DESIRE: Entwurf einer Datenstruktur zur symbolischen Beschreibung von Bildern

Rainer Sprengel

IfI-HH-M-142/86

29. November 1986

Fachbereich Informatik
der Universität Hamburg

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die vorliegende Arbeit entstand im Rahmen des SISSY-Projekts (Stereo Image Sequence SYstem) [*Dreschler-Fischer 86*]. Es wird die Datenstruktur DESIRE[1] vorgestellt, die die Ergebnisse der Segmentation einer Bildfolge [*Bartsch u.a. 86*] aufnehmen, aufbereiten und in geeigneter Form zur Weiterverarbeitung anbieten kann. Die Datenstruktur wird damit zum Bindeglied zwischen der ikonisch orientierten Segmentation und einer mehr symbolischen Bildverarbeitung (Korrespondenzanalyse, 3-D Rekonstruktion, Objekterkennung). Gerade weil im SISSY-Projekt kein Vorwissen in Form von Objektmodellen in die Verarbeitung eingehen soll, ist es wichtig, die wesentliche Information der Bilder kompakt weiterzugeben, ohne daß dabei zuviel Information verloren geht. Auf dieser frühen Stufe der Verarbeitung können Fehlentscheidungen gravierende Folgen haben, besonders wenn man bedenkt, daß das System Voraussagen für spätere Bilder der Folge aus dem bisher Gesehenen ableiten soll (learning by observation).

Außer den Merkmalen eines Bildes soll die Datenstruktur auch Informationen über die Güte dieser Merkmale speichern. Ein Merkmal ist für die Korrespondenzanalyse dann gut geeignet, wenn es besonders auffällig ist. Um die Algorithmen zur Lösung des Korrespondenzproblems bei der Auswahl geeigneter Merkmale zu unterstützen, werden diese klassifiziert. Das Ergebnis sind dann Klassen von Merkmalen, die man als Metamerkmale des Bildes ansehen kann. Wie wir später sehen werden, hat der Klassifikator bezüglich der Datenstruktur eine besondere Position.

Aus diesem Umfeld heraus ergeben sich verschiedene Anforderungen an die Datenstruktur:

[1] DESIRE steht für Database Environment for Symbolic Image REpresentation

große Datenmenge: Es werden Farbbildfolgen verarbeitet. Daraus resultiert eine große Datenmenge, die in der Datenstruktur abgelegt werden soll.

Parallelität: Die Datenstruktur ist die Grundlage für die Algorithmen zur Lösung des Korrespondenzproblems. Dabei sollen viele Wissensquellen parallel arbeiten können. Auch sollen die Prozeduren der 3-D-Rekonstruktion jederzeit auf die Daten zugreifen können. Die Datenstruktur sollte deshalb eine echte Parallelität der Zugriffe ermöglichen.

wahlfreier Zugriff: Die verschiedenen Algorithmen (Merkmalsfinder, Korrespondenzanalyse, 3-D-Rekonstruktion) sollen auf verschiedenen Bildern arbeiten können. Der Zugriff auf die Daten ist also nicht bildsequenziell.

räumliche Struktur: Zur Segmentierung werden verschiedene Verfahren verwendet (Punktefinder, Kantenfinder, Bereichsfinder). Die Datenstruktur muß in der Lage sein, alle diese Merkmale sowie die räumlichen Beziehungen zwischen diesen zu speichern.

verschiedene Sichten: Sowohl die Merkmalsfinder als auch die Wissensquellen sollen sich nicht um die interne Struktur der Datenhaltung kümmern müssen. Sie wollen die Daten so angeboten bekommen, wie sie sie brauchen. Merkmalsfindern wird nur schreibender Zugriff erlaubt. Der Klassifizierer muß sowohl Daten lesen als auch schreiben können. Eine Wissensquelle für das Korrespondenzproblem benötigt andere Attribute und Beziehungen als die 3-D-Rekonstruktion. Dazu müssen verschiedene Sichten auf die Daten implementiert werden.

Im Gegensatz zu allgemein gehaltenen Datenbanksystemen können jedoch Einschränkungen gemacht werden, die die Effizienz der Implementierung verbessern helfen:

2-Phasen-Ablauf: Sind die Merkmale eines Bildes einmal vollständig in die Datenstruktur eingetragen, so kommen Änderungen nicht mehr vor. Eine Segmentation wird nicht wiederholt. Es gibt also zu jeder Zeit nur einen schreibenden Prozeß, denn die Ergebnisse der Merkmalsfinder werden sequentiell in die Datenstruktur eingetragen. Daten werden bildweise zum Lesen freigegeben. Der schreibende Prozeß arbeitet auf einem Bild, welches noch nicht freigegeben ist. Wird das

Bild nach Beendigung des Schreibvorgangs dann zum Lesen freigegeben, so sind UPDATE-, INSERT- oder DELETE-Operationen auf diesem Bild nicht mehr erlaubt.

Eingeschränkter Dialog: Die Daten in der Datenstruktur sind zur Weiterverarbeitung mit anderen Programmen gedacht. Deshalb weisen auch *Benn + Radig 82* darauf hin, daß man mehr Wert auf schnellen Datenaustausch als auf Dialogorientiertheit legen sollte. Auf eine Sprache zur interaktiven Benutzung der Datenstruktur wurde verzichtet. Es gibt allerdings ein menügesteuertes, graphikorientiertes Zugriffsprogramm, mit dem man die Segmentation darstellen und analysieren kann.

nur eine Auflösung: Eine Auflösungshierarchie wird nicht implementiert. In [*Marr + Hildreth 80*] wird vorgeschlagen, die Korrespondenzanalyse mit einer Auflösungshierarchie durchzuführen. Dieser Ansatz hat breiten Anklang gefunden, und so wundert es vielleicht, daß eine solche Hierarchie hier nicht implementiert wird. Das SISSY-Projekt untersucht jedoch nur einen Teil der Ansätze zur Lösung des Korrespondenzproblems. Das Hauptziel ist es, zu klären, inwieweit sich Orts- und Bewegungs-Stereo ergänzen und beeinflußen können.

In Abbildung 1.1 ist der interne Aufbau der Datenstruktur als Petrinetz gezeichnet. Die Pfeile deuten dabei auf den Datenfluß hin. Die dynamischen Teile, die Daten berechnen oder transformieren, sind in eckigen Kästchen (Transitionen) gezeichnet, während die statischen Datenhaltungsteile (Zustände) oval umrandet sind.

Die untere Hälfte der Abbildung zeigt die zentralen Teile der eigentlichen Datenstruktur. Dies sind die interne Strukturierung der Daten in Dateien und die Schnittstellen zu den folgenden Verarbeitungsalgorithmen. Da der Kern von SISSY die Korrespondenzanalyse ist, richtet sich der Aufbau der Datenstruktur hauptsächlich nach ihren Bedürfnissen.

