}x_i+ & \cdots & +a_{1n}x_n & = & b_1 \\
\ddots & \vdots & & & \vdots & & \vdots \\
& a_{ii}x_i+ & \cdots & +a_{in}x_n & = & b_i \\
& & \ddots & \vdots & & \vdots \\
& & & a_{nn}x_n & = & b_n
\end{array}$$

This can be solved by the so called *back substitution* algorithm.

BACK-SUBSTITUTION(A, b, n)

1 $x_n \leftarrow b_n/a_{nn}$
2 **for** $i \leftarrow n-1$ **downto** 1
3 **do** $x_i \leftarrow (b_i - \sum_{j=i+1}^{n} a_{ij}x_j)/a_{ii}$
4 **return** x

The solution of lower triangular systems is similar.

The Gauss method

The Gauss method or Gaussian elimination (GE) consists of two phases:
I. The linear system $Ax = b$ is transformed to an equivalent upper triangular system using elementary operations (see Figure 19.2).
II. The obtained upper triangular system is then solved by the back substitution algorithm.

The first phase is often called the elimination or forward phase. The second phase of GE is called the backward phase. The elementary operations are of the following three types:

1. Add a multiple of one equation to another equation.
2. Interchange two equations.
3. Multiply an equation by a nonzero constant.

The elimination phase of GE is based on the following observation. Multiply equation k by $\gamma \neq 0$ and subtract it from equation i:

$$(a_{i1} - \gamma a_{k1})\, x_1 + \cdots + \left(a_{ij} - \gamma a_{kj}\right) x_j + \cdots + (a_{in} - \gamma a_{kn})\, x_n = b_i - \gamma b_k.$$

If $a_{kj} \neq 0$, then by choosing $\gamma = a_{ij}/a_{kj}$, the coefficient of x_j becomes 0 in the new equivalent equation, which replaces equation i. Thus we can eliminate variable x_j (or coefficient a_{ij}) from equation i.

The Gauss method eliminates the coefficients (variables) under the main diagonal of

A in a systematic way. First variable x_1 is eliminated from equations $i = 2, \ldots, n$ using equation 1, then x_2 is eliminated from equations $i = 3, \ldots, n$ using equation 2, and so on.

Assume that the unknowns are eliminated in the first $(k-1)$ columns under the main diagonal and the resulting linear system has the form

$$\begin{array}{cccccccccc}
a_{11}x_1+ & \cdots & \cdots & +a_{1k}x_k & + & \cdots & + & a_{1n}x_n & = & b_1 \\
 & \ddots & & \vdots & & & & \vdots & & \vdots \\
 & & \ddots & \vdots & & & & \vdots & & \vdots \\
 & & & a_{kk}x_k & + & \cdots & + & a_{kn}x_n & = & b_k \\
 & & & \vdots & & & & \vdots & & \vdots \\
 & & & a_{ik}x_k & + & \cdots & + & a_{in}x_n & = & b_i \\
 & & & \vdots & & & & \vdots & & \vdots \\
 & & & a_{nk}x_k & + & \cdots & + & a_{nn}x_n & = & b_n
\end{array}$$

If $a_{kk} \neq 0$, then multiplying row k by γ and subtracting it from equation i we obtain

$$(a_{ik} - \gamma a_{kk})x_k + (a_{i,k+1} - \gamma a_{k,k+1})x_{k+1} + \cdots + (a_{in} - \gamma a_{kn})x_n = b_i - \gamma b_k.$$

Since $a_{ik} - \gamma a_{kk} = 0$ for $\gamma = a_{ik}/a_{kk}$, we eliminated the coefficient a_{ik} (variable x_k) from equation $i > k$. Repeating this process for $i = k+1, \ldots, n$ we can eliminate the coefficients under the main diagonal entry a_{kk}. Next we denote by $A[i, j]$ the element a_{ij} of matrix A and by $A[i, j : n]$ the vector $\left[a_{ij}, a_{i,j+1}, \ldots, a_{in}\right]$. The Gauss method has the following form (where the *pivoting* discussed later is also included):

GAUSS-METHOD(A, b)

```
1  ▷ Forward phase:
2  n ← rows[A]
3  for k ← 1 to n − 1
4      do { pivoting and interchange of rows and columns}
5         for i ← k + 1 to n
6             do γ_ik ← A[i,k] /A[k,k]
7                A[i,k+1 : n] ← A[i,k+1 : n] − γ_ik ∗ A[k,k+1 : n]
8                b_i ← b_i − γ_ik b_k
9  ▷ Backward phase: see the back substitution algorithm.
10 return x
```

The algorithm overwrites the original matrix A and vector b. It does not write however the zero entries under the main diagonal since these elements are not necessary for the second phase of the algorithm. Hence the lower triangular part of matrix A can be used to store information for the LU decomposition of matrix A.

The above version of the Gauss method can be performed only if the elements a_{kk} occurring in the computation are not zero. For this and numerical stability reasons we use the Gaussian elimination with *pivoting*.

The Gauss method with pivoting

If $a_{kk} = 0$, then we can interchange row k with another row, say i, so that the new entry (a_{ki}) at position (k,k) should be nonzero. If this is not possible, then all the coefficients $a_{kk}, a_{k+1,k}, \ldots, a_{nk}$ are zero and $\det(A) = 0$. In the latter case $Ax = b$ has no unique solution. The element a_{kk} is called the k^{th} ***pivot element***. We can always select new pivot elements by interchanging the rows. The selection of the pivot element has a great influence on the reliability of the computed results. The simple fact that we divide by the pivot element indicates this influence. We recall that $\delta(a/b)$ is proportional to $1/|b|^2$. It is considered advantageous if the pivot element is selected so that it has the greatest possible modulus. The process of selecting the pivot element is called *pivoting*. We mention the following two pivoting processes.

Partial pivoting: At the k^{th} step, interchange the rows of the matrix so the largest remaining element, say a_{ik}, in the k^{th} column is used as pivot. After the pivoting we have

$$|a_{kk}| = \max_{k \le i \le n} |a_{ik}| .$$

Complete pivoting: At the k^{th} step, interchange both the rows and columns of the matrix so that the largest element, say a_{ij}, in the remaining matrix is used as pivot After the pivoting we have

$$|a_{kk}| = \max_{k \le i,j \le n} |a_{ij}| .$$

Note that the interchange of two columns implies the interchange of the corresponding unknowns. The significance of pivoting is well illustrated by the following

19.6. Example. The exact solution of the linear system

$$\begin{array}{rcrcl} 10^{-17}x & + & y & = & 1 \\ x & + & y & = & 2 \end{array}$$

is $x = 1/(1-10^{-17})$ and $y = 1 - 10^{-17}/(1-10^{-17})$. The MATLAB program gives the result $x = 1, y = 1$ and this is the best available result in standard double precision arithmetic. Solving this system with the Gaussian elimination without pivoting (also in double precision) we obtain the catastrophic result $x = 0$ and $y = 1$. Using partial pivoting with the Gaussian elimination we obtain the best available numerical result $x = y = 1$.

Remark 19.5 *Theoretically we do not need pivoting in the following cases: 1. If A is symmetric and positive definite ($A \in \mathbb{R}^{n\times n}$ is positive definite $\Leftrightarrow x^T Ax > 0$, $\forall x \in \mathbb{R}^n$, $x \neq 0$). 2. If A is diagonally dominant in the following sense:*

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \quad (1 \le i \le n) .$$

In case of symmetric and positive definite matrices we use the Cholesky method which is a special version of the Gauss-type methods.

During the Gaussian elimination we obtain a sequence of equivalent linear systems

$$A^{(0)}x = b^{(0)} \to A^{(1)}x = b^{(1)} \to \cdots \to A^{(n-1)}x = b^{(n-1)},$$

where

$$A^{(0)} = A, \quad A^{(k)} = \left[a_{ij}^{(k)}\right]_{i,j=1}^{n}.$$

Note that matrices $A^{(k)}$ are stored in the place of $A = A^{(0)}$. The last coefficient matrix of phase I has the form

$$A^{(n-1)} = \begin{bmatrix} a_{11}^{(0)} & a_{12}^{(0)} & \cdots & a_{1n}^{(0)} \\ 0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & \cdots & a_{nn}^{(n-1)} \end{bmatrix},$$

where $a_{kk}^{(k-1)}$ is the k^{th} pivot element. The ***growth factor of pivot elements*** is given by

$$\rho = \rho_n = \max_{1 \le k \le n} \left| a_{kk}^{(k-1)} / a_{11}^{(0)} \right|.$$

Wilkinson proved that the error of the computed solution is proportional to the growth factor ρ and the bounds

$$\rho \le \sqrt{n}\left(2 \cdot 3^{\frac{1}{2}} \cdots n^{\frac{1}{n-1}}\right)^{\frac{1}{2}} \sim c n^{\frac{1}{2}} n^{\frac{1}{4} log(n)}$$

and

$$\rho \le 2^{n-1}$$

hold for complete and partial pivoting, respectively. Wilkinson conjectured that $\rho \le n$ for complete pivoting. This has been proved by researchers for small values of n. Statistical investigations on random matrices ($n \le 1024$) indicate that the average of ρ is $\Theta\left(n^{2/3}\right)$ for the partial pivoting and $\Theta\left(n^{1/2}\right)$ for the complete pivoting. Hence the case $\rho > n$ hardly occurs in the statistical sense.

We remark that Wilkinson constructed a linear system on which $\rho = 2^{n-1}$ for the partial pivoting. Hence Wilkinson's bound for ρ is sharp in the case of partial pivoting. There also exist examples of linear systems concerning discretizations of differential and integral equations, where ρ is increasing exponentially if Gaussian elimination is used with partial pivoting.

The growth factor ρ can be very large, if the Gaussian elimination is used without pivoting. For example, $\rho = \rho_4\,(A) = 1.23 \times 10^5$, if

$$A = \begin{bmatrix} 1.7846 & -0.2760 & -0.2760 & -0.2760 \\ -3.3848 & 0.7240 & -0.3492 & -0.2760 \\ -0.2760 & -0.2760 & 1.4311 & -0.2760 \\ -0.2760 & -0.2760 & -0.2760 & 0.7240 \end{bmatrix}.$$

Operations counts

The Gauss method gives the solution of the linear system $Ax = b$ ($A \in \mathbb{R}^{n \times n}$) in a finite number of steps and arithmetic operations ($+, -, *, /$). The amount of necessary arithmetic operations is an important characteristic of the direct linear system solvers, since the CPU time is largely proportional to the number of arithmetic operations. It was also observed that the number of additive and multiplicative operations are nearly the same in the numerical algorithms of linear algebra. For measuring the cost of such algorithms C.B. Moler introduced the concept of flop.

Definition 19.6 ***One (old) flop*** *is the computational work necessary for the operation* $s = s + x * y$ *(1 addition + 1 multiplication).* ***One (new)*** *flop is the computational work necessary for any of the arithmetic operations* $+, -, *, /$.

The new flop can be used if the computational time of additive and multiplicative operations are approximately the same. Two new flops equals to one old flop. Here we use the notion of old flop.

For the Gauss method a simple counting gives the number of additive and multiplicative operations.

Theorem 19.7 *The computational cost of the Gauss method is* $n^3/3 + \Theta(n^2)$ *flops.*

V.V. Klyuyev and N. Kokovkin-Shcherbak proved that if only elementary row and column operations (multiplication of row or column by a number, interchange of rows or columns, addition of a multiple of row or column to another row or column) are allowed, then the linear system $Ax = b$ cannot be solved in less than $n^3/3 + \Omega(n^2)$ flops.

Using fast matrix inversion procedures we can solve the $n \times n$ linear system $Ax = b$ in $O(n^{2.808})$ flops. These theoretically interesting algorithms are not used in practice since they are considered as numerically unstable.

The LU-decomposition

In many cases it is easier to solve a linear system if the coefficient matrix can be decomposed into the product of two triangular matrices.

Definition 19.8 *The matrix* $A \in \mathbb{R}^{n\times n}$ *has an* ***LU-decomposition****, if* $A = LU$*, where* $L \in \mathbb{R}^{n\times n}$ *is lower and* $U \in \mathbb{R}^{n\times n}$ *is upper triangular matrix.*

The LU-decomposition is not unique. If a nonsingular matrix has an LU-decomposition, then it has a particular LU-decomposition, where the main diagonal of a given component matrix consists of 1's. Such triangular matrices are called ***unit (upper or lower) triangular*** matrices. The LU decomposition is unique, if L is set to be lower unit triangular or U is set to be unit upper triangular.

The LU-decomposition of nonsingular matrices is closely related to the Gaussian elimination method. If $A = LU$, where L is unit lower triangular, then $l_{ik} = \gamma_{ik}$ $(i > k)$, where γ_{ik} is given by the Gauss algorithm. The matrix U is the upper triangular part of the matrix we obtain at the end of the forward phase. The matrix L can also be derived from this matrix, if the columns of the lower triangular part are divided by the corresponding main diagonal elements. We remind that the first phase of the Gaussian elimination does not annihilate the matrix elements under the main diagonal. It is clear that a nonsingular matrix has LU-decomposition if and only if $a_{kk}^{(k-1)} \neq 0$ holds for each pivot element for the Gauss method without pivoting.

Definition 19.9 *A matrix* $P \in \mathbb{R}^{n\times n}$ *whose every row and column has one and only one non-zero element, that element being* 1*, is called a* ***permutation matrix****.*

In case of partial pivoting we permute the rows of the coefficient matrix (multiply A by a permutation matrix on the left) so that $a_{kk}^{(k-1)} \neq 0$ $(k = 1, \ldots, n)$ holds for a nonsingular matrix. Hence we have

Theorem 19.10 *If $A \in \mathbb{R}^{n\times n}$ is nonsingular then there exists a permutation matrix P such that PA has an LU-decomposition.*

The the algorithm of LU-decomposition is essentially the Gaussian elimination method. If pivoting is used then the interchange of rows must also be executed on the elements under the main diagonal and the permutation matrix P must be recorded. A vector containing the actual order of the original matrix rows is obviously sufficient for this purpose.

The LU- and Cholesky-methods

Let $A = LU$ and consider the equation $Ax = b$. Since $Ax = LUx = L(Ux) = b$, we can decompose $Ax = b$ into the equivalent linear system $Ly = b$ and $Ux = b$, where L is lower triangular and U is upper triangular.

LU-METHOD(A, b)

1 Determine the LU-decomposition $A = LU$.
2 Solve $Ly = b$.
3 Solve $Ux = y$.
4 **return** x

Remark. In case of partial pivoting we obtain the decomposition $\hat{A} = PA = LU$ and we set $\hat{b} = Pb$ instead of b.

In the first phase of the Gauss method we produce decomposition $A = LU$ and the equivalent linear system $Ux = L^{-1}b$ with upper triangular coefficient matrix. The latter is solved in the second phase. In the LU-method we decompose the first phase of the Gauss method into two steps. In the first step we obtain only the decomposition $A = LU$. In the second step we produce the vector $y = L^{-1}b$. The third step of the algorithm is identical with the second phase of the original Gauss method.

The LU-method is especially advantageous if we have to solve several linear systems with the same coefficient matrix:

$$Ax = b_1,\ Ax = b_2, \ldots,\ Ax = b_k.$$

In such a case we determine the LU-decomposition of matrix A only once, and then we solve the linear systems $Ly_i = b_i$, $Ux_i = y_i$ $(x_i, y_i, b_i \in \mathbf{R}^n,\ i = 1, \ldots, k)$. The computational cost of this process is $n^3/3 + kn^2 + \Theta(kn)$ flops.

The *inversion of a matrix* $A \in \mathbb{R}^{n\times n}$ can be done as follows:

1. Determine the LU-decomposition $A = LU$. .

2. Solve $Ly_i = e_i$, $Ux_i = y_i$ (e_i is the i^{th} unit vector $i = 1, \ldots, n$).

The inverse of A is given by $A^{-1} = [x_1, \ldots, x_n]$. The computational cost of the algorithm is $4n^3/3 + \Theta\left(n^2\right)$ flops.

The LU-method with pointers

This implementation of the LU-method is known since the 60's. Vector P contains the indices of the rows. At the start we set $P[i] = i$ $(1 \le i \le n)$. When exchanging rows we exchange only those components of vector P that correspond to the rows.

LU-METHOD-WITH-POINTERS(A, b)

```
 1  n ← rows[A]
 2  P ← [1, 2, ..., n]
 3  for k ← 1 to n − 1
 4      do compute index t such that |A[P[t], k]| = max_{k≤i≤n} |A[P[i], k]| .
 5         if k < t
 6            then exchange the components P[k] and P[t].
 7         for i ← k + 1 to n
 8             do A[P[i], k] ← A[P[i], k] /A[P[k], k]
 9                A[P[i], k + 1 : n] ← A[P[i], k + 1 : n] − A[P[i], k] ∗ A[P[k], k + 1 : n]
10  for i ← 1 to n
11      do s ← 0
12         for j ← 1 to i − 1
13             do s ← s + A[P[i], j] ∗ x[j]
14         x[i] ← b[P[i]] − s
15  for i ← n downto 1
16      do s ← 0
17         for j ← i + 1 to n
18             s ← s + A[P[i], j] ∗ x[j]
19         x[i] ← (x[i] − s) /A[P[i], i]
20  return x
```

If $A \in \mathbb{R}^{n\times n}$ is *symmetric and positive definite*, then it can be decomposed in the form $A = LL^T$, where L is lower triangular matrix. The LL^T-decomposition is called the ***Cholesky-decomposition*** . In this case we can save approximately half of the storage place for A and half of the computational cost of the LU-decomposition (LL^T-decomposition). Let

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 0 & l_{22} & \cdots & l_{n2} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & l_{nn} \end{bmatrix}.$$

Observing that only the first k elements may be nonzero in the k^{th} column of L^T we obtain that

$$\begin{aligned} a_{kk} &= l_{k1}^2 + l_{k2}^2 + \cdots + l_{k,k-1}^2 + l_{kk}^2, \\ a_{ik} &= l_{i1}l_{k1} + l_{i2}l_{k2} + \cdots + l_{i,k-1}l_{k,k-1} + l_{ik}l_{kk} \qquad (i = k+1, \ldots, n). \end{aligned}$$

This gives the formulae

$$\begin{aligned} l_{kk} &= (a_{kk} - \sum_{j=1}^{k-1} l_{kj}^2)^{1/2}, \\ l_{ik} &= (a_{ik} - \sum_{j=1}^{k-1} l_{ij}l_{kj})/l_{kk} \qquad (i = k+1, \ldots, n). \end{aligned}$$

Using the notation $\sum_{j=i}^{k} s_j = 0$ $(k < i)$ we can formulate the Cholesky-method as follows.

CHOLESKY-METHOD(A)

```
1  n ← rows[A]
2  for k ← 1 to n
3      do a_kk ← (a_kk − Σ_{j=1}^{k−1} a_kj^2)^{1/2}
4          for i ← k + 1 to n
5              do a_ik ← (a_ik − Σ_{j=1}^{k−1} a_ij a_kj)/a_kk
6  return A
```

The lower triangular part of A contains L. The computational cost of the algorithm is $n^3/6 + \Theta(n^2)$ flops and n square roots. The algorithm, which can be considered as a special case of the Gauss-methods, does not require pivoting, at least in principle.

The LU- and Cholesky-methods on banded matrices

It often happens that linear systems have *banded coefficient matrices*.

Definition 19.11 *Matrix $A \in \mathbb{R}^{n\times n}$ is banded with lower bandwidth p and upper bandwidth q if*

$$a_{ij} = 0, \quad \text{if } i > j + p \text{ or } j > i + q.$$

The possibly non-zero elements a_{ij} $(i - p \le j \le i + q)$ form a band like structure. Schematically A has the form

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1,1+q} & 0 & \cdots & \cdots & 0 \\ a_{21} & a_{22} & & & & \ddots & & & \vdots \\ \vdots & & \ddots & & & & \ddots & & \vdots \\ a_{1+p,1} & & & \ddots & & & & \ddots & 0 \\ 0 & \ddots & & & \ddots & & & & a_{n-q,n} \\ \vdots & & \ddots & & & \ddots & & & \vdots \\ \vdots & & & \ddots & & & \ddots & & \vdots \\ \vdots & & & & \ddots & & & \ddots & a_{n-1,n} \\ 0 & \cdots & \cdots & \cdots & 0 & a_{n,n-p} & \cdots & a_{n,n-1} & a_{nn} \end{bmatrix}.$$

The *banded matrices* yield very efficient algorithms if p and q are significantly less than n. If a banded matrix A with lower bandwidth p and upper bandwidth q has an LU-decomposition, then both L and U are banded with lower bandwidth p and upper bandwidth q, respectively.

Next we give the LU-method for banded matrices in three parts.

THE-LU-DECOMPOSITION-OF-BANDED-MATRIX(A, n, p, q)

```
1  for k ← 1 to n − 1
2      do for i ← k + 1 to min {k + p, n}
3             do a_ik ← a_ik / a_kk
4                for j ← k + 1 to min {k + q, n}
5                    do a_ij ← a_ij − a_ik a_kj
6  return A
```

Entry a_{ij} is overwritten by l_{ij}, if $i > j$ and by u_{ij}, if $i \leq j$. The computational cost of is $c(p, q)$ flops, where

$$c(p,q) = \begin{cases} npq - \frac{1}{2}pq^2 - \frac{1}{6}p^3 + pn, & p \leq q \\ npq - \frac{1}{2}qp^2 - \frac{1}{6}q^3 + qn, & p > q \end{cases}$$

The following algorithm overwrites b by the solution of equation $Ly = b$.

SOLUTION OF BANDED UNIT LOWER TRIANGULAR SYSTEM(L, b, n, p)

```
1  for i ← 1 to n
2      do b_i ← b_i − Σ_{j=max{1,i−p}}^{i−1} l_ij b_j
3  return b
```

The total cost of the algorithm is $np - p^2/2$ flops. The next algorithm overwrites vector b by the solution of $Ux = b$.

SOLUTION-OF-BANDED-UPPER-TRIANGULAR-SYSTEM(U, b, n, q)

```
1  for i ← n downto 1
2      do b_i ← ( b_i − Σ_{j=i+1}^{min{i+q,n}} u_ij b_j ) / u_ii
3  return b
```

The computational cost is $n(q + 1) - q^2/2$ flops.

Assume that $A \in \mathbb{R}^{n \times n}$ is symmetric, positive definite and banded with lower bandwidth p. The banded version of the Cholesky-methods is given by

CHOLESKY-DECOMPOSITION-OF-BANDED-MATRICES(A, n, p)

```
1  for i ← 1 to n
2      do for j ← max {1, i − p} to i − 1
3             do a_ij ← ( a_ij − Σ_{k=max{1,i−p}}^{j−1} a_ik a_jk ) / a_jj
4         a_ii ← ( a_ii − Σ_{k=max{1,i−p}}^{i−1} a_ik^2 )^{1/2}
5  return A
```

The elements a_{ij} are overwritten by l_{ij} $(i \geq j)$. The total amount of work is given by $\left(np^2/2\right) - \left(p^3/3\right) + (3/2)\left(np - p^2\right)$ flops és n square roots.

Remark. If $A \in \mathbb{R}^{n \times n}$ has lower bandwidth p and upper bandwidth q and partial pivoting takes place, then the upper bandwidth of U increases up to $\hat{q} = p + q$.

19.2.2. Iterative methods for linear systems

There are several iterative methods for solving linear systems of algebraic equations. The best known iterative algorithms are the classical *Jacobi*-, the *Gauss-Seidel*- and the *relaxation methods*. The greatest advantage of these iterative algorithms is their easy implementation to large systems. At the same time they usually have slow convergence. However for parallel computers the *multisplitting* iterative algorithms seem to be efficient.

Consider the iteration

$$x_i = Gx_{i-1} + b \quad (i = 1, 2, \ldots)$$

where $G \in \mathbb{R}^{n\times n}$ és $x_0, b \in \mathbb{R}^n$. It is known that $\{x_i\}_{i=0}^{\infty}$ converges for all $x_0, b \in \mathbb{R}^n$ if and only if the spectral radius of G satisfies $\rho(G) < 1$ ($\rho(G) = \max|\lambda| \mid \lambda$ is an eigenvalue of G). In case of convergence $x_i \to x^* = (I-G)^{-1}b$, that is we obtain the solution of the equation $(I-G)x = b$. The speed of convergence depends on the spectral radius $\rho(G)$. Smaller the spectral radius$\rho(G)$, faster the convergence.

Consider now the linear system

$$Ax = b,$$

where $A \in \mathbb{R}^{n\times n}$ is nonsingular. The matrices $M_l, N_l, E_l \in \mathbf{R}^{n\times n}$ form a multisplitting of A if

(i) $A = M_i - N_i$, $i = 1, 2, \ldots, L$,
(ii) M_i is nonsingular, $i = 1, 2, \ldots, L$,
(iii) E_i is non-negative diagonal matrix, $i = 1, 2, \ldots, L$,
(iv) $\sum_{i=1}^{L} E_i = I$.

Let $x_0 \in \mathbb{R}^n$ be a given initial vector. The multisplitting iterative method is the following.

MULTISPLITTING-ITERATION($x_0, b, L, M_l, N_l, E_l, l = 1, \ldots, L$)

```
1  i ← 0
2  while exit condition=FALSE
3        do i ← i + 1
4           for l ← 1 to L
5                do M_l y_l ← N_l x_{i-1} + b
6           x_i ← Σ_{l=1}^{L} E_l y_l
7  return x_i
```

It is easy to show that $y_l = M_l^{-1}N_l x_{i-1} + M_l^{-1}b$ and

$$\begin{aligned} x_i &= \sum_{l=1}^{L} E_l y_l = \sum_{l=1}^{L} E_l M_l^{-1} N_l x_{i-1} + \sum_{l=1}^{L} E_l M_l^{-1} b \\ &= Hx_{i-1} + c. \end{aligned}$$

Thus the condition of convergence is $\rho(H) < 1$. The *multisplitting iteration* is a true *parallel algorithm* because we can solve L linear systems parallel in each iteration (synchronized parallelism). The bottleneck of the algorithm is the computation of iterate x_i.

The selection of matrices M_i and E_i is such that the solution of the linear system $M_i y = c$ should be cheap. Let $S_1, S_2, \ldots, S_L$ be a partition of $\{1, \ldots, n\}$, that is $S_i \neq \emptyset$, $S_i \cap S_j = \emptyset$ $(i \neq j)$ and $\cup_{i=1}^{L} S_i = \{1, \ldots, n\}$. Furthermore let $S_i \subseteq T_i \subseteq \{1, \ldots, n\}$ $(i = 1, \ldots, L)$ be such

that $S_l \neq T_l$ for at least one l.

The ***non-overlapping block Jacobi splitting*** of A is given by

$$M_l = \left[M_{ij}^{(l)}\right]_{i,j=1}^{n}, \quad M_{ij}^{(l)} = \begin{cases} a_{ij}, & \text{if } i,j \in S_l \\ a_{ii}, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases},$$

$$N_l = M_l - A,$$

$$E_l = \left[E_{ij}^{(l)}\right]_{i,j=1}^{n}, \quad E_{ij}^{(l)} = \begin{cases} 1, & \text{if } i = j \in S_l \\ 0, & \text{otherwise} \end{cases}$$

for $l = 1, \ldots, L$.

Define now the simple splitting

$$A = M - N,$$

where M is nonsingular,

$$M = \left[M_{ij}\right]_{i,j=1}^{n}, \quad M_{ij} = \begin{cases} a_{ij}, & \text{if } i,j \in S_l \text{ for some } l \in \{1, \ldots, n\} \\ 0, & \text{otherwise} \end{cases}.$$

It can be shown that

$$H = \sum_{l=1}^{L} E_l M_l^{-1} N_l = M^{-1} N$$

holds for the non-overlapping block Jacobi multisplitting.

