Rheinisch-Westfälische Technische Hochschule
Institut für Geometrie und Praktische Mathematik

Mathematisches Praktikum (MaPra) — Sommersemester 2011

Prof. Dr. Wolfgang Dahmen — M.Sc. Mathieu Bachmann, Dipl.-Math. Jens Berger, M.Sc. Liang Zhang

Aufgabe 6

Bearbeitungszeit: fünf Wochen (bis Freitag, den 1. Juli 2011)

Testattermin: Donnerstag, der 7. Juli 2011

Mathematischer Hintergrund: punktweise und lokale Bildverarbeitung, Faltung, Bäume, Datenkompression

Elemente von C++: Klassen, Lesen und Schreiben von Bildformaten

Elemente von Linux: Grafikformate, man pages

30 Punkte

Aufgabenstellung

Schreiben Sie ein Programm, das es dem Benutzer ermöglicht, auf ein Bild interaktiv verschiedene Bildverarbeitungsoperationen anzuwenden. Das Originalbild wird aus einer Grafikdatei im ASCII-PGM-Format geladen; das veränderte Bild soll auch wieder in diesem Format abgespeichert werden können. Außerdem soll das Bild auch in einem verlustfrei komprimierten Bildformat gespeichert und gelesen werden. Schreiben Sie zu diesem Zweck eine Bildklasse, die diese Funktionalität bietet.

Datentypen für Bilder

Ein Graustufenbild B ist horizontal und vertikal in Bildpunkte (*Pixel*) eingeteilt. Die Helligkeit eines Bildpunktes liegt im Intervall $[0, 1]$; dabei entspricht die Helligkeit 0 der Farbe „Schwarz“ und 1 der Farbe „Weiß“. In einer Bildklasse muss also neben den Ausmaßen des Bildes in einer vollbesetzten Matrix M_B die Helligkeitsverteilung abgespeichert werden, wobei hierfür der Datentyp `float` eingesetzt werden soll.

PGM-Format

Damit Bilder zwischen verschiedenen Programmen ausgetauscht werden können, müssen die Programme ein gemeinsames Format zum Speichern dieser Bilder verarbeiten können. Ein besonders einfaches Format für Graustufenbilder ist hierbei das (ASCII-)PGM-Format (*portable greymap*) [2].

Eine Datei in diesem Format hat folgende Struktur: Zuerst kommen die beiden Zeichen `P2`. Sie bilden die sogenannte *magic number*, eine Kennung, an der man das Dateiformat identifizieren kann. Dann folgen zwei Zahlen x und y für die horizontale und vertikale Ausdehnung des Bildes. Die nächste Zahl d gibt die Farbtiefe an, d.h., $d + 1$ ist die Anzahl der möglichen Graustufen, wobei „Schwarz“ wieder durch 0, aber „Weiß“ hier durch den Wert d repräsentiert wird. Die restlichen $x \cdot y$ Werte sind die (ganzzahligen) Grauwerte im Bereich zwischen 0 und d, die die eigentliche Pixelinformation enthalten. Daneben erlaubt das Format noch Kommentare (an jeder beliebigen Stelle nach der *magic number*), die durch `#` eingeleitet werden und jeweils bis zum Zeilenende reichen. Mehr Informationen zum Dateiformat finden Sie in den entsprechenden *manual pages*, die Sie mit `man pgm` aufrufen können.

Bildverarbeitungsoperatoren

Bildverarbeitungsoperatoren sind Abbildungen $T : E \to A$ von einem Eingabebild E zu einem Ausgabebild A. Ein Operator auf einem Schwarzweißbild transformiert die Intensitätsmatrix M_E. Das Resultat ist die Intensitätsmatrix M_A, die das Ausgabebild A definiert. In dieser Aufgabe werden im wesentlichen zwei Klassen von Operatoren behandelt: pixelweise und lokale Operatoren.

Pixelweise Operationen

Der Wert eines Bildpunktes des Ausgabebildes A am Ort (x, y) ist allein durch den Wert $M_E(x, y)$ des Eingabebildes bestimmt. Deswegen ist ein punktweiser Operator T durch die Angabe einer entsprechenden Funktion $f : [0, 1] \to [0, 1]$ bereits vollständig definiert:

$$M_A(x, y) := f(M_E(x, y)) \qquad \text{für alle } x, y.$$

Beispiele für solche punktweisen Operatoren sind folgende:

- **Schwellwertbinarisierung:** Zu einem festen Schwellwert $c \in [0, 1]$ sei f_c die stückweise konstante Funktion
$$f_c(z) := \begin{cases} 0 & \text{für } z < c, \\ 1 & \text{sonst.} \end{cases}$$
Das Ausgabebild enthält so nur noch die Pixelwerte 0 und 1.
- **Affin-lineare Abbildungen (Helligkeit, Kontrast):** Der konstante Anteil b einer affinen Abbildung
$$f_{a,b}(z) := az + b$$
steuert hauptsächlich die Helligkeit des Ausgabebildes, während a für den Kontrast verantwortlich ist. Man kann die Parameter a und b beispielsweise so wählen, dass der minimale Helligkeitswert des Ausgangsbildes auf 0 und der maximale auf 1 abgebildet wird.