Die obere Hälfte der Abbildung zeigt die Schnittstellen zu den Segmentationsalgorithmen. Diese sollen ebenfalls den dort gelieferten Strukturen angepaßt sein. Auf die internen Sichten (Punkte-Sicht, Kanten-Sicht und Bereichs-Sicht) wird deshalb noch eine Stufe aufgesetzt, die Algorithmen enthält, die die Rohdaten in die Datenstruktur einträgt und dabei die Relationen explizit macht. Dieser Eintrageprozeß erfordert noch ein Steuerungsprogramm, welches nicht in der Abbildung eingezeichnet ist. Insgesamt soll

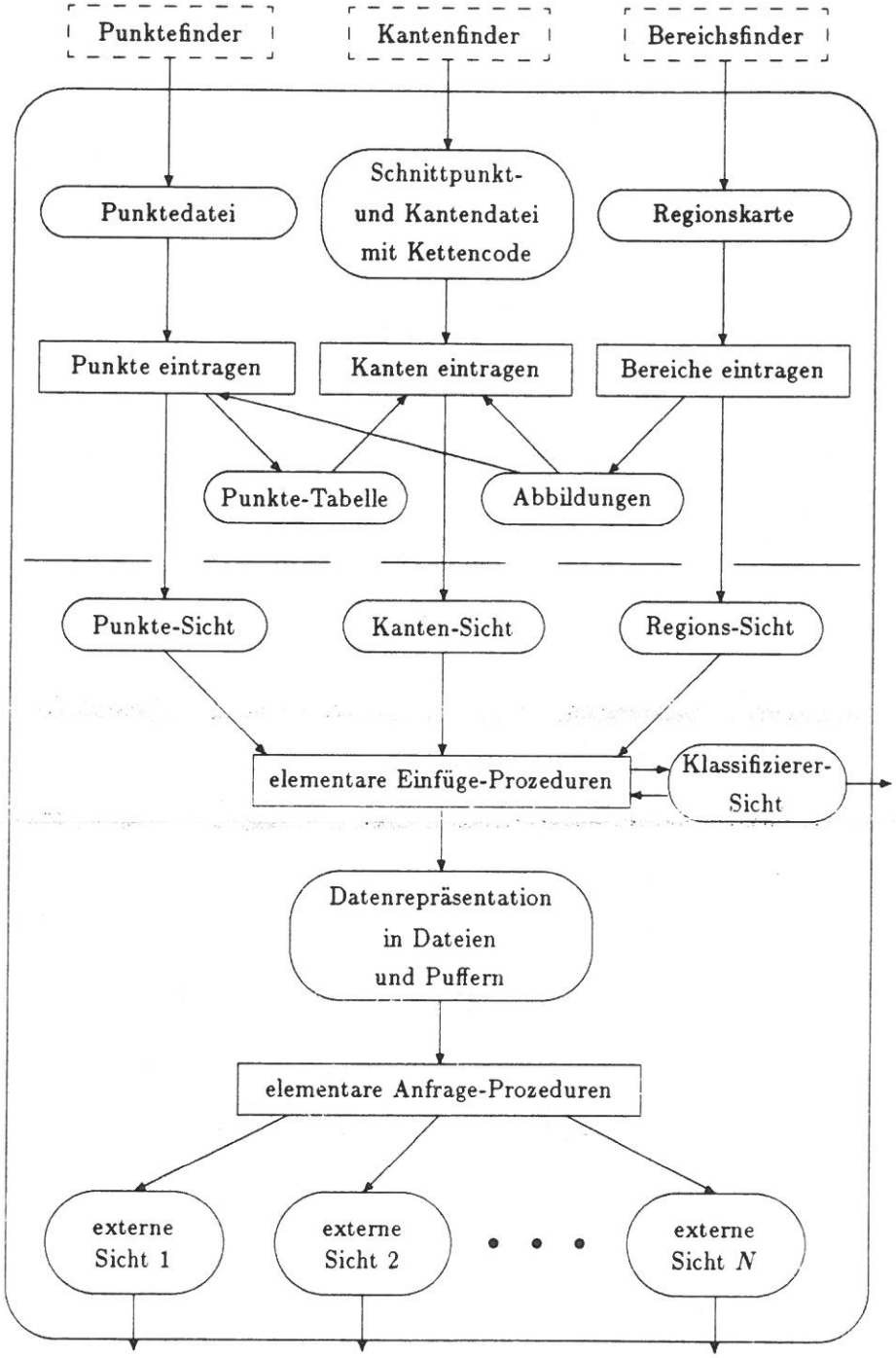


Abbildung 1.1: Übersichtsdiagramm

es damit möglich werden, die Segmentations-Verfahren leicht auszutauschen, und so verschiedene Segmentierer auf ihre Brauchbarkeit hin zu untersuchen.

Im zweiten Kapitel wird das konzeptuelle Modell der Datenstruktur vorgestellt. Es folgen im dritten Kapitel einige Hinweise zur Implementation. Mit Hilfsmitteln zum Entwurf eigener externer Schnittstellen zur Datenstruktur beschäftigt sich das vierte Kapitel.

Der Aufbau des Transformationsalgorithus für die Regionskarte wird im fünften Kapitel erläutert. Schließlich wird im sechsten Kapitel auf Probleme bei der Berechnung von Relationen zwischen gleichartigen Merkmalen, die von verschiedenen Algorithmen stammen, eingegangen. Die in der Übersicht auftauchenden Module "Abbildungen" und "Punkte-Tabellen" werden ebenfalls in diesen beiden Kapiteln behandelt.

Der Aufbau der Datenstruktur versucht sich an das Konzept der Unabhängigkeit von internem, externem und konzeptionellem Schema zu halten, wie es in der Literatur über Datenbanken vorgeschlagen wird (siehe z.B. [*Schlageter + Stucky 83*]). Auch die Begriffe wurden an die Sprechweise der Datenbankliteratur angelehnt: So werden wir z.B. die externen Schnittstellen auch Sichten auf die Daten nennen, sowie die gespeicherten Objekte als Entities bezeichnen.

Kapitel 2

Globales Konzept

Zunächst hatten wir geplant, für die genannte Aufgabe eine Datenbank einsetzen zu können. Und es gibt in der Tat für sog. Bilddatenbanksysteme (Image-Data-Base-Systems) einige Ansätze. Die meisten dieser Systeme beschränken sich aber auf die Verwaltung von Grauwert- oder Farbbildern. Eine Bilddatenbank wird oft als System definiert, in dem große Mengen von Bilddaten und relevanten Informationen darüber integriert gespeichert sind. Wir brauchen aber eine Datenbank, die geeignet ist, die räumliche Beziehungen zwischen symbolischen Bildmerkmalen aufzunehmen. *Tamura + Yokoya 83* verweisen für die Aufgaben der symbolischen Beschreibung von Bildern auf die Literatur über "Knowledge Base Systems".

In [*Kraasch + Zach 78*] wird darauf hingewiesen, daß es kaum Arbeiten über die Darstellung und weitere Verarbeitung von Segmentationsergebnissen gibt. Jedoch wurden gerade in Hamburg, insbesondere durch die Arbeiten von B. Radig und W. Benn, Forschungen zum Thema "Repräsentation von Bildern" betrieben. Ausgehend von der relationalen Beschreibung von Bildern und Objektmodellen, wie sie erstmals in [*Barrow + Popplestone 71*] entwickelt wurde, wurden Algorithmen zum Vergleich solcher Strukturen vorgeschlagen, um z.B. Objekte auch bei Verdeckungen wiedererkennen zu können. Solche Teilinstanziierungen lassen sich als Cliquen im Kompatibilitätsgraphen ansehen (siehe z.B. [*Radig 82*] oder [*Radig 84*]). Wegen der großen Datenmengen, die von verschiedenen Analyseprogrammen aus erreichbar sein sollten, schlugen *Mühle + Radig 81* ein erstes Konzept für eine Datenbank vor. Die Komplexität der Strukturvergleichs-Algorithmen führte schließlich zu der Idee, solche Vergleiche auch als Anfragen an eine Datenbank zuzulassen. Dazu wurde in [*Benn + Radig 82*] und