The ***overlapping block Jacobi multisplitting*** of A is defined by

$$\widetilde{M}_l = \left[\widetilde{M}_{ij}^{(l)}\right]_{i,j=1}^{n}, \quad \widetilde{M}_{ij}^{(l)} = \begin{cases} a_{ij}, & \text{if } i,j \in T_l \\ a_{ii}, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases},$$

$$\widetilde{N}_l = \widetilde{M}_l - A,$$

$$\widetilde{e}_l = \left[\widetilde{e}_{ij}^{(l)}\right]_{i,j=1}^{n}, \quad E_{ii}^{(l)} = 0, \text{if } i \notin T_l$$

for $l = 1, \ldots, L$.

A nonsingular matrix $A \in \mathbb{R}^{n \times n}$ is called an M-matrix, if $a_{ij} \leq 0$ $(i \neq j)$ and all the elements of A^{-1} are nonnegative.

Theorem 19.12 *Assume that $A \in \mathbb{R}^{n \times n}$ is nonsingular M-matrix, $\{M_i, N_i, E_i\}_{i=1}^{L}$ is a non-overlapping, $\left\{\widetilde{M}_i, \widetilde{N}_i, E_i\right\}_{i=1}^{L}$ is an overlapping block Jacobi multisplitting of A, where the weighting matrices E_i are the same. The we have*

$$\rho\left(\widetilde{H}\right) \leq \rho\left(H\right) < 1,$$

where $H = \sum_{l=1}^{L} E_l M_l^{-1} N_l$ and $\widetilde{H} = \sum_{l=1}^{L} E_l \widetilde{M}_l^{-1} \widetilde{N}_l$.

We can observe that both iteration procedures are convergent and the convergence of the overlapping multisplitting is not slower than that of the non-overlapping procedure. The theorem remains true if we use block Gauss-Seidel multisplittings instead of the block Jacobi multisplittings. In this case we replace the above defined matrices M_i and $\widetilde{M}_i$ with their lower triangular parts.

The multisplitting algorithm has multi-stage and asynchronous variants as well.

19.2.3. Error analysis of linear algebraic systems

We analyze the direct and inverse errors. We use the following notations and concepts. The exact (theoretical) solution of $Ax = b$ is denoted by x, while any approximate solution is denoted by $\hat{x}$. The direct error of the approximate solution is given by $\Delta x = \hat{x} - x$. The quantity $r = r(y) = Ay - b$ is called the ***residual error***. For the exact solution $r(x) = 0$, while for the approximate solution

$$r(\hat{x}) = A\hat{x} - b = A(\hat{x} - x) = A\Delta x.$$

We use various models to estimate the inverse error. In the most general case we assume that the computed solution $\hat{x}$ satisfies the linear system$\hat{A}\hat{x} = \hat{b}$, where $\hat{A} = A + \Delta A$ and $\hat{b} = b + \Delta b$. The quantities ΔA and Δb are called the inverse errors.

One has to distinguish between the sensitivity of the problem and the stability of the solution algorithm. By ***sensitivity of a problem*** we mean the sensitivity of the solution to changes in the input parameters (data). By the ***stability (or sensitivity) of an algorithm*** we mean the influence of computational errors on the computed solution. We measure the sensitivity of a problem or algorithm in various ways. One such characterization is the *condition number*„condition number", which compares the relative errors of the input and output values.

The following general principles are used when applying any algorithm:

- We use only stable or well-conditioned algorithms.
- We cannot solve an unstable (ill-posed or ill-conditioned) problem with a general purpose algorithm, in general.

Sensitivity analysis

Assume that we solve the perturbed equation

$$A\hat{x} = b + \Delta b \tag{19.16}$$

instead of the original $Ax = b$. Let $\widehat{x} = x + \Delta x$ and investigate the differene of the two solutions.

Theorem 19.13 *If A is nonsingular and $b \neq 0$, then*

$$\frac{\|\Delta x\|}{\|x\|} \leq \operatorname{cond}(A)\frac{\|\Delta b\|}{\|b\|} = \operatorname{cond}(A)\frac{\|r(\hat{x})\|}{\|b\|}, \tag{19.17}$$

where $\operatorname{cond}(A) = \|A\|\left\|A^{-1}\right\|$ *is the condition number of A.*

Here we can see that the condition number of A may strongly influence the relative error of the perturbed solution $\widehat{x}$. A linear algebraic system is said to be *well-conditioned*

if cond(A) is small, and *ill-conditioned*, if cond(A) is big. It is clear that the terms „small" and „big" are relative and the condition number depends on the norm chosen. We identify the applied norm if it is essential for some reason. For example $\text{cond}_\infty(A) = \|A\|_\infty \left\|A^{-1}\right\|_\infty$. The next example gives possible geometric characterization of the condition number.

19.7. Example. The linear system

$$\begin{array}{rcrcl} 1000x_1 & + & 999x_2 & = & b_1 \\ 999x_1 & + & 998x_2 & = & b_2 \end{array}$$

is ill-conditioned ($\text{cond}_\infty(A) = 3.99 \times 10^6$). The two lines, whose meshpoint defines the system, are almost parallel. Therefore if we perturb the right hand side, the new meshpoint of the two lines will be far from the previous meshpoint.

The inverse error is Δb in the sensitivity model under investigation. Theorem 19.13 gives an estimate of the direct error which conforms with the thumb rule. It follows that we can expect a small relative error of the perturbed solution $\widehat{x}$, if the condition number of A is small.

19.8. Example. Consider the linear system $Ax = b$ with

$$A = \begin{bmatrix} 1+\epsilon & 1 \\ 1 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad x = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Let $\hat{x} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$. Then $r = \begin{bmatrix} 2\epsilon \\ 0 \end{bmatrix}$ and $\|r\|_\infty / \|b\|_\infty = 2\epsilon$, but $\|\hat{x} - x\|_\infty / \|x\|_\infty = 2$.

Consider now the perturbed linear system

$$(A + \Delta A)\,\hat{x} = b \tag{19.18}$$

instead of $Ax = b$. It can be proved that for this perturbation model there exist more than one *inverse errors*„inverse error" among which $\Delta A = -r(\widehat{x})\,\widehat{x}^T / \widehat{x}^T\widehat{x}$ is the inverse error with minimal spectral norm, provided that $\widehat{x}, r(\widehat{x}) \neq 0$.

The following theorem establish that for small relative residual error the relative inverse error is also small.

Theorem 19.14 *Assume that $\hat{x} \neq 0$ is the approximate solution of $Ax = b$, $\det(A) \neq 0$ and $b \neq 0$. If $\|r(\hat{x})\|_2 / \|b\|_2 = \alpha < 1$, the the matrix $\Delta A = -r(\hat{x})\,\hat{x}^T / \hat{x}^T\hat{x}$ satisfies $(A + \Delta A)\,\hat{x} = b$ and $\|\Delta A\|_2 / \|A\|_2 \leq \alpha / (1 - \alpha)$.*

If the relative inverse error and the condition number of A are small, then the relative residual error is small.

Theorem 19.15 *If $r(\hat{x}) = A\hat{x} - b$, $(A + \Delta A)\,\hat{x} = b$, $A \neq 0$, $b \neq 0$ and $\text{cond}(A)\,\frac{\|\Delta A\|}{\|A\|} < 1$, then*

$$\frac{\|r(\hat{x})\|}{\|b\|} \leq \frac{\text{cond}(A)\,\frac{\|\Delta A\|}{\|A\|}}{1 - \text{cond}(A)\,\frac{\|\Delta A\|}{\|A\|}}. \tag{19.19}$$

If A is ill-conditioned, then Theorem 19.15 is not true.

19.9. Example. Let $A = \begin{bmatrix} 1+\epsilon & 1 \\ 1 & 1-\epsilon \end{bmatrix}$, $\Delta A = \begin{bmatrix} 0 & 0 \\ 0 & \epsilon^2 \end{bmatrix}$ and $b = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, $(0 < \epsilon \ll 1)$. Then $\operatorname{cond}_\infty(A) = (2+\epsilon)^2/\epsilon^2 \approx 4/\epsilon^2$ and $\|\Delta A\|_\infty / \|A\|_\infty = \epsilon^2/(2+\epsilon) \approx \epsilon^2/2$. Let

$$\hat{x} = (A+\Delta A)^{-1} b = \frac{1}{\epsilon^3}\begin{bmatrix} 2-\epsilon+\epsilon^2 \\ -2-\epsilon \end{bmatrix} \approx \begin{bmatrix} 2/\epsilon^3 \\ -2/\epsilon^3 \end{bmatrix}.$$

Then $r(\hat{x}) = A\hat{x} - b = \begin{bmatrix} 0 \\ 2/\epsilon + 1 \end{bmatrix}$ and $\|r(\hat{x})\|_\infty / \|b\|_\infty = 2/\epsilon + 1$, which is not small.

In the most general case we solve the perturbed equation

$$(A+\Delta A)\,\hat{x} = b + \Delta b \tag{19.20}$$

instead of $Ax = b$. The following general result holds.

Theorem 19.16 *If A is nonsingular, $\operatorname{cond}(A)\frac{\|\Delta A\|}{\|A\|} < 1$ and $b \neq 0$, then*

$$\frac{\|\Delta x\|}{\|x\|} \leq \frac{\operatorname{cond}(A)\left(\frac{\|\Delta A\|}{\|A\|} + \frac{\|\Delta b\|}{\|b\|}\right)}{1 - \operatorname{cond}(A)\frac{\|\Delta A\|}{\|A\|}}. \tag{19.21}$$

This theorem implies the following „thumb rule".

Thumb rule. *Assume that $Ax = b$. If the entries of A and b are accurate to about s decimal places and $\operatorname{cond}(A) \sim 10^t$, where $t < s$, then the entries of the computed solution are accurate to about $s - t$ decimal places.*

The assumption $\operatorname{cond}(A)\|\Delta A\|/\|A\| < 1$ of Theorem 19.16 guarantees that that matrix $A + \Delta A$ is nonsingular. The inequality $\operatorname{cond}(A)\|\Delta A\|/\|A\| < 1$ is equaivalent with the inequality $\|\Delta A\| < \frac{1}{\|A^{-1}\|}$ and the distance of A from the nearest singular matrix is just $1/\left\|A^{-1}\right\|$. Thus we can give a new characterization of the condition number:

$$\frac{1}{\operatorname{cond}(A)} = \min_{A+\Delta A \text{ is singular}} \frac{\|\Delta A\|}{\|A\|}. \tag{19.22}$$

Thus if a matrix is ill-conditioned, then it is close to a singular matrix. Earlier we defined the condition numbers of matrices as the condition number of the mapping $F(x) = A^{-1}x$.

Let us introduce the following definition.

Definition 19.17 *A linear system solver is said to be **weakly stable** on a matrix class H, if for all well-conditioned $A \in H$ and for all b, the computed solution $\widehat{x}$ of the linear system $Ax = b$ has small relative error $\|\hat{x} - x\|/\|x\|$.*

Putting together Theorems 19.13–19.16 we obtain the following.

Theorem 19.18 (Bunch). *A linear system solver is weakly stable on a matrix class H, if for all well-conditioned $A \in H$ and for all b, the computed solution $\widehat{x}$ of the linear system $Ax = b$ satisfies any of the following conditions:*
(1) $\|\hat{x} - x\|/\|x\|$ is small;
(2) $\|r(\hat{x})\|/\|b\|$ is small;
(3) There exists ΔA such that $(A+\Delta A)\,\hat{x} = b$ and $\|\Delta A\|/\|A\|$ are small.

The estimate of Theorem 19.16 can be used in practice if we know estimates of $\Delta b, \Delta A$ and $\operatorname{cond}(A)$. If no estimates are available, then we can only make a posteriori error estimates.

In the following we study the componentwise error estimates. We first give an estimate for the absolute error of the approximate solution using the components of the inverse error.

Theorem 19.19 (Bauer, Skeel). *Let $A \in \mathbb{R}^n$ be nonsingular and assume that the approximate solution $\widehat{x}$ of $Ax = b$ satisfies the linear system $(A+E)\widehat{x} = b + e$. If $S \in \mathbb{R}^{n\times n}$, $s \in \mathbb{R}^n$ and $\varepsilon > 0$ are such that $S \geq 0$, $s \geq 0$, $|E| \leq \varepsilon S$, $|e| \leq \varepsilon s$ and $\varepsilon \left\| \left| A^{-1} \right| S \right\|_\infty < 1$, then*

$$\left\| \widehat{x} - x \right\|_\infty \leq \frac{\varepsilon \left\| \left| A^{-1} \right| (S\,|x| + s) \right\|_\infty}{1 - \varepsilon \left\| \left| A^{-1} \right| S \right\|_\infty}. \tag{19.23}$$

If $e = 0$ ($s = 0$), $S = |A|$ and

$$k_r(A) = \left\| \left| A^{-1} \right| |A| \right\|_\infty < 1, \tag{19.24}$$

then we obtain the estimate

$$\left\| \widehat{x} - x \right\|_\infty \leq \frac{\varepsilon k_r(A)}{1 - \varepsilon k_r(A)}. \tag{19.25}$$

The quantity $k_r(A)$ is said to be ***Skeel-norm***, although it is not a norm in the earlier defined sense. The Skeel-norm satisfies the inequality

$$k_r(A) \leq \operatorname{cond}_\infty(A) = \|A\|_\infty \left\| A^{-1} \right\|_\infty. \tag{19.26}$$

Therefore the above estimate is not worse than the traditional one that uses the standard condition number.

The inverse error can be estimated componentwise by the following result of Oettli and Prager. Let $A, \delta A \in \mathbb{R}^{n\times n}$ and $b, \delta b \in \mathbb{R}^n$. Assume that $\delta A \geq 0$ and $\delta b \geq 0$. Furthermore let

$$D = \{\Delta A \in \mathbb{R}^{n\times n} : |\Delta A| \leq \delta A\}, \quad G = \{\Delta b \in \mathbb{R}^n : |\Delta b| \leq \delta b\}.$$

Theorem 19.20 (Oettli, Prager). *The computed solution $\hat{x}$ satisfies a perturbed equation $(A + \Delta A)\hat{x} = b + \Delta b$ with $\Delta A \in D$ and $\Delta b \in G$, if*

$$|r(\hat{x})| = |A\hat{x} - b| \leq \delta A\,|\hat{x}| + \delta b. \tag{19.27}$$

We do not need the condition number to apply this theorem. In practice the entries δA and δb are proportional to the machine epsilon.

Theorem 19.21 (Wilkinson). *The approximate solution $\widehat{x}$ of $Ax = b$ obtained by the Gauss method in floating point arithmetic satisfies the perturbed linear equation*

$$(A + \Delta A)\widehat{x} = b \tag{19.28}$$

with

$$\|\Delta A\|_\infty \leq 8n^3 \rho_n \|A\|_\infty u + O(u^2), \tag{19.29}$$

where ρ_n denotes the groth factor of the pivot elements and u is the unit roundoff.

Since ρ_n is small in practice, the realtive error

$$\frac{\|\Delta A\|_\infty}{\|A\|_\infty} \le 8n^3\rho_n u + O(u^2)$$

is also small. Therefore Theorem19.18 implies that the Gauss method is weakly stable both for full and partial pivoting.

Wilkinson's theorem implies that

$$\mathrm{cond}_\infty(A)\frac{\|\Delta A\|_\infty}{\|A\|_\infty} \le 8n^3\rho_n \mathrm{cond}_\infty(A)\,u + O\left(u^2\right).$$

For a small condition number we can assume that $1-\mathrm{cond}_\infty(A)\,\|\Delta A\|_\infty / \|A\|_\infty \approx 1$. Using Theorems 19.21 and 19.16 (case $\Delta b = 0$) we obtain the following estimate of the direct error:

$$\frac{\|\Delta x\|_\infty}{\|x\|_\infty} \le 8n^3\rho_n \mathrm{cond}_\infty(A)\,u. \tag{19.30}$$

The obtained result supports the thumb rule in the case of the Gauss method.

19.10. Example. Consider the following linear system whose coefficients can be represented exactly:

$$\begin{aligned} 888445x_1 + 887112x_2 &= 1,\\ 887112x_1 + 885781x_2 &= 0. \end{aligned}$$

Here $\mathrm{cond}(A)_\infty$ is big, but $\mathrm{cond}_\infty(A)\|\Delta A\|_\infty/\|A\|_\infty$ is negligible. The exact solution of the problem is $x_1 = 885781$, $x_2 = -887112$. The MATLAB gives the approximate solution $\hat{x}_1 = 885827.23$, $\hat{x}_2 = -887158.30$ with the relative error

$$\frac{\|x-\hat{x}\|_\infty}{\|x\|_\infty} = 5.22\times 10^{-5}.$$

Since $s \approx 16$ and $\mathrm{cond}\,(A)_\infty \approx 3.15\times 10^{12}$, the result essentially corresponds to the Wilkinson theorem or the thumb rule. The Wilkinson theorem gives the bound

$$\|\Delta A\|_\infty \le 1.26\times 10^{-8}$$

for the inverse error. If we use the Oettli-Prager theorem with the choice $\delta A = \epsilon_M |A|$ and $\delta b = \epsilon_M |b|$, then we obtain the estimate $|r(\hat{x})| \le \delta A\,|\hat{x}| + \delta b$. Since $\||\delta A|\|_\infty = 3.94\times 10^{-10}$, this estimate is better than that of Wilkinson.

Scaling and preconditioning

Several matrices that occur in applications are ill-conditioned if their order n is large. For example the famous Hilbert-matrix

$$H_n = \left[\frac{1}{i+j-1}\right]_{i,j=1}^{n} \tag{19.31}$$

has $\mathrm{cond}_2\,(H_n) \approx e^{3.5n}$, if $n \to \infty$. There exist $2n\times 2n$ matrices with integer entries that can be represented exactly in standard IEEE754 floating point arithmetic while their condition number is approximately 4×10^{32n}.

We have two main techniques to solve linear systems with large condition numbers.

Either we use multiple precision arithmetic or decrease the condition number. There are two known forms of decreasing the condition number.

1. Scaling We replace the linear system $Ax = b$ with the equation

$$(RAC)\, y = (Rb)\,, \tag{19.32}$$

where R and C are diagonal matrices.

We apply the Gauss method to this scaled system and get the solution y. The quantity $x = Cy$ defines the requested solution. If the condition number of the matrix RAC is smaller then we expect a smaller error in y and consequently in x. Various strategies are given to choose the scaling matrices R and C. One of the best known strategies is the ***balancing*** which forces every column and row of RAC to have approximately the same norm. For example, if

$$\hat{D} = \operatorname{diag}\left(\frac{1}{\left\|a_1^T\right\|_2}, \ldots, \frac{1}{\left\|a_n^T\right\|_2}\right)$$

where a_i^T is the i^{th} row vector of A, the Euclidean norms of the rows of $\hat{D}A$ will be 1 and the estimate

$$\operatorname{cond}_2\left(\hat{D}A\right) \leq \sqrt{n} \min_{D \in D_+} \operatorname{cond}_2(DA)$$

holds with $D_+ = \{\operatorname{diag}(d_1, \ldots, d_n) \mid d_1, \ldots, d_n > 0\}$. This means that $\hat{D}$ optimally scales the rows of A in an approximate sense.

The next example shows that the scaling may lead to bad results.

19.11. Example. Consider the matrix

$$A = \begin{bmatrix} \epsilon/2 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

for $0 < \epsilon \ll 1$. It is easy to show that $\operatorname{cond}_\infty(A) = 12$. Let

$$R = C = \begin{bmatrix} 2/\sqrt{\epsilon} & 0 & 0 \\ 0 & \sqrt{\epsilon}/2 & 0 \\ 0 & 0 & \sqrt{\epsilon}/2 \end{bmatrix}.$$

Then the scaled matrix

$$RAR = \begin{bmatrix} 2 & 1 & 1 \\ 1 & \epsilon/4 & \epsilon/4 \\ 1 & \epsilon/4 & \epsilon/2 \end{bmatrix},$$

has the condition number $\operatorname{cond}_\infty(RAR) = 32/\epsilon$, which a very large value for small ϵ.

2. Preconditioning The preconditioning is very close to scaling. We rewrite the linear system $Ax = b$ with the equivalent form

$$\widetilde{A}x = (MA)\, x = Mb = \tilde{b}, \tag{19.33}$$

where matrix M is such that $\operatorname{cond}\left(M^{-1}A\right)$ is smaller and $Mz = y$ is easily solvable.

The preconditioning is often used with iterative methods on linear systems with symmetric and positive definite matrices.

A posteriori error estimates

The a posteriori estimate of the error of an approximate solution is necessary to get some information on the reliability of the obtained result. There are plenty of such estimates. Here we show three estimates whose computational cost is $\Theta\left(n^2\right)$ flops. This cost is acceptable when comparing to the cost of direct or iterative methods ($\Theta\left(n^3\right)$ or $\Theta\left(n^2\right)$ per iteration step).

The estimate of the direct error with the residual error

Theorem 19.22 (Auchmuty). *Let $\widehat{x}$ be the approximate solution of $Ax = b$. Then*

$$\|x - \hat{x}\|_2 = \frac{c\,\|r(\hat{x})\|_2^2}{\left\|A^T r(\hat{x})\right\|_2},$$

where $c \geq 1$.

The error constant c depends on A and the direction of error vector $\hat{x} - x$. Furthermore

$$\frac{1}{2}\mathrm{cond}_2(A) \approx C_2(A) = \frac{1}{2}\left(\mathrm{cond}_2(A) + \frac{1}{\mathrm{cond}_2(A)}\right) \leq \mathrm{cond}_2(A).$$

The error constant c takes the upper value $C_2(A)$ only in exceptional cases. The computational experiments indicate that the average value of c grows slowly with the order of A and it depends more strongly on n than the condition number of A. The following experimental estimate

$$\left\|x - \widehat{x}\right\|_2 \lessapprox 0.5 \dim(A) \left\|r\left(\widehat{x}\right)\right\|_2^2 / \left\|A^T r\left(\widehat{x}\right)\right\|_2 \tag{19.34}$$

seems to hold with a high degree of probability.

The LINPACK estimate of $\left\|A^{-1}\right\|$

The famous LINPACK program package uses the following process to estimate $\left\|A^{-1}\right\|$. We solve the linear systems $A^T y = d$ and $Aw = y$. Then the estimate of $\left\|A^{-1}\right\|$ is given by

$$\left\|A^{-1}\right\| \approx \frac{\|w\|}{\|y\|} \quad \left(\leq \left\|A^{-1}\right\|\right). \tag{19.35}$$

Since

$$\frac{\|w\|}{\|y\|} = \frac{\left\|A^{-1}\left(A^{-T} d\right)\right\|}{\left\|A^{-T} d\right\|},$$

we can interpret the process as an application of the power method of the eigenvalue problem. The estimate can be used with the $1-$, $2-$ and ∞-norms. The entries of vector d are ± 1 possibly with random signs.

If the linear system $Ax = b$ is solved by the LU-method, then the solution of further linear systems costs $\Theta(n^2)$ flops per system. Thus the total cost of the LINPACK estimate remains small. Having the estimate $\left\|A^{-1}\right\|$ we can easily estimate $\mathrm{cond}(A)$ and the error of the approximate solution (cf. Theorem 19.16 or the thumb rule). We remark that several similar processes are known in the literature.

The Oettli-Prager estimate of the inverse error

We use the Oettli-Prager theorem in the following form. Let $r(\hat{x}) = A\hat{x} - b$ be the residual error, $E \in \mathbb{R}^{n\times n}$ and $f \in \mathbb{R}^n$ are given such that $E \geq 0$ and $f \geq 0$. Let

$$\omega = \max_i \frac{|r(\hat{x})_i|}{(E|\hat{x}| + f)_i},$$

where $0/0$ is set to 0-nak, $\rho/0$ is set to ∞, if $\rho \neq 0$. Symbol $(y)_i$ denotes the i^{th} component of the vector y. If $\omega \neq \infty$, then there exist a matrix ΔA and a vector Δb for which

$$|\Delta A| \leq \omega E, \quad |\Delta b| \leq \omega f$$

holds and

$$(A + \Delta A)\hat{x} = b + \Delta b.$$

Moreover ω is the smallest number for which ΔA and Δb exist with the above properties. The quantity ω measures the relative inverse error in terms of E and f. If for a given E, f and $\hat{x}$, the quantity ω is small, then the perturbed problem (and its solution) are close to the original problem (and its solution). In practice, the choice $E = |A|$ and $f = |b|$ is preferred

Iterative refinement

Denote by $\hat{x}$ the approximate solution of $Ax = b$ and let $r(y) = Ay - b$ be the residual error at the point y. The precision of the approximate solution $\hat{x}$ can be improved with the following method.

ITERATIVE-REFINEMENT$(A, b, \hat{x}, tol)$

```
1  k ← 1
2  x_1 ← x̂
3  d̂ ← inf
4  while = ||d̂|| / ||x_k|| > tol
5      do r ← Ax_k − b
6          Compute the approximate solution d̂ of Ad = r with the LU-method.
7          x_{k+1} ← x_k − d̂
8          k ← k + 1
9  return x_k
```

There are other variants of this process. We can use other linear solvers instead of the LU-method.

Let η be the smallest bound of relative inverse error with

$$(A + \Delta A)\hat{x} = b + \Delta b, \quad |\Delta A| \leq \eta |A|, \quad |\Delta b| \leq \eta |b|.$$

Furthermore let

$$\sigma(A, x) = \max_k (|A||x|)_k / \min_k (|A||x|)_k, \quad \min_k (|A||x|)_k > 0.$$

Theorem 19.23 (Skeel). *If $k_r\left(A^{-1}\right)\sigma(A, x) \leq c_1 < 1/\epsilon_M$, then for sufficiently large k we have*

$$(A + \Delta A)\, x_k = b + \Delta b, \quad |\Delta A| \leq 4\eta\epsilon_M |A|, \quad |\Delta b| \leq 4\eta\epsilon_M |b|. \tag{19.36}$$

This result often holds after the first iteration, i.e. for $k = 2$. Jankowski and Wozniakowski investigated the iterative refinement for any method ϕ which produces an approximate solution $\widehat{x}$ with relative error less than 1. They showed that the iterative refinement improves the precision of the approximate solution even in single precision arithmetic and makes method ϕ to be weakly stable.

Exercises

19.2-1 Prove Theorem 19.7.

19.2-2 Consider the linear systems $Ax = b$ and $Bx = b$, where

$$A = \begin{bmatrix} 1 & 1/2 \\ 1/2 & 1/3 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & -1/2 \\ 1/2 & 1/3 \end{bmatrix}$$

and $b \in \mathbb{R}^2$. Which equation is more sensitive to the perturbation of b? What should be the *relative error* of b in the more sensitive equation in order to get the solutions of both equations with the same precision?

19.2-3 Let $\chi = 3/2^{29}$, $\zeta = 2^{14}$ and

$$A = \begin{bmatrix} \chi\zeta & -\zeta & \zeta \\ \zeta^{-1} & \zeta^{-1} & 0 \\ \zeta^{-1} & -\chi\zeta^{-1} & \zeta^{-1} \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1+\varepsilon \\ 1 \end{bmatrix}.$$

Solve the linear systems $Ax = b$ for $\varepsilon = 10^{-1}, 10^{-3}, 10^{-5}, 10^{-7}, 10^{-10}$. Explain the results.

19.2-4 Let A be a 10×10 matrix and choose the band matrix consisting of the main and the neighboring two subdiagonals of A as a preconditioning matrix. How much does the *condition number* of A improves if (i) A is a random matrix; (ii) A is a Hilbert matrix?

19.2-5 Let

$$A = \begin{bmatrix} 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \\ 1/4 & 1/5 & 1/6 \end{bmatrix},$$

and assume that ε is the common error bound of every component of $b \in \mathbb{R}^3$. Give the sharpest possible error bounds for the solution $[x_1, x_2, x_3]^T$ of the equation $Ax = b$ and for the sum $(x_1 + x_2 + x_3)$.