Lokale Faltungs-Operationen

Der Wert eines Bildpunktes des Ausgabebildes am Ort (x, y) hängt hier nicht nur vom Punkt (x, y) des Eingabebildes, sondern von den M_E-Werten in einer gewissen Umgebung von (x, y) ab.

Eine besondere Rolle spielen dabei die sogenannten Faltungen, wie sie in kontinuierlicher Form aus der Analysis bekannt sind, zum Beispiel

$$A(x) := (E * k)(x) := \int_{\mathrm{supp}(k)} E(x - t) \cdot k(t)\, dt$$

mit $t, x \in \mathbb{R}^n$, wobei die Funktion k der sogenannte Integralkern mit Träger $\mathrm{supp}(k)$ und E bzw. A wieder die Eingabe-/Ausgabefunktion ist. Das (zweidimensionale) diskrete Analogon dazu hat die Form

$$M_A(x, y) := (M_E * K)(x, y) := \sum_{i,j \in I} M_E(x + i, y + j) \cdot K(i, j),$$

wobei $I = \{-s, -s + 1, \ldots, s - 1, s\}$ für ein geeignetes $s \in \mathbb{N}$ ist. [1]

Für Randpunkte (x, y) der Bildmatrix M_E ergibt sich dabei das Problem, dass gewisse Nachbarpunkte $(x+i, y+j)$ gar nicht existieren; aus diesem Grund stelle man sich das Bild über den Rand hinaus fortgesetzt vor, und zwar mit dem Wert des nächstgelegenen Bildpunktes: Für die linke obere Ecke $(0, 0)$ beispielsweise seien $M_E(-1, -1) := M_E(-1, 0) := M_E(0, -1) := M_E(0, 0)$.

[1] Es hat sich die Vorzeichenkonvention $M_E(x + i, y + j)$ statt $M_E(x - i, y - j)$ durchgesetzt, was letztlich nur einer Umnummerierung der $K(i, j)$ entspricht.

Die $K(i,j)$-Werte kann man in einer $(2s+1) \times (2s+1)$-Matrix darstellen, wobei man im Fall $s = 1$ so den 3×3-Filterkern

$$K = \begin{pmatrix} K(-1,-1) & K(0,-1) & K(1,-1) \\ K(-1,0) & K(0,0) & K(1,0) \\ K(-1,1) & K(0,1) & K(1,1) \end{pmatrix}.$$

erhält. Die Filteroperationen, die den Filterkernen

$$K_L = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}, \quad K_T = \frac{1}{5}\begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \text{und} \quad K_K = \begin{pmatrix} 1 & 3 & 3 \\ -1 & 0 & 1 \\ -3 & -3 & -1 \end{pmatrix}$$

entsprechen, haben folgende Wirkung:

- **Laplace-Filter:** $M_A := M_E * K_L$. Der Laplace-Filter ist ein sogenannter Hochpassfilter, der hochfrequente Signalanteile verstärkt und niederfrequente unterdrückt: Da K_L (bis auf Normierung) eine Diskretisierung des Laplace-Operators $\Delta = \partial_{xx} + \partial_{yy}$ darstellt, werden konstante und lineare Intensitätsverläufe (niedrige Frequenzen) ausgelöscht.
- **Tiefpass-Filter:** $M_A := M_E * K_T$. Er nimmt das arithmetische Mittel der umliegenden Punkte und „verschmiert“ so abrupte Grauwertübergänge, d.h., hochfrequente Bildanteile werden unterdrückt. Andere Filterkerne mit $\sum_{i,j} K(i,j) = 1$ und $K(i,j) \geq 0$ liefern vergleichbare Ergebnisse. Auch im Kontinuierlichen glättet man oft Funktionen durch Faltung mit einem Kern, der $\int_{\mathrm{supp}(k)} k(t)\,dt = 1$ und $k \geq 0$ erfüllt.
- **Kirsch-Filter:** $M_A := M_E * K_K$. Da K_K Summe von diskretisierten Ableitungen in Richtung 0, 45, 90 und 135 Grad ist, werden vorzugsweise Grauwertänderungen (also Kanten) in diesen Richtungen erfasst.