[*Benn 86b*] ein neues Datenbankkonzept vorgeschlagen, welches sog. dynamische, nichtnormalisierte Relationen (NF^{2D}-Relationen) zur Darstellung der Objekte benutzt. Dadurch ist es möglich, beliebig komplexe Strukturen als eigene Objekte der Datenbank anzusehen, d.h. solche Strukturen können mit einer Anfrage als Ganzes aus der Datenbank ausgelesen werden. Dabei sind sowohl strukturtolerante als auch attributtolerante Anfragen vorgesehen, d.h. man kann mit einem Referenzobjekt, welches Nullwerte enthalten kann, konkrete Objekte anfragen, die auch eine etwas andere Struktur haben können. Diese Datenbanksprache wurde deshalb auch "Query by Structure Example" (kurz QSE) genannt [*Benn 86a*]. Ein erster eingeschränkter Prototyp (mit Namen DRAGON [1]) ist an der Universität Hamburg verfügbar. Die Benutzung dieser Datenbank erschien uns für das SISSY-Projekt allerdings problematisch zu sein, denn:

- DRAGON ist integriert in eine Ada-Programmierumgebung und benutzt die DIANA[2]-Darstellung, die unter anderem auch vom ADA-HH-Compiler erzeugt wird. Leider erzeugt das VAX-Ada-System diese Zwischenstruktur nicht, so daß Benutzerprogramme nur auf den Dietz-Rechnern des KOGS-Labors der Universität Hamburg geschrieben werden können. Die Implementierung von SISSY auf diesen Rechnern ist aber nicht möglich. Eine Umstellung von DRAGON ist in absehbarer Zeit nicht zu erwarten.

- Die angebotene Allgemeinheit des Ansatzes ist für das Projekt nicht nötig. Die oben erwähnten Einschränkungen der Datenbankoperationen können nicht ausgenutzt werden. Darunter leidet die Effizienz.

- Das neue Konzept hat Vorteile bei der Suche nach vorgegebenen Modellstrukturen oder auch bei stark hierarchisch strukturierten Objekten (z.B. Polygon — Linie — Endpunkt). Die Struktur der in SISSY auf dieser Ebene abgelegten Relationen ist hingegen stark vernetzt (siehe z.B. die Nachbarschaftsrelation in Abschnitt 2.1.2). Das Problem besteht darin, Teilstrukturen aus einem Bild so abzugrenzen, daß diese im zu vergleichenden Bild auch zuverläßig wiedergefunden werden können. Eine Unterstützung dieser Abgrenzung durch die Da-

[1] Datenbank für Relationengebilde mit Aggregation und Generalisierung von Objektbeschreibungen in NF^{2D}-Relationen

[2] Descriptive Intermediate Attributed Notation for Ada programs, genaue Spezifikationen finden sich in [*Goos + Wulf 81*]

tenbank ist kaum möglich, wodurch der Vorteil der Strukturanfrage eingeschränkt wird.

In Abschnitt 2.1.1 werden die Entities mit ihren Attributen eingeführt. Die Spezifikation der Relationen wird dann in Abschnitt 2.1.2 erfolgen. *Sties u.a. 76* zeigen, welche topologischen Relationen zwischen punktartigen, linieanartigen und bereichsartigen Objekten bestehen können. Es werden 14 solcher Beziehungen aufgeführt. Die Relationen, die in unsere Datenstruktur Eingang gefunden haben, entstammen hauptsächlich dem Aufsatz [*Hanson + Riseman 78*]. Dort werden die ersten Ergebnisse der Segmentierung in einem sog. RSV-Graphen[3] gespeichert. Aber auch andere Systeme haben sich für diese Attribute und Relationen entschieden. Erwähnt sei hier noch die Sketch-Struktur, welche in [*Bertelsmeier + Radig 78*] beschrieben wird, und die die Ergebnisse der Segmentation, die das SERF-System durchführt (siehe dazu [*Kraasch + Zach 78*]), beschreibt.

Einige Merkmale unserer Struktur sind:

- Alle Daten werden auf Platte gehalten. Zugriffe können entweder direkt von der Platte oder über einen internen Puffer erfolgen.
- Es werden Schlüssel für die einzelnen Bildmerkmale und Klassen vergeben. Mit diesem Schlüssel kann jederzeit effizient auf ein Entity zugegriffen werden. Benutzer brauchen also lediglich die Verweise auf die Merkmale zu speichern.
- Es können ganze Bildfolgen abgelegt werden. Jedes Bild wird über einen Schlüssel identifiziert. Entities eines Bildes können zwischengepuffert werden. Assoziativer Zugriff ist in begrenztem Rahmen möglich.
- Es existiert ein menügesteuertes Zugriffsprogramm, mit dem sich Merkmale graphisch darstellen und analysieren lassen.

2.1 Konzeptuelles Modell

Im konzeptuellen Modell soll die Gesamtsicht der Daten definiert werden. Dieses Modell ist dann ein Bezugspunkt für alle Anwendungen mit ihren speziellen Sichten, denn alle Beziehungen und Entities der einzelnen Benutzer müssen sich aus diesem konzeptuellen Schema herleiten lassen.

[3]RSV steht für *Regions, Segments, Vertices*

2.1.1 Attributbeschreibung der Entities

Es gibt 5 Gruppen von Entities, die aufgrund der Semantik noch einmal unterteilt sind, obwohl sie teilweise die gleichen Attribute haben. Jedoch bestehen verschiedene Relationen für die verschiedenen Elemente einer Gruppe. Die Gruppen sind:

Bilder: Im allgemeinen sind dies zweidimensionale Arrays. Darunter kann man die Grauwert- bzw. Farbbilder rechnen, aber auch abgeleitete Bilder wie die Regionskarte oder der optische Fluß.

Punkte: Es gibt zwei Arten von Punkten. Ein spezieller Punktefinder sucht nach Grauwert- bzw. Farbecken sowie isolierten Flecken, wie z.B. Glanzlichtern. Diese Punkte werden markante Punkte genannt. Auch der Kantenfinder soll Punkte liefern. Hier sind die Schnittpunkte zwischen mehreren Kanten interessant. Solche Punkte werden im folgenden Schnittpunkte genannt.

Kanten: Dies sind zum einen Kanten, die ein spezieller Kantenfinder im Bild findet, und zum anderen Grenzen zwischen Bereichen, wie sie der Bereichsfinder liefert. Alle Kanten werden durch einen Kettencode beschrieben.

Bereiche: Bereiche sind Pixelmengen, deren Elemente eine gemeinsame Eigenschaft haben (z.B. Farbe, Textur). Wichtig für die Datenstruktur ist, daß es sich um flächige Gebiete handelt, die durch ihren Rand beschrieben werden können. Genauere Randbedingungen finden sich im Abschnitt 5.1.

Klassen: Alle Objekte werden klassifiziert. Dabei entstehen dann Partitionen über den obigen Objekttypen (außer Bildern). Ein Element dieser Partition wird Klasse genannt und ist ein eigenes Entity der Datenstruktur.

Im folgenden werden die symbolischen Objekte zusammen mit den Attributen aufgelistet, die in der Datenstruktur abgelegt sind und die in dieser Arbeit noch wichtig werden. Es gibt aber noch weitere verfahrensspezifische Attribute, die hier nicht explizit angegeben werden, da hier jederzeit Erweiterungen möglich sind. Zu jedem Attribut ist der Wertebereich in den theoretisch größten Grenzen für Geo-Bilder ($192 \cdot 256$ Pixel) angegeben.

1. **Bild**: Bildname : string

   Durch Anhängen von entsprechenden Extensionen an den Bildnamen kann man auf die Dateien mit den ikonischen Daten zugreifen. Möglich sind dadurch Zugriffe auf das Ursprungsbild oder die Regionskarte. Man könnte die Attributmenge noch erweitern. Denkbar sind z.B. Kameraparameter oder der Zeitpunkt der Aufnahme.