19.2-6 Consider the linear system $Ax = b$ with the approximate solution $\hat{x}$.
(i) Give an error bound for $\hat{x}$, if $(A + E)\hat{x} = b$ holds exactly and both A and $A + E$ is nonsingular.
(ii) Let

$$A = \begin{bmatrix} 10 & 7 & 8 \\ 7 & 5 & 6 \\ 8 & 6 & 10 \end{bmatrix}, \quad b = \begin{bmatrix} 25 \\ 18 \\ 24 \end{bmatrix}$$

and consider the solution of $Ax = b$. Give (if possible) a relative error bound for the entries of A such that the integer part of every solution component remains constant within the range of this relative error bound.

19.3. Eigenvalue problems

The set of complex n-vectors will be denoted by $\mathbb{C}^n$. Similarly, $\mathbb{C}^{m\times n}$ denotes the set of *complex $m \times n$ matrices.*

Definition 19.24 *Let $A \in \mathbb{C}^{n\times n}$ be an arbitrary matrix. The number $\lambda \in \mathbb{C}$ is the* ***eigenvalue*** *of A if there is vector $x \in \mathbb{C}^n$ ($x \neq 0$) such that*

$$Ax = \lambda x. \tag{19.37}$$

Vector x is called the (right) eigenvector of A that belongs to the eigenvalue λ.

Equation $Ax = \lambda x$ can be written in the equivalent form $(A-\lambda I)x = 0$, where I is the unit matrix of appropriate size. The latter homogeneous linear system has a nonzero solution x if and only if

$$\phi(\lambda) = \det(A - \lambda I) = \det\left(\begin{bmatrix} a_{11}-\lambda & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22}-\lambda & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn}-\lambda \end{bmatrix}\right) = 0\,. \tag{19.38}$$

Equation (19.38) is called the ***characteristic equation*** of matrix A. The roots of this equation are the eigenvalues of matrix A. Expanding $\det(A - \lambda I)$ we obtain a polynomial of degree n:

$$\phi(\lambda) = (-1)^n(\lambda^n - p_1\lambda^{n-1} - \ldots - p_{n-1}\lambda - p_n).$$

This polynomial called the ***characteristic polynomial*** of A. It follows from the fundamental theorem of algebra that any matrix $A \in \mathbb{C}^{n\times n}$ has exactly n eigenvalues with multiplicities. The eigenvalues may be complex or real. Therefore one needs to use complex arithmetic for eigenvalue calculations. If the matrix is real and the computations are done in real arithmetic, the complex eigenvalues and eigenvectors can be determined only with special techniques.

If $x \neq 0$ is an eigenvector, $t \in \mathbb{C}$ ($t \neq 0$), then tx is also eigenvector. The number of linearly independent eigenvectors that belong to an eigenvalue λ_k does not exceed the multiplicity of λ_k in the characteristic equation (19.38). The eigenvectors that belong to different eigenvalues are linearly independent.

The following results give estimates for the size and location of the eigenvalues.

Theorem 19.25 *Let λ be any eigenvalue of matrix A. The upper estimate $|\lambda| \leq \|A\|$ holds in any induced matrix norm.*

Theorem 19.26 (Gersgorin). *Let $A \in \mathbb{C}^{n\times n}$,*

$$r_i = \sum_{j=1, j\neq i}^{n} \left|a_{ij}\right| \quad (i = 1, \ldots, n)$$

and

$$D_i = \{z \in \mathbf{C} \,|\, |z - a_{ii}| \leq r_i\} \quad (i = 1, \ldots, n)\,.$$

Then for any eigenvalue λ of A we have $\lambda \in \cup_{i=1}^{n} D_i$.

For certain matrices the solution of the characteristic equation (19.38) is very easy. For example, if A is a triangular matrix, then its eigenvalues are entries of the main diagonal. In most cases however the computation of all eigenvalues and eigenvectors is a very difficult task. Those transformations of matrices that keeps the eigenvalues unchanged have practical significance for this problem. Later we see that the eigenvalue problem of transformed matrices is simpler.

Definition 19.27 *The matrices $A, B \in \mathbb{C}^{n\times n}$ are similar if there is a matrix T such that $B = T^{-1}AT$. The mapping $A \to T^{-1}AT$ is said to be* ***similarity transformation*** *of A.*

Theorem 19.28 *Assume that* $\det(T) \neq 0$. *Then the eigenvalues of A and $B = T^{-1}AT$ are the same. If x is the eigenvector of A, then $y = T^{-1}x$ is the eigenvector of B.*

Similar matrices have the same eigenvalues.

The difficulty of the eigenvalue problem also stems from the fact that the eigenvalues and eigenvectors are very sensitive (unstable) to changes in the matrix entries. The eigenvalues of A and the perturbed matrix $A + \delta A$ may differ from each other significantly. Besides the multiplicity of the eigenvalues may also change under perturbation. The following theorems and examples show the very sensitivity of the eigenvalue problem.

Theorem 19.29 (Ostrowski, Elsner). *For every eigenvalue λ_i of matrix $A \in \mathbb{C}^{n\times n}$ there exists an eigenvalue μ_k of the perturbed matrix $A + \delta A$ such that*

$$|\lambda_i - \mu_k| \leq (2n-1)\left(\|A\|_2 + \|A + \delta A\|_2\right)^{1-\frac{1}{n}} \|\delta A\|_2^{\frac{1}{n}}.$$

We can observe that the eigenvalues are changing continuously and the size of change is proportional to the n^{th} root of $\|\delta A\|_2$.

19.12. Example. Consider the following perturbed *Jordan matrix* of the size $r \times r$:

$$\begin{bmatrix} \mu & 1 & 0 & \dots & 0 \\ 0 & \mu & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & & \ddots & \mu & 1 \\ \epsilon & 0 & \dots & 0 & \mu \end{bmatrix}.$$

The characteristic equation is $(\lambda - \mu)^r = \epsilon$, which gives the r different eigenvalues

$$\lambda_s = \mu + \epsilon^{1/r}\left(\cos(2s\pi/r) + i\sin(2s\pi/r)\right) \quad (s = 0, \dots, r-1)$$

instead of the original eigenvalue μ with multiplicity r. The size of change is $\epsilon^{1/r}$, which corresponds to Theorem (19.29). If $|\mu| \approx 1$, $r = 16$ and $\epsilon = \epsilon_M \approx 2.2204 \times 10^{-16}$, then the perturbation size of the eigenvalues is ≈ 0.1051. This is a significant change relative to the input perturbation ϵ.

For special matrices and perturbations we may have much better perturbation bounds.

Theorem 19.30 (Bauer, Fike). *Assume that $A \in \mathbb{C}^{n\times n}$ is* ***diagonalizable*** *, that is a matrix X exists such that $X^{-1}AX = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$. Denote μ an eigenvalue of $A + \delta A$. Then*

$$\min_{1\leq i\leq n} |\lambda_i - \mu| \leq \operatorname{cond}_2(X)\,\|\delta A\|_2. \tag{19.39}$$

This result is better than that of Ostrowski and Elsner. Nevertheless $\mathrm{cond}_2(X)$, which is generally unknown, can be very big.

The eigenvalues are continuous functions of the matrix entries. This is also true for the normalized eigenvectors if the eigenvalues are simple. The following example shows that this property does not hold for multiple eigenvalues.

19.13. Example. Let

$$A(t) = \begin{bmatrix} 1 + t\cos(2/t) & -t\sin(2/t) \\ -t\sin(2/t) & 1 - t\cos(2/t) \end{bmatrix} \quad (t \neq 0).$$

The eigenvalues of $A(t)$ are $\lambda_1 = 1 + t$ and $\lambda_2 = 1 - t$. Vector $[\sin(1/t), \cos(1/t)]^T$ is the eigenvector belonging to λ_1. Vector $[\cos(1/t), -\sin(1/t)]^T$ is the eigenvector belonging to λ_2. If $t \to 0$, then

$$A(t) \to I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \lambda_1, \lambda_2 \to 1,$$

while the eigenvectors do not have limit.

We study the numerical solution of the eigenvalue problem in the next section. Unfortunately it is very difficult to estimate the goodness of numerical approximations. From the fact that $Ax - \lambda x = 0$ holds with a certain error we cannot conclude anything in general.

19.14. Example. Consider the matrix

$$A(\epsilon) = \begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix},$$

where $\epsilon \approx 0$ is small. The eigenvalues of $A(\epsilon)$ are $1 \pm \sqrt{\epsilon}$, while the corresponding eigenvectors are $[1, \pm\sqrt{\epsilon}]^T$. Let $\mu = 1$ be an approximation of the eigenvalues and let $x = [1, 0]^T$ be the approximate eigenvector. Then

$$\|Ax - \mu x\|_2 = \left\| \begin{bmatrix} 0 \\ \epsilon \end{bmatrix} \right\|_2 = \epsilon.$$

If $\epsilon = 10^{-10}$, then the residual error under estimate the true error 10^{-5} by five order.

Remark 19.31 *We can define the* ***condition number of eigenvalues*** *for simple eigenvalues:*

$$\nu(\lambda_1) \approx \frac{\|x\|_2 \|y\|_2}{\left|x^H y\right|},$$

where x and y are the right and left eigenvectors, respectively. For multiple eigenvalues the condition number is not finite.

19.3.1. Iterative solutions of the eigenvalue problem

We investigate only the real eigenvalues and eigenvectors of real matrices. The methods under consideration can be extended to the complex case with appropriate modifications.

The power method

This method is due to von Mieses. Assume that $A \in \mathbb{R}^{n\times n}$ has exactly n different real eigenvalues. Then the eigenvectors $x_1, \ldots, x_n$ belonging to the corresponding eigenvalues $\lambda_1, \ldots, \lambda_n$ are linearly independent. Assume that the eigenvalues satisfy the condition

$$|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$$

and let $v^{(0)} \in \mathbb{R}^n$ be a given vector. This vector is a unique linear combination of the eigenvectors, that is $v^{(0)} = \alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_n x_n$. Assume that $\alpha_1 \neq 0$ and compute the sequence $v^{(k)} = Av^{(k-1)} = A^k v^{(0)}$ $(k = 1, 2, \ldots)$. The initial assumptions imply that

$$\begin{aligned} v^{(k)} = Av^{(k-1)} &= A(\alpha_1 \lambda_1^{k-1} x_1 + \alpha_2 \lambda_2^{k-1} x_2 + \ldots + \alpha_n \lambda_n^{k-1} x_n) \\ &= \alpha_1 \lambda_1^k x_1 + \alpha_2 \lambda_2^k x_2 + \ldots + \alpha_n \lambda_n^k x_n \\ &= \lambda_1^k \left(\alpha_1 x_1 + \alpha_2 \left(\tfrac{\lambda_2}{\lambda_1}\right)^k x_2 + \ldots + \alpha_n \left(\tfrac{\lambda_n}{\lambda_1}\right)^k x_n \right). \end{aligned}$$

Let $y \in \mathbb{R}^n$ be an arbitrary vector such that $y^T x_1 \neq 0$. Then

$$\frac{y^T A v^{(k)}}{y^T v^{(k)}} = \frac{y^T v^{(k+1)}}{y^T v^{(k)}} = \frac{\lambda_1^{k+1} \left(\alpha_1 y^T x_1 + \sum_{i=2}^n \alpha_i \left(\frac{\lambda_i}{\lambda_1}\right)^{k+1} y^T x_i \right)}{\lambda_1^k \left(\alpha_1 y^T x_1 + \sum_{i=2}^n \alpha_i \left(\frac{\lambda_i}{\lambda_1}\right)^k y^T x_i \right)} \to \lambda_1 .$$

Given the initial vector $v^{(0)} \in \mathbb{R}^n$, the power method has the following form.

POWER-METHOD$(A, v^{(0)})$

```
1  k ← 0
2  while exit condition = FALSE
3        do k ← k + 1
4           z^(k) ← A v^(k−1)
5           Select vector y such that y^T v^(k−1) ≠ 0
6           γ_k ← y^T z^(k) / y^T v^(k−1)
7           v^(k) ← z^(k) / ‖z^(k)‖_∞
8  return γ_k, v^(k)
```

It is clear that

$$v^{(k)} \to x_1, \quad \gamma_k \to \lambda_1.$$

The convergence $v^{(k)} \to x_1$ here means that $\left(v^{(k)}, x_1\right)_\measuredangle \to 0$, that is the action line of $v^{(k)}$ tends to the action line of x_1. There are various strategies to select y. We can select $y = e_i$, where i is defined by $\left|v_i^{(k)}\right| = \left\|v^{(k)}\right\|_\infty$. If we select $y = v^{(k-1)}$, then $\gamma_k = v^{(k-1)T} A v^{(k-1)} / \left(v^{(k-1)T} v^{(k-1)}\right)$ will be identical with the *Rayleigh quotient* $\mathbf{R}\left(v^{(k-1)}\right)$. This choice gives an approximation of λ_1 that have the minimal residual norm (Example 19.14. shows that this choice is not necessarily the best option).

The speed of convergence depends on the quotient $|\lambda_2/\lambda_1|$. The method is very sensitive to the choice of the initial vector $v^{(0)}$. If $\alpha_1 = 0$, then the process does not converge to the dominant eigenvalue λ_1. For certain matrix classes the power method converges with probability 1 if the initial vector $v^{(0)}$ is randomly chosen. In case of complex eigenvalues or multiple λ_1 we have to use modifications of the algorithm. The speed of convergence can be

accelerated if the method is applied to the shifted matrix $A-\sigma I$, where σ is an appropriately chosen number. The shifted matrix $A-\sigma I$ has the eigenvalues $\lambda_1-\sigma, \lambda_2-\sigma, \ldots, \lambda_n-\sigma$ and the corresponding convergence factor $|\lambda_2-\sigma| / |\lambda_1-\sigma|$. The latter quotient can be made smaller than $|\lambda_2/\lambda_1|$ with the proper selection of σ.

The usual exit condition of the power method is

$$\|E_k\|_2 = \frac{\|r_k\|_2}{\left\|v^{(k)}\right\|_2} = \frac{\left\|Av^{(k)} - \gamma_k v^{(k)}\right\|_2}{\left\|v^{(k)}\right\|_2} \leq \epsilon.$$

If we simultaneously apply the power method to the transposed matrix A^T and $w_k = \left(A^T\right)^k w_0$, then the quantity

$$\nu(\lambda_1) \approx \frac{\left\|w^{(k)}\right\|_2 \left\|v^{(k)}\right\|_2}{\left|w_k^T v_k\right|}$$

gives an estimate for the condition number of λ_1 (see Remark 19.31). In such a case we use the exit condition

$$\nu(\lambda_1)\,\|E_k\|_2 \leq \epsilon.$$

The power method is very useful for large sparse matrices. It is often used to determine the largest and the smallest eigenvalue. We can approximate the smallest eigenvalue as follows. The eigenvalues of A^{-1} are $1/\lambda_1, \ldots, 1/\lambda_n$. The eigenvalue $1/\lambda_n$ will be the eigenvalue with the largest modulus. We can approximate this value by applying the power method to A^{-1}. This requires only a small modification of the algorithm. We replace line 4. with the following:

$$\text{Solve equation } Az^{(k)} = v^{(k-1)} \text{ for } z^{(k)}$$

The modified algorithm is called the ***inverse power method***. It is clear that $\gamma_k \to 1/\lambda_n$ and $v^{(k)} \rightharpoonup x_n$ hold under appropriate conditions. If we use the LU-method to solve $Az^{(k)} = v^{(k-1)}$, we can avoid the inversion of A.

If the inverse power method is applied to the shifted matrix $A-\mu I$, then the eigenvalues of $(A-\mu I)^{-1}$ are $(\lambda_i-\mu)^{-1}$. If μ approaches, say, to λ_t, then $\lambda_i-\mu \to \lambda_i-\lambda_t$. Hence the inequality

$$|\lambda_t-\mu|^{-1} > |\lambda_i-\mu|^{-1} \qquad (i \neq t)$$

holds for the eigenvalues of the shifted matrix. The speed of convergence is determined by the quotient

$$q = |\lambda_t-\mu| / \{\max|\lambda_i-\mu|\}.$$

If μ is close enough to λ_t, then q is very small and the inverse power iteration converges very fast. This property can be exploited in the calculation of approximate eigenvectors if an approximate eigenvalue, say μ, is known. Assuming that $\det(A-\mu I) \neq 0$, we apply the inverse power method to the shifted matrix $A-\mu I$. In spite of the fact that matrix $A-\mu I$ is nearly singular and the linear equation $(A-\mu I)\,z^{(k)} = v^{(k)}$ cannot be solved with high precision, the algorithm gives very often good approximations of the eigenvectors.

Finally we note that in principle the von Mieses method can be modified to determine all eigenvalues and eigenvectors.

Orthogonalization processes
We need the following definition and theorem.

Definition 19.32 *The matrix $Q \in \mathbb{R}^{n\times n}$ is said to be **orthogonal** if $Q^T Q = I$.*

Theorem 19.33 (QR-decomposition). *Every matrix $A \in \mathbb{R}^{n\times m}$ having linearly independent column vectors can be decomposed in the product form $A = QR$, where Q is orthogonal and R is upper triangular matrix.*

We note that the QR-decomposition can be applied for solving linear systems of equations, similarly to the LU-decomposition. If the QR-decomposition of A is known, then the equation $Ax = QRx = b$ can be written in the equivalent form $Rx = Q^T b$. Thus we have to solve only an upper triangular linear system.

There are several methods to determine the QR-decomposition of a matrix. In practice the Givens-, the Householder- and the MGS-methods are used.

The MGS (Modified Gram-Schmidt) method is a stabilized, but algebraically equivalent version of the classical Gram-Schmidt orthogonalization algorithm. The basic problem is the following: We seek for an *orthonormal basis* $\left\{q_j\right\}_{j=1}^{m}$ of the subspace

$$\mathcal{L}\{a_1,\ldots,a_m\} = \left\{\sum_{j=1}^{m} \lambda_j a_j \mid \lambda_j \in \mathbb{R},\ j = 1,\ldots,m\right\},$$

where $a_1,\ldots,a_m \in \mathbb{R}^n$ $(m \le n)$ are linearly independent vectors. That is we determine the linearly independent vectors $q_1,\ldots,q_m$ such that

$$q_i^T q_j = 0\ \ (i \ne j), \|q_i\|_2 = 1\ \ (i = 1,\ldots,m)$$

and

$$\mathcal{L}\{a_1,\ldots,a_m\} = \mathcal{L}\{q_1,\ldots,q_m\}\ .$$

The basic idea of the *classical Gram-Schmidt method* is the following:

Let $r_{11} = \|a_1\|_2$ and $q_1 = a_1/r_{11}$. Assume that vectors $q_1,\ldots,q_{k-1}$ are already computed and orthonormal. Assume that vector $\tilde{q}_k = a_k - \sum_{j=1}^{k-1} r_{jk} q_j$ is such that $\tilde{q}_k \perp q_i$, that is $\tilde{q}_k^T q_i = a_k^T q_i - \sum_{j=1}^{k-1} r_{jk} q_j^T q_i = 0$ holds for $i = 1,\ldots,k-1$. Since $q_1,\ldots,q_{k-1}$ are orthonormal, $q_j^T q_i = 0$ $(i \ne j)$ and $r_{ik} = a_k^T q_i \quad (i = 1,\ldots,k-1)$. After normalization we obtain $q_k = \tilde{q}_k / \|\tilde{q}_k\|_2$.

The algorithm is formalized as follows.

CGS-ORTHOGONALIZATION$(m, a_1,\ldots,a_m)$

```
1  for k ← 1 to m
2      do for i ← 1 to k − 1
3             do r_ik ← a_k^T a_i
4                a_k ← a_k − r_ik a_i
5        r_kk ← ‖a_k‖_2
6        a_k ← a_k / r_kk
7  return a_1, ..., a_m
```

The algorithm overwrites vectors a_i by the orthonormal vectors q_i. The connection with the QR-decomposition follows from the relation $a_k = \sum_{j=1}^{k-1} r_{jk} q_j + r_{kk} q_k$. Since

$$\begin{aligned} a_1 &= q_1 r_{11}, \\ a_2 &= q_1 r_{12} + q_2 r_{22}, \\ a_3 &= q_1 r_{13} + q_2 r_{23} + q_3 r_{33}, \\ &\vdots \\ a_m &= q_1 r_{1m} + q_2 r_{2m} + \ldots + q_m r_{mm}, \end{aligned}$$

we can write that

$$A = [a_1, \ldots, a_m] = \underbrace{[q_1, \ldots, q_m]}_{Q} \underbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & \ldots & r_{1m} \\ 0 & r_{22} & r_{23} & \ldots & r_{2m} \\ 0 & 0 & r_{33} & \ldots & r_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & r_{mm} \end{bmatrix}}_{R} = QR\,.$$

The *numerically stable MGS method* is given in the following form

MGS-ORTHOGONALIZATION$(m, a_1, \ldots, a_m)$

```
1  for k ← 1 to m
2      do r_kk ← ||a_k||_2
3         a_k ← a_k / r_kk
4         for j ← k + 1 to m
5             do r_kj ← a_j^T a_k
6                a_j ← a_j − r_kj a_k
7  return a_1, ..., a_m
```

The algorithm overwrites vectors a_i by the orthonormal vectors q_i. The MGS method is more stable than the CGS algorithm. Björck proved that for $m = n$ the computed matrix $\hat{Q}$ satisfies

$$\hat{Q}^T \hat{Q} = I + E, \qquad \|E\|_2 \cong \text{cond}\,(A)\, u,$$

where u is the unit roundoff.

The QR-method

Today the QR-method is the most important numerical algorithm to compute all eigenvalues of a general matrix. It can be shown that the QR-method is a generalization of the power method. The basic idea of the method is the following: Starting from $A_1 = A$ we compute the sequence $A_{k+1} = Q_k^{-1} A_k Q_k = Q_k^T A_k Q_k$, where Q_k is orthogonal, A_{k+1} is orthogonally similar to A_k (A) and the lower triangular part of A_k tends to a diagonal matrix, whose entries will be the eigenvalues of A. Here Q_k is the orthogonal factor of the QR-decomposition $A_k = Q_k R_k$. Therefore $A_{k+1} = Q_k^T (Q_k R_k) Q_k = R_k Q_k$. The basic algorithm is given in the following form.

QR-METHOD(A)

```
1  k ← 1
2  A_1 ← A
3  while exit condition=FALSE
4        do Compute the QR-decomposition A_k = Q_k R_k
5           A_{k+1} ← R_k Q_k
6           k ← k + 1
7  return A_k
```

The following result holds.

Theorem 19.34 (Parlett). *If the matrix A is diagonalizable, $X^{-1}AX = diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$, the eigenvalues satisfy*

$$|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n| > 0$$

and X has an LU-decomposition, then the lower triangular part of A_k converges to a diagonal matrix whose entries are the eigenvalues of A.

In general, matrices A_k do not necessarily converge to a given matrix. If A has p eigenvalues of the same modulus, the form of matrices A_k converge to the form

$$\begin{bmatrix} \times & & & & & & & \times \\ 0 & \ddots & & & & & & \\ & & \times & & & & & \\ 0 & & 0 & * & \cdots & * & & \\ & & & \vdots & & \vdots & & \\ & & & * & \cdots & * & & \\ 0 & & & & & 0 & \times & \\ & & & & & & & \ddots \\ 0 & & & & & & 0 & \times \end{bmatrix}, \tag{19.40}$$

where the entries of the submatrix denoted by $*$ do not converge. However the eigenvalues of this submatrix will converge. This submatrix can be identified and properly handled. A real matrix may have real and complex eigenvalues. If there is a complex eigenvalues, than there is a corresponding conjugate eigenvalue as well. For pairs of complex conjugated eigenvalues p is at least 2. Hence the sequence A_k will show this phenomenon .

The QR-decomposition is very expensive. Its cost is $\Theta(n^3)$ flops for general $n \times n$ matrices. If A has upper Hessenberg form, the cost of QR-decomposition is $\Theta(n^2)$ flops.

Definition 19.35 *The matrix $A \in \mathbb{R}^{n \times n}$ has **upper Hessenberg form**, , if*

$$A = \begin{bmatrix} a_{11} & & & \ldots & & a_{1n} \\ a_{21} & & & & & \\ 0 & a_{32} & & & & \vdots \\ \vdots & 0 & \ddots & & & \\ & & \ddots & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & & \ldots & 0 & a_{n,n-1} & a_{nn} \end{bmatrix}.$$

The following theorem guarantees that if A has upper Hessenberg form, then every A_k of the QR-method has also upper Hessenberg form.

Theorem 19.36 *If A has upper Hessenberg form and $A = QR$, then RQ has also upper Hessenberg form.*

We can transform a matrix A to a similar matrix of upper Hessenberg form in many ways. One of the cheapest ways, that costs about $5/6n^3$ flops, is based on the Gauss elimination method. Considering the advantages of the upper Hessenberg form the efficient implementation of the QR-method requires first the similarity transformation of A to upper Hessenberg form.

The convergence of the QR-method, similarly to the power method, depends on the quotients $|\lambda_{i+1}/\lambda_i|$. The eigenvalues of the shifted matrix $A-\sigma I$ are $\lambda_1-\sigma, \lambda_2-\sigma, \ldots, \lambda_n-\sigma$. The corresponding eigenvalue ratios are $|(\lambda_{i+1}-\sigma)/(\lambda_i-\sigma)|$. A proper selection of σ can fasten the convergence.

The usual form of the QR-method includes the transformation to upper Hessenberg form and the shifting.

SHIFTEDQR-METHOD(A)

1 $H_1 \leftarrow U^{-1}AU$ (H_1 is of upper Hessenberg form)
2 $k \leftarrow 1$
3 **while** *exit condition*=FALSE
4 **do** Compute the QR-decomposition $H_k - \sigma_k I = Q_k R_k$
5 $H_{k+1} \leftarrow R_k Q_k + \sigma_k I$
6 $k \leftarrow k+1$
7 **return** H_k

In practice the QR-method is used in shifted form. There are various strategies to select σ_i. The most often used selection is given by $\sigma_k = h_{nn}^{(k)}$ $\left(H_k = \left[h_{ij}^{(k)}\right]_{i,j=1}^{n}\right)$.

The eigenvectors of A can also be determined by the QR-method. For this we refer to the literature.

Exercises

19.3-1 Apply the *power method* to the matrix $A = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$ with the initial vector $v^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. What is the result of the 20th step?

19.3-2 Apply the *power method*, the *inverse power method* and the QR-method to the matrix

$$\begin{bmatrix} -4 & -3 & -7 \\ 2 & 3 & 2 \\ 4 & 2 & 7 \end{bmatrix}.$$

19.3-3 Apply the shifted QR-method to the matrix of the previous exercise with the choice $\sigma_i = \sigma$ (σ is fixed).

19.4. Numerical program libraries and software tools

We have plenty of devices and tools that support efficient coding and implementation of numerical algorithms. One aim of such developments is to free the programmers from writing the programs of frequently occurring problems. This is usually done by writing safe, reliable and standardized routines that can be downloaded from (public) program libraries. We just mention the LINPACK, EISPACK, LAPACK, VISUAL NUMERICS (former IMSL) and NAG libraries. Another way of developments is to produce software that work as a programming language and makes the programming very easy. Such software systems are the MATLAB and the SciLab.

19.4.1. Standard linear algebra subroutines

The main purpose of the BLAS (Basic Linear Algebra Subprograms) programs is the standardization and efficient implementation the most frequent matrix-vector operations. Although the BLAS routines were published in FORTRAN they can be accessed in optimized machine code form as well. The BLAS routines have three levels:

- BLAS 1 (1979),
- BLAS 2 (1988),
- BLAS 3 (1989).

These levels corresponds to the computation cost of the implemented matrix operations. The BLAS routines are considered as the best implementations of the given matrix operations. The selection of the levels and individual BLAS routines strongly influence the efficiency of the program. A sparse version of BLAS also exists.