Weitere lokale Operationen

Ein anderer sehr oft zur Kantendetektion eingesetzter Filter ist der

- **Sobel-Filter**, der die euklidische Norm des diskretisierten Gradienten misst:

  $$M_A(x,y) := \sqrt{[(M_E * DX)(x,y)]^2 + [(M_E * DY)(x,y)]^2},$$

  wobei

  $$DX = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad \text{und} \quad DY = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

  die Ableitungen in x- und y-Richtung diskretisieren.

Ähnliche Eigenschaften wie der Tiefpass-Filter hat der

- **Median-Filter**: Zur Berechnung von $M_A(x,y)$ werden die Pixelwerte der 8 Nachbarn des Bildpunktes (x,y) und der Wert des Punktes (x,y) selbst (insgesamt also neun Werte) der Reihe nach sortiert und der in der Rangfolge mittlere (also der fünfte) Wert als Ergebnis zurückgegeben. Gegenüber dem arithmetischen Mittel beim Tiefpass-Filter hat der Median den Vorteil, dass er einzelne fehlerbehaftete Bildpunkte komplett eliminiert statt sie nur zu verschmieren.

Zum Sortieren der 9 Punkte können Sie eine eigene Sortierfunktion schreiben[2] oder auch geeignete Algorithmen aus der STL [1] benutzen.

[2] Bei den geringen Datenmengen kann man zum Beispiel auf Bubblesort zurückgreifen.

Anforderungen an die Bildklasse

Entwerfen Sie eine Bildklasse namens `GreyScale`, die jeweils ein Grauwertbild speichern kann. Der Konstruktor soll mit der Breite und Höhe des Bildes aufgerufen werden können, dabei soll der Wert 0 für diese beiden Größen ausdrücklich zugelassen sein. Ruft man den Konstruktor ohne Argumente auf, zum Beispiel `GreyScale Bild;`, so soll dies dieselbe Wirkung wie `GreyScale Bild(0,0);` haben. Ferner soll es eine Funktion `Resize` geben, mit der man beide Dimensionen des Bildes gleichzeitig ändern kann, wobei der Bildinhalt verloren geht. Die Dimensionen des Bildes sollen sich über die Funktionen `GetWidth` und `GetHeight` abfragen lassen.

Der Wert des Pixels (x, y) soll über die Funktion `operator()` ausgelesen und geschrieben werden können. Um bei den Faltungen etc. keine Sonderbehandlungen für die Randpunkte durchführen zu müssen, sollten Sie beim Lesen auch den Zugriff auf Bildpunkte außerhalb des eigentlichen Bildes ermöglichen. Weiterhin müssen die Operationen `=`, `+=` und `-=` zur Verfügung gestellt werden, die Bilder zuweisen, aufaddieren bzw. voneinander subtrahieren. Stimmen bei einer Zuweisung die Dimensionen der Bilder nicht überein, so soll keine Fehlermeldung erzeugt, sondern das Bild auf der linken Seite des Operators umdimensioniert werden.

Die eigentlichen Filterfunktionen soll die Klasse als *member* beinhalten, die das Original nicht verändern und als Ergebnis das transformierte Bild zurückgeben:

- `Binarize(float c)`: Binarisiere mit Schwellenwert `c`.
- `Blur()`: Tiefpassfilter
- `Clamp()`: Setze alle Grauwerte kleiner 0 auf 0 und alle größer 1 auf 1.
- `Contrast()`: Transformiere den Grauwertbereich des Bildes durch eine lineare Transformation auf das maximale Intervall $[0, 1]$.
- `Convolve(const float mask[], int size=3)`: Falte das Bild mit dem `size`×`size`-Kern `mask`. Dabei stehen die Einträge des Kerns zeilenweise in dem Feld `mask`. Der Nullpunkt des Kerns liegt in der Mitte.
- `Kirsch()`: Kirsch-Filter
- `Laplace()`: Laplace-Filter
- `LinTrans(float a, float b)`: Lineare Transformation $f(z) = \mathtt{a}z + \mathtt{b}$
- `Invert()`: Invertieren des Graustufenbildes, entspricht einem Aufruf von `LinTrans(-1, 1)`.
- `Median()`: Median-Filter
- `Sobel()`: Sobel-Filter

Schließlich soll ihre Klasse die *stream*-Operatoren „>>“ und „<<“ zur Ein- und Ausgabe im PGM-Format implementieren. Bei der Ausgabe soll immer ein Bild mit 256 Graustufen geschrieben werden. Bei der Eingabe aus einem Eingabestream `s` können Ihnen die folgenden Funktionen helfen:

- `s.get()` liest das nächste Zeichen aus dem Stream `s`, d.h. `ch=s.get()` ist mit `s >> ch` vergleichbar, wenn `ch` vom Typ `char` ist.
- `s.peek()` gibt dasselbe Ergebnis zurück, allerdings ohne das Zeichen aus dem Stream zu entfernen. Damit kann man gut auf das Kommentarzeichen `#` testen.
- `s >> ws` entfernt allen „*whitespace*“, d.h. alle Leerzeichen, Tabulatorzeichen und Zeilenenden, vom Anfang des Streams.
- `s.rdstate()` gibt den aktuellen (Fehler-)Zustand des Streams zurück und löscht den Zustand anschließend. Im Ergebnis können die Bits `std::ios::eofbit`, `std::ios::failbit` sowie `std::ios::badbit` gesetzt sein. Das erste signalisiert, dass das Ende des Streams erreicht wurde, während die anderen beiden einen Fehler anzeigen. Nachdem Sie ein Bild eingelesen haben, sollte `s.rdstate()` daher den Wert `std::ios::eofbit` liefern, sonst ist irgendetwas nicht in Ordnung. Ihr Programm soll dies überprüfen.

Bildkompression

Zwar ist Festplattenplatz heutzutage einigermaßen günstig, dennoch fallen bei der Bildverarbeitung leicht Datenmengen an, die auch große Festplatten schnell füllen: Ein Farbbild mit 8 Megapixeln und 24 Bit Farbtiefe (d.h. 256 Helligkeitsstufen pro Grundfarbe (rot, grün, blau) im additiven Farbmodell) beispielsweise verschlingt unkomprimiert 24 MB Speicher. Daher möchte man Bilder komprimieren, um sie möglichst platzsparend abzulegen. Auch bei der (manchmal recht langsamen) Übertragung von Bilddaten über das Internet möchte man nicht Zeit und damit Geld für unnötig lange Dateien verschwenden.

Grundsätzlich sind dabei zwei Kategorien der Bildkompression zu unterscheiden: verlustfreie und verlustbehaftete Komprimierung. Bei der ersten Variante kann man das ursprüngliche Bild wieder exakt rekonstruieren. Prinzipiell lassen sich diese Verfahren auf beliebige Daten — nicht nur Bilder — anwenden. Eine Komprimierung erzielt man jedoch meist nur, wenn die Daten gewisse statistische Annahmen erfüllen, zum Beispiel, dass einige (Grau-)Werte häufiger vorkommen als andere oder benachbarte (Grau-)Werte meist nur wenig voneinander abweichen. Mit dieser Kategorie befasst sich die vorliegende Aufgabe.

Bei der zweiten Variante lässt sich das Originalbild nur näherungsweise rekonstruieren. Je geringer die Anforderungen an die Treue zum Original dabei sind, desto größer kann offensichtlich die Kompressionsrate werden. Typische Vertreter dieser Kategorie sind das JPEG-Format, bei dem eine diskrete Kosinustransformation eingesetzt wird, und das JPEG2000-Format, das eine Wavelet-Kompression benutzt. Hier müssen Annahmen darüber getroffen werden, welche Teile eines Bildes verändert werden können, ohne dass das Bild beeinträchtigt wird. Meist sind es die hochfrequenten Anteile eines Bildes, die nur ungenügend wiedergegeben werden. Ob man bei einem Bild wirklich auf diese Information verzichten kann, ist allerdings vom Kontext (d.h. der späteren Interpretation der Daten) abhängig: Bei einem eingescannten Foto eines Autos (mit großen glatten Flächen) werden einzelne Bildpunkte, die sich stark von ihren Nachbarn unterscheiden, hauptsächlich von Bildfehlern herrühren, die man getrost wegretuschieren kann — bei einem Foto des Nachthimmels würde man mit derselben Strategie allerdings sämtliche Sterne aus dem Bild entfernen.