2. **Markanter Punkt**: row : (0 - 191)
   col : (0 - 255)

   Diese Punkte liegen direkt auf den Pixeln. Markante Punkte werden von einem speziellen Punktefinder geliefert. Als verfahrensspezifisches Attribut ist z.B. noch Farbe, gaußsche Krümmung oder gerichtete Varianz denkbar. Bei analytischen Verfahren ist auch eine Ortsangabe auf Subpixelgenauigkeit möglich.

3. **Schnitt-Punkt**: row : (0 - 192)
   col : (0 - 256)

   Diese Punkte liegen zwischen den Pixeln, deshalb auch der andere Wertebereich für die Koordinaten. Schnittpunkte sind Gabelungs- oder Kreuzungspunkte von Kanten. Als zusätzliches Attribut könnte man sich die Anzahl der Kanten vorstellen, die in dem Punkt münden.

4. **Kante**: startrow, endrow : (0 - 192)
   startcol, endcol : (0 - 256)
   length : (1 - 98752)
   shape : $\{\text{up,down,left,right}\}^{\text{len}}$

   Diese Kanten sollen von einem speziellen Kantenfinder in die Datenstruktur eingetragen werden. *(startrow, startcol)* ist der Anfangs-, *(endrow, endcol)* der Endpunkt der Kante. Das Attribut *shape* ist eine Kodierung des Kantenverlaufs in Form eines vierwertigen Kettencodes. Zur groben Klassifizierung könnte man die Form aber auch noch durch einen mehr qualitativen Wert wie "gerade", "rund" oder "eckig" charakterisieren.

5. **Bereichs-Grenze**:

startrow, endrow	: (0 - 192)
startcol, endcol	: (0 - 256)
length	: (1 - 98752)
shape	: $\{\text{up,down,left,right}\}^{\text{len}}$

Bereichsgrenzen sind die Grenzlinien zwischen zwei Bereichen, nicht die Umrandung eines ganzen Bereichs. *(startrow, startcol)* ist der Anfangs-, *(endrow, endcol)* der Endpunkt der Bereichsgrenze. Das Attribut *shape* ist eine Kodierung des Grenzverlaufs in Form eines vierwertigen Kettencodes. Die Gesamtheit der Grenzen eines Bereichs beschreibt dessen Form.

6. **Bereich**:

size	: (0 - 49152)	$= \sum_{(x,y)\in\Re} 1$
rowsum	: (1 - 4694016)	$= \sum_{(x,y)\in\Re} y$
colsum	: (1 - 6266880)	$= \sum_{(x,y)\in\Re} x$
rowqsum	: (1 - 599269376)	$= \sum_{(x,y)\in\Re} y^2$
colqsum	: (1 - 1067458560)	$= \sum_{(x,y)\in\Re} x^2$
rowcolsum	: (1 - 598487040)	$= \sum_{(x,y)\in\Re} xy$
minrow	: (0 - 191)	
maxrow	: (0 - 191)	
mincol	: (0 - 255)	
maxcol	: (0 - 255)	
perimeter	: (4 - 98752)	

Mit diesen Attributen lassen sich Schwerpunkt, Flächenträgheitsmomente und Orientierung eines Bereichs berechnen. Siehe dazu den Abschnitt 4.2. Weitere Attribute, wie *Farbmittelwert* und *Varianz*, sind vorgesehen.

7. (a) **Punkte-Klasse**:

size	: (1 - 49152)
maxdistance	: (0 - ∞)
mindistance	: (0 - ∞)

(b) **Kanten-Klasse**:

size	: (1 - 98752)
maxdistance	: (0 - ∞)
mindistance	: (0 - ∞)

(c) **Bereichs-Klasse**:

size	: (1 - 49152)
maxdistance	: (0 - ∞)
mindistance	: (0 - ∞)

Die angegebenen Abstände sind reelle Zahlen[4]. Sie geben einen Eindruck von der Homogenität der Klasse, indem sie den Bereich der vorkommenden Abstände in einer normierten Metrik angeben. Weiterhin könnte man *Mittelwert* und *Varianz* als Attribute aufnehmen. Es ist vorgesehen, einen typischen Repräsentanten der jeweiligen Klasse anzugeben.

2.1.2 Relationen zwischen Entities

Die folgenden Beziehungen werden in der Datenstruktur realisiert. Dabei ist zu beachten, daß nicht alle dieser Relationen gleich schnelle Zugriffe erlauben (siehe das Kapitel 3.1 über die Implementierung). Die Zahlen unter den Namen verweisen auf die Nummer der Beschreibung im vorherigen Abschnitt.

1. **Bild-von** (Bild, Merkmal)
   1 2-7(c)

Zu jedem Merkmal ist bekannt, in welchem Bild es liegt. Ebenso kann man über diese Relationen an alle Merkmale eines bestimmten Bildes herankommen.

2. **Nachbarschaft** (Region 1, Grenze, Region 2)
   6 5 6

Zwei Bereiche $\mathcal{A}$ und $\mathcal{B}$ sind benachbart, wenn sie eine gemeinsame Grenze ϕ haben. Bereiche, die sich lediglich in einem Punkt berühren, sind nicht benachbart. Eine Grenze wird gerade durch die Bereiche definiert, die sie trennt. Somit gibt es zu je zwei benachbarten Bereichen genau eine Grenze, und umgekehrt. Eine Grenze kann also auch als Attribut der Nachbarschaft zweier Bereiche aufgefasst werden. Es

[4]Die oberen Grenzen sind abhängig vom verwendeten Klassifikator

ist klar, daß die Nachbarschaftsbeziehung symmetrisch ist. Wir führen noch folgende Kurzschreibweise ein:

$$\text{Benachbart}(\mathcal{A},\mathcal{B}) \overset{\text{def}}{\Longleftrightarrow} \bigvee_{\phi} : \text{Nachbarschaft}(\mathcal{A},\phi,\mathcal{B})$$

3. **Teil-von** (Region 1, Region 2)
   6 6

Zwei Bereiche $\mathcal{A}$ und $\mathcal{B}$ stehen in der Teil-von-Beziehung zueinander, wenn Bereich $\mathcal{A}$ in Bereich $\mathcal{B}$ enthalten ist, und beide Bereiche benachbart sind. Mit der gerade eingeführten Nachbarschaftsrelation kann man dies auch folgendermaßen ausdrücken:

Es sei $Pf_{\mathcal{A}}$ die Menge der Pfade, die vom Rand des Bildes bis zum Bereich $\mathcal{A}$ laufen. Ein Pfad P besteht aus einem Tupel von Pixeln, wobei zwei aufeinanderfolgende Pixel jeweils zusammenhängen. Es gilt:

$$\text{Innerhalb}(\mathcal{A},\mathcal{B}) \overset{\text{def}}{\Longleftrightarrow} \bigwedge_{P\in Pf_{\mathcal{A}}} \bigvee_{p_i\in P} : p_i \in \mathcal{B}$$

und schließlich

$$\text{Teil-von}(\mathcal{A},\mathcal{B}) \overset{\text{def}}{\Longleftrightarrow} \text{Innerhalb}(\mathcal{A},\mathcal{B}) \wedge \text{Benachbart}(\mathcal{A},\mathcal{B})$$

4. **Liegt-in** (Region, Punkt)
   6 2

Zu jedem markanten Punkt gibt es genau einen Bereich, in dem er liegt. Weil Schnittpunkte zwischen den Pixeln liegen, ist eine Relation zwischen diesen Punkten und Bereichen nicht aufgenommen worden.

5. **Punkt-auf** (Schnitt-Punkt, Kante)
   3 4

Die Kante mündet in dem Punkt. In die Punkte sollen mehrere Kanten einlaufen. Der Ort muß nicht immer genau mit den Endpunkten der Kanten übereinstimmen, falls zwischen mehreren Endpunkten gemittelt werden mußte.