We note that the BLAS 3 routines were developed mainly for block parallel algorithms. The standard linear algebra packages LINPACK, EISPACK and LAPACK are built from BLAS routines. The parallel versions can be found in the SCALAPACK package. These programs can be found in the public NETLIB library:

`http:/www.netlib.org/index.html`

BLAS 1 routines

Let $\alpha \in \mathbb{R}$, $x, y \in \mathbb{R}^n$. The BLAS 1 routines are the programs of the most important vector operations ($z = \alpha x$, $z = x+y$, $dot = x^T y$), the computation of $\|x\|_2$, the swapping of variables, rotations and the *saxpy* operation which is defined by

$$z = \alpha x + y.$$

The word *saxpy* means that „scalar alpha x plus y". The *saxpy* operation is implemented in the following way.

SAXPY(α, x, y)

```
1  n ← elements[x]
2  for i ← 1 to n
3      do z[i] = αx[i] + y[i]
4  return z
```

The *saxpy* is a software driven operation. The cost of BLAS 1 routines is $\Theta(n)$ flops.

BLAS 2 routines
The matrix-vector operations of BLAS 2 requires $\Theta\left(n^2\right)$ flops. These operations are $y = \alpha Ax + \beta y$, $y = Ax$, $y = A^{-1}x$, $y = A^T x$, $A \leftarrow A + xy^T$ and their variants. Certain operations work only with triangular matrices. We analyze two operations in detail. The „outer or dyadic product" update

$$A \leftarrow A + xy^T \quad (A \in \mathbb{R}^{m \times n},\ x \in \mathbb{R}^m,\ y \in \mathbb{R}^n)$$

can be implemented in two ways.

The rowwise or „*ij*" variant:

OUTER-PRODUCT-UPDATE-VERSION „IJ" (A, x, y)

```
1  m ← rows[A]
2  for i ← 1 to m
3       do A[i, :] ← A[i, :] + x[i] y^T
4  return A
```

The notation „:" denotes all allowed indices. In our case this means the indices $1 \leq j \leq n$. Thus $A[i, :]$ denotes the i^{th} row of matrix A.

The columnwise or „*ji*" variant:

OUTER-PRODUCT-UPDATE-VERSION „JI"(A, x, y)

```
1  n ← columns[A]
2  for j ← 1 to n
3       do A[:, j] ← A[:, j] + y[j] x
4  return A
```

Here $A[:, j]$ denotes the j^{th} column of matrix A. Observe that both variants are based on the saxpy operation.

The *gaxpy* operation is defined by

$$z = y + Ax \quad (x \in \mathbb{R}^n,\ y \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}).$$

The word *gaxpy* means that „general A x plus y". The *gaxpy* operation is also software driven and implemented in the following way:

GAXPY(A, x, y)

```
1  n ← columns[A]
2  z ← y
3  for j ← 1 to n
4       do z ← z + x[j] A[:, j]
5  return z
```

Observe that the computation is done columnwise and the *gaxpy* operation is essentially a generalized *saxpy*.

BLAS 3 routines
These routines are the implementations of $\Theta\left(n^3\right)$ matrix-matrix and matrix-vector operati-

ons such as the operations $C \leftarrow \alpha AB + \beta C$, $C \leftarrow \alpha AB^T + \beta C$, $B \leftarrow \alpha T^{-1}B$ (T is upper triangular) and their variants. BLAS 3 operations can be implemented in several forms. For example, the matrix product $C = AB$ can be implemented at least in three ways. Let $A \in \mathbb{R}^{m\times r}$, $B \in \mathbb{R}^{r\times n}$.

MATRIX-PRODUCT-DOT-VERSION(A, B)

```
1  m ← rows[A]
2  r ← columns[A]
3  n ← columns[B]
4  C[1 : m, 1 : n] ← 0
5  for i ← 1 to m
6      do for j ← 1 to n
7              do for k ← 1 to r
8                      do C[i, j] ← C[i, j] + A[i, k] B[k, j]
9  return C
```

This algorithm computes c_{ij} as the dot (inner) product of the i^{th} row of A and the j^{th} column of B. This corresponds to the original definition of matrix products.

Now let A, B and C be partitioned columnwise as follows

$$\begin{aligned} A &= [a_1, \ldots, a_r] \quad (a_i \in \mathbb{R}^m), \\ B &= [b_1, \ldots, b_n] \quad (b_i \in \mathbb{R}^r), \\ C &= [c_1, \ldots, c_n] \quad (c_i \in \mathbb{R}^m). \end{aligned}$$

Then we can write c_j as the linear combination of the columns of A, that is

$$c_j = \sum_{k=1}^{r} b_{kj} a_k \quad (j = 1, \ldots, n).$$

Hence the product can be implemented with *saxpy* operations.

MATRIX-PRODUCT-GAXPY-VARIANT(A, B)

```
1  m ← rows[A]
2  r ← columns[A]
3  n ← columns[B]
4  C[1 : m, 1 : n] ← 0
5  for j ← 1 to n
6      do for k ← 1 to r
7              do for i ← 1 to m
8                      do C[i, j] ← C[i, j] + A[i, k] B[k, j]
9  return C
```

The following equivalent form of the „*jki*"-algorithm shows that it is indeed a *gaxpy* based process.

MATRIX-PRODUCT-WITH-GAXPY-CALL(A, B)

```
1  m ← rows[A]
2  n ← columns[B]
3  C[1 : m, 1 : n] ← 0
4  for j ← 1 to n
5      do C[:, j] = gaxpy(A, B[:, j], C[:, j])
6  return C
```

Consider now the partitions $A = [a_1, \ldots, a_r]$ $(a_i \in R^m)$ and

$$B = \begin{bmatrix} b_1^T \\ \vdots \\ b_r^T \end{bmatrix} \quad (b_i \in R^n).$$

Then $C = AB = \sum_{k=1}^{r} a_k b_k^T$.

MATRIX-PRODUCT-OUTER-PRODUCT-VARIANT(A, B)

```
1  m ← rows[A]
2  r ← columns[A]
3  n ← columns[B]
4  C[1 : m, 1 : n] = 0
5  for k ← 1 to r
6      do for j ← 1 to n
7             do for i ← 1 to m
8                    C[i, j] ← C[i, j] + A[i, k] B[k, j]
9  return C
```

The inner loop realizes a *saxpy* operation: it gives the multiple of a_k to the j^{th} column of matrix C.

19.4.2. Mathematical software

These are those programming tools that help easy programming in concise (possibly mathematical) form within an integrated program development system. Such systems were developed primarily for solving mathematical problems. By now they have been extended so that they can be applied in many other fields. For example, Nokia uses MATLAB in the testing and quality control of mobile phones. We give a short review on MATLAB in the next section. We also mention the widely used MAPLE, DERIVE and MATEMATICA systems.

The MATLAB system

The MATLAB software was named after the expression MATrix LABoratory. The name indicates that the matrix operations are very easy to make. The initial versions of MATLAB had only one data type: the complex matrix. In the later versions high dimension arrays, cells, records and objects also appeared. The MATLAB can be learned quite easily and even a beginner can write programs for relatively complicated problems.

The coding of matrix operations is similar to their standard mathematical form. For

example if A and B are two matrices of the same size, then their sum is given by the command $C = A + B$. As a programming language the MATLAB contains only four control structures known from other programing languages:

– the simple statement Z =*expression*,

– the if statement of the form

if *expression, commands* {**else**/**elseif** *commands*} **end**,

– the for loop of the form

for *the values of the loop variable, commands* **end**

– the while loop of the form

while *expression, commands* **end**.

The MATLAB has an extremely large number of built in functions that help efficient programming. We mention the following ones as a sample.

– max(A) selects the maximum element in every column of A,

– [v, s] =eig(A) returns the approximate eigenvalues and eigenvectors of A,

– The command $A\backslash b$ returns the numerical solution of the linear system $Ax = b$.

The entrywise operations and partitioning of matrices can be done very efficiently in MATLAB. For example, the statement

$$A([2,3],:) = 1./A([3,2],:)$$

exchange the second and third rows of A while it takes the reciprocal of each element.

The above examples only illustrate the possibilities and easy programming of MATLAB. These examples require much more programming effort in other languages, say e.g. in PASCAL. The built in functions of MATLAB can be easily supplemented by other programs.

The higher number versions of MATLAB include more and more functions and special libraries (tool boxes) to solve special problems such as optimization, statistics and so on.

There is a built in automatic technique to store and handle sparse matrices that makes the MATLAB competitive in solving large computational problems. The recent versions of MATLAB offer very rich graphic capabilities as well. There is an extra interval arithmetic package that can be downloaded from the WEB site

`http:/www.ti3.tu-harburg.de\%7Erump\intlab`

There is a possibility to build certain C and FORTRAN programs into MATLAB. Finally we mention that the system has an extremely well written help system.

Problems

19-1. Without overflow

Write a MATLAB program that computes the norm $\|x\|_2 = \left(\sum_{i=1}^{n} x_i^2\right)^{1/2}$ without *overflow* in all cases when the result does not make overflow. It is also required that the error of the final result can not be greater than that of the original formula.

19-2. Estimate

Equation $x^3 - 3.330000x^2 + 3.686300x - 1.356531 = 0$ has the solution $x_1 = 1.01$. The

perturbed equation $x^3 - 3.3300x^2 + 3.6863x - 1.3565 = 0$ has the solutions $y_1,\ y_2,\ y_3$. Give an estimate for the perturbation $\min_i |x_1 - y_i|$.

19-3. Double word length

Consider an arithmetic system that has double word length such that every number represented with $2t$ digits are stored in two t digit word. Assume that the computer can only add numbers with t digits. Furthermore assume that the machine can recognize overflow.
(i) Find an algorithm that add two positive numbers of $2t$ digit length.
(ii) If the representation of numbers requires the sign digit for all numbers, then modify algorithm (i) so that it can add negative and positive numbers both of the same sign. We can assume that the sum does not overflow.

19-4. Auchmuty theorem

Write a MATLAB program for the Auchmuty error estimate (see Theorem 19.22) and perform the following numerical testing.
(i) Solve the linear systems $Ax = b_i$, where $A \in \mathbb{R}^{n\times n}$ is a given matrix, $b_i = Ay_i$, $y_i \in \mathbb{R}^n$ $(i = 1, \ldots, N)$ are random vectors such that $\|y_i\|_\infty \le \beta$. Compare the true errors $\|\widetilde{x_i} - y_i\|$, $(i = 1, \ldots, N)$ and the estimated errors $EST_i = \|r(\widetilde{x_i})\|_2^2 / \left\|A^T r(\widetilde{x_i})\right\|_2$, where $\widetilde{x_i}$ is the approximate solution of $Ax = b_i$. What is the minimum, maximum and average of numbers c_i? Use graphic for the presentation of the results. Suggested values are $n \le 200, \beta = 200$ and $N = 40$.
(ii) Analyze the effect of condition number and size.
(iii) Repeat problems (i) and (ii) using LINPACK and BLAS.

19-5. Hilbert matrix

Consider the linear system $Ax = b$, where $b = [1, 1, 1, 1]^T$ and A is the fourth order Hilbert matrix, that is $a_{i,j} = 1/(i + j)$. A is ill-conditioned. The inverse of A is approximated by

$$B = \begin{bmatrix} 202 & -1212 & 2121 & -1131 \\ -1212 & 8181 & -15271 & 8484 \\ 2121 & -15271 & 29694 & -16968 \\ -1131 & 8484 & -16968 & 9898 \end{bmatrix}.$$

Thus an x_0 approximation of the true solution x is given by $x_0 = Bb$. Although the true solution is also integer x_0 is not an acceptable approximation. Apply the *iterative refinement* with B instead of A^{-1} to find an acceptable integer solution.

19-6. Consistent norm

Let $\|A\|$ be a *consistent norm* and consider the linear system $Ax = b$
(i) Prove that if $A + \Delta A$ is singular, then $\text{cond}(A) \ge \|A\| / \|\Delta A\|$.
(ii) Show that for the 2-norm equality holds in (i), if $\Delta A = -bx^T/(b^t x)$ and $\left\|A^{-1}\right\|_2 \|b\|_2 = \left\|A^{-1}b\right\|_2$.
(iii) Using the result of (i) give a lower bound to $\text{cond}_\infty(A)$, if

$$A = \begin{bmatrix} 1 & -1 & 1 \\ -1 & \varepsilon & \varepsilon \\ 1 & \varepsilon & \varepsilon \end{bmatrix}.$$

19-7. Cholesky-method

Use the Cholesky-method to solve the linear system $Ax = b$, where

$$A = \begin{bmatrix} 5.5 & 0 & 0 & 0 & 0 & 3.5 \\ 0 & 5.5 & 0 & 0 & 0 & 1.5 \\ 0 & 0 & 6.25 & 0 & 3.75 & 0 \\ 0 & 0 & 0 & 5.5 & 0 & 0.5 \\ 0 & 0 & 3.75 & 0 & 6.25 & 0 \\ 3.5 & 1.5 & 0 & 0.5 & 0 & 5.5 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Also give the exact *Cholesky-decomposition* $A = LL^T$ and the true solution of $Ax = b$. The approximate Cholesky-factor $\widetilde{L}$ satisfies the relation $\tilde{L}\tilde{L}^T = A + F$. It can proved that in a *floating point arithmetic* with t-digit mantissa and base β the entries of F satisfy the inequality $|f_{i,j}| \leq e_{i,j}$, where

$$E = \beta^{1-t} \begin{bmatrix} 11 & 0 & 0 & 0 & 0 & 3.5 \\ 0 & 11 & 0 & 0 & 0 & 1.5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 11 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 3.5 & 1.5 & 0 & 0.5 & 0 & 11 \end{bmatrix}.$$

Give a bound for the relative error of the approximate solution $\tilde{x}$, if $\beta = 16$ and $t = 14$ (IBM3033).

19-8. Bauer-Fike theorem

Let

$$A = \begin{bmatrix} 10 & 10 & & & & & \\ & 9 & 10 & & & & \\ & & 8 & 10 & & & \\ & & & \ddots & \ddots & & \\ & & & & 2 & 10 & \\ \varepsilon & & & & & 1 & \end{bmatrix}$$

(i) Analyze the perturbation of the eigenvalues for $\varepsilon = 10^{-5}, 10^{-6}, 10^{-7}, 0$.
(ii) Compare the estimate of Bauer-Fike theorem to the matrix $A = A(0)$.

19-9. Eigenvalues

Using the MATLAB eig routine compute the eigenvalues of $B = AA^T$ for various (random) matrices $A \in \mathbb{R}^{n\times n}$ and order n. Also compute the eigenvalues of the perturbed matrices $B + R_i$, where R_i are random matrices with entries from the interval $\left[-10^{-5}, 10^{-5}\right]$ ($i = 1, \ldots, N$). What is the maximum perturbation of the eigenvalues? How precise is the Bauer-Fike estimate? Suggested values are $N = 10$ and $5 \leq n \leq 200$. How do the results depend on the condition number and the order n? Display the maximum perturbations and the Bauer-Fike estimates graphically.

Chapter notes

The a posteriori error estimates of linear algebraic systems are not completely reliable. Demmel, Diament és Malajovich [43] showed that for the $\Theta\left(n^2\right)$ number estimators there

are always cases when the estimate is unreliable (the error of the estimate exceeds a given order). The first appearance of the iterative improvement is due to Fox, Goodwin, Turing and Wilkinson (1946). The experiences show that the decrease of the residual error is not monotone.

Young [127], Hageman and Young [**?**] give an excellent survey of the theory and application of iterative methods. Barett, Berry et al. [7] give a software oriented survey of the subject. Frommer [52] concentrates on the parallel computations.

The convergence of the *QR*-method is a delicate matter. It is analyzed in great depth and much better results than Theorem 19.34 exist in the literature. There are *QR*-like methods that involve double shifting. Batterson [9] showed that there exists a 3×3 Hessenberg matrix with complex eigenvalues such that convergence cannot be achieved even with multiple shifting.

Several other methods are known for solving the eigenvalue problems (see, e.g. [121], [119]). The *LR*-method is one of the best known ones. It is very effective on positive definite Hermitian matrices. The *LR*-method computes the Cholesky-decomposition $A_k = LL^*$ and sets $A_{k+1} = L^*L$.

20. Semi-structured databases

The use of the internet and the development of the theory of databases mutually affect each other. The contents of web sites are usually stored by databases, while the web sites and the references between them can also be considered a database which has no fixed schema in the usual sense. The contents of the sites and the references between sites are described by the sites themselves, therefore we can only speak of semi-structured data, which can be best characterized by directed labeled graphs. In case of semi-structured data, recursive methods are used more often for giving data structures and queries than in case of classical relational databases. Different problems of databases, e.g. restrictions, dependencies, queries, distributed storage, authorities, uncertainty handling, must all be generalized according to this. Semi-structuredness also raises new questions. Since queries not always form a closed system like they do in case of classical databases, that is, the applicability of queries one after another depends on the type of the result obtained, therefore the problem of checking types becomes more emphasized.

The theoretical establishment of relational databases is closely related to finite modelling theory, while in case of semi-structured databases, automata, especially tree automata are most important.

20.1. Semi-structured data and XML

By semi-structured data we mean a directed rooted labeled graph. The root is a special node of the graph with no entering edges. The nodes of the graph are objects distinguished from each other using labels. The objects are either atomic or complex. Complex objects are connected to one or more objects by directed edges. Values are assigned to atomic objects. Two different models are used: either the vertices or the edges are labeled. The latter one is more general, since an edge-labeled graph can be assigned to all vertex-labeled graphs in such a way that the label assigned to the edge is the label assigned to its endpoint. This way we obtain a directed labeled graph for which all inward directed edges from a vertex have the same label. Using this transformation, all concepts, definitions and statements concerning edge-labeled graphs can be rewritten for vertex-labeled graphs.

The following method is used to gain a vertex-labeled graph from an edge-labeled graph. If edge (u, v) has label c, then remove this edge, and introduce a new vertex w with

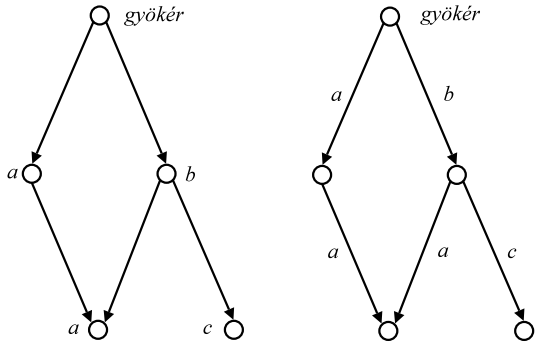


Figure 20.1. Edge-labeled graph assigned to a vertex-labeled graph.

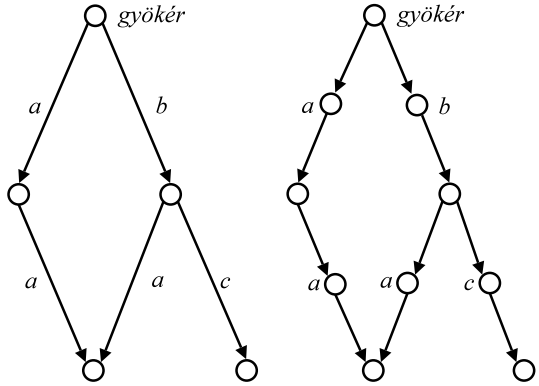


Figure 20.2. An edge-labeled graph and the corresponding vertex-labeled graph.

label c, then add edges (u, w) and (w, v). This way we can obtain a vertex-labeled graph of $m + n$ nodes and $2m$ edges from an edge-labeled graph of n vertices and m edges. Therefore all algorithms and cost bounds concerning vertex-labeled graphs can be rewritten for edge-labeled graphs.

Since most books used in practice use vertex-labeled graphs, we will also use vertex-labeled graphs in this chapter.

The ***XML*** (e**X**tensible **M**arkup **L**anguage) language was originally designed to describe embedded ordered labeled elements, therefore it can be used to represent trees of semi-structured data. In a wider sense of the XML language, references between the elements can also be given, thus arbitrary semi-structured data can be described using the XML language.

The medusa.inf.elte.hu/forbidden site written in XML language is as follows. We can obtain the vertex-labeled graph of Figure 20.3 naturally from the structural characteristics of the code.

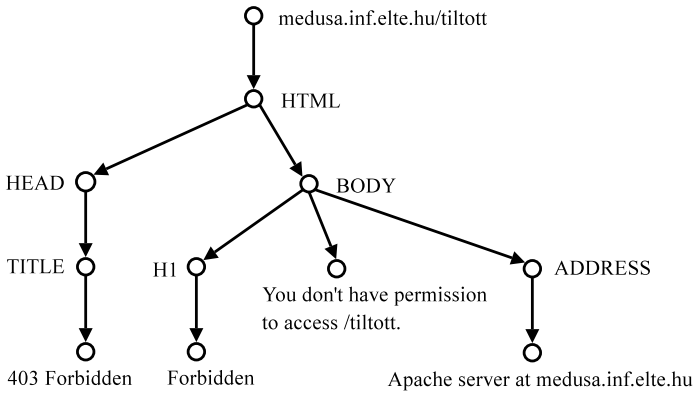


Figure 20.3. The graph corresponding to the XML file „forbidden".

```
<HTML>
        <HEAD>
                <TITLE>403 Forbidden</TITLE>
        </HEAD>
        <BODY>
                <H1>Forbidden</H1>
                You don't have permission to access /forbidden.
                <ADDRESS>Apache Server at medusa.inf.elte.hu </ADDRESS>
        </BODY>
</HTML>
```

Exercises

20.1-1 Give a vertex-labeled graph that represents the structure and formatting of this chapter.

20.1-2 How many different directed vertex-labeled graphs exist with n vertices, m edges and k possible labels? How many of these graphs are not isomorphic? What values can be obtained for $n = 5$, $m = 7$ and $k = 2$?

20.1-3 Consider a tree in which all children of a given node are labeled with different numbers. Prove that the nodes can be labeled with pairs (a_v, b_v), where a_v and b_v are natural numbers, in such a way that

a. $a_v < b_v$ for every node v.

b. If u is a descendant of v, then $a_v < a_u < b_u < b_v$.

c. If u and v are siblings and $number(u) < number(v)$, then $b_u < a_v$.

20.2. Schemas and simulations

In case of relational databases, schemas play an important role in coding and querying data, query optimization and storing methods that increase efficiency. When working with semi-

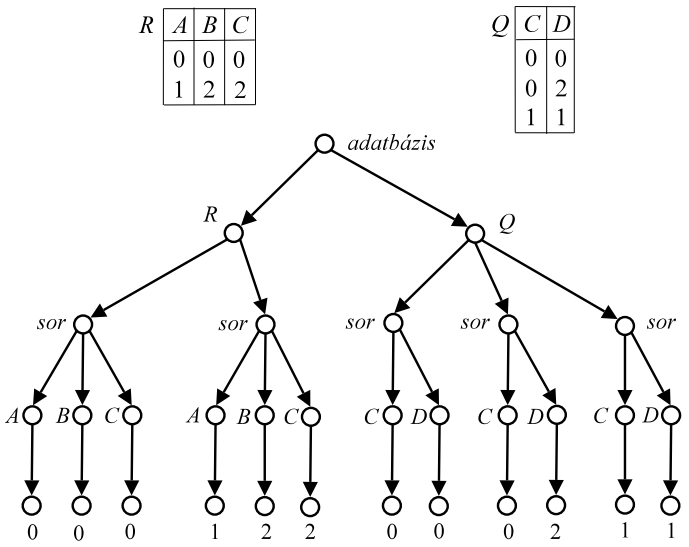


Figure 20.4. A relational database in the semi-structured model.

structured databases, the schema must be obtained from the graph. The schema restricts the possible label strings belonging to the paths of the graph.

Figure 20.4 shows the relational schemas with relations $R(A, B, C)$ and $Q(C, D)$, respectively, and the corresponding semi-structured description. The labels of the leaves of the tree are the components of the tuples. The directed paths leading from the root to the values contain the label strings *database.R.tuple.A, database.R.tuple.B, database.R.tuple.C, database.Q.tuple.C, database.Q.tuple.D*. This can be considered the schema of the semi-structured database. Note that the schema is also a graph, as it can be seen on Figure 20.5. The disjoint union of the two graphs is also a graph, on which a simulation mapping can be defined as follows. This way we create a connection between the original graph and the graph corresponding to the schema.

Definition 20.1 *Let $G = (V, E, A, label())$ be a vertex-labeled directed graph, where V denotes the set of nodes, E the set of edges, A the set of labels, and $label(v)$ is the label belonging to node v. Denote by $E^{-1}(v) = \{u \mid (u, v) \in E\}$ the set of the start nodes of the edges leading to node v. A binary relation s ($s \subseteq V \times V$) is a* ***simulation****, if, for $s(u, v)$,*
i) $label(u) = label(v)$ and
ii) for all $u' \in E^{-1}(u)$ there exists a $v' \in E^{-1}(v)$ such that $s(u', v')$
Node v simulates node u*, if there exists a simulation s such that $s(u, v)$.* ***Node u and node v are similar****, $u \approx v$, if u simulates v and v simulates u.*

It is easy to see that the empty relation is a simulation, that the union of simulations is a simulation, that there always exists a maximal simulation and that similarity is an equivalence relation. We can write E instead of E^{-1} in the above definition, since that only means that the direction of the edges of the graph is reversed.

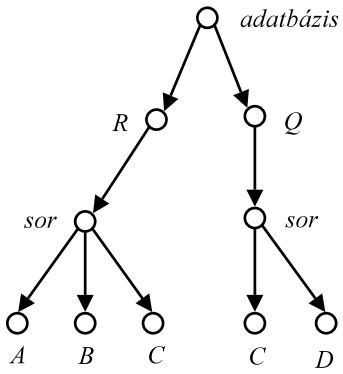


Figure 20.5. The schema of the semi-structured database given in Figure 20.4.

We say that graph D simulates graph S if there exists a mapping $f : V_S \mapsto V_D$ such that the relation $(v, f(v))$ is a simulation on the set $V_S \times V_D$.

Two different schemas are used, a lower bound and an upper bound. If the data graph D simulates the schema graph S, then ***S is a lower bound of D***. Note that this means that all label strings belonging to the directed paths in S appear in D at some directed path. If S simulates D, then ***S is an upper bound of D***. In this case, the label strings of D also appear in S.

In case of semi-structured databases, the schemas which are greatest lower bounds or lowest upper bounds play an important role.

A map between graphs S and D that preserves edges is called a morphism. Note that f is a morphism if and only if D simulates S. To determine whether a morphism from D to S exists is an NP-complete problem. We will see below, however, that the calculation of a maximal simulation is a PTIME problem.

Denote by $sim(v)$ the nodes that simulate v. The calculation of the maximal simulation is equivalent to the determination of all sets $sim(v)$ for $v \in V$. First, our naive calculation will be based on the definition.

NAIVE-MAXIMAL-SIMULATION(G)

1 **for** all $v \in V$
2 **do** $sim(v) \leftarrow \{u \in V \mid label(u) = label(v)\}$
3 **while** $\exists\, u, v, w \in V :\ v \in E^{-1}(u)\ \wedge\ w \in sim(u)\ \wedge\ E^{-1}(w) \cap sim(v) = \emptyset$
4 **do** $sim(u) \leftarrow sim(u) \setminus \{w\}$
5 **return** $\{sim(u) \mid u \in V\}$

Proposition 20.2 *The algorithm* NAIVE-MAXIMAL-SIMULATION *computes the maximal simulation in* $O(m^2n^3)$ *time if* $m \geq n$.

Proof. Let us start with the elements of $sim(u)$. If an element w of $sim(u)$ does not simulate u by definition according to edge (v, u), then we remove w from set $sim(u)$. In this case, we say that we improved set $sim(u)$ according to edge (v, u). If set $sim(u)$ cannot be improved

according to any of the edges, then all elements of $sim(u)$ simulate u. To complete the proof, notice that the **while** cycle consists of at most n^2 iterations. ■

The efficiency of the algorithm can be improved using special data structures. First, introduce a set *sim-candidate(u)*, which contains $sim(u)$, and of the elements of whom we want to find out whether they simulate u.