Huffman-Code

Der *Huffman-Code* basiert darauf, dass einige Grauwerte in einem Bild häufiger vorkommen als andere. Werden beim PGM-Raw-Format (eine kompaktere, aber unkomprimierte Variante des Formats, siehe unten) alle Grauwerte durch eine 8 Bit-Zahl repräsentiert, so verwendet man beim Huffman-Code für häufigere Grauwerte kürzere Bitfolgen — auf Kosten der selteneren Grauwerte, die durch längere Folgen dargestellt werden. Bei der Auswahl der Codes muss allerdings darauf geachtet werden, dass die Bitfolgen „präfixfrei“ sind: Angenommen, man hätte für die drei Helligkeitswerte „schwarz“, „weiß“ und „grau“ die Codes „1“, „0“ und „10“. Dann könnte man nicht entscheiden, ob mit der Bitfolge „10“ ein schwarzes und ein weißes Pixel oder nur ein graues gemeint ist. Es darf daher nicht vorkommen, dass ein Anfangsteil (Präfix) eines Codes — hier die „1“ von „10“ für „grau“ bereits einen anderen Code darstellt — nämlich den von „schwarz“. Genau dies leistet der Huffman-Code, von dem sich sogar nachweisen lässt, dass er derjenige präfixfreie Code ist, der die höchste Kompressionsrate liefert.

Um den Huffman-Code zu bestimmen, benötigt man zunächst die Häufigkeitsverteilung der Grauwerte, das sogenannte *Histogramm*. Aus ihm wird von den Blättern zur Wurzel ein Baum konstruiert, bei dem jedes Blatt einem Grauwert entspricht: Dazu ordnet man dem Histogramm der n Grauwerte zunächst einen Wald von n Bäumen zu. Jeder Baum besteht dabei nur aus einem einzigen Knoten, der den Grauwert und seine Häufigkeit enthält. Nun sucht man sich die zwei Bäume mit der kleinsten Häufigkeit (an der Wurzel) heraus und vereint sie durch Anhängen eines gemeinsamen Vaterknotens an den Wurzeln zu einem neuen Baum. Der Vaterknoten (die Wurzel des neuen Baums) bekommt als Häufigkeit die Summe der beiden Häufigkeiten der Kinder zugewiesen. Die entstehenden Kanten bekommen die Markierung „0“ und „1“. Aus den nun $n-1$ Bäumen sucht man nun wieder diejenigen mit den kleinsten Häufigkeiten an der Wurzel etc. Abbildung 1 verdeutlicht diesen Prozess.