6. **Punkteentsprechung** (Markanter-Punkt, Schnitt-Punkt)
   2 3

Da markante Punkte und Schnittpunkte durch verschiedene Algorithmen unabhängig voneinander gefunden werden, ist es möglich, daß zwei Punkte Abbildungen desselben Objektpunktes der Szene sind. Solche Punkte sollten in dieser Relation miteinander stehen. Von der Semantik her sollte es sich um eine partielle 1:1 Abbildung handeln, d.h. zu jedem Punkt der einen Art gibt es höchstens einen Punkt der anderen Art. Dies wird vom Algorithmus, der die Relation herstellen soll, jedoch nicht mit vollständiger Sicherheit gewährleistet (siehe Kapitel 6). Aus diesem Grund handelt es sich um eine partielle 1:n-Abbildung von Schnittpunkten zu markanten Punkten. Zu jedem Schnittpunkt gibt es also nur höchstens einen markanten Punkt, während ein markanter Punkt auch mehrere zugeordnete Schnittpunkte haben kann.

7. **Kantenentsprechung** (Kante, Grenze)
   4 5

Da Kanten und Bereiche durch verschiedene Algorithmen unabhängig voneinander gefunden werden, ist es möglich, daß eine Kante an einer Bereichsgrenze entlangläuft. Solche Kanten sollten in dieser Relation mit der Bereichsgrenze stehen. Von der Semantik her sollte es sich um eine partielle 1:1 Abbildung handeln, d.h. zu jeder Kante der einen Art gibt es höchstens eine Kante der anderen Art. Dies wird vom Algorithmus, der die Relation herstellen soll, jedoch nicht mit vollständiger Sicherheit gewährleistet (siehe Kapitel 6). Aus diesem Grund handelt es sich um eine partielle 1:n-Abbildung von Kanten zu Bereichsgrenzen. Zu jeder Kante gibt es also nur höchstens eine Grenze, während eine Grenze auch mehrere zugeordnete Kanten haben kann.

8. **Punkt-in**(Klasse, Punkt)
   7(a) 2
   7(a) 3

Alle Punkte werden nach ihrer Ähnlichkeit klassifiziert. Als Merkmal wird dazu die lokale Umgebung verwendet. Da es sich um eine Partition handelt, liegt jeder Punkt in genau einer Klasse. Es handelt sich also um eine hierarchische Relation.

9. **Kante-in**(Klasse, Kante)
   7(b) 4
   7(b) 5

Alle Kanten werden nach ihrer Ähnlichkeit klassifiziert. Da es sich um eine Partition handelt, liegt jede Kante in genau einer Klasse. Es handelt sich also um eine hierarchische Relation.

10. **Bereich-in**(Klasse, Bereich)
    7(c) 6

Alle Bereiche werden nach ihrer Ähnlichkeit klassifiziert. Da es sich um eine Partition handelt, liegt jeder Bereich in genau einer Klasse. Es handelt sich also um eine hierarchische Relation.

Kapitel 3

Implementation

In diesem Kapitel geht es um das interne Datenmodell. Hier interessieren vor allem zwei Punkte:

- Die Implementation insbesondere der komplexeren Relationen.
- Die Realisierung der Parallelität von Zugriffen. Hier muß noch zwischen Zugriffen auf Dateien und Puffer unterschieden werden.

Bei der Erörterung dieser Punkte soll auf spezielle Eigenheiten, die auf die Eigenschaften der Rechenanlage zurückgehen, möglichst verzichtet werden. An einigen Stellen werden aber Andeutungen gemacht.

3.1 Zugriffspfade

Zugriffe auf die Entities in der Datenstruktur erfolgen auf verschiedenen Wegen. Zunächst gibt es die Möglichkeit, über einen Primärschlüssel direkt auf Entities zuzugreifen. Dann kann man aber natürlich auch die vorhandenen Relationen verwenden. So möchte man z.B. alle Nachbarn einer Region wissen. Manchmal ist es auch nötig, auf Mengen von Entities über ein Nicht-Schlüssel-Attribut assoziativ zuzugreifen. Wie die Anfragen eines Benutzers nachher wirklich aussehen, soll hier zunächst noch offen bleiben. In diesem Teil wird erst einmal geklärt, wie die Relationen eigentlich abgelegt werden. Diese Frage ist aber aus Effizienzgründen nicht ganz von der Anwendung zu trennen, und so machen wir hier einige Annahmen über das Zugriffsverhalten der Anwender.

Die Datenstruktur vergibt bei der Erzeugung eines jeden Entities einen Primärschlüssel, über den auf das Entity effizient direkt zugegriffen werden

kann. Diese Primärschlüssel werden sowohl innerhalb der Struktur zum Speichern der Relationen verwendet, als auch an die Benutzer weitergegeben, um ihnen zu ermöglichen, Verweise auf die einzelnen Merkmale zu speichern.

Bei den Schlüsseln handelt es sich einfach um Nummern, die in aufsteigender Reihenfolge den Entities zugewiesen werden. Es gibt einen vordefinierten **Null**-Schlüssel. Aus dem 2-Phasen-Ablauf von Schreiben und Lesen folgt dann, daß alle Schlüssel der Merkmale und Klassen eines Bildes ein zusammenhängendes Intervall bilden. Die **Bild-von**-Relationen werden nun dadurch implementiert, daß zu jedem Bild die Bereiche seiner Merkmals- und Klassen-Schlüssel gespeichert werden.

Die Relationen **Nachbarschaft**, **Teil-von** und **Punkt-auf** sind als Mengen von Primärschlüsseln realisiert. Das bedeutet: Jedes Merkmal enthält einen Verweis auf eine Liste von Primärschlüsseln derjenigen Entities, die mit ihm in der entsprechenden Relation stehen.

Die Beziehungen zwischen Punkten und Kanten der verschiedenen Merkmalsfinder (also die Relationen **Punkteentsprechung**, **Kantenentsprechung**) werden in den beiden Richtungen verschieden realisiert. Aus der Semantik der Beziehung sollte folgen, daß es nur 1:1 Beziehungen dieser Art gibt. Dies wird durch den Algorithmus, der diese Beziehungen herstellt, aber nicht garantiert. Es kann sein, daß z.B. ein markanter Punkt mit mehreren Schnitt-Punkten korrespondiert. Hingegen kann es zu jedem Schnitt-Punkt nur einen markanten Punkt geben. In der einen Richtung ist die Beziehung somit durch einen einfachen Verweis realisiert, in der anderen Richtung durch Verkettung. Damit ist zwar nur ein sequentielles Durchlaufen durch die Relation möglich, doch sollten, wegen der Semantik, die Ketten nicht allzu lang werden.

Auch die Elemente einer jeden Klasse sind miteinander verkettet, denn von der Anwendung wird nur ein sequentieller Durchlauf durch die Elemente einer Klasse gefordert. Allerdings ist in jedem Merkmal ein direkter Verweis auf die zugehörige Klasse gespeichert.

Weiterhin gibt es natürlich die Möglichkeit einen assoziativen Zugriff durch eine sequentielle Suche zu realisieren. Hier wurden zwei Wege beschritten. Zum einen gibt es für bestimmte Attribute, wie z.B. Koordinaten bei Punkten oder Größe bei Regionen, schon fest implementierte Prozeduren. Hier kann ein Intervall angegeben werden, innerhalb dessen die Attributwerte liegen sollen. Zum anderen werden aber auch generische Prozeduren angeboten, die mit einer Operation, die dann sequentiell für jedes Entity des entsprechenden Bildes ausgeführt wird, instanziiert werden kann. Dieser sehr viel allgemeinere Ansatz ermöglicht es dann, beliebige FOR_EACH-

Schleifen effizienter zu realisieren, als es durch jeweilige Zugriffe über die **Bild-von**-Relation möglich ist. Die Instanziierung der generischen Prozeduren sollte dann, nach den Bedürfnissen der Benutzer, in den externen Schnittstellen erfolgen.