IMPROVED-MAXIMAL-SIMULATION(G)

```
1  for all v ∈ V
2      do sim-candidate(u) ← V
3         if E^{-1}(v) = ∅
4            then sim(v) ← {u ∈ V | label(u) = label(v)}
5            else sim(v) ← {u ∈ V | label(u) = label(v) ∧ E^{-1}(u) ≠ ∅}
6  while ∃ v ∈ V : sim(v) ≠ sim-candidate(v)
7        do removal-candidate ← E(sim-candidate(v)) \ E(sim(v))
8           for all u ∈ E(v)
9               do sim(u) ← sim(u) \ removal-candidate
10          sim-candidate(v) ← sim(v)
11 return {sim(u) | u ∈ V}
```

The **while** cycle of the improved algorithm possesses the following invariant characteristics.

I_1: $\forall\, v \in V :\ sim(v) \subseteq sim\text{-}candidate(v)$.
I_2: $\forall\, u, v, w \in V : (v \in E^{-1}(u) \ \wedge\ w \in sim(u)) \ \Rightarrow\ (E^{-1}(w) \cap sim\text{-}candidate(v) \neq \emptyset)$.

When improving the set $sim(u)$ according to edge (v, u), we check whether an element $w \in sim(u)$ has parents in $sim(v)$. It is sufficient to check that for the elements of $sim\text{-}candidate(v)$ instead of $sim(v)$ because of I_2. Once an element $w' \in sim\text{-}candidate(v) \setminus sim(v)$ was chosen, it is removed from set $sim\text{-}candidate(v)$.

We can further improve the algorithm if we do not compute the set *removal-candidate* in the iterations of the **while** cycle but refresh the set dynamically.

EFFICIENT-MAXIMAL-SIMULATION(G)

```
 1 for all v ∈ V
 2     do sim-candidate(v) ← V
 3        if E^{-1}(v) = ∅
 4           then sim(v) ← {u ∈ V | label(u) = label(v)}
 5           else sim(v) ← {u ∈ V | label(u) = label(v) ∧ E^{-1}(u) ≠ ∅}
 6     removal-candidate(v) ← E(V) \ E(sim(v))
 7 while ∃ v ∈ V : removal-candidate(v) ≠ ∅
 8        do for all u ∈ E(v)
 9           do for all w ∈ removal-candidate(v)
10              do if w ∈ sim(u)
11                    then sim(u) ← sim(u) \ {w}
12                         for all w'' ∈ E(w)
13                            do if E^{-1}(w'') ∩ sim(u) = ∅
14                                  then removal-candidate(u) ← removal-candidate(u) ∪ {w''}
15           sim-candidate(v) ← sim(v)
16           removal-candidate(v) ← ∅
17 return {sim(u) | u ∈ V}
```

The above algorithm possesses the following invariant characteristic with respect to the **while** cycle.

I_3: $\forall\, v \in V$: *removal-candidate*$(v) = E($*sim-candidate*$(v)) \setminus E($*sim*$(v))$.

Use an $n \times n$ array as a *counter* for the realization of the algorithm. Let the value *counter*$[w'', u]$ be the nonnegative integer $|E^{-1}(w'') \cap$ *sim*$(u)|$. The initial values of the counter are set in $O(mn)$ time. When element w is removed from set *sim*(u), the values *counter*$[w'', u]$ must be decreased for all children w'' of w. By this we ensure that the innermost **if** condition can be checked in constant time. At the beginning of the algorithm, the initial values of the sets *sim*(v) are set in $O(n^2)$ time if $m \geq n$. The setting of sets *removal-candidate*(v) takes altogether $O(mn)$ time. For arbitrary nodes v and w, if $w \in$ *removal-candidate*(v) is true in the i-th iteration of the **while** cycle, then it will be false in the j-th iteration for $j > i$. Since $w \in$ *removal-candidate*(v) implies $w \notin E($*sim*$(v))$, the value of *sim-candidate*(v) in the j-th iteration is a subset of the value of *sim*(v) in the i-th iteration, and we know that invariant I_3 holds. Therefore $w \in$ *sim*(u) can be checked in $\sum_v \sum_w |E(v)| = O(mn)$ time. $w \in$ *sim*(u) is true at most once for all nodes w and u, since once the condition holds, we remove w from set *sim*(u). This implies that the computation of the outer **if** condition of the **while** cycle takes $\sum_v \sum_w (1 + |E(v)|) = O(mn)$ time.

Thus we have proved the following proposition.

Proposition 20.3 *The algorithm* EFFECTIVE-MAXIMAL-SIMULATION *computes the maximal simulation in $O(mn)$ time if $m \geq n$.*

If the inverse of a simulation is also a simulation, then it is called a bisimulation. The empty relation is a bisimulation, and there always exist a maximal bisimulation. The maximal bisimulation can be computed more efficiently than the simulation. The maximal bisimulation can be computed in $O(m \lg n)$ time using the PT algorithm. In case of edge-labeled

graphs, the cost is $O(m \lg(m+n))$.

We will see that bisimulations play an important role in indexing semi-structured databases, since the quotient graph of a graph with respect to a bisimulation contains the same label strings as the original graph. Note that in practice, instead of simulations, the so-called ***DTD*** descriptions are also used as schemas. DTD consists of data type definitions formulated in regular language.

Exercises

20.2-1 Show that simulation does not imply bisimulation.

20.2-2 Define the operation *turn-tree* for a directed, not necessarily acyclic, vertex-labeled graph G the following way. The result of the operation is a not necessarily finite graph G', the vertices of which are the directed paths of G starting from the root, and the labels of the paths are the corresponding label strings. Connect node p_1 with node p_2 by an edge if p_1 can be obtained from p_2 by deletion of its endpoint. Prove that G and *turn-tree*(G) are similar with respect to the bisimulation.

20.3. Queries and indexes

The information stored in semi-structured databases can be retrieved using queries. For this, we have to fix the form of the questions, so we give a ***query language***, and then define the meaning of questions, that is, the ***query evaluation*** with respect to a semi-structured database. For efficient evaluation we usually use indexes. The main idea of ***indexing*** is that we reduce the data stored in the database according to some similarity principle, that is, we create an index that reflects the structure of the original data. The original query is executed in the index, then using the result we find the data corresponding to the index values in the original database. The size of the index is usually much smaller than that of the original database, therefore queries can be executed faster. Note that the inverted list type index used in case of classical databases can be integrated with the schema type indexes introduced below. This is especially advantageous when searching XML documents using keywords.

First we will get acquainted with the query language consisting of regular expressions and the index types used with it.

Definition 20.4 *Given a directed vertex-labeled graph $G = (V, E, root, \Sigma, label, id, value)$, where V denotes the set of vertices, $E \subseteq V \times V$ the set of edges and Σ the set of labels. Σ contains two special labels, ROOT and VALUE. The label of vertex v is $label(v)$, and the identifier of vertex v is $id(v)$. The root is a node with label ROOT, and from which all nodes can be reached via directed paths. If v is a leaf, that is, if it has no outgoing edges, then its label is VALUE, and $value(v)$ is the value corresponding to leaf v. Under the term path we always mean a directed path, that is, a sequence of nodes $n_0, \ldots, n_p$ such that there is an edge from n_i to n_{i+1} if $0 \le i \le p-1$. A sequence of labels $l_0, \ldots, l_p$ is called a* ***label sequence*** *or* ***simple expression***. ***Path $n_0, \ldots, n_p$ fits to the label sequence $l_0, \ldots, l_p$*** *if $label(n_i) = l_i$ for all $0 \le i \le p$.*

We define regular expressions recursively.

Definition 20.5 *Let $R ::= \varepsilon \mid \Sigma \mid \_ \mid R.R \mid R|R \mid (R) \mid R? \mid R^*$, where* ***R*** *is a* ***regular expression***,

*and ε is the empty expression, _ denotes an arbitrary label, . denotes succession, | is the logical OR operation, ? is the optional choice, and * means finite repetition. Denote by* ***L(R)*** *the regular language consisting of the label sequences determined by R.* ***Node n fits to a label sequence*** *if there exists a path from the root to node n such that fits to the label sequence.* ***Node n fits to the regular expression*** *R if there exists a label sequence in the language L(R), to which node n fits. The* ***result of the query on graph G*** *determined by the regular expression R is the set R(G) of nodes that fit to expression R.*

Since we are always looking for paths starting from the root when evaluating regular expressions, the first element of the label sequence is always ROOT, which can therefore be omitted.

Note that the set of languages $L(R)$ corresponding to regular expressions is closed under intersection, and the problem whether $L(R) = \emptyset$ is decidable.

The result of the queries can be computed using the nondeterministic automaton A_R corresponding to the regular expression R. The algorithm given recursively is as follows.

NAIVE-EVALUATION(G, A_R)

1 *Visited* $\leftarrow \emptyset$ ▷ If we were in node u in state s, then (u, s) was put in set *Visited*.
2 TRAVERSE (*root*(G), *starting-state*(A_R))

TRAVERSE(u, s)

1 **if** $(u, s) \in$ *Visited*
2 **then return** *result*$[u, s]$
3 *Visited* $\leftarrow$ *Visited* $\cup \{(u, s)\}$
4 *result*$[u, s] \leftarrow \emptyset$
5 **for** all $s \xrightarrow{\epsilon} s'$ ▷ If we get to state s' from state s by reading sign ϵ.
6 **do if** $s' \in$ *final-state*(A_R)
7 **then** *result*$[u, s] \leftarrow \{u\} \cup$ *result*$[u, s]$
8 *result*$[u, s] \leftarrow$ *result*$[u, s] \cup$ TRAVERSE(u, s')
9 **for** all $s \xrightarrow{label(u)} s'$ ▷ If we get to state s' from state s by reading sign *label*(u).
10 **do if** $s' \in$ *final-state*(A_R)
11 **then** *result*$[u, s] \leftarrow \{u\} \cup$ *result*$[u, s]$
12 **for** all v, where $(u, v) \in E(G)$ ▷ Continue the traversal for the children of node u recursively.
13 **do** *result*$[u, s] \leftarrow$ *result*$[u, s] \cup$ TRAVERSE(v, s')
14 **return** *result*$[u, s]$

Proposition 20.6 *Given a regular query R and a graph G, the calculation cost of R(G) is a polynomial of the number of edges of G and the number of different states of the finite nondeterministic automaton corresponding to R.*

Proof. The sketch of the proof is the following. Let A_R be the finite nondeterministic automaton corresponding to R. Denote by $|A_R|$ the number of states of A_R. Consider the breadth-first traversal corresponding to the algorithm TRAVERSE of graph G with m edges, starting

from the root. During the traversal we get to a new state of the automaton according to the label of the node, and we store the state reached at the node for each node. If the final state of the automaton is acceptance, then the node is a result. During the traversal, we sometimes have to step back on an edge to ensure we continue to places we have not seen yet. It can be proved that during a traversal every edge is used at most once in every state, so this is the number of steps performed by that automaton. This means $O(|A_R|m)$ steps altogether, which completes the proof. ■

Two nodes of graph G are ***indistinguishable with regular expressions*** if there is no regular R for which one of the nodes is among the results and the other node is not. Of course, if two nodes cannot be distinguished, then their labels are the same. Let us categorize the nodes in such a way that nodes with the same label are in the same class. This way we produce a partition P of the set of nodes, which is called the ***basic partition***. It can also be seen easily that if two nodes are indistinguishable, then it is also true for the parents. This implies that the set of label sequences corresponding to paths from the root to the indistinguishable nodes is the same. Let $L(n) = \{l_0, \ldots, l_p \mid n$ fits to the label sequence $l_0, \ldots, l_p\}$ for all nodes n. Nodes n_1 and n_2 are indistinguishable if and only if $L(n_1) = L(n_2)$. If the nodes are assigned to classes in such a way that the nodes having the same value $L(n)$ are arranged to the same class, then we get a refinement P' of partition P. For this new partition, if a node n is among the results of a regular query R, then all nodes from the equivalence class of n are also among the results of the query.

Definition 20.7 *Given a graph $G = (V, E, root, \Sigma, label, id, value)$ and a partition P of V that is a refinement of the basic partition, that is, for which the nodes belonging to the same equivalence class have the same label. Then the graph $I(G) = (P, E', root', \Sigma, label', id', value')$ is called an* ***index****. The nodes of the index graph are the equivalence classes of partition P, and $(I, J) \in E'$ if and only if there exist $i \in I$ and $j \in J$ such that $(i, j) \in E$. If $I \in P$, then $id'(I)$ is the identifier of index node I, $label'(I) = label(n)$, where $n \in I$, and root' is the equivalence class of partition P that contains the root of G. If $label(I) = VALUE$, then $label'(I) = \{value(n) \mid n \in I\}$.*

Given a partition P of set V, denote by ***class(n)*** the equivalence class of P that contains n for $n \in V$. In case of indexes, the notation $I(n)$ can also be used instead of $class(n)$.

Note that basically the indexes can be identified with the different partitions of the nodes, so partitions can also be called indexes without causing confusion. Those indexes will be good that are of small size and for which the result of queries is the same on the graph and on the index. Indexes are usually given by an equivalence relation on the nodes, and the partition corresponding to the index consists of the equivalence classes.

Definition 20.8 *Let P be the partition for which $n, m \in I$ for a class I if and only if $L(n) = L(m)$. Then the index $I(G)$ corresponding to P is called a* ***naive index****.*

In case of naive indexes, the same language $L(n)$ is assigned to all elements n of class I in partition P, which will be denoted by $L(I)$.

Proposition 20.9 *Let I be a node of the naive index and R a regular expression. Then $I \cap R(G) = \emptyset$ or $I \subseteq R(G)$.*

Proof. Let $n \in I \cap R(G)$ and $m \in I$. Then there exists a label sequence $l_0, \ldots, l_p$ in $L(R)$ to

which n fits, that is, $l_0, \ldots, l_p \in L(n)$. Since $L(n) = L(m)$, m also fits to this label sequence, so $m \in I \cap R(G)$. ■

NAIVE-INDEX-EVALUATION(G, R)

```
1 let I_G be the naive index of G
2 Q ← ∅
3 for all I ∈ NAIVE-EVALUATION(I_G, A_R)
4     do Q ← Q ∪ I
5 return Q
```

Proposition 20.10 *Set Q produced by the algorithm* NAIVE-INDEX-EVALUATION *is equal to $R(G)$.*

Proof. Because of the previous proposition either all elements of a class I are among the results of a query or none of them. ■

Using naive indexes we can evaluate queries, but, according to the following proposition, not efficiently enough. The proposition was proved by Stockmeyer and Meyer in 1973.

Proposition 20.11 *The creation of the naive index I_G needed in the algorithm* NAIVE-INDEX-EVALUATION *is PSPACE-complete.*

The other problem with using naive indexes is that the sets $L(I)$ are not necessary disjoint for different I, which might cause redundancy in storing.

Because of the above we will try to find a refinement of the partition corresponding to the naive index, which can be created efficiently and can still be used to produce $R(G)$.

Definition 20.12 *Index $I(G)$ is **safe** if for any $n \in V$ and label sequence $l_0, \ldots, l_p$ such that n fits to the label sequence $l_0, \ldots, l_p$ in graph G, class(n) fits to the label sequence $l_0, \ldots, l_p$ in graph $I(G)$. Index $I(G)$ is **exact** if for any class I of the index and label sequence $l_0, \ldots, l_p$ such that I fits to the label sequence $l_0, \ldots, l_p$ in graph $I(G)$, arbitrary node $n \in I$ fits to the label sequence $l_0, \ldots, l_p$ in graph G.*

Safety means that the nodes belonging to the result we obtain by evaluation using the index contain the result of the regular query, that is, $R(G) \subseteq R(I(G))$, while exactness means that the evaluation using the index does not provide false results, that is, $R(I(G)) \subseteq R(G)$. Using the definitions of exactness and of the edges of the index the following proposition follows.

Proposition 20.13 *1. Every index is safe.*
2. The naive index is safe and exact.

If I is a set of nodes of G, then the language $L(I)$, to the label strings of which the elements of I fit, was defined using graph G. If we wish to indicate this, we use the notation $L(I, G)$. However, $L(I)$ can also be defined using graph $I(G)$, in which I is a node. In this case, we can use the notation $L(I, I(G))$ instead of $L(I)$, which denotes all label sequences to which node I fits in graph $I(G)$. $L(I, G) = L(I, I(G))$ for safe and exact indexes, so in this case we can write $L(I)$ for simplicity. Then $L(I)$ can be computed using $I(G)$, since the size

of $I(G)$ is usually smaller than that of G.

Arbitrary index graph can be queried using the algorithm NAIVE-EVALUATION. After that join the index nodes obtained. If we use an exact index, then the result will be the same as the result we would have obtained by querying the original graph.

INDEX-EVALUATION$(G, I(G), A_R)$

1 let $I(G)$ be the index of G
2 $Q \leftarrow \emptyset$
3 **for** all $I \in$ NAIVE-EVALUATION$(I(G), A_R)$
4 **do** $Q \leftarrow Q \cup I$
5 **return** Q

First, we will define a safe and exact index that can be created efficiently, and is based on the similarity of nodes. We obtain the 1-index this way. Its size can be decreased if we only require similarity locally. The A(k)-index obtained this way lacks exactness, therefore using the algorithm INDEX-EVALUATION we can get results that do not belong to the result of the regular query R, so we have to test our results to ensure exactness.

Definition 20.14 *Let $\approx$ be an equivalence relation on set V such that, for $u \approx v$,*
i) $label(u) = label(v)$,
ii) if there is an edge from node u' to node u, then there exists a node v' for which there is an edge from node v' to node v and $u' \approx v'$.
iii) if there is an edge from node v' to node v, then there exists a node u' for which there is an edge from node u' to node u and $u' \approx v'$.
*The above equivalence relation is called a **bisimulation**. Nodes u and v of a graph are **bisimilar** if and only if there exists a bisimulation $\approx$ such that $u \approx v$.*

Definition 20.15 *Let P be the partition consisting of the equivalence classes of a bisimulation. The index defined using partition P is called **1-index**.*

Proposition 20.16 *The 1-index is a refinement of the naive index. If the labels of the ingoing edges of the nodes in graph G are different, that is, $label(x) \neq label(x')$ for $x \neq x'$ and $(x, y), (x', y) \in E$, then $L(u) = L(v)$ if and only if u and v are bisimilar.*

Proof. $label(u) = label(v)$ if $u \approx v$. Let node u fit to the label sequence $l_0, \ldots, l_p$, and let u' be the node corresponding to label l_{p-1}. Then there exists a v' such that $u' \approx v'$ and $(u', u), (v', v) \in E$. u' fits to the label sequence $l_0, \ldots, l_{p-1}$, so, by induction, v' also fits to the label sequence $l_0, \ldots, l_{p-1}$, therefore v fits to the label sequence $l_0, \ldots, l_p$. So, if two nodes are in the same class according to the 1-index, then they are in the same class according to the naive index as well.

To prove the second statement of the proposition, it is enough to show that the naive index corresponds to a bisimulation. Let u and v be in the same class according to the naive index. Then $label(u) = label(v)$. If $(u', u) \in E$, then there exists a label sequence $l_0, \ldots, l_p$ such that the last two nodes corresponding to the labels are u' and u. Since we assumed that the labels of the parents are different, $L(u) = L' \cup L''$, where L' and L'' are disjoint, and $L' = \{l_0, \ldots, l_p \mid u' \text{ fits to the sequence } l_0, \ldots, l_{p-1}, \text{ and } l_p = label(u)\}$, while $L'' = L(u) \setminus L'$. Since $L(u) = L(v)$, there exists a v' such that $(v', v) \in E$ and $label(u') = label(v')$. $L' =$

$\{l_0, \ldots, l_p \mid v'$ fits to the sequence $l_0, \ldots, l_{p-1}$, and $l_p = label(v)\}$ because of the different labels of the parents, so $L(u') = L(v')$, and $u' \approx v'$ by induction, therefore $u \approx v$. ■

Proposition 20.17 *The 1-index is safe and exact.*

Proof. If x_p fits to the label sequence $l_0, \ldots, l_p$ in graph G because of nodes $x_0, \ldots, x_p$, then, by the definition of the index graph, there exists an edge from $class(x_i)$ to $class(x_{i+1})$, $0 \leq i \leq p-1$, that is, $class(x_p)$ fits to the label sequence $l_0, \ldots, l_p$ in graph $I(G)$. To prove exactness, assume that I_p fits to the label sequence $l_0, \ldots, l_p$ in graph $I(G)$ because of $I_0, \ldots, I_p$. Then there are $u' \in I_{p-1}$, $u \in I_p$ such that $u' \approx v'$ and $(v', v) \in E$, that is, $v' \in I_{p-1}$. We can see by induction that v' fits to the label sequence $l_0, \ldots, l_{p-1}$ because of nodes $x_0, \ldots, x_{p-2}, v'$, but then v fits to the label sequence $l_0, \ldots, l_p$ because of nodes $x_0, \ldots, x_{p-2}, v', v$ in graph G. ■

If we consider the bisimulation in case of which all nodes are assigned to different partitions, then the graph $I(G)$ corresponding to this 1-index is the same as graph G. Therefore the size of $I(G)$ is at most the size of G, and we also have to store the elements of I for the nodes I of $I(G)$, which means we have to store all nodes of G. For faster evaluation of queries we need to find the smallest 1-index, that is, the coarsest 1-index. It can be checked that x and y are in the same class according to the coarsest 1-index if and only if x and y are bisimilar.

1-INDEX-EVALUATION(G, R)

1 let I_1 be the coarsest 1-index of G
2 **return** INDEX-EVALUATION(G, I_1, A_R)

In the first step of the algorithm, the coarsest 1-index has to be given. This can be reduced to finding the coarsest stable partition, what we will discuss in the next section of this chapter. Thus using the efficient version of the PT-algorithm, the coarsest 1-index can be found with computation cost $O(m \lg n)$ and space requirement $O(m + n)$, where n and m denote the number of nodes and edges of graph G, respectively.

Since graph I_1 is safe and exact, it is sufficient to evaluate the query in graph I_1, that is, to find the index nodes that fit to the regular expression R. Using Proposition 20.6, the cost of this is a polynomial of the size of graph I_1.

The size of I_1 can be estimated using the following parameters. Let p be the number of different labels in graph G, and k ***the diameter of graph*** G, that is, the length of the longest directed path. (No node can appear twice in the directed path.) If the graph is a tree, then the diameter is the depth of the tree. We often create websites that form a tree of depth d, then we add a navigation bar consisting of q elements to each page, that is, we connect each node of the graph to q chosen pages. It can be proved that in this case the diameter of the graph is at most $d + q(d-1)$. In practice, d and q are usually very small compared to the size of the graph. The proof of the following proposition can be found in the paper of Milo and Suciu.

Proposition 20.18 *Let the number of different labels in graph G be at most p, and let the diameter of G be less than k. Then the size of the 1-index I_1 defined by an arbitrary bisimulation can be bounded from above with a bound that only depends on k and p but does not depend on the size of G.*

Exercises

20.3-1 Show that the index corresponding to the maximal simulation is between the 1-index and the naive index with respect to refinement. Give an example that shows that both inclusions are proper.

20.3-2 Denote by $I_s(G)$ the index corresponding to the maximal simulation. Does $I_s(I_s(G)) = I_s(G)$ hold?

20.3-3 Represent graph G and the state transition graph of the automaton corresponding to the regular expression R with relational databases. Give an algorithm in a relational query language, for example in PL/SQL, that computes $R(G)$.

20.4. Stable partitions and the PT-algorithm

Most index structures used for efficient evaluation of queries of semi-structured databases are based on a partition of the nodes of a graph. The problem of creating indexes can often be reduced to finding the coarsest stable partition.

Definition 20.19 *Let E be a binary relation on the finite set V, that is, $E \subseteq V \times V$. Then V is the set of* ***nodes****, and E is the set of* ***edges****. For arbitrary $S \subseteq V$, let $E(S) = \{y \mid \exists x \in S, (x,y) \in E\}$ and $E^{-1}(S) = \{x \mid \exists\, y \in S, (x,y) \in E\}$. We say that B* ***is stable with respect to*** *S for arbitrary $S \subseteq V$ and $B \subseteq V$, if $B \subseteq E^{-1}(S)$ or $B \cap E^{-1}(S) = \emptyset$. Let P be a partition of V, that is, a decomposition of V into disjoint sets, or in other words, blocks. Then P* ***is stable with respect to*** *S, if all blocks of P are stable with respect to S. P* ***is stable with respect to partition*** *P', if all blocks of P are stable with respect to all blocks of P'. If P is stable with respect to all of its blocks, then partition P is* ***stable****. Let P and Q be two partitions of V. Q* ***is a refinement of*** *P, or P* ***is coarser than*** *Q, if every block of P is the union of some blocks of Q. Given V, E and P,* ***the coarsest stable partition*** *is the coarsest stable refinement of P, that is, the stable refinement of P that is coarser than any other stable refinement of P.*

Note that stability is sometimes defined the following way. B is stable with respect to S if $B \subseteq E(S)$ or $B \cap E(S) = \emptyset$. This is not a major difference, only the direction of the edges is reversed. So in this case stability is defined with respect to the binary relation E^{-1} instead of E, where $(x,y) \in E^{-1}$ if and only if $(y,x) \in E$, since $(E^{-1})^{-1}(S) = \{x \mid \exists\, y \in S, (x,y) \in E^{-1})\} = \{x \mid \exists\, y \in S, (y,x) \in E\} = E(S)$.

Let $|V| = n$ and $|E| = m$. We will prove that there always exists a unique solution of the problem of finding the coarsest stable partition, and there is an algorithm that finds the solution in $O(m \lg n)$ time with space requirement $O(m+n)$. This algorithm was published by R. Paige and R. E. Tarjan in 1987, therefore it will be called the ***PT-algorithm***.

The main idea of the algorithm is that if a block is not stable, then it can be split into two in such a way that the two parts obtained are stable. First we will show a naive method. Then, using the properties of the split operation, we will increase its efficiency by continuing the procedure with the smallest part.

Definition 20.20 *Let E be a binary relation on V, $S \subseteq V$ and Q a partition of V. Furthermore, let* ***split****(S, Q) be the refinement of Q which is obtained by splitting all blocks B of Q*

that are not disjoint from $E^{-1}(S)$, that is, $B \cap E^{-1}(S) \neq \emptyset$ and $B \setminus E^{-1}(S) \neq \emptyset$. In this case, add blocks $B \cap E^{-1}(S)$ and $B \setminus E^{-1}(S)$ to the partition instead of B. S is a ***splitter*** *of Q if $split(S, Q) \neq Q$.*

Note that Q is not stable with respect to S if and only if S is a splitter of Q.

Stability and splitting have the following properties, the proofs are left to the Reader.

Proposition 20.21 *Let S and T be two subsets of V, while P and Q two partitions of V. Then*
1. Stability is preserved under refinement, that is, if Q is a refinement of P, and P is stable with respect to S, then Q is also stable with respect to S.
2. Stability is preserved under unification, that is, if P is stable with respect to both S and T, then P is stable with respect to $S \cup T$.
3. The split operation is monotonic in its second argument, that is, if P is a refinement of Q, then $split(S, P)$ is a refinement of $split(S, Q)$.
4. The split operation is commutative in the following sense. For arbitrary S, T and P, $split(S, split(T, P)) = split(T, split(S, P))$, and the coarsest partition of P that is stable with respect to both S and T is $split(S, split(T, P))$.

In the naive algorithm, we refine partition Q starting from partition P, until Q is stable with respect to all of its blocks. In the refining step, we seek a splitter S of Q that is a union of some blocks of Q. Note that finding a splitter among the blocks of Q would be sufficient, but this more general way will help us in improving the algorithm.