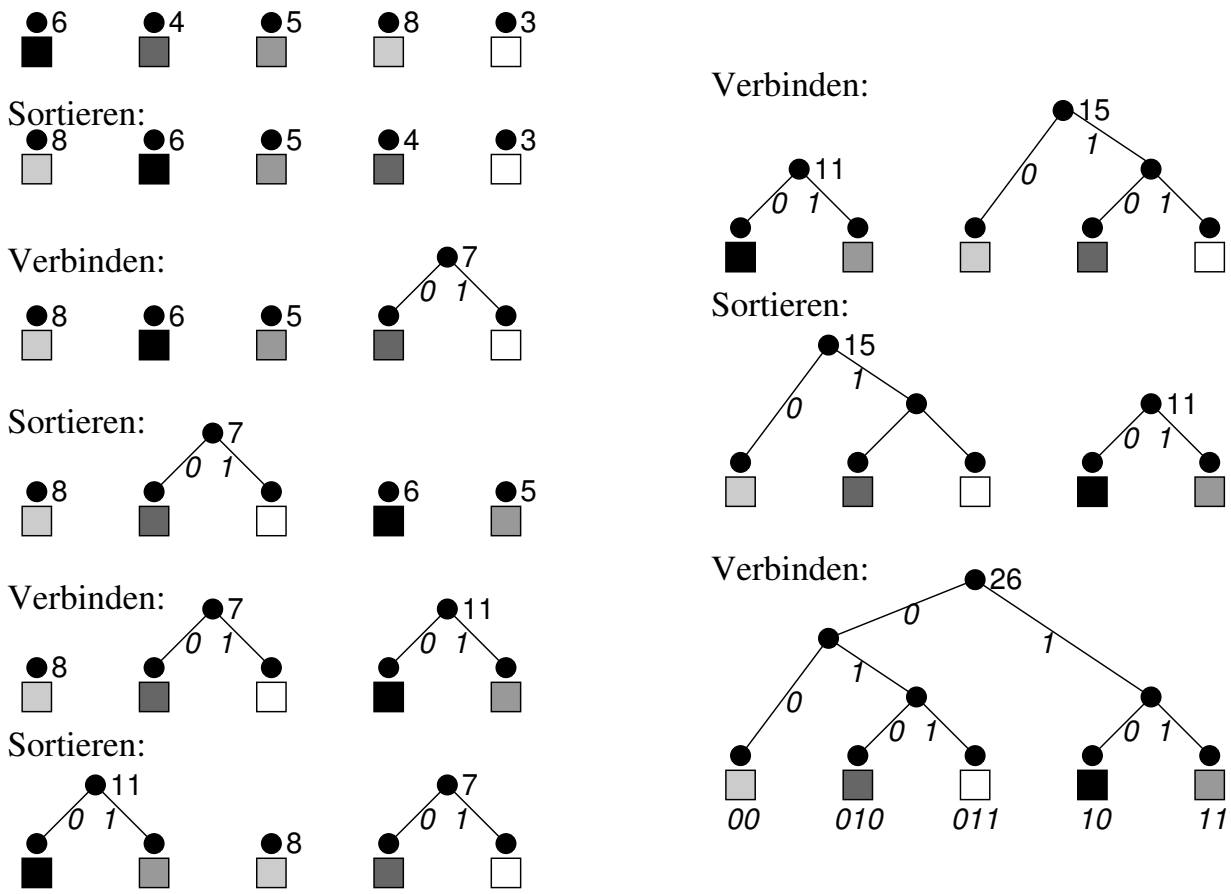

Abbildung 1: Huffman-Coding

Zum Schluss erhält man einen einzigen Baum, aus dem man nun den Huffman-Code eines Grauwertes ablesen kann: Jeder Grauwert lässt sich durch einen eindeutigen Weg von der Wurzel aus erreichen. Die Markierungen der Kanten, die man dabei durchläuft, ergeben dann den Code.

Wenn man diesen Algorithmus genau betrachtet, erkennt man, dass der Code nicht eindeutig bestimmt ist: Erstens spielt es keine Rolle, welche der beiden Kanten die Aufschrift „0“ und welche die Aufschrift „1“ bekommt, wenn man zwei Teilbäume vereinigt. Zweitens sind die beiden Bäume mit der niedrigsten Häufigkeit nicht unbedingt eindeutig bestimmt. Es hängt also vom Sortieralgorithmus und der Wahl der Kantenmarkierung ab, wie der Code aussieht. Auf die Länge des komprimierten Bildes hat dies aber keinerlei Einfluss. Man sollte aber aufpassen, dass beim Komprimieren und Dekomprimieren derselbe Code verwendet wird.

Bemerkungen zur Implementierung des Algorithmus

Für Grauwerte, die im Bild gar nicht vorkommen, sollte kein Code erzeugt werden. Ansonsten würde der Algorithmus nicht nur langsamer; unter Umständen würden die anderen Codes unnötig lang, um die Präfixfreiheit zu wahren.

Der Datentyp für einen Knoten eines Baums könnte wie folgt aussehen:

```
struct Knoten {
  byte   Grauwert;
  int    Haeufigkeit;
  Knoten *P0, *P1;
};
```

Damit der Code eindeutig wird, auch wenn es mehrere Bäume mit der gleichen Häufigkeit an der Wurzel gibt, führen wir ein zweites Sortierkriterium ein: Sind die Häufigkeiten an der Wurzel gleich, so ist derjenige Baum der „kleinste“, dessen Wurzel den niedrigsten Grauwert trägt. Beim Vereinen zweier Bäume bekommt der neu hinzukommende Vaterknoten als Grauwert das Minimum der Grauwerte seiner beiden Söhne zugewiesen. Ferner bekommt diejenige Kante die Aufschrift „0“ die zum „kleineren“ Knoten (gemäß Häufigkeit bzw. Grauwert) führt.

Es brauchen nicht immer alle Bäume korrekt sortiert zu werden, es reicht, in jedem Schritt die beiden „kleinsten“ zu lokalisieren.

Kodieren und Dekodieren eines Bildes

Wenn man den Code bestimmt hat, kann man jeden Grauwert durch seinen Code ersetzen. Sollte die Länge der Bitfolge, die so entsteht, nicht durch 8 teilbar sein, so füllt man noch Nullbits auf, bis eine durch 8 teilbare Länge erreicht ist. Jeweils 8 Bit kann man dann zu einem Byte zusammenfassen.

Auch das Dekodieren eines Bildes lässt sich am einfachsten mit Hilfe des Baums durchführen: Man startet bei der Wurzel und wählt je nach gelesenem Bit die Kante mit der entsprechenden Markierung, bis man schließlich ein Blatt, d.h. einen Grauwert, erreicht. Danach startet man wieder bei der Wurzel, um den nächsten Grauwert zu bestimmen. Da man den Baum wieder zum Dekodieren benötigt, speichern wir einfach das Histogramm zu Beginn des Bildes ab (jede Häufigkeit als 32 Bit-Zahl) — aus ihm lässt sich der Baum wieder rekonstruieren.