3.2 Parallele Anfragen

Parallele Operationen auf Dateien

Die Algorithmen, die später mit der Datenstruktur arbeiten, sollen möglichst parallel ablaufen können, denn die Ergebnisse verschiedener Wissensquellen sollen sich gegenseitig stützen können, ohne daß eine Reihenfolge festgelegt werden muß. Um dies zu ermöglichen, gibt es verschiedene Alternativen, die sich in der realisierten Parallelität unterscheiden:

1. Greift ein Prozeß (in Ada ein Task) auf die Datenstruktur zu, so wird der Zugriff aller anderen Prozeße verzögert. Dies bedeutet also, daß die gesamte Datenstruktur gesperrt wird, falls auch nur ein Benutzer lesend oder schreibend auf einen Teil zugreift. Diese Lösung läßt sich sehr einfach implementieren, indem die Datenverwaltung zu einem Ada-Task gemacht wird. Jede Anfrage wird dann zu einem Rendezvous und findet im wechselseitigen Ausschluß statt. Allerdings ist hier die denkbar kleinste Parallelität erreicht. Etwas abmildern kann man dieses Verhalten, indem man disjunkte Mengen von Zugriffen, die sich nicht gegenseitig stören, als eigene Tasks implementiert.

2. Allen Prozessen wird der parallele lesende Zugriff auf die Datenstruktur gestattet, gleichzeitig darf ein schreibender Prozeß neue Daten in die Datenstruktur einfügen, die dann aber für Leser noch nicht freigegeben sind. Diese Lösung nutzt den 2-Phasen-Ablauf der Bilddaten aus, und erlaubt so größtmögliche Parallelität.

   Bei der Implementierung treten jedoch Schwierigkeiten auf. Der Zugriff auf eine physikalische Datei erfolgt immer über eine logische File-Variable, die vorher mit der physikalischen Datei eröffnet worden ist. Auch wenn es sich nur um lesende Zugriffe auf die physikalische Datei handelt, wird der interne Zustand der logischen File-Variable verändert; der Index zeigt auf ein anderes Record. Hier sei auf eine Inkonsistenz der Ada-Input-Output-Pakete verwiesen, die bei `READ`, `WRITE` und `SET_INDEX` Prozeduren die Datei trotzdem als **in**-Parameter be-

zeichnen. Lesen alle Prozeße über dieselbe File-Variable, so können sie sich also durchaus gegenseitig stören.

Das Problem kann umgangen werden, wenn jeder Prozeß, der die Datenstruktur benutzt, seine eigenen File-Variablen bekommt. Alle diese Variablen werden mit ein und derselben physikalischen Datei eröffnet[1]. Die File-Variablen werden als **limited private**-Typen nach außen weitergegeben. Sie können dann in der externen Schnittstelle versteckt werden.

Um nun noch einen schreibenden Prozeß zulassen zu können, muß das Paket `RELATIVE_IO` benutzt werden. Die Daten eines Bildes müssen vom schreibenden Prozeß zum Lesen freigegeben werden. Danach können keine Veränderungen mehr vorgenommen werden. Da somit sicher gestellt ist, daß ein schreibender Prozeß nicht in Konflikt mit einem lesenden Prozeß geraten kann, wird die Record-Locking-Option des Betriebssystems ausgeschaltet.

Implementiert wurde hier die zweite Version. Ein kleiner Nachteil ist, daß jeder Task, der Zugriff auf die Datenstruktur hat, eine gewisse Zeit für das Öffnen der Dateien investieren muß.

Paralleler Zugriff auf Puffer

Um nicht bei jedem Zugriff erneut I/O-Operationen durchführen zu müssen, ist eine benutzergesteuerte Pufferung der Daten implementiert worden. Hier ergeben sich ebenfalls Schwierigkeiten beim parallelen Zugriff. Der Benutzer kann Teile der Datenstruktur jeweils bildweise im Hauptspeicher puffern lassen. Ist ein Bild einmal gepuffert, so sollen alle Prozesse (auch diejenigen, die den Puffer nicht eröffnet haben) auf diesen Puffer zugreifen können. Wird auf dem Bild nicht mehr gearbeitet, so kann der entsprechende Prozeß den Puffer wieder freigeben.

Puffer können dynamisch erzeugt und wieder freigegeben werden. Sie werden von Puffervariablen verwaltet, die angeben, welche Puffer mit welchen Bildern von wievielen Prozessen belegt sind. Jede Operation, die einen Puffer beschreibt oder freigibt, muß diese Variablen auf dem laufenden Stand halten. Eine Operation, die ein Objekt lesen will, kann dann anhand dieser

[1]Dies ist in VAX-Ada erlaubt, falls man entsprechende Angaben im Form-String übergibt. Informationen über die verwendete File-Description-Language (FDL) findet man im "VAX Ada Programmer's Run-Time Reference Manual" und im "Guide to VAX/VMS File Application"

Variablen feststellen, ob das Objekt gepuffert ist oder aber von der Platte eingelesen werden muß. Diese Operationen zum Lesen aus - und Schreiben in einen Puffer werden nun in ihre einzelnen Bestandteile zerlegt:

(1)	Puffer beschreiben:	(a) Lesen der Puffervariablen
		(b) Schreiben in den Puffer
		(c) Puffervariable verändern
(2)	Puffer freigeben:	(a) Lesen und verändern der Puffervariablen
		(b) Freigeben des Puffer-Speicherplatzes
(3)	Lesen	(a) Lesen der Puffervariablen
		(b1) Lesen aus dem Puffer
		(b2) Lesen von der Datei

Da diese Gesamtoperationen nicht unbedingt als eine Einheit ausgeführt werden, ist eine Synchronisation erforderlich, die dann die Konsistenz der Puffervariablen sicherstellt.

Wir haben es hier mit einem Leser/Schreiber-Problem zu tun, d.h.

- Wenn gelesen wird, soll kein Schreiber arbeiten.
- Wenn geschrieben wird, soll kein lesender Zugriff erfolgen.
- Schreibende Prozeße sollen sich gegenseitig ausschließen.

Dabei sind natürlich (1) und (2) die schreibenden Operationen, sowie (3) die lesende Operation. Existieren mehrere Puffer, so kann man diese Einschränkungen auf jeden Puffer und die Puffervariablen einzeln, oder auf alle Puffer zusammen anwenden.

Zur Lösung dieses Problems gibt es wieder mehrere Möglichkeiten. Im folgenden sei n die Anzahl der Puffer.

- Jeder Benutzer bekommt seine eigenen Puffer, die nur er benutzen kann. Der Vorteil der Effizienz und einfachen Implementierung wird durch den Nachteil der Speicherplatzverschwendung bei mehrfacher Pufferung der gleichen Daten wieder aufgehoben.
- Man führt $n + 1$ Scheduler-Tasks ein, die folgendes leisten:
  Der erste Scheduler-Task verwaltet die Puffervariablen und stellt hier die genannten Bedingungen sicher. Die anderen Tasks machen das

gleiche für die Puffer selbst. Ein Beispiel, wie solche Scheduler in Ada realisiert werden können, findet man z.B. in [*Habermann + Perry 83*].

Will also ein Leser auf die Datenstruktur zugreifen, so reiht er sich zunächst in die Schlange für die Zugriffe auf die Puffervariablen ein. Schließlich erfährt er dann, ob die gebrauchten Daten überhaupt gepuffert sind. Ist dies nicht der Fall, so war die bisherige Wartezeit umsonst, es muß sowieso auf die Datei direkt zugegriffen werden. Der Nachteil an diesem Verfahren ist also der Flaschenhals, den die Verwaltung der Puffervariablen darstellt.