NAIVE-PT(V, E, P)

```
1 Q ← P
2 while Q is not stable
3       do let S be a splitter of Q that is the union of some blocks of Q
4          Q ← split(S, Q)
5 return Q
```

Note that the same set S cannot be used twice during the execution of the algorithm, since stability is preserved under refinement, and the refined partition obtained in step 4 is stable with respect to S. The union of the sets S used can neither be used later, since stability is also preserved under unification. It is also obvious that a stable partition is stable with respect to any S that is a union of some blocks of the partition. The following propositions can be proved easily using these properties.

Proposition 20.22 *In any step of the algorithm* NAIVE-PT*, the coarsest stable refinement of P is a refinement of the actual partition stored in Q.*

Proof. The proof is by induction on the number of times the cycle is executed. The case $Q = P$ is trivial. Suppose that the statement holds for Q before using the splitter S. Let R be the coarsest stable refinement of P. Since S consists of blocks of Q, and, by induction, R is a refinement of Q, therefore S is the union of some blocks of R. R is stable with respect to all of its blocks and the union of any of its blocks, thus R is stable with respect to S, that is, $R = split(S, R)$. On the other hand, using that the split operation is monotonic, $split(S, R)$ is a refinement of $split(S, Q)$, which is the actual value of Q. ■

Proposition 20.23 *The algorithm* NAIVE-PT *determines the unique coarsest stable refinement of P, while executing the cycle at most $n-1$ times.*

Proof. The number of blocks of Q is obviously at least 1 and at most n. Using the split operation, at least one block of Q is divided into two, so the number of blocks increases. This implies that the cycle is executed at most $n-1$ times. Q is a stable refinement of P when the algorithm terminates, and, using the previous proposition, the coarsest stable refinement of P is a refinement of Q. This can only happen if Q is the coarsest stable refinement of P. ■

Proposition 20.24 *If we store the set $E^{-1}(\{x\})$ for all elements x of V, then the cost of the algorithm* NAIVE-PT *is at most $O(mn)$.*

Proof. We can assume, without restricting the validity of the proof, that there are no sinks in the graph, that is, every node has outgoing edges. Then $1 \leq |E(\{x\})|$ for arbitrary x in V. Consider a partition P, and split all blocks B of P. Let B' be the set of the nodes of B that have at least one outgoing edge. Then $B' = B \cap E^{-1}(V)$. Now let $B'' = B \setminus E^{-1}(V)$, that is, the set of sinks of B. Set B'' is stable with respect to arbitrary S, since $B'' \cap E^{-1}(S) = \emptyset$, so B'' does not have to be split during the algorithm. Therefore, it is enough to examine partition P' consisting of blocks B' instead of P, that is, a partition of set $V' = E^{-1}(V)$. By adding blocks B'' to the coarsest stable refinement of P' we obviously get the coarsest stable refinement of P. This means that there is a preparation phase before the algorithm in which P' is obtained, and a processing phase after the algorithm in which blocks B'' are added to the coarsest stable refinement obtained by the algorithm. The cost of preparation and processing can be estimated the following way. V' has at most m elements. If, for all x in V we have $E^{-1}(\{x\})$, then the preparation and processing requires $O(m+n)$ time.

From now on we will assume that $1 \leq |E(\{x\})|$ holds for arbitrary x in V, which implies that $n \leq m$. Since we store sets $E^{-1}(\{x\})$, we can find a splitter among the blocks of partition Q in $O(m)$ time. This, combined with the previous proposition, means that the algorithm can be performed in $O(mn)$ time. ■

The algorithm can be executed more efficiently using a better way of finding splitter sets. The main idea of the improved algorithm is that we work with two partitions besides P, Q and a partition X that is a refinement of Q in every step such that Q is stable with respect to all blocks of X. At the start, let $Q = P$ and let X be the partition consisting only one block, set V. The refining step of the algorithm is repeated until $Q = X$.

PT(V, E, P)

1 $Q \leftarrow P$
2 $X \leftarrow \{V\}$
3 **while** $X \neq Q$
4 **do** let S be a block of X that is not a block of Q,
and B a block of Q in S for which $|B| \leq |S|/2$
5 $X \leftarrow (X \setminus \{S\}) \cup \{B, S \setminus B\}$
6 $Q \leftarrow split(S \setminus B, split(B, Q))$
7 **return** Q

Proposition 20.25 *The result of the* PT*-algorithm is the same as that of algorithm* Naive-PT.

Proof. At the start, Q is a stable refinement of P with respect to the blocks of X. In step 5, a block of X is split, thus we obtain a refinement of X. In step 6, by refining Q using splits we ensure that Q is stable with respect to two new blocks of X. The properties of stability mentioned in Proposition 20.21 and the correctness of algorithm Naive-PT imply that the PT-algorithm also determines the unique coarsest stable refinement of P. ■

In some cases one of the two splits of step 6 can be omitted. A sufficient condition is that E is a function of x.

Proposition 20.26 *If* $|E(\{x\})| = 1$ *for all* x *in* V*, then step 6 of the* PT*-algorithm can be exchanged with* $Q \leftarrow split(B, Q)$.

Proof. Suppose that Q is stable with respect to a set S which is the union of some blocks of Q. Let B be a block of Q that is a subset of S. It is enough to prove that $split(B, Q)$ is stable with respect to $(S \setminus B)$. Let B_1 be a block of $split(B, Q)$. Since the result of a split according to B is a stable partition with respect to B, either $B_1 \subseteq E^{-1}(B)$ or $B_1 \subseteq E^{-1}(S) \setminus E^{-1}(B)$. Using $|E(\{x\})| = 1$, we get $B_1 \cap E^{-1}(S \setminus B) = \emptyset$ in the first case, and $B_1 \subseteq E^{-1}(S \setminus B)$ in the second case, which means that we obtained a stable partition with respect to $(S \setminus B)$. ■

Note that the stability of a partition with respect to S and B generally does not imply that it is also stable with respect to $(S \setminus B)$. If this is true, then the execution cost of the algorithm can be reduced, since the only splits needed are the ones according to B because of the reduced sizes.

The two splits of step 6 can cut a block into four parts in the general case. According to the following proposition, one of the two parts gained by the first split of a block remains unchanged at the second split, so the two splits can result in at most three parts. Using this, the efficiency of the algorithm can be improved even in the general case.

Proposition 20.27 *Let* Q *be a stable partition with respect to* S*, where* S *is the union of some blocks of* Q*, and let* B *be a block of* Q *that is a subset of* S*. Furthermore, let* D *be a block of* Q *that is cut into two (proper) parts* D_1 *and* D_2 *by the operation* $split(B, Q)$ *in such a way that none of these is the empty set. Suppose that block* D_1 *is further divided into the nonempty sets* D_{11} *and* D_{12} *by* $split(S \setminus B, split(B, Q))$*. Then*
1. $D_1 = D \cap E^{-1}(B)$ *and* $D_2 = D \setminus D_1$ *if and only if* $D \cap E^{-1}(B) \neq \emptyset$ *and* $D \setminus E^{-1}(B) \neq \emptyset$.
2. $D_{11} = D_1 \cap E^{-1}(S \setminus B)$ *and* $D_{12} = D_1 \setminus D_{11}$ *if and only if* $D_1 \cap E^{-1}(S \setminus B) \neq \emptyset$ *and* $D_1 \setminus E^{-1}(S \setminus B) \neq \emptyset$.
3. The operation $split(S \setminus B, split(B, Q))$ *leaves block* D_2 *unchanged.*
4. $D_{12} = D_1 \cap (E^{-1}(B) \setminus E^{-1}(S \setminus B))$.

Proof. The first two statements follow using the definition of the split operation. To prove the third statement, suppose that D_2 was obtained from D by a proper decomposition. Then $D \cap E^{-1}(B) \neq \emptyset$, and since $B \subseteq S$, $D \cap E^{-1}(S) \neq \emptyset$. All blocks of partition Q, including D, are stable with respect to S, which implies $D \subseteq E^{-1}(S)$. Since $D_2 \subseteq D$, $D_2 \subseteq E^{-1}(S) \setminus E^{-1}(B) = E^{-1}(S \setminus B)$ using the first statement, so D_2 is stable with respect to the set $S \setminus B$, therefore a split according to $S \setminus B$ does not divide block D_2. Finally, the fourth statement follows from $D_1 \subseteq E^{-1}(B)$ and $D_{12} = D_1 \setminus E^{-1}(S \setminus B)$. ■

Denote by $counter(x, S)$ the number of nodes in S that can be reached from x, that is, $counter(x, S) = |S \cap E(\{x\})|$. Note that if $B \subseteq S$, then $E^{-1}(B) \setminus E^{-1}(S \setminus B) = \{x \in E^{-1}(B) \mid counter(x, B) = counter(x, S)\}$.

Since sizes are always halved, an arbitrary x in V can appear in at most $\lg n + 1$ different sets B that were used for refinement in the PT-algorithm. In the following, we will give an execution of the PT algorithm in which the determination of the refinement according to block B in steps 5 and 6 of the algorithm costs $O(|B| + \sum_{y \in B} |E^{-1}(\{y\})|)$. Summing this for all blocks B used in the algorithm and for all elements of these blocks, we get that the complexity of the algorithm EFFICIENT-PT is at most $O(m \lg n)$. To give such a realization of the algorithm, we have to choose good data structures for the representation of our data.

- Attach node x to all edges (x, y) of set E, and attach the list $\{(x, y) \mid (x, y) \in E\}$ to all nodes y. Then the cost of reading set $E^{-1}(\{y\})$ is proportional to the size of $E^{-1}(\{y\})$.

- Let partition Q be a refinement of partition X. Represent the blocks of the two partitions by records. A block S of partition X is ***simple*** if it consists of one block of Q, otherwise it is ***compound***.

- Let C be the list of all compound blocks in partition X. At start, let $C = \{V\}$, since V is the union of the blocks of P. If P consists of only one block, then P is its own coarsest stable refinement, so no further computation is needed.

- For any block S of partition P, let $Q\text{-}blocks(S)$ be the double-chained list of the blocks of partition Q the union of which is set S. Furthermore, store the values $counter(x, S)$ for all x in set $E^{-1}(S)$ to which one pointer points from all edges (x, y) such that y is an element of S. At start, the value assigned to all nodes x is $counter(x, V) = |E(\{x\})|$, and make a pointer to all nodes (x, y) that points to the value $counter(x, V)$.

- For any block B of partition Q, let $X\text{-}block(B)$ be the block of partition X in which B appears. Furthermore, let $size(B)$ be the cardinality of B, and $elements(B)$ the double-chained list of the elements of B. Attach a pointer to all elements that points to the block of Q in which this element appears. Using double chaining any element can be deleted in $O(1)$ time.

Using the proof of Proposition 20.24, we can suppose that $n \leq m$ without restricting the validity. It can be proved that in this case the space requirement for the construction of such data structures is $O(m)$.

EFFICIENT-PT(V, E, P)

```
 1 if |P| = 1
 2   then return P
 3 Q ← P
 4 X ← {V}
 5 C ← {V}                                   ▷ C is the list of the compound blocks of X.
 6 while C ≠ ∅
 7       do let S be an element of C
 8          let B be the smaller of the first two elements of S
 9          C ← C \ {S}
10          X ← (X \ {S}) ∪ {{B}, S \ {B}}
11          S ← S \ {B}
12          if |S| > 1
13            then C ← C ∪ {S}
14          Generate set E^{-1}(B) by reading the edges (x, y) of set E for which y
            is an element of B, and for all elements x of this set, compute the value
            counter(x, B).
15          Find blocks D_1 = D ∩ E^{-1}(B) and D_2 = D \ D_1 for all blocks
            D of Q by reading set E^{-1}(B)
16          By reading all edges (x, y) of set E for which y is an element of B,
            create set E^{-1}(B) \ E^{-1}(S \ B) checking the condition
            counter(x, B) = counter(x, S)
17          Reading set E^{-1}(B) \ E^{-1}(S \ B), for all blocks D of Q,
            determine the sets D_12 = D_1 ∩ (E^{-1}(B) \ E^{-1}(S \ B))
            and D_11 = D_1 \ D_12
18          for all blocks D of Q for which D_11 ≠ ∅, D_12 ≠ ∅ and D_2 ≠ ∅
19              do if D is a simple block of X
20                   then C ← C ∪ {D}
21                 Q ← (Q \ {D}) ∪ {D_11, D_12, D_2}
22          Compute the value counter(x, S) by reading
            the edges (x, y) of E for which y is an element of B.
23 return Q
```

Proposition 20.28 *The algorithm* EFFICIENT-PT *determines the coarsest stable refinement of P. The computation cost of the algorithm is $O(m \lg n)$, and its space requirement is $O(m+n)$.*

Proof. The correctness of algorithm follows from the correctness of the PT-algorithm and Proposition 20.27. Because of the data structures used, the computation cost of the steps of the cycle is proportional to the number of edges examined and the number of elements of block B, which is $O(|B| + \sum_{y \in B} |E^{-1}(\{y\})|)$ altogether. Sum this for all blocks B used during the refinement and all elements of these blocks. Since the size of B is at most half the size of S, arbitrary x in set V can be in at most $\lg n + 1$ different sets B. Therefore, the total computation cost of the algorithm is $O(m \lg n)$. It can be proved easily that a space of $O(m+n)$ size is enough for the storage of the data structures used in the algorithm and their maintenance. ■

Note that the algorithm could be further improved by contracting some of its steps but that would only decrease computation cost by a constant factor.

Let $G^{-1} = (V, E^{-1})$ be the graph that can be obtained from G by changing the direction of all edges of G. Consider a 1-index in graph G determined by the bisimulation $\approx$. Let I and J be two classes of the bisimulation, that is, two nodes of $I(G)$. Using the definition of bisimulation, $J \subseteq E(I)$ or $E(I) \cap J = \emptyset$. Since $E(I) = (E^{-1})^{-1}(I)$, this means that J is stable with respect to I in graph G^{-1}. So the coarsest 1-index of G is the coarsest stable refinement of the basic partition of graph G^{-1}.

Corollary 20.29 *The coarsest 1-index can be determined using the algorithm* Efficient-PT. *The computation cost of the algorithm is at most $O(m \lg n)$, and its space requirement is at most $O(m + n)$.*

Exercises

20.4-1 Prove Proposition 29.21.

20.4-2 Partition P is size-stable with respect to set S if $|E(\{x\}) \cap S| = |E(\{y\}) \cap S|$ for arbitrary elements x, y of a block B of P. A partition is size-stable if it is size-stable with respect to all its blocks. Prove that the coarsest size-stable refinement of an arbitrary partition can be computed in $O(m \lg n)$ time.

20.4-3 The 1-index is minimal if no two nodes I and J with the same label can be contracted, since there exists a node K for which $I \cup J$ is not stable with respect to K. Give an example that shows that the minimal 1-index is not unique, therefore it is not the same as the coarsest 1-index.

20.4-4 Prove that in case of an acyclic graph, the minimal 1-index is unique and it is the same as the coarsest 1-index.

20.5. A(k)-indexes

In case of 1-indexes, nodes of the same class fit to the same label sequences starting from the root. This means that the nodes of a class cannot be distinguished by their ancestors. Modifying this condition in such a way that indistinguishability is required only locally, that is, nodes of the same class cannot be distinguished by at most k generations of ancestors, we obtain an index that is coarser and consists of less classes than the 1-index. So the size of the index decreases, which also decreases the cost of the evaluation of queries. The 1-index was safe and exact, which we would like to preserve, since these guarantee that the result we get when evaluating the queries according to the index is the result we would have obtained by evaluating the query according to the original graph. The A(k)-index is also safe, but it is not exact, so this has to be ensured by modification of the evaluation algorithm.

Definition 20.30 *The **k-bisimulation** $\approx^k$ is anequivalence relation on the nodes V of a graph defined recursively as*
i) $u \approx^0 v$ if and only if label(u) = label(v),
ii) $u \approx^k v$ if and only if $u \approx^{k-1} v$ and if there is an edge from node u' to node u, then there is a node v' from which there is an edge to node v and $u' \approx^{k-1} v'$, also, if there is an edge from node v' to node v, then there is a node u' from which there is an edge to node u and

$u' \approx^{k-1} v'$.
In case $u \approx^k v$ u and v are ***k-bisimilar****. The classes of the partition according to the* ***A(k)-index*** *are the equivalence classes of the k-bisimulation.*

The „A" in the notation refers to the word „approximative".

Note that the partition belonging to $k = 0$ is the basic partition, and by increasing k we refine this, until the coarsest 1-index is reached.

Denote by $L(u, k, G)$ the label sequences of length at most k to which u fits in graph G. The following properties of the A(k)-index can be easily checked.

Proposition 20.31
1. If u and v are k-bisimilar, then $L(u, k, G) = L(v, k, G)$.
2. If I is a node of the $A(k)$-index and $u \in I$, then $L(I, k, I(G)) = L(u, k, G)$.
3. The $A(k)$-index is exact in case of simple expressions of length at most k.
4. The $A(k)$-index is safe.
5. The $(k+1)$-bisimulation is a (not necessarily proper) refinement of the k-bisimulation.

The A(k)-index compares the k-distance half-neighbourhoods of the nodes which contain the root, so the equivalence of the nodes is not affected by modifications outside this neighbourhood, as the following proposition shows.

Proposition 20.32 *Suppose that the shortest paths from node v to nodes x and y contain more than k edges. Then adding or deleting an edge from u to v does not change the k-bisimilarity of x and y.*

We use a modified version of the PT-algorithm for creating the A(k)-index. Generally, we can examine the problem of approximation of the coarsest stable refinement.

Definition 20.33 *Let P be a partition of V in the directed graph $G = (V, E)$, and let $P_0, P_1, \ldots, P_k$ be a sequence of partitions such that $P_0 = P$ and P_{i+1} is the coarsest refinement of P_i that is stable with respect to P_i. In this case, partition P_k is the* ***k-step approximation of the coarsest stable refinement of*** *P.*

Note that every term of sequence P_i is a refinement of P, and if $P_k = P_{k-1}$, then P_k is the coarsest stable refinement of P. It can be checked easily that an arbitrary approximation of the coarsest stable refinement of P can be computed greedily, similarly to the PT-algorithm. That is, if a block B of P_i is not stable with respect to a block S of P_{i-1}, then split B according to S, and consider the partition $split(S, P_i)$ instead of P_i.

NAIVE-APPROXIMATION(V, E, P, k)

```
1  P_0 ← P
2  for i ← 1 to k
3      do P_i ← P_{i-1}
4          for all S ∈ P_{i-1} such that split(S, P_i) ≠ P_i
5              do P_i ← split(S, P_i)
6  return P_k
```

Note that the algorithm NAIVE-APPROXIMATION could also be improved similarly to the PT-algorithm.

Algorithm NAIVE-APPROXIMATION can be used to compute the A(k)-index, all we have to notice is that the partition belonging to the A(k)-index is stable with respect to the partition belonging to the A($k-1$)-index in graph G^{-1}. It can be shown that the computation cost of the A(k)-index obtained this way is $O(km)$, where m is the number of edges in graph G.

A(k)-INDEX-EVALUATION(G, A_R, k)

```
1 let I_k be the A(k)-index of G
2 Q ← INDEX-EVALUATION(G, I_k, A_R)
3 for all u ∈ Q
4     do if L(u) ∩ L(A_R) = ∅
5           then Q ← Q \ {u}
6 return Q
```

The A(k)-index is safe, but it is only exact for simple expressions of length at most k, so in step 4, we have to check for all elements u of set Q whether it satisfies query R, and we have to delete those from the result that do not fit to query R. We can determine using a finite nondeterministic automaton whether a given node satisfies expression R as in Proposition 20.6, but the automaton has to run in the other way. The number of these checks can be reduced according to the following proposition, the proof of which is left to the Reader.

Proposition 20.34 *Suppose that in the graph I_k belonging to the A(k)-index, index node I fits to a label sequence that ends with $s = l_0, \ldots, l_p$, $p \leq k-1$. If all label sequences of the form s'.s that start from the root satisfy expression R in graph G, then all elements of I satisfy expression R.*

Exercises

20.5-1 Denote by $A_k(G)$ the A(k)-index of G. Determine whether $A_k(A_l(G)) = A_{k+l}(G)$.

20.5-2 Prove Proposition 20.31.

20.5-3 Prove Proposition 20.32.

20.5-4 Prove Proposition 20.34.

20.5-5 Prove that the algorithm NAIVE-APPROXIMATION generates the coarsest k-step stable approximation.

20.5-6 Let $A = \{A_0, A_1, \ldots, A_k\}$ be a set of indexes, the elements of which are A(0)-, A(1)-, ..., A(k)-indexes, respectively. A is *minimal*, if by uniting any two elements of A_i, A_i is not stable with respect to A_{i-1}, $1 \leq i \leq k$. Prove that for arbitrary graph, there exists a unique minimal A the elements of which are coarsest A(i)-indexes, $0 \leq i \leq k$.

20.6. D(k)- and M(k)-indexes

When using A(k)-indexes, the value of k must be chosen appropriately. If k is too large, the size of the index will be too big, and if k is too small, the result obtained has to be checked too many times in order to preserve exactness. Nodes of the same class are similar locally, that is, they cannot be distinguished by their k distance neighbourhoods, or, more precisely,

by the paths of length at most k leading to them. The same k is used for all nodes, even though there are less important nodes. For instance, some nodes appear very rarely in results of queries in practice, and only the label sequences of the paths passing through them are examined. There is no reason for using a better refinement on the less important nodes. This suggests the idea of using the dynamic D(k)-index, which assigns different values k to the nodes according to queries. Suppose that a set of queries is given. If there is an $R.a.b$ and an $R.a.b.c$ query among them, where R and R' are regular queries, then a partition according to at least 1-bisimulation in case of nodes with label b, and according to at least 2-bisimulation in case of nodes with label c is needed.

Definition 20.35 *Let $I(G)$ be the index graph belonging to graph G, and to all index node I assign a nonnegative integer $k(I)$. Suppose that the nodes of block I are $k(I)$-bisimilar. Let the values $k(I)$ satisfy the following condition: if there is an edge from I to J in graph $I(G)$, then $k(I) \geq k(J) - 1$. The index $I(G)$ having this property is called a* ***D*(k)*-index*.

The „D" in the notation refers to the word „dynamic". Note that the A(k)-index is a special case of the D(k)-index, since in case of A(k)-indexes, the elements belonging to any index node are exactly k-bisimilar.

Since classification according to labels, that is, the basic partition is an A(0)-index, and in case of finite graphs, the 1-index is the same as an A(k)-index for some k, these are also special cases of the D(k)-index. The D(k)-index, just like any other index, is safe, so it is sufficient to evaluate the queries on them. Results must be checked to ensure exactness. The following proposition states that exactness is guaranteed for some queries, therefore checking can be omitted in case of such queries.

Proposition 20.36 *Let $I_1, I_2, \ldots, I_s$ be a directed path in the D(k)-index, and suppose that $k(I_j) \geq j - 1$ if $1 \leq j \leq s$. Then all elements of I_s fit to the label sequence label(I_1), label(I_2), ..., label(I_s).*

Proof. The proof is by induction on s. The case $s = 1$ is trivial. By the inductive assumption, all elements of I_{s-1} fit to the label sequence label(I_1), label(I_2), ..., label(I_{s-1}). Since there is an edge from node I_{s-1} to node I_s in graph $I(G)$, there exist $u \in I_s$ and $v \in I_{s-1}$ such that there is an edge from v to u in graph G. This means that u fits to the label sequence label(I_1), label(I_2), ..., label(I_s) of length $s - 1$. The elements of I_s are at least $(s - 1)$-bisimilar, therefore all elements of I_s fit to this label sequence. ■

Corollary 20.37 *The D(k)-index is exact with respect to label sequence $l_0, \ldots, l_m$ if $k(I) \geq m$ for all nodes I of the index graph that fit to this label sequence.*

When creating the D(k)-index, we will refine the basic partition, that is, the A(0)-index. We will assign initial values to the classes consisting of nodes with the same label. Suppose we use t different values. Let K_0 be the set of these values, and denote the elements of K_0 by $k_1 > k_2 > \ldots > k_t$. If the elements of K_0 do not satisfy the condition given in the D(k)-index, then we increase them using the algorithm WEIGHT-CHANGER, starting with the greatest value, in such a way that they satisfy the condition. Thus, the classes consisting of nodes with the same label will have good k values. After this, we refine the classes by splitting them, until all elements of a class are k-bisimilar, and assign this k to all terms of the split. During this process the edges of the index graph must be refreshed according to the partition obtained

by refinement.

WEIGHT-CHANGER(G, K_0)

```
1  K ← ∅
2  K_1 ← K_0
3  while K_1 ≠ ∅
4        do for all I, where I is a node of the A(0)-index and k(I) = max(K_1)
5              do for all J, where J is a node of the A(0)-index and there is an edge from J to I
6                    k(J) ← max(k(J), max(K_1) − 1)
7        K ← K ∪ {max(K_1)}
8        K_1 ← {k(A) | A is a node of the A(0)-index } \ K
9  return K
```

It can be checked easily that the computation cost of the algorithm WEIGHT-CHANGER is $O(m)$, where m is the number of edges of the A(0)-index.

D(k)-INDEX-CREATOR(G, K_0)

```
1  let I(G) be the A(0)-index belonging to graph G, let V_I be the set of nodes of I(G),
   let E_I be the set of edges of I(G)
2  K ← Weight-Changer(G, K_0)                    ▷ Changing the initial weights
                                                   according to the condition of the D(k)-index.
3  for k ← 1 to max(K)
4     do for all I ∈ V_I
5           do if k(I) ≥ k
6                 then for all J, where (J, I) ∈ E_I
7                       do V_I ← (V_I \ {I}) ∪ {I ∩ E(J), I \ E(J)}
8                          k(I ∩ E(J)) ← k(I)
9                          k(I \ E(J)) ← k(I)
10                         E_I ← {(A, B) | A, B ∈ V_I, ∃ a ∈ A, ∃ b ∈ B, (a, b) ∈ E}
11                         I(G) ← (V_I, E_I)
12 return I(G)
```

In step 7, a split operation is performed. This ensures that the classes consisting of $(k-1)$-bisimilar elements are split into equivalence classes according to k-bisimilarity. It can be proved that the computation cost of the algorithm D(k)-INDEX-CREATOR is at most $O(km)$, where m is the number of edges of graph G, and $k = max(K_0)$.

In some cases, the D(k)-index results in a partition that is too fine, and it is not efficient enough for use because of its huge size. Over-refinement can originate in the following. The algorithm D(k)-INDEX-CREATOR assigns the same value k to the nodes with the same label, although some of these nodes might be less important with respect to queries, or appear more often in results of queries of length much less than k, so less fineness would be enough for these nodes. Based on the value k assigned to a node, the algorithm WEIGHT-CHANGER will not decrease the value assigned to the parent node if it is greater than $k-1$. Thus, if these parents are not very significant nodes considering frequent queries, then this can cause over-refinement. In order to avoid over-refinement, we introduce the M(k)-index and the M*(k)-index, where the „M" refers to the word „mixed", and the „*" shows that

not one index is given but a finite hierarchy of gradually refined indexes. The M(k)-index is a D(k)-index the creation algorithm of which not necessarily assigns nodes with the same label to the same k-bisimilarity classes.

Let us first examine how a D(k)-index $I(G) = (V_I, E_I)$ must be modified if the initial weight k_I of index node I is increased. If $k(I) \geq k_I$, then $I(G)$ does not change. Otherwise, to ensure that the conditions of the D(k)-index on weights are satisfied, the weights on the ancestors of I must be increased recursively until the weight assigned to the parents is at least $k_I - 1$. Then, by splitting according to the parents, the fineness of the index nodes obtained will be at least k_I, that is, the elements belonging to them will be at least k_I-bisimilar. This will be achieved using the algorithm WEIGHT-INCREASER.

WEIGHT-INCREASER($I, k_I, I(G)$)

```
1  if k(I) ≥ k_I
2     then return I(G)
3  for all (J, I) ∈ E_I
4     do I(G) ← WEIGHT-INCREASER(J, k_I − 1, I(G))
5  for all (J, I) ∈ E_I
6     do V_I ← (V_I \ {I}) ∪ {I ∩ E(J), I \ E(J)}
7        E_I ← {(A, B) | A, B ∈ V_I, ∃ a ∈ A, ∃ b ∈ B, (a, b) ∈ E}
8        I(G) ← (V_I, E_I)
9  return I(G)
```

The following proposition can be easily proved, and with the help of this we will be able to achieve the appropriate fineness in one step, so we will not have to increase step by step anymore.

Proposition 20.38 *$u \approx^k v$ if and only if $u \approx^0 v$, and if there is an edge from node u' to node u, then there is a node v', from which there is an edge to node v and $u' \approx^{k-1} v'$, and, conversely, if there is an edge from node v' to node v, then there is a node u', from which there is an edge to node u and $u' \approx^{k-1} v'$.*

Denote by *FRE* the set of simple expressions, that is, the label sequences determined by the frequent regular queries. We want to achieve a fineness of the index that ensures that it is exact on the queries belonging to *FRE*. For this, we have to determine the significant nodes, and modify the algorithm D(k)-INDEX-CREATOR in such a way that the not significant nodes and their ancestors are always deleted at the refining split.

Let $R \in FRE$ be a frequent simple query. Denote by S and T the set of nodes that fit to R in the index graph and data graph, respectively, that is $S = R(I(G))$ and $T = R(G)$. Denote by $k(I)$ the fineness of index node I in the index graph $I(G)$, then the nodes belonging to I are at most $k(I)$-bisimilar.

REFINE(R, S, T)

```
1  for all I ∈ S
2      do I(G) ← REFINE-INDEX-NODE(I, length(R), I ∩ T)
3  while ∃ I ∈ V_I such that k(I) < length(R) and I fits to R
4        do I(G) ← WEIGHT-INCREASER(I, length(R), I(G))
5  return I(G)
```

The refinement of the index nodes will be done using the following algorithm. First, we refine the significant parents of index node I recursively. Then we split I according to its significant parents in such a way that the fineness of the new parts is k. The split parts of I are kept in set H. Lastly, we unite those that do not contain significant nodes, and keep the original fineness of I for this united set.

REFINE-INDEX-NODE(I, k, *significant-nodes*)

```
 1  if k(I) ≥ k
 2    then return I(G)
 3  for all (J, I) ∈ E_I
 4      do significant-parents ← E^{-1}(significant-nodes) ∩ J
 5         if significant-parents ≠ ∅
 6           then REFINE-INDEX-NODE(J, k − 1, significant-parents)
 7  k-previous ← k(I)
 8  H ← {I}
 9  for all (J, I) ∈ E_I
10      do if E^{-1}(significant-parents) ∩ J ≠ ∅
11           then for all F ∈ H
12                    do V_I ← (V_I \ {F}) ∪ {F ∩ E(J), F \ E(J)}
13                       E_I ← {(A, B) | A, B ∈ V_I, ∃ a ∈ A, ∃ b ∈ B, (a, b) ∈ E}
14                       k(F ∩ E(J)) ← k
15                       k(F \ E(J)) ← k
16                       I(G) ← (V_I, E_I)
17                       H ← (H \ {F}) ∪ {F ∩ E(J), F \ E(J)}
18  remaining ← ∅
19  for all F ∈ H
20      do if significant-nodes ∩ F = ∅
21           then remaining ← remaining ∪ F
22                V_I ← (V_I \ {F})
23  V_I ← V_I ∪ {remaining}
24  E_I ← {(A, B) | A, B ∈ V_I, ∃ a ∈ A, ∃ b ∈ B, (a, b) ∈ E}
25  k(remaining) ← k-previous
26  I(G) ← (V_I, E_I)
27  return I(G)
```

The algorithm REFINE refines the index graph $I(G)$ according to a frequent simple expression in such a way that it splits an index node into not necessarily equally fine parts, and

thus avoids over-refinement. If we start from the A(0)-index, and create the refinement for all frequent queries, then we get an index graph that is exact with respect to frequent queries. This is called the M(k)-index. The set *FRE* of frequent queries might change during the process, so the index must be modified dynamically.

Definition 20.39 *The* ***M**(k)**-index** is a D(k)-index created using the following* M(k)-INDEX-CREATOR *algorithm.*

M(k)-INDEX-CREATOR(G, *FRE*)

```
1  I(G) ← the A(0) index belonging to graph G
2  V_I ← the nodes of I(G)
3  for all I ∈ V_I
4      do k(I) ← 0
5  E_I ← the set of edges of I(G)
6  for all R ∈ FRE
7      do I(G) ← REFINE(R, R(I(G)), R(G))
8  return I(G)
```

The M(k)-index is exact with respect to frequent queries. In case of a not frequent query, we can do the following. The M(k)-index is also a D(k)-index, therefore if an index node fits to a simple expression R in the index graph $I(G)$, and the fineness of the index node is at least the length of R, then all elements of the index node fit to the query R in graph G. If the fineness of the index node is less, then for all of its elements, we have to check according to NAIVE-EVALUATION whether it is a solution in graph G.

When using the M(k)-index, over-refinement is the least if the lengths of the frequent simple queries are the same. If there are big differences between the lengths of frequent queries, then the index we get might be too fine for the short queries. Create the sequence of gradually finer indexes with which we can get from the A(0)-index to the M(k)-index in such a way that, in every step, the fineness of parts obtained by splitting an index node is greater by at most one than that of the original index node. If the whole sequence of indexes is known, then we do not have to use the finest and therefore largest index for the evaluation of a simple query, but one whose fineness corresponds to the length of the query.

Definition 20.40 *The* ***M*****(k)**-index** *is a sequence of indexes I_0, I_1, ..., I_k such that*

1. *Index I_i is an M(i)-index, where $i = 0, 1, \ldots, k$.*
2. *The fineness of all index nodes in I_i is at most i, where $i = 0, 1, \ldots, k$.*
3. *I_{i+1} is a refinement of I_i, where $i = 0, 1, \ldots, k-1$.*
4. *If node J of index I_i is split in index I_{i+1}, and J' is a set obtained by this split, that is, $J' \subseteq J$, then $k(J) \leq k(J') \leq k(J) + 1$.*
5. *Let J be a node of index I_i, and $k(J) < i$. Then $k(J) = k(J')$ for $i < i'$ and for all J' index nodes of $I_{i'}$ such that $J' \subseteq J$.*

It follows from the definition that in case of M*(k)-indexes I_0 is the A(0)-index. The last property says that if the refinement of an index node stops, then its fineness will not

change anymore. The M*(k)-index possesses the good characteristics of the M(k)-index, and its structure is also similar: according to frequent queries the index is further refined if it is necessary to make it exact on frequent queries, but now we store and refresh the coarser indexes as well, not only the finest.

When representing the M*(k)-index, we can make use of the fact that if an index node is not split anymore, then we do not need to store this node in the new indexes, it is enough to refer to it. Similarly, edges between such nodes do not have to be stored in the sequence of indexes repeatedly, it is enough to refer to them. Creation of the M*(k)-index can be done similarly to the M(k)-INDEX-CREATOR algorithm. The detailed description of the algorithm can be found in the paper of He and Yang.

With the help of the M*(k)-index, we can use several strategies for the evaluation of queries. Let R be a frequent simple query.

The simplest strategy is to use the index the fineness of which is the same as the length of the query.

M*(k)-INDEX-NAIVE-EVALUATION(G, FRE, R)

1 $\{I_0, I_1, \ldots, I_k\} \leftarrow$ the M*(k)-index corresponding to graph G
2 $h \leftarrow$ length(R)
3 **return** INDEX-EVALUATION(G, I_h, A_R)

The evaluation can also be done by gradually evaluating the longer prefixes of the query according to the index the fineness of which is the same as the length of the prefix. For the evaluation of a prefix, consider the partitions of the nodes found during the evaluation of the previous prefix in the next index and from these, seek edges labeled with the following symbol. Let $R = l_0, l_1, \ldots, l_h$ be a simple frequent query, that is, $length(R) = h$.

M*(*k*)-INDEX-EVALUATION-TOP-TO-BOTTOM(*G*, *FRE*, *R*)

```
1  {I_0, I_1, ..., I_k} ← the M*(k)-index corresponding to graph G
2  R_0 ← l_0
3  H_0 ← ∅
4  for all C ∈ E_{I_0}(root(I_0))                ▷ The children of the root in graph I_0.
5      do if label(C) = l_0
6            then H_0 ← H_0 ∪ {C}
7  for j ← 1 to length(R)
8      do H_j ← ∅
9         R_j ← R_{j-1}.l_j
10        H_{j-1} ← M*(k)-INDEX-EVALUATION-TOP-TO-BOTTOM(G,FRE, R_{j-1})
11        for all A ∈ H_{j-1}                    ▷ Node A is a node of graph I_{j-1}.
12            do if A = ∪B_m, where B_m ∈ V_{I_j}   ▷ The partition of node A in graph I_j.
13                  then for minden B_m
14                          do for all C ∈ E_{I_j}(B_m)   ▷ For all children of
                                                          B_m in graph I_j.
15                                do if label(C) = l_j
16                                      then H_j ← H_j ∪ {C}
17 return H_h
```

Our strategy could also be that we first find a subsequence of the label sequence corresponding to the simple query that contains few nodes, that is, its selectivity is large. Then find the fitting nodes in the index corresponding to the length of the subsequence, and using the sequence of indexes see how these nodes are split into new nodes in the finer index corresponding to the length of the query. Finally, starting from these nodes, find the nodes that fit to the remaining part of the original query. The detailed form of the algorithm M*(*k*)-INDEX-PREFILTERED-EVALUATION is left to the Reader.

Exercises

20.6-1 Find the detailed form of the algorithm M*(*k*)-INDEX-PREFILTERED-EVALUATION . What is the cost of the algorithm?

20.6-2 Prove Proposition 20.38.

20.6-3 Prove that the computation cost of the algorithm WEIGHT-CHANGER is $O(m)$, where m is the number of edges of the A(0)-index.

20.7. Branching queries

With the help of regular queries we can select the nodes of a graph that are reached from the root by a path the labels of which fit to a given regular pattern. A natural generalization is to add more conditions that the nodes of the path leading to the node have to satisfy. For example, we might require that the node can be reached by a label sequence from a node with a given label. Or, that a node with a given label can be reached from another node by a path with a given label sequence. We can take more of these conditions, or use their negation or composition. To check whether the required conditions hold, we have to step not

only forward according to the direction of the edges, but sometimes also backward. In the following, we will give the description of the language of branching queries, and introduce the forward-backward indexes. The forward-backward index which is safe and exact with respect to all branching queries is called FB-index. Just like the 1-index, this is also usually too large, therefore we often use an FB(f, b, d)-index instead, which is exact if the length of successive forward steps is at most f, the length of successive backward steps is at most b, and the depth of the composition of conditions is at most d. In practice, values f, b and d are usually small. In case of queries for which the value of one of these parameters is greater than the corresponding value of the index, a checking step must be added, that is, we evaluate the query on the index, and only keep those nodes of the resulted index nodes that satisfy the query.

If there is a directed edge from node n to node m, then this can be denoted by n/m or $m\backslash n$. If node m can be reached from node n by a directed path, then we can denote that by $n//m$ or $m\backslash\backslash n$. (Until now we used . instead of /, so // represents the regular expression _* or * in short.)

From now on, a label sequence is a sequence in which separators are the forward signs (/, //) and the backward signs (\, \\). A sequence of nodes fit to a label sequence if the relation of successive nodes is determined by the corresponding separator, and the labels of the nodes come according to the label sequence.

There are only forward signs in ***forward label sequences***, and only backward signs in ***backward label sequences***.

Branching queries are defined by the following grammar .

branching_query	::=	forward_label sequence [or_expression] forward_sign branching_expression
		| forward_label_sequence [or_expression]
		| forward_label_sequence
or_expression	::=	and_expression or or_expression
		| and_expressnion
and_expression	::=	branching_condition and and_expression
		| not_branching_condition and and_expression
		| branching_condition
		| not_branching_condition
not_branching_condition	::=	not branching_condition
branching_condition	::=	condition_label_sequence [or_expression] branching_condition
		| condition_label_sequence [or_expression]
		| condition_label_sequence
condition_label_sequence	::=	forward_sign label_sequence
		| backward_sign label_sequence

In branching queries, a condition on a node with a given label holds if there exists a label sequence that fits to the condition. For example, the *root*//*a*/*b*[*c*//*d* and not *e*/*f*]/*g* query seeks nodes with label g such that the node can be reached from the root in such a way that the labels of the last two nodes are a and b, furthermore, there exists a parent of the node with label b whose label is c, and among the descendants of the node with label c there is one with label d, but it has no children with label e that has a parent with label f.

If we omit all conditions written between signs [ ] from a branching query, then we get the ***main query*** corresponding to the branching query. In our previous example, this is the

query *root//a/b/g*. The main query always corresponds to a forward label sequence.

A directed graph can be assigned naturally to branching queries. Assign nodes with the same label to the label sequence of the query, in case of separators / and \, connect the successive nodes with a directed edge according to the separator, and in case of separators // and \\, draw the directed edge and label it with label // or \\. Finally, the logic connectives are assigned to the starting edge of the corresponding condition as a label. Thus, it might happen that an edge has two labels, for example // and „and". Note that the graph obtained cannot contain a directed cycle because of the definition of the grammar.

A simple degree of complexity of the query can be defined using the tree obtained. Assign 0 to the nodes of the main query and to the nodes from which there is a directed path to a node of the main query. Then assign 1 to the nodes that can be reached from the nodes with sign 0 on a directed path and have no sign yet. Assign 2 to the nodes from which a node with sign 1 can be reached and have no sign yet. Assign 3 to the nodes that can be reached from nodes with sign 2 and have no sign yet, etc. Assign $2k + 1$ to the nodes that can be reached from nodes with sign $2k$ and have no sign yet, then assign $2k + 2$ to the nodes from which nodes with sign $2k + 1$ can be reached and have no sign yet. The value of the greatest sign in the query is called the ***depth of the tree***. The depth of the tree shows how many times the direction changes during the evaluation of the query, that is, we have to seek children or parents according to the direction of the edges. The same query could have been given in different ways by composing the conditions differently, but it can be proved that the value defined above does not depend on that, that is why the complexity of a query was not defined as the number of conditions composed.

The 1-index assigns the nodes into classes according to incoming paths, using bisimulations. The concept of stability used for computations was ***descendant-stability***. A set A of the nodes of a graph is ***descendant-stable*** with respect to a set B of nodes if $A \subseteq E(B)$ or $A \cap E(B) = \emptyset$, where $E(B)$ is the set of nodes that can be reached by edges from B. A partition is stable if any two elements of the partition are descendant-stable with respect to each other. The 1-index is the coarsest descendant-stable partition that assigns nodes with the same label to same classes, which can be computed using the PT-algorithm. In case of branching queries, we also have to go backwards on directed edges, so we will need the concept of ***ancestor-stability*** as well. A set A of nodes of a graph is ***ancestor-stable*** with respect to a set B of the nodes if $A \subseteq E^{-1}(B)$ or $A \cap E^{-1}(B) = \emptyset$, where $E^{-1}(B)$ denotes the nodes from which a node of B can be reached.

Definition 20.41 *The **FB-index** is the coarsest refinement of the basic partition that is ancestor-stable and descendant-stable.*

Note that if the direction of the edges of the graph is reversed, then an ancestor-stable partition becomes a descendant-stable partition and vice versa, therefore the PT-algorithm and its improvements can be used to compute the coarsest ancestor-stable partition. We will use this in the following algorithm. We start with classes of nodes with the same label, compute the 1-index corresponding to this partition, then reverse the direction of the edges, and refine this by computing the 1-index corresponding to this. When the algorithm stops, we get a refinement of the initial partition that is ancestor-stable and descendant-stable at the same time. This way we obtain the coarsest such partition. The proof of this is left to the Reader.

FB-INDEX-CREATOR(V, E)

```
1  P ← A(0)                                  ▷ Start with classes of nodes with the same label.
2  while P changes
3        do P ← PT(V, E^{-1}, P)             ▷ Compute the 1-index.
4           P ← PT(V, E, P)                  ▷ Reverse the direction of edges, and
                                             ▷ compute the 1-index.
5  return P
```

The following corollary follows simply from the two stabilities.

Corollary 20.42 *The FB-index is safe and exact with respect to branching queries.*

The complexity of the algorithm can be computed from the complexity of the PT-algorithm. Since P is always the refinement of the previous partition, in the worst case refinement is done one by one, that is, we always take one element of a class and create a new class consisting of that element. So in the worst case, the cycle is repeated $O(n)$ times. Therefore, the cost of the algorithm is at most $O(mn \lg n)$.

The partition gained by executing the cycle only once is called the ***F+B-index***, the partition obtained by repeating the cycle twice is the ***F+B+F+B-index***, etc.

The following proposition can be proved easily.

Proposition 20.43 *The F+B+F+B+$\cdots$+F+B-index, where F+B appears d times, is safe and exact with respect to the branching queries of depth at most d.*

Nodes of the same class according to the FB-index cannot be distinguished by branching queries. This restriction is usually too strong, therefore the size of the FB-index is usually much smaller than the size of the original graph. Very long branching queries are seldom used in practice, so we only require local equivalence, similarly to the A(k)-index, but now we will describe it with two parameters depending on what we want to restrict: the length of the directed paths or the length of the paths with reversed direction. We can also restrict the depth of the query. We can introduce the FB(f, b, d)-index, with which such restricted branching queries can be evaluated exactly. We can also evaluate branching queries that do not satisfy the restrictions, but then the result must be checked.

FB(f, b, d)-INDEX-CREATOR(V, E, f, b, d)

```
1  P ← A(0)                                  ▷ start with classes of nodes with the same label.
2  for i ← 1 to d
3     do P ← NAIVE-APPROXIMATION(V, E^{-1}, P, f)   ▷ Compute the A(f)-index.
4        P ← NAIVE-APPROXIMATION(V, E, P, b)  ▷ Reverse the direction of the edges, and
                                             ▷ compute the A(b)-index.
5  return P
```

The cost of the algorithm, based on the computation cost of the A(k)-index, is at most $O(dm \max(f, b))$, which is much better than the computation cost of the FB-index, and the index graph obtained is also usually much smaller.

The following proposition obviously holds for the index obtained.

Proposition 20.44 *The FB(f, b, d)-index is safe and exact for the branching queries in which the length of forward-sequences is at most f, the length of backward-sequences is at most b, and the depth of the tree corresponding to the query is at most d.*

As a special case we get that the FB(∞, ∞, ∞)-index is the FB-index, the FB(∞, ∞, d)-index is the F+B+$\cdots$+F+B-index, where F+B appears d times, the FB($\infty, 0, 1$)-index is the 1-index, and the FB($k, 0, 1$)-index is the A(k)-index.

Exercises

20.7-1 Prove that the algorithm FB-Index-Creator produces the coarsest ancestor-stable and descendant-stable refinement of the basic partition.

20.7-2 Prove Proposition 20.44.

20.8. Index refresh

In database management we usually have three important aspects in mind. We want space requirement to be as small as possible, queries to be as fast as possible, and insertion, deletion and modification of the database to be as quick as possible. Generally, a result that is good with respect to one of these aspects is worse with respect to another aspect. By adding indexes of typical queries to the database, space requirement increases, but in return we can evaluate queries on indexes which makes them faster. In case of dynamic databases that are often modified we have to keep in mind that not only the original data but also the index has to be modified accordingly. The most costly method which is trivially exact is that we create the index again after every modification to the database. It is worth seeking procedures to get the modified indexes by smaller modifications to those indexes we already have.

Sometimes we index the index or its modification as well. The index of an index is also an index of the original graph, although formally it consists of classes of index nodes, but we can unite the elements of the index nodes belonging to the same class. It is easy to see that by that we get a partition of the nodes of the graph, that is, an index.

In the following, we will discuss those modifications of semi-structured databases when a new graph is attached to the root and when a new edges is added to the graph, since these are the ones we need when creating a new website or a new reference.

Suppose that $I(G)$ is the 1-index of graph G. Let H be a graph that has no common node with G. Denote by $I(H)$ the 1-index of H. Let $F = G + H$ be the graph obtained by uniting the roots of G and H. We want to create $I(G+H)$ using $I(G)$ and $I(H)$. The following proposition will help us.

Proposition 20.45 *Let $I(G)$ be the 1-index of graph G, and let J be an arbitrary refinement of $I(G)$. Then $I(J) = I(G)$.*

Proof. Let u and v be two nodes of G. We have to show that u and v are bisimilar in G with respect to the 1-index if and only if $J(u)$ and $J(v)$ are bisimilar in the index graph $I(G)$ with respect to the 1-index of $I(G)$. Let u and v be bisimilar in G with respect to the 1-index. We will prove that there is a bisimulation according to which $J(u)$ and $J(v)$ are bisimilar in $I(G)$. Since the 1-index is the partition corresponding to the coarsest bisimulation, the given bisimulation is a refinement of the bisimulation corresponding to the 1-index, so $J(u)$

and $J(v)$ are also bisimilar with respect to the bisimulation corresponding to the 1-index of $I(G)$. Let $J(a) \approx' J(b)$ if and only if a and b are bisimilar in G with respect to the 1-index. Note that since J is a refinement of $I(G)$, all elements of $J(a)$ and $J(b)$ are bisimilar in G if $J(a) \approx' J(b)$. To show that the relation $\approx'$ is a bisimulation, let $J(u')$ be a parent of $J(u)$, where u' is a parent of u_1, and u_1 is an element of $J(u)$. Then u_1, u and v are bisimilar in G, so there is a parent v' of v for which u' and v' are bisimilar in G. Therefore $J(v')$ is a parent of $J(v)$, and $J(u') \approx' J(v')$. Since bisimulation is symmetric, relation $\approx'$ is also symmetric. We have proved the first part of the proposition.

Let $J(u)$ and $J(v)$ be bisimilar in $I(G)$ with respect to the 1-index of $I(G)$. It is sufficient to show that there is a bisimulation on the nodes of G according to which u and v are bisimilar. Let $a \approx' b$ if and only if $J(a) \approx J(b)$ with respect to the 1-index of $I(G)$. To prove bisimilarity, let u' be a parent of U. Then $J(u')$ is also a parent of $J(u)$. Since $J(u)$ and $J(v)$ are bisimilar if $u \approx' v$, there is a parent $J(v'')$ of $J(v)$ for which $J(u')$ and $J(v'')$ are bisimilar with respect to the 1-index of $I(G)$, and v'' is a parent of an element v_1 of $J(v)$. Since v and v_1 are bisimilar, there is a parent v' of v such that v' and v'' are bisimilar. Using the first part of the proof, it follows that $J(v')$ and $J(v'')$ are bisimilar with respect to the 1-index of $I(G)$. Since bisimilarity is transitive, $J(u')$ and $J(v')$ are bisimilar with respect to the 1-index of $I(G)$, so $u' \approx' v'$. Since relation $\approx'$ is symmetric by definition, we get a bisimulation. ■

As a consequence of this proposition, $I(G + H)$ can be created with the following algorithm for disjoint G and H.

GRAPHADDITION-1-INDEX(G, H)

1 $P_G \leftarrow A_G(0)$ ▷ P_G is the basic partition according to labels.
2 $P_H \leftarrow A_H(0)$ ▷ P_H is the basic partition according to labels.
3 $I_1 \leftarrow PT(V_G, E_G^{-1}, P_G)$ ▷ I_1 is the 1-index of G.
4 $I_2 \leftarrow PT(V_H, E_H^{-1}, P_H)$ ▷ I_2 is the 1-index of H.
5 $J \leftarrow I_1 + I_2$ ▷ The 1-indexes are joined at the roots.
6 $P_J \leftarrow A_J(0)$ ▷ P_J is the basic partition according to labels.
7 $I \leftarrow PT(V_J, E_J^{-1}, PJ)$ ▷ I is the 1-index of J.
8 **return** I

To compute the cost of the algorithm, suppose that the 1-index $I(G)$ of G is given. Then the cost of the creation of $I(G + H)$ is $O(m_H \lg n_H + (m_{I(H)} + m_{I(G)}) \lg(n_{I(G)} + n_{I(H)}))$, where n and m denote the number of nodes and edges of the graph, respectively.

To prove that the algorithm works, we only have to notice that $I(G)+I(H)$ is a refinement of $I(G+H)$ if G and H are disjoint. This also implies that index $I(G)+I(H)$ is safe and exact, so we can use this as well if we do not want to find the minimal index. This is especially useful if new graphs are added to our graph many times. In this case we use the ***lazy method***, that is, instead of computing the minimal index for every pair, we simply sum the indexes of the addends and then minimize only once.

Proposition 20.46 *Let $I(G_i)$ be the 1-index of graph G_i, $i = 1, \ldots, k$, and let the graphs be disjoint. Then $I(G_1 + \cdots + G_k) = I(I(G_1) + \cdots + I(G_k))$ for the 1-index $I(G_1 + \cdots + G_k)$ of the union of the graphs joined at the roots.*

In the following we will examine what happens to the index if a new edge is added to the

graph. Even an operation like this can have significant effects. It is not difficult to construct a graph that contains two identical subgraphs at a distant of 2 from the root which cannot be contracted because of a missing edge. If we add this critical edge to the graph, then the two subgraphs can be contracted, and therefore the size of the index graph decreases to about the half of its original size.

Suppose we added a new edge to graph G from u to v. Denote the new graph by G', that is, $G' = G + (u, v)$. Let partition $I(G)$ be the 1-index of G. If there was an edge from $I(u)$ to $I(v)$ in $I(G)$, then the index graph does not have to be modified, since there is a parent of the elements of $I(v)$, that is, of all elements bisimilar to v, in $I(u)$ whose elements are bisimilar to u. Therefore $I(G') = I(G)$.

If there was no edge from $I(u)$ to $I(v)$, then we have to add this edge, but this might cause that $I(v)$ will no longer be stable with respect to $I(u)$. Let Q be the partition we get from $I(G)$ by splitting $I(v)$ in such a way that v is in one part and the other elements of $I(v)$ are in the other, and leaving all other classes of the partition unchanged. Q defines its edges the usual way, that is, if there is an edge from an element of a class to an element of another class, then we connect the two classes with an edge directed the same way.

Let partition X be the original $I(G)$. Then Q is a refinement of X, and Q is stable with respect to X according to G'. Note that the same invariant property appeared in the PT-algorithm for partitions X and Q. Using Proposition 20.45 it is enough to find a refinement of $I(G')$. If we can find an arbitrary stable refinement of the basic partition of G', then, since the 1-index is the coarsest stable partition, this will be a refinement of $I(G')$. X is a refinement of the basic partition, that is, the partition according to labels, and so is Q. So if Q is stable, then we are done. If it is not, then we can stabilize it using the PT-algorithm by starting with the above partitions X and Q. First we have to examine those classes of the partition that contain a children of v, because these might lost their stability with respect to the two new classes gained by the split. The PT-algorithm stabilizes these by splitting them, but because of this we now have to check their children, since they might have lost stability because of the split, etc. We can obtain a stable refinement using this stability-propagator method. Since we only walk through the nodes that can be reached from v, this might not be the coarsest stable refinement. We have shown that the following algorithm computes the 1-index of the graph $G + (u, v)$.

EDGEADDITION-1-INDEX$(G, (u, v))$

1 $P_G \leftarrow A_G(0)$ ▷ P_G is the basic partition according to labels.
2 $I \leftarrow PT(V_G, E_G^{-1}, P_G)$ ▷ I is the 1-index of G.
3 $G' \leftarrow G + (u, v)$ ▷ Add edge (u, v).
4 **if** $(I(u), I(v)) \in E_I$ ▷ If there was an edge from $I(u)$ to $I(v)$, then no modification is needed.
5 **then return** I
6 $I' \leftarrow \{v\}$ ▷ Split $I(v)$.
7 $I'' \leftarrow I(v) \setminus \{v\}$
8 $X \leftarrow I$ ▷ X is the old partition.
9 $E_I \leftarrow E_I \cup \{(I(u), I(v))\}$ ▷ Add an edge from $I(u)$ to $I(v)$.
10 $Q \leftarrow (I \setminus \{I(v)\}) \cup \{I', I''\}$ ▷ Replace $I(v)$ with I' and I''.
11 $E \leftarrow E_Q$ ▷ Determine the edges of Q.
12 $J \leftarrow PT(V_{G'}, E_{G'}^{-1}, P_{G'}, X, Q)$ ▷ Execute the PT-algorithm starting with X and Q.
13 $J \leftarrow PT(V_J, E_J^{-1}, P_J)$ ▷ J is the coarsest stable refinement.
14 **return** J

Step 13 can be omitted in practice, since the stable refinement obtained in step 12 is a good enough approximation of the coarsest stable partition, there is only 5% difference between them in size.

In the following we will discuss how FB-indexes and A(k)-indexes can be refreshed. The difference between FB-indexes and 1-indexes is that in the FB-index, two nodes are in the same similarity class if not only the incoming but also the outgoing paths have the same label sequences. We saw that in order to create the FB-index we have to execute the PT-algorithm twice, using it on the graph with the edges reversed at the second time. The FB-index can be refreshed similarly to the 1-index. The following proposition can be proved similarly to Proposition 20.45, therefore we leave it to the Reader.

Proposition 20.47 *Let $I(G)$ be the FB-index of graph G, and let J be an arbitrary refinement of $I(G)$. Denote by $I(J)$ the FB-index of J. Then $I(J) = I(G)$.*

As a consequence of the above proposition, the FB-index of $G + H$ can be created using the following algorithm for disjoint G and H.

GRAPHADDITION-FB-INDEX(G, H)

1 $I_1 \leftarrow$ FB-INDEX-CREATOR(V_G, E_G) ▷ I_1 is the FB-index of G.
2 $I_2 \leftarrow$ FB-INDEX-CREATOR(V_H, E_H) ▷ I_2 is the FB-index of H.
3 $J \leftarrow I_1 + I_2$ ▷ Join the FB-indexes at their roots.
4 $I \leftarrow$ FB-INDEX-CREATOR(V_J, E_J) ▷ I is the FB-index of J.
5 **return** I

When adding edge (u, v), we must keep in mind that stability can be lost in both directions, so not only $I(v)$ but also $I(u)$ has to be split into $\{v\}$, $(I \setminus \{v\})$ and $\{u\}$, $(I(u) \setminus \{u\})$, respectively. Let X be the partition before the modification, and Q the partition obtained

after the splits. We start the PT-algorithm with X and Q in step 3 of the algorithm FB-INDEX-CREATOR. When stabilizing, we will now walk through all descendants of v and all ancestors of u.

EDGEADDITION-FB-INDEX(G, (u, v))

```
1  I ← FB-INDEX-CREATOR(V_G, E_G)                ▷ I is the FB-index of G.
2  G' ← G + (u, v)                               ▷ Add edge (u, v).
3  if (I(u), I(v)) ∈ E_I                         ▷ If there was an edge from I(u) to I(v),
                                                   then no modification is needed.
4     then return I
5  I_1 ← {v}                                     ▷ Split I(v).
6  I_2 ← I(v) \ {v}
7  I_3 ← {u}                                     ▷ Split I(u).
8  I_4 ← I(u) \ {u}
9  X ← I                                         ▷ X is the old partition.
10 E_I ← E_I ∪ {(I(u), I(v))}                    ▷ Add an edge form I(u) to I(v).
11 Q ← (I \ {I(v), I(u)}) ∪ {I_1, I_2, I_3, I_4} ▷ Replace I(v) with I_1 and I_2,
                                                   I(u) with I_3 and I_4.
12 E ← E_Q                                       ▷ Determine the edges of Q.
13 J ← FB-INDEX-CREATOR(V_G', E_G', X, Q)        ▷ Start the PT-algorithm with X and Q
                                                   in the algorithm FB-INDEX-CREATOR.
14 J ← FB-INDEX-CREATOR(V_J, E_J)                ▷ J is the coarsest ancestor-stable
                                                   and descendant-stable refinement.
15 return J
```

Refreshing the A(k)-index after adding an edge is different than what we have seen. There is no problem with adding a graph though, since the following proposition holds, the proof of which is left to the Reader.

Proposition 20.48 *Let $I(G)$ be the A(k)-index of graph G, and let J be an arbitrary refinement of $I(G)$. Denote by $I(J)$ the A(k)-index of $I(J)$. Then $I(J) = I(G)$.*

As a consequence of the above proposition, the A(k)-index of $G + H$ can be created using the following algorithm for disjoint G and H.

GRAPHADDITION-A(k)-INDEX(G, H)

```
1  P_G ← A_G(0)                                  ▷ P_G is the basic partition according to labels.
2  I_1 ← NAIVE-APPROXIMATION(V_G, E_G^-1, P_G, k)  ▷ I_1 is the A(k)-index of G.
3  P_H ← A_H(0)                                  ▷ P_H is the basic partition according to labels.
4  I_2 ← NAIVE-APPROXIMATION(V_H, E_H^-1, P_H, k)  ▷ I_1 is the A(k)-index of H.
5  J ← I_1 + I_2                                 ▷ Join the A(k)-indexes.
6  P_J ← A_J(0)                                  ▷ P_J is the basic partition according to labels.
7  I ← NAIVE-APPROXIMATION(V_J, E_J^-1, P_J, k)  ▷ I is the A(k)-index of J.
8  return I
```

If we add a new edge (u, v) to the graph, then, as earlier, first we split $I(v)$ into two parts, one of which is $I' = \{v\}$, then we have to repair the lost k-stabilities walking through the descendants of v, but only within a distant of k. What causes the problem is that the A(k)-index contains information only about k-bisimilarity, it tells us nothing about $(k-1)$-bisimilarity. For example, let v_1 be a child of v, and let $k = 1$. When stabilizing according to the 1-index, v_1 has to be detached from its class if there is an element in this class that is not a children of v. This condition is too strong in case of the A(1)-index, and therefore it causes too many unnecessary splits. In this case, v_1 should only be detached if there is an element in its class that has no 0-bisimilar parent, that is, that has the same label as v. Because of this, if we refreshed the A(k)-index the above way when adding a new edge, we would get a very bad approximation of the A(k)-index belonging to the modification, so we use a different method. The main idea is to store all A(i)-indexes not only the A(k)-index, where $i = 1, \ldots, k$. Yi et al. give an algorithm based on this idea, and creates the A(k)-index belonging to the modification. The given algorithms can also be used for the deletion of edges with minor modifications, in case of 1-indexes and A(k)-indexes.

Exercises

20.8-1 Prove Proposition 20.47.

20.8-2 Give an algorithm for the modification of the index when an edge is deleted from the data graph. Examine different indexes. What is the cost of the algorithm?

20.8-3 Give algorithms for the modification of the D(k)-index when the data graph is modified.

Problems

20-1. Implication problem regarding constraints

Let R and Q be regular expressions, x and y two nodes. Let predicate $R(x, y)$ mean that y can be reached from x by a label sequence that fits to R. Denote by $R \subseteq Q$ the constraint $\forall x(R(\text{root}, x) \to Q(\text{root}, x))$. $R = Q$ if $R \subseteq Q$ and $Q \subseteq R$. Let C be a finite set of constraints, and c a constraint.

a. Prove that the implication problem $C \models c$ is a 2-EXPSPACE problem.

b. Denote by $R \subseteq Q@u$ the constraint $\forall v(R(u, v) \to Q(u, v))$. Prove that the implication problem is undecidable with respect to this class.

20-2. Transformational distance of trees

Let the *transformational distance* of vertex-labeled trees be the minimal number of basic operations with which a tree can be transformed to the other. We can use three basic operations: addition of a new node, deletion of a node, and renaming of a label.

a. Prove that the transformational distance of trees T and T' can be computed in $O(n_T n_{T'} d_T d_{T'})$ time, with storage cost of $O(n_T n_{T'})$, where n_T is the number of nodes of the tree and d_T is the depth of the tree.

b. Let S and S' be two trees. Give an algorithm that generates all pairs (T, T'), where T and T' simulates graphs S and S', respectively, and the transformational distance of T and T' is less then a given integer n. (This operation is called *approximate join*.)

20-3. Queries of distributed databases
A distributed database is a vertex-labeled directed graph the nodes of which are distributed in m partitions (servers). The edges between different partitions are *cross references*. Communication is by message broadcasting between the servers. An algorithm that evaluates a query is *efficient*, if the number of communication steps is constant, that is, it does not depend on the data and the query, and the size of the data transmitted during communication only depends on the size of the result of the query and the number of cross references. Prove that an efficient algorithm can be given for the regular query of distributed databases in which the number of communication steps is 4, and the size of data transmitted is $O(n^2) + O(k)$, where n is the size of the result of the query, and k is the number of cross references. (*Hint:* Try to modify the algorithm NAIVE-EVALUATION for this purpose.)

Chapter notes

This chapter examined those fields of the world of semi-structured databases where the morphisms of graphs could be used. Thus we discussed the creation of schemas and indexes from the algorithmic point of view. The world of semi-structured databases and XML is much broader than that. A short summary of the development, current issues and the possible future development of semi-structured databases can be found in the paper of Vianu [115].

The paper of M. Henzinger, T. Henzinger and Kopke [64] discusses the computation of the maximal simulation. They extend the concept of simulation to infinite graphs that can be represented efficiently (these are called effective graphs), and prove that for such graphs, it can be determined whether two nodes are similar. In their paper, Corneil and Gotlieb [34] deal with quotient graphs and the determination of isomorphism of graphs. Arenas and Libkin [4] extend normal forms used in the relational model to XML documents. They show that arbitrary DTD can be rewritten without loss as XNF, a normal form they introduced.

Buneman, Fernandez and Suciu [21] introduce a query language, the UnQL, based on structural recursion, where the data model used is defined by bisimulation. Gottlob, Koch and Pichler [57] examine the classes of the query language XPath with respect to complexity and parallelization. For an overview of complexity problems we recommend the classical work of Garey and Johnson [55] and the paper of Stockmeyer and Meyer [103].

The PT-algorithm was first published in the paper of Paige and Tarjan [89]. The 1-index based on bisimulations is discussed in detail by Milo and Suciu [85], where they also introduce the 2-index, and as a generalization of this, the T-index

The A(k)-index was introduced by Kaushik, Shenoy, Bohannon and Gudes [69]. The D(k)-index first appeared in the work of Chen, Lim and Ong [27]. The M(k)-index and the M*(k)-index, based on frequent queries, are the results of He and Yang [63]. FB-indexes of branching queries were first examined by Kaushik, Bohannon, Naughton and Korth [71].

The algorithms of the modifications of 1-indexes, FB-indexes and A(k)-indexes were summarized by Kaushik, Bohannon, Naughton and Shenoy [72]. The methods discussed here are improved and generalized in the work of Yi, He, Stanoi and Yang [126]. Polyzotis and Garafalakis use a probability model for the study of the selectivity of queries [92].

Kaushik, Krishnamurthy, Naughton and Ramakrishnan [70] suggest the combined use of structural indexes and inverted lists.

The book of Tucker [113] and the encyclopedia edited by Khosrow-Pour [73] deal with the use of XML in practice.

The theory of XML has no literature in Hungarian yet, but several books discuss its practical use [8, 19, 84].

Bibliography

[1] S. Abiteboul, V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995. 730

[2] R. Ahlswede, B. Balkenhol, L. Khachatrian. Some properties of fix-free codes. In *Proceedings of the 1st International Seminarium on Coding Theory and Combinatorics*, 1996, pp. 20–23. 618

[3] A. Aho, C. Beeri, J. D. Ullman. The theory of joins in relational databases. *ACM Transactions on Database Systems*, 4(3):297–314, 1979. 730

[4] M. Arenas, L. Libkin. A normal form for XML documents. In *Proceedings of the 21st Symposium on Principles of Database Systems*, 2002, pp. 85–96. 819

[5] W. Armstrong. Dependency structures of database relationships. In *Proceedings of IFIP Congress*, 580–583. o. North Holland, 1974. 730

[6] B. Balkenhol, S. Kurtz. Universal data compression based on the Burrows-Wheeler transform: theory and practice. *IEEE Transactions on Computers*, 49(10):1043–1053–953, 2000. 619

[7] R. Barett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozzo, C. Romine, H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994. 780

[8] C. F. Bates. *XML in Theory and Praxis*. John Wiley & Sons, 2003. 820

[9] S. Batterson. Convergence of the shifted QR algorithm on 3×3 normal matrices. , 58, 1990. 780

[10] C. Beeri. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Transactions on Database Systems*, 5:241–259, 1980. 730

[11] C. Beeri, P. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems*, 4(1):30–59, 1979. 730

[12] C. Beeri, M. Dowd. On the structure of armstrong relations for functional dependencies. *Journal of ACM*, 31(1):30–46, 1984. 730

[13] C. Beeri, R. Fagin, J. Howard. A complete axiomatization for functional and multivalued dependencies. In *ACM SIGMOD Symposium on the Management of Data*, 47–61. o., 1977. 730

[14] T. C. Bell, I. H. Witten, J. G. Cleary. Modeling for text compression. *Communications of the ACM*, 21:557–591, 1989. 619

[15] T. C. Bell, I. H. Witten, J. G. Cleary. *Text Compression*. Prentice Hall, 1990. 619

[16] A. Békéssy, J. Demetrovics. Contribution to the theory of data base relations. *Discrete Mathematics*, 27(1):1–10, 1979. 730

[17] J. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):135–256, 1982. 696

[18] J. Bloomenthal. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, 1997. 696

[19] N. Bradley. *The XML Companion (3. edition)*. Addison-Wesley, 2004. 820

[20] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965. 697

[21] P. Buneman, M. Fernandez, D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The International Journal on Very Large Data Bases*, 9(1):76–110, 2000. 819

[22] M. Burrows, D. J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html, 1994. 619

[23] Calgary. The Calgary/Canterbury Text Compression. ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus, 2004. 619

[24] Canterbury. The Canterbury Corpus. http://corpus.canterbury.ac.nz, 2004. 619

[25] E. Catmull, J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10:350–355, 1978. 697

[26] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6(5):353–363, 1991. 696

[27] Q. Chen, A. Lim, K. W. Ong. An adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, pp. 134–144. 819

[28] E. F. Codd. A relational model of large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. 730

[29] E. F. Codd. Recent investigations in relational data base systems. In *Information Processing 74*. North-Holland, pp. 1017–1021,1974. 730

[30] E. F. Codd. Normalized database structure: A brief tutorial. In *ACM SIGFIDET Workshop on Data Description, Access and Control*, pp. 24–30, 1971. 730

[31] E. F. Codd. Further normalization of the data base relational model. In R. Rustin (szerkesztő), *Courant Computer Science Symposium 6: Data Base Systems*. Prentice Hall, pp. 33–64, 1972. 730

[32] E. F. Codd. Relational completeness of database sublanguages. In *Courant Computer Science Symposium 6: Data Base Systems*. Prentice Hall, pp. 65–98, 1972, editor =. 730

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 2004 (Fifth corrected printing of 2. edition. 618

[34] D. G. Corneil, C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17(1):51–64, 1970. 819

[35] T. M. Cover, J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991. 619

[36] H. S. M. Coxeter. *Projective Geometry*. University of Toronto Press, 1974 (2. edition). 696

[37] M. de Berg. *Efficient Algorithms for Ray Shooting and Hidden Surface Removal*. PhD thesis, Rijksuniversiteit te Utrecht, 1992. 696, 697

[38] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000. 696

[39] C. Delobel. Normalization and hierarchical dependencies in the relational data model. *ACM Transactions on Database Systems*, 3(3):201–222, 1978. 730

[40] J. Demetrovics, Gy. O. H. Katona, A. Sali. Minimal representations of branching dependencies. *Discrete Applied Mathematics*, 40:139–153, 1992. 730

[41] J. Demetrovics, Gy. O. H. Katona, A. Sali. Minimal representations of branching dependencies. *Acta Scientiarum Mathematicorum (Szeged)*, 60:213–223, 1995. 730

[42] J. Demetrovics, Gy. O. H. Katona, A. Sali. Design type problems motivated by database theory. *Journal of Statistical Planning and Inference*, 72:149–164, 1998. 730

[43] J. Demmel, D. Malajovich. On the complexity of computing error bounds. *Foundations of Computational Mathematics*, 1:101–125, 2001. 779

[44] N. Dyn, J. Gregory, D. Levin. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics*, 9:160–169, 1990. 697

[45] M. Effros, K. Viswesvariah, S. Kulkarni, S. Verdú. Universal lossless source coding with the Burrows-Wheeler transform. *IEEE Transactions on Information Theory*, 48(5):1061–1081, 2002. 619

[46] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems*, 2:262–278, 1977. 730

[47] R. Fagin. Armstrong databases. In *Proceedings of IBM Symposium on Mathematical Foundations of Computer Science*, 1982. 730

[48] R. Fagin. Horn clauses and database dependencies. *Journal of ACM*, 29(4):952–985, 1982. 730

[49] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Morgan Kaufmann Publishers, 1998. 696

[50] R. Fernando. *GPUGems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley, 2004. 697

[51] J. D. Fooley, A., S. K. Feiner, J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990. 697

[52] A. Frommer. *Lösung linearer Gleichungssysteme auf Parallelrechnern*. Vieweg Verlag, 1990. 780

[53] H. Fuchs, Z. M. Kedem, B. F. Naylor. On visible surface generation by a priory tree structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, pp. 124–133, 1980. 697

[54] A. Fujimoto, T. Takayuki, I. Kansey. ARTS: accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986. 697

[55] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 819

[56] A. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989. 697

[57] G. Gottlob, C. Koch, R. Pichler. The complexity of XPath query evaluation. In *Proceedings of the 22nd Symposium on Principles of Database Systems*, 2003, pp. 179–190. 819

[58] J. Grant, J. Minker. Inferences for numerical dependencies. *Theoretical Computer Science*, 41:271–287, 1985. 730

[59] J. Grant, J. Minker. Normalization and axiomatization for numerical dependencies. *Information and Control*, 65:1–17, 1985. 730

[60] T. S. Han, K. Kobayashi. *Mathematics of Information and Coding*. American Mathematical Society, 2002. 619

[61] D. Hankerson, G. A. Harris, P. D. Johnson. *Introduction to Information Theory and Data Compression*. CRC Press, 2003 (2. edition). 619

[62] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, 2001. 697

[63] H. He, J. Yang. Multiresolution indexing of XML for frequent queries. In *Proceedings of the 20th International Conference on Data Engineering*, 2004, pp. 683–694. 819

[64] M. R. Henzinger, T. A. Henzinger, P. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1995, pp. 453–462. 819

[65] I. Herman. *The Use of Projective Geometry in Computer Graphics*. Springer-Verlag, 1991. 696

[66] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. 618

[67] P. Jiménez, F. Thomas, C. Torras. 3D collision detection: A survey. *Computers and Graphics*, 25(2):269–285, 2001. 697

[68] S. Karlin, M. T. Taylor. *A First Course in Stochastic Processes*. Academic Press, 1975. 697

[69] R. Kaushik, R. Krishnamurthy, J. F. Naughton, R. Ramakrishnan. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 18th International Conference on Data Engineering*. 819

[70] R. Kaushik, R. Shenoy, P. F. Bohannon, E. Gudes. On the integration of structure indexes and inverted lists. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, pp. 779–790. 820

[71] R. R. Kaushik, P. Bohannon, J. F. Naughton, H. Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002, pp. 133–144. 819

[72] R. R. Kaushik, P. Bohannon, J. F. Naughton, H. Korth, P. Shenoy. Updates for structure indexes. In *Proceedings of Very Large Data Bases*, 2002, pp. 239–250. 819

[73] M. Khosrow-Pour (szerkesztő). *Encyclopedia of Information Science and Technology, Vol. 1, Vol. 2, Vol. 3, Vol. 4, Vol. 5*. Idea Group Inc., 2005. 820

[74] G. Krammer. Notes on the mathematics of the PHIGS output pipeline. *Computer Graphics Forum*, 8(8):219–226, 1989. 696

[75] R. Krichevsky, V. Trofimov. The performance of universal encoding. *IEEE Transactions on Information Theory*, 27:199–207, 1981. 619

[76] S. Kurtz, B. Balkenhol. Space efficient linear time computation of the Burrows and Wheeler transformation. in I. Althöfer, N. Cai, G. Dueck, L. Khachatrian, M. Pinsker, A. Sárközy, I. Wegener, Z. Zhang (editors) *Numbers, Information and Complexity*. Kluwer Academic Publishers, 2000, pp. 375–383. 619

[77] J. Lamperti. *Stochastic Processes*. Springer-Verlag, 1972. 697

[78] G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28:135–149, 1984. 618

[79] C. Lucchesi. Candidate keys for relations. *Journal of of Computer and System Sciences*, 17(2):270–279, 1978. 730

[80] D. Maier. Minimum covers in the relational database model. *Journal of the ACM*, 27(4):664–674, 1980. 730

[81] D. Maier, A. O. Mendelzon, Y. Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4(4):455–469, 1979. 730

[82] E. A. Maxwell. *Methods of Plane Projective Geometry Based on the Use of General Homogenous Coordinates*. Cambridge University Press, 1946. 696

[83] E. A. Maxwell. *General Homogenous Coordinates in Space of Three Dimensions*. Cambridge University Press, 1951. 696

[84] B. McLaughlin. *Java and XML*. O'Reilly, 2000. 820

[85] T. Milo, D. Suciu. Index structures for path expressions. Lecture Notes in Computer Science, Vol. 1540. Springer-Verlag, 1999, pp. 277–295. 819

[86] M. Nelson, J. L. Gailly. *The Data Compression Book*. M&T Books, 1996. 619

[87] M. E. Newell, R. G. Newell, T. L. Sancha. A new approach to the shaded picture problem. In *Proceedings of the ACM National Conference*, pp. 443–450, 1972. 697

[88] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987. 697

[89] R. Paige, R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987. 819

[90] R. Pasco. *Source Coding Algorithms for Fast Data Compression*. PhD thesis, Stanford University, 1976. 618

[91] S. Petrov. Finite axiomatization of languages for representation of system properties. *Information Sciences*, 47:339–372, 1989. 730

[92] N. Polyzotis, M. N. Garofalakis. Statistical synopses for graph-structured XML databases. In *Proceedings of the 2002 ACM SIGMOD international Conference on Management of Data*, 2002, pp. 358–369. 819

[93] F. P. Preparata, M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985. 696

[94] J. J. Rissanen. Generalized Kraft inequality and arithmetic coding. *http://www.research.ibm.com/journal/ Journal of Research and Development*, 20:198–203, 1976. 618

[95] D. F. Rogers, J. Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill Book Co., 1989. 696

[96] A. Sali, Sr., A. Sali. Generalized dependencies in relational databases. *Acta Cybernetica*, 13:431–438, 1998. 730

[97] D. Salomon. *Data Compression*. Springer-Verlag, 2004 (3. edition). 619

[98] L. A. Santaló. *Integral Geometry and Geometric Probability*. Addison-Wesley, 1976. 697

[99] K. Sayood. *Introduction to Data Compression*. Morgan Kaufman Publisher, 2000 (2. edition). 619

[100] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1(1):51–64, 1991. 696

[101] B. Sharp. Implementing subdivision theory. *Game Developer*, 7(2):40–45, 2000. 697

[102] B. Sharp. Subdivision Surface theory. *Game Developer*, 7(1):34–42, 2000. 697

[103] L. Stockmeyer, A. R. Meyer. Word problems requiring exponential time. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*. ACM Press, 1973, pp. 1–9. 819

[104] I. Sutherland, G. Hodgeman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974. 697

[105] I. E. Sutherland, R. Sproull, R. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974. 697

[106] L. Szécsi. An effective kd-tree implementation. in j. lauder (editor) *Graphics Programming Methods*, pp. 315–326. Charles River Media, 2003. 697

[107] L. Szirmay-Kalos, Gy. Antal, F. Csonka. *Háromdimenziós grafika, animáció és játékfejlesztés + CD (Three Dimensional Graphics, Animation and Game Development)*. Computerbooks, 2003. 697

[108] L. Szirmay-Kalos, G. Márton. Worst-case versus average-case complexity of ray-shooting. *Computing*, 61(2):103–131, 1998. 697

[109] L. Szirmay-Kalos (editor). *Theory of Three Dimensional Computer Graphics*. Akadémiai Kiadó, 1995. 697

[110] D. S. Taubman, M. W. Marcelin. *JPEG 2000 – Image Compression, Fundamentals, Standards and Practice*. Society for Industrial and Applied Mathematics, 1983. 619

[111] B. Thalheim. *Dependencies in Relational Databases*. B. G. Teubner, 1991. 730

[112] D. M. Tsou, P. C. Fischer. Decomposition of a relation scheme into Boyce–Codd normal form. *SIGACT News*, 14(3):23–29, 1982. 730

[113] A. Tucker. *Handbook of Computer Science*. Chapman & Hall/CRC, 2004. 820

[114] J. D. Ullman. *Principles of Database and Knowledge Base Systems. Vol. 1*. Computer Science Press, 1989 (2. edition). 730

[115] V. Vianu. A Web Odyssey: from Codd to XML. In *Proceedings of the 20th Symposium on Principles of Database Systems*, 2001, pp. 1–5. 819

[116] T. Várady, R. R. Martin, J. Cox. Reverse engineering of geometric models - an introduction. *Computer-Aided Design*, 29(4):255–269, 1997. 696

[117] G. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34:30–44, 1991. 619

[118] J. Warren, H. Weimer. *Subdivision Methods for Geometric Design: A Constructive Approach*. Morgan Kaufmann Publishers, 2001. 697

[119] D. Watkins. Bulge exchanges in algorithms of QR type. *SIAM Journal on Matrix Analysis and Application*, 19(4):1074–1096, 1998. 780

[120] D. Welsh. *Codes and Cryptography*. Oxford University Press, 1988. 618

[121] J. Wilkinson. Convergence of the LR, QR, and related algorithms. *The Computer Journal*, 8(1):77–84, 1965. 780

[122] F. M. J. Willems, Y. M. Shtarkov, T. J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Transactions on Information Theory*, 47:653–664, 1995. 619

[123] F. M. J. Willems, Y. M. Shtarkov, T. J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Information Theory Society Newsletter*, 1:1 and 20–27, 1997. 619

[124] I. H. Witten, R. M. Neal, J. G. Cleary. Arithmetic coding for sequential data compression. *Communications of the ACM*, 30:520–540, 1987. 618

[125] G. Wyvill, C. McPheeters, B. Wyvill. Data structure for soft objects. *The Visual Computer*, 4(2):227–234, 1986. 696

[126] K. Yi, H. He, I. Stanoi, J. Yang. Incremental maintenance of XML structural indexes. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, pp. 491–502. 819

[127] D. M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, 1971. 780

[128] C. Yu, D. Johnson. On the complexity of finding the set of candidate keys for a given set of functional dependencies. In *Information Processing 74*. North-Holland, pp. 580–583, 1974. 730

[129] B. Zalik, G. Clapworthy. A universal trapezoidation algorithms for planar polygons. *Computers and Graphics*, 23(3):353–363, 1999. 696

[130] C. Zaniolo. Analysis and design of relational schemata for database systems. Technical Report UCLA–Eng–7669, Department of Computer Science, University of California at Los Angeles, 1976. 730

[131] C. Zaniolo. A new normal form for the design of relational database schemata. *ACM Transactions on Database Systems*, 7:489–499, 1982. 730

[132] J. Ziv, A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977. 619

[133] J. Ziv, A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978. 619

Name index

Subject Index

Contents