Verbesserung der Kompressionsrate

Je ungleichmäßiger das Histogramm ist, desto höher ist die Kompressionsrate bei der Huffman-Kodierung, weil sich dann der Längenunterschied der einzelnen Codewörter umso stärker bemerkbar macht. Um die Kompressionsrate zu verbessern, kann man versuchen, durch eine invertierbare Transformation des Bildes das Histogramm zu beeinflussen: Statt zu jedem Pixel (x, y) den Grauwert abzuspeichern, speichert man die Differenz zwischen dem Grauwert und dem (gerundeten) arithmetischen Mittel aus den Grauwerten der Pixel $(x-1, y-1)$, $(x, y-1)$, $(x+1, y-1)$ und $(x-1, y)$. Dies ist eine Zahl zwischen -255 und 255. Da die Grauwerte im Bereich 0 bis 255 liegen, kann man getrost modulo 256 rechnen, so dass man einen Wert im Bereich $\{0, \ldots, 255\}$ erhält. Das ursprüngliche Bild kann man aus diesen Daten zeilenweise von oben nach unten rekonstruieren.

Das transformierte Bild hat jetzt ein anderes Histogramm als das Original: Verändern sich die Grauwerte beim Originalbild einigermaßen stetig, dann sind die Differenzen zwischen den durch Mittelung geschätzten und den echten Grauwerten relativ klein. Beim transformierten Bild sind daher die Werte in der Nähe von Null (wegen der Rechnung modulo 256 auch die Werte in der Nähe von 255) besonders häufig. Man kann es auch so sehen: Durch die Transformation werden die langwelligen Anteile des Bildes unterdrückt, während die hochfrequenten unangetastet bleiben. Hat ein Bild nur geringe hochfrequente Anteile, so wird das Histogramm durch diese Transformation zu kleinen Werten hin verschoben, was dann zu höheren Kompressionsraten führt.

Bildformate mit und ohne Kompression

Ihre Aufgabe ist es, die Lese- und Schreiboperatoren „`<<`“ und „`>>`“ aus der `GreyScale`-Klasse so zu erweitern, dass sie neben dem bereits implementierten ASCII-PGM-Format die folgenden drei Bildformate verarbeiten können: die platzsparende Variante „PGM-Raw“-Format (siehe „pgm“ Manual Page), wo pro Pixel nur ein Byte benötigt wird, sowie die Huffman-Codierung und die Huffman-Codierung mit der gerade beschriebenen Transformation.

Das Format der beiden Varianten mit Huffman-Codierung soll wie folgt sein: Zuerst kommt die *magic number* „`MHa`“ (ohne Transformation) bzw. „`MHb`“ (mit Transformation), direkt daran anschließend jeweils als 16 Bit-Zahl (das höherwertige Byte zuerst) die Anzahl der Pixel in x-Richtung und in y-Richtung. Danach kommt das Histogramm, d.h., für jede der 256 Grauwerte eine 32 Bit-Zahl, und schließlich das kodierte Bild.

Der Lese-Operator soll anhand der magic number erkennen, um welchen Typ es sich handelt (und ihn korrekt laden). Der Schreiboperator soll durch eine statische Variable der `GreyScale`-Klasse gesteuert werden, die man über die Memberfunktion

```
void SetFormat(int);
```

setzen kann. Die Werte `0` bis `3` des Arguments stehen dann jeweils für eines der hier angeführten Bildformate (in der angegebenen Reihenfolge).

MapraView und BildTest

Die Schnittstelle zwischen der Klasse `GreyScale` und dem Anwender brauchen Sie nicht selbst zu schreiben, Sie wird Ihnen in Form der Datei `mapraview.cpp` zur Verfügung gestellt — Sie können aber auch gerne selbst eine solche Schnittstelle verfassen. Schreiben Sie eine Header-Datei `greyscale.h` mit Ihrer Klassendefinition, um sie `mapraview.cpp` zur Verfügung stellen zu können. In `unit6.o` sind die Routinen enthalten, die Ihre Bilder auf den Bildschirm bringen; sie müssen zu Ihrem Programm hinzugelinkt werden.

Zum Anzeigen der Grafiken wird das Programm `display` aus dem Paket `ImageMagick` verwendet, das auf dem Computer installiert sein muss.

Mit Hilfe des fertigen Programms sollen Sie einige Bildtransformationen auf verschiedenen Bildern durchführen und die Resultate abspeichern. Mit Hilfe des Programms `bildtest` können Sie überprüfen, ob Ihre Ergebnisse korrekt waren. Mit `bildtest -h` erhalten Sie eine Hilfestellung zu diesem Programm.

Aufgaben

Die folgenden Aufgaben sollen von Ihnen bearbeitet werden, um die Funktionsfähigkeit Ihre Implementierung der Klasse `GreyScale` zu testen. Speichern Sie die Ergebnisse der Tests jeweils in den Dateien `result1.pgm`, `result2.pgm`, ... ab.