- Eine Analyse der möglichen Reihenfolgen der elementaren Operationen soll einen Ausweg aufzeigen. Zunächst ist klar, daß nichts passieren kann, wenn alle drei Operationen jeweils als Einheit ausgeführt werden. Auch die Leser stören sich gegenseitig nicht. Wenn wir zunächst den wechselseitigen Ausschluß aller Schreiber annehmen, so kommen wir zu folgenden Fällen:

  - Operation (1) wird ausgeführt. Ein Leser benutzt den neuen Puffer nur, wenn die Puffervariablen anzeigen, daß er die richtigen Daten enthält. Die Puffervariablen werden aber erst gesetzt, wenn der schreibende Prozeß fertig ist. Ein Fehler kann also nicht auftreten.
  - Operation (2) wird ausgeführt. Es bleiben dann folgende Reihenfolgen mit lesenden Prozeßen übrig:
    1. (3a)(2a)(3b1/2)(2b): Kein Problem, denn der Puffer ist gelesen worden, bevor er freigegeben wird.
    2. (2a)(3a)(2b)(3b1/2): Entweder liest der Leser von einem anderen Puffer, oder er stellt fest, daß die Daten nicht gepuffert sind. Das ist zwar in diesem Fall nicht richtig (denn der Puffer existiert zu diesem Zeitpunkt ja noch), aber ein Fehler entsteht dadurch nicht.
    3. (3a)(2a)(2b)(3b1) : Hier tritt ein Fehler auf. In dem Moment, wo der Leser auf den Puffer zugreifen will, existiert dieser schon nicht mehr. Ein `CONSTRAINT_ERROR` wird ausgelöst.

Falls man also dafür Sorge trägt, daß sich die (relativ seltenen) Schreiber wechselseitig ausschließen, so kann es höchstens vorkommen, daß ein Puffer nicht mehr vorhanden ist, wenn auf ihn zugegriffen wird. In diesem Fall kann man aber die entsprechende Exception abfangen und dann direkt von der Datei lesen.

Kapitel 4

Externe Schnittstellen

Die Gestaltung der externen Schnittstellen oder Sichten auf die Datenstruktur, geschieht nach Gesichtspunkten der Benutzer. Es muß dabei gewährleistet sein, daß alle angebotenen Daten auch aus dem konzeptuellen Datenmodell abgeleitet werden können. An dieser Stelle wird zunächst eine weitere abstrakte Datenstruktur "Liste" eingeführt, die bei Anfragen eingesetzt werden kann, die eine variable Anzahl von Elementen zurückliefern. In Ada kann man zwar variabel lange sogenannte "unconstraint arrays" als Parameter verwenden, doch muß die Länge bei einer konkreten Anfrage, in der das Array als Out-Parameter auftritt, schon vorher festgelegt sein.

Im zweiten Teil wird gezeigt, wie die im konzeptuellen Modell gegebenen Attribute der Bereiche in eine aussagekräftigere Form umgerechnet werden können.

Zum Schluß werden die, für die Zusammenstellung einer eigenen Schnittstelle wichtigen, Pakete genannt. Ein kurzes Beispiel für eine Schnittstellenimplementation ist angefügt.

4.1 Listen als Datenstruktur für Anfragen

Eine abstrakte Datenstruktur wird definiert über die Operationen, die mit ihr möglich sind. Ich halte mich hier an die Art der Definition, wie sie Wirth für die Datenstruktur "File" in [*Wirth 79*] verwendet hat:

Notation:

$\langle\rangle$	ist die leere Liste
$\langle x \rangle$	ist die Liste mit x als einzigem Element

Mit $X = \langle x_1, \ldots, x_n \rangle$ und $Y = \langle y_1, \ldots, y_m \rangle$ sei:

$\|X\|$	$= n$	(Länge der Liste)
$X \diamond Y$	$= \langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$	
$firstX$	$= x_1$	($first\langle\rangle$ ist undefiniert)
$lastX$	$= x_n$	
$restX$	$= \langle x_2, \ldots, x_n \rangle$	($rest\langle\rangle$ ist $\langle\rangle$)

Jede Liste hat ein aktuelles Element.
Dies sei folgendermassen beschrieben:

X_L ist die Liste links vom aktuellen Element.
X_R ist die Liste rechts vom aktuellen Element (einschließlich).
$firstX_R$ ist das aktuelle Element.

Operationen:

IS_EMPTY(X)	liefert	$X = \langle\rangle$
NUM_OF_ELEM(X)	liefert	$\|X\|$
END_OF_LIST(X)	liefert	$\|X_R\| \leq 1$
NO_CURRENT_ELEMENT(X)	liefert	$X_R = \langle\rangle$
EMPTY_LIST(X)	ergibt	$X \leftarrow \langle\rangle$;
RESET(X)	ergibt	$X_L \leftarrow \langle\rangle$;
		$X_R \leftarrow X$;
GET(X, e)	ergibt	$e \leftarrow firstX_R$;
GET_FIRST(X, e)	ergibt	$X_L \leftarrow \langle\rangle$;
		$X_R \leftarrow X$;
		$e \leftarrow firstX$;
GET_LAST(X, e)	ergibt	$e \leftarrow lastX$;
NEXT(X)	ergibt	$X_L \leftarrow X_L \diamond \langle firstX_R \rangle$;
		$X_R \leftarrow restX_R$;
GET_NEXT(X, e)	ergibt	$X_L \leftarrow X_L \diamond \langle firstX_R \rangle$;
		$X_R \leftarrow restX_R$;
		$e \leftarrow firstX_R$;
INSERT(X, e)	ergibt	if $X_R = \langle\rangle$
		then $X_R \leftarrow \langle e \rangle$;

		else $X_L \leftarrow X_L \diamond \langle firstX_R \rangle$;
		$X_R \leftarrow \langle e \rangle \diamond restX_R$;
		end if;
INSERT_BEFORE(X, e)	ergibt	$X_L \leftarrow X_L \diamond \langle e \rangle$;
UPDATE_CURRENT_ELEMENT(X, e)	ergibt	$firstX_R \leftarrow e$;
DELETE_CURRENT_ELEMENT(X)	ergibt	**if** $X_R \neq \langle\rangle$;
		then $X_R \leftarrow restX_R$;
		end if;

Exceptions:

LIST_ACCESS_ERROR
wird ausgelöst, wenn versucht wird $first\langle\rangle$ in einer der folgenden Operationen auszuführen:
GET, GET_FIRST, GET_NEXT, NEXT, UPDATE_CURRENT_ELEMENT

Die Listenstruktur wird in einem *generischen* Ada-Paket angeboten, welches mit dem Element-Typ instanziiert werden muß. Die Repräsentation der Listen ist im *private*-Teil des Paketes verborgen. Somit kann der Benutzer den Zustand der Listen nur über die angebotenen Prozeduren verändern.

Leider kann das Paket nicht mit einem *private type* instanziiert werden, obwohl in der Implementation der Operationen keinerlei Annahmen über die Struktur der Elemente der Listen gemacht wird. Auch mit *unconstraint types* können keine Listen instanziiert werden. Hier gibt es zwei Möglichkeiten zur Abhilfe:

- Man instanziiert die Listen mit einem *access type* auf die entsprechenden *unconstraint* Element-Typen.
- Falls es sich bei dem *unconstraint type* um einen varianten Record-Typ handelt, kann man diesen als Feld in einen neuen Record-Typ einbauen. Dieser neue Record-Typ ist dann ein *constraint type* und man kann die Listen dann mit diesem Typ instanziieren.

Auch bei der Instanziierung eines I/O-Pakets mit *unconstraint record types* ist die zweite Methode zu verwenden. Zwar stört den Compiler eine Instanziierung in diesem Fall nicht, doch treten Laufzeitprobleme auf. Beim *create* eines neuen Files muß im Form-String die maximale Länge des Typs

angegeben werden. Lesen kann man vom File dann nur *constraint* Records mit der richtigen Länge.

Daß die Liste auch bei Lese-Operationen wie GET_FIRST oder GET_NEXT als **in out**-Parameter auftritt, zeigt, daß sich der interne Zustand (nämlich das aktuelle Element) ändert. Dies ist auch der Grund, weshalb diese Operationen nicht als Funktionen angeboten werden können.

Um alle Elemente einer Liste durchzugehen, kann eines der folgenden Programm-Schemata verwenden werden. Dabei ist L eine Listenvariable und E eine Variable vom Elementtyp dieser Liste:

```
(a)  if not IS_EMPTY (L)
       then GET_FIRST (L,E);
            loop <statements>
                 exit when END_OF_LIST (L);
                 GET_NEXT (L,E);
            end loop;
     end if;

(b)  RESET (L);
     while not NO_CURRENT_ELEMENT (L)
     loop
       GET (L,E);
       <statements>
       NEXT (L);
     end loop;
```

4.2 Umrechnung von Attributen

Die Attribute **rowsum, colsum, rowqsum, colqsum** und **rowcolsum**, wie sie auf Seite 13 im konzeptuellen Modell angegeben sind, sagen in dieser Form noch nicht allzuviel über die Bereiche aus, denn in jedem dieser Attribute sind Lage- und Form-Informationen gemischt enthalten. Es ist zweckmäßiger, diese Informationen getrennt vorliegen zu haben. Wir wollen versuchen, Orts-, Orientierungs- und Form-Information zu trennen, indem wir uns von einer physikalischen Analogie leiten lassen. Stellen wir uns einen Bereich als flachen, gleichmäßig mit Masse belegten, starren Körper

vor. Der Ort kann dann durch den **Schwerpunkt**, die Orientierung durch die **Hauptträgheitsachsen** und die Form durch die **Trägheitsmomente** in Richtung der Hauptträgheitsachsen ausgedrückt werden. Bei Bereichen reicht, im Gegensatz zum allgemeinen starren Körper, ein zweidimensionales Koordinatensystem zur Beschreibung aus.

Die Schwerpunktskoordinaten lassen sich nach den Formeln

$$\text{center}_{row} = \frac{\text{rowsum}}{\text{size}}$$

$$\text{center}_{col} = \frac{\text{colsum}}{\text{size}}$$

berechnen.

Um zu verstehen, was die Hauptträgheitsachsen eines starren Körpers sind, muß etwas weiter ausgeholt werden:

Das Trägheitsmoment eines starren Körpers sagt etwas über den "Widerstand" aus, den dieser Körper einer Drehung entgegensetzt. Jedes Trägheitsmoment muß darum auf die Drehachse bezogen werden, um die diese Drehung ausgeführt wird. Die Trägheitsmomente verschiedener Achsen, die alle durch denselben Punkt gehen, können zu einem Trägheitstensor zusammengefaßt werden. Ein Tensor ist eine Größe, die durch eine Matrix bezüglich eines bestimmten Koordinatensystems ausgedrückt wird. Beim Wechsel des Koordinatensystems ändern sich die Werte in der Matrix, der Tensor ist hingegen noch die gleiche Größe. Dies ist ähnlich wie bei Vektoren, die ja auch immer bezüglich eines bestimmten Koordinatensystems angegeben werden. Der Trägheitstensor bezüglich des Bildnullpunktes und des Koordinatensystems entlang der Bildachsen hat den Wert:

$$\mathbf{I} = \begin{pmatrix} \text{colqsum} & -\text{rowcolsum} \\ -\text{rowcolsum} & \text{rowqsum} \end{pmatrix}$$

Um den Tensor translationsinvariant zu machen, wird er auf das jeweilige Schwerpunktskoordinatensystem des Bereichs bezogen. Wir geben diesem Tensor den Index c. Eine Umrechnung des obigen Tensors in das Schwerpunktskoordinatensystem kann mit dem *Steinerschen Satz* durchgeführt werden:

$$\mathbf{I}_c = \begin{pmatrix} I_{11} & I_{12} \\ I_{21} & I_{22} \end{pmatrix} = \mathbf{I} - \begin{pmatrix} \text{center}_{col}^2 & \text{center}_{col}\text{center}_{row} \\ \text{center}_{col}\text{center}_{row} & \text{center}_{row}^2 \end{pmatrix} \cdot \text{size}$$

Das Trägheitsmoment bezüglich einer speziellen Achse in Richtung des Einheitsvektors $\vec{e}$ durch den Schwerpunkt erhält man aus diesem Tensor durch zweimalige Multiplikation:

$$T_{\vec{e}} = \vec{e}^{\,t}\, \mathbf{I}_c\, \vec{e}$$

Dabei sei $\vec{e}^{\,t}$ der transponierte Zeilenvektor des Spaltenvektors $\vec{e}$.

Dies ist eine Ellipsengleichung, die eine besonders einfache Form bekommt, wenn wir das Koordinatensystem entlang der Hauptachsen dieser Ellipse legen, denn dann hat die Matrix Diagonalfrom, d.h. $I_{12} = I_{21} = 0$. Die Gleichung geht dann über in:

$$T_{\vec{e}} = I_{11} e_x^2 + I_{22} e_y^2$$

Diese Achsen nennt man Hauptträgheitsachsen. Die Richtung der Achsen legt die Orientierung des Bereichs fest. Probleme bereitet allerdings der Fall, in dem die Ellipse zu einem Kreis entartet ist. Dann ist jede Achse auch Hauptträgheitsachse. Dies gilt für alle Flächen, die für mindestens zwei Symmetrieachsen das gleiche Trägheitsmoment haben, wie z.B. Kreis oder Quadrat. In diesen Fällen kann man über das Trägheitsmoment keine Orientierung angeben.

Wir haben nun ein körperfestes Koordinatensystem, und alle Größen, die wir auf dieses Koordinatensystem beziehen, sind unabhängig von der Lage des Bereichs, also sowohl translations- als auch rotationsinvariant. Dazu gehören auch die Trägheitsmomente entlang der Hauptachsen. Wie man aus der Ellipsengleichung leicht ablesen kann, sind dies auch das minimale bzw. maximale Trägheitsmoment der Fläche.

Fassen wir nochmals zusammen: Jedem Bereich im Bild wird eine Ellipse zugeordnet. Der Schwerpunkt des Bereichs entspricht dann dem Mittelpunkt der Ellipse. Die Orientierung ist durch die Schräglage der Ellipse bestimmt. Schließlich sagt die Form der Ellipse (Länge der Achsen oder Exzentrizität) etwas über die Form des Bereichs aus.

Nun zur Berechnung dieser verschiedenen Größen. Der Vektor $\mathbf{I}_c \vec{v}$ ist, physikalisch gesehen, der "Drehimpuls" des Bereichs bei einer Drehung mit der Winkelgeschwindigkeit $\vec{v}$. Man kann zeigen, daß der Drehimpulsvektor eines starren Körpers nur dann in Richtung der Drehachse liegt, wenn die Drehachse eine der Hauptträgheitsachsen ist. Dies kann zur Berechnung der Größen ausgenutzt werden. Wir müssen nur die Eigenwertgleichung $\mathbf{I}_c \vec{v}_i = I_{max/min}\ \vec{v}_{max/min}$ lösen. Die Eigenwerte $I_{max/min}$ dieser Gleichung sind dann das minimale bzw. maximale Trägheitsmoment und die Vektoren $