Die ersten beiden Aufgaben dienen lediglich zum Test des Einlese-Operators $\gg$. Dabei enthält `mapra.pgm` keinerlei Kommentarzeilen, `puppenbrunnen.pgm` dagegen schon.

1. `mapra.pgm` laden und in `result1.pgm` abspeichern.
2. `puppenbrunnen.pgm` laden und in `result2.pgm` abspeichern.

Die Datei `dom.pgm` enthält ein leicht verrauschtes Bild des Aachener Domes. Durch Medianfilter bzw. Verwischen (Blur) kann dieses Rauschen vermindert werden, allerdings auf Kosten der Details bei Medianfilterung bzw. der Schärfe bei Blur. Mit beiden Methoden werden so stark unterschiedliche Ergebnisse erzielt.

3. `dom.pgm` laden, 5x Median anwenden.
4. `dom.pgm` laden, 2x Blur anwenden.

Folgende Aufgaben verdeutlichen die Wirkungsweise der verschiedenen Kantenerkennungsfilter. Offensichtlich erzeugt der Sobelfilter dickere Kanten als der Laplacefilter.

5. `shuttle.pgm` laden, Kirsch, Contrast anwenden.
6. `shuttle.pgm` laden, Laplace anwenden.
7. `shuttle.pgm` laden, Sobel anwenden.

Die Bildverarbeitung findet u.a. Anwendung bei der vollautomatisierten Qualitätskontrolle oder bei Robotern, die mit Hilfe einer Kamera ein bestimmtes Objekt lokalisieren sollen, um es zum Beispiel zu greifen: Für die Mustererkennung ist es ratsam, ein Bild so zu vereinfachen, dass nur noch wichtige Elemente wie die Konturen erhalten bleiben.

8. `bauteil.pgm` laden, Contrast, Binarize anwenden.
9. `bauteil.pgm` laden, Sobel, Invert anwenden.

Die Schärfe eines Bildes kann verbessert werden, indem die Kanten zusätzlich verstärkt werden.

10. `roentgen.pgm` laden, 3x Sobel+ (beachte das '+'!), 1x Contrast anwenden.
11. `result4.pgm` laden (der unscharfe Dom, das Ergebnis von Aufgabe 4), 5x Laplace+ anwenden.

Überprüfen Sie das Schreiben und Lesen in den verschiedenen Formaten.

12. `roentgen.pgm` laden, Format 1 wählen, abspeichern und wieder einlesen.
13. `roentgen.pgm` laden, Format 2 wählen, abspeichern und wieder einlesen.
14. `roentgen.pgm` laden, Format 3 wählen, abspeichern und wieder einlesen.
15. `shuttle.pgm` laden, Format 1 wählen, abspeichern und wieder einlesen.
16. `shuttle.pgm` laden, Format 2 wählen, abspeichern und wieder einlesen.
17. `shuttle.pgm` laden, Format 3 wählen, abspeichern und wieder einlesen.

Neben diesen Aufgaben, die als Anregung dienen sollen, sollten Sie ruhig selbst einmal mit den verschiedenen Bildern und Operationen herumexperimentieren. Viel Spaß dabei!

Überprüfung der Ergebnisse

Benutzen Sie das Programm `bildtest`, um Ihre Ergebnisse, die Sie als PGM-Dateien gespeichert haben, auf Korrektheit hin prüfen zu lassen. Um zum Beispiel die Ergebnisse der Aufgabe 7 in `result7.pgm` zu überprüfen, rufen Sie `bildtest -a7 result7.pgm` auf: `-a7` bedeutet, dass Aufgabe 7 gerechnet werden soll, die Ergebnisse der Musterlösung werden anschließend mit der Datei `result7.pgm` (also Ihrem Ergebnis) verglichen.

`bildtest` ohne Parameter startet das Programm im interaktiven Modus, so dass Sie sich auch schon vor der kompletten Implementierung Ihrer Klasse GreyScale einmal einen Eindruck davon machen können, wie Ihr Programm später laufen sollte...

Durch Aufruf von `bildtest -h` erhalten Sie einen Hilfetext mit weiteren Informationen.

Literatur

[1] *Dokumentation der C++ Standard Template Library (STL).* `http://www.sgi.com/tech/stl/`.

[2] *Informationen zum PGM-Format.* `http://de.wikipedia.org/wiki/Portable_Graymap`.

[3] Zimmer, W. D. und E. Bonz: *Objektorientierte Bildverarbeitung.* Carl-Hanser-Verlag, München, 1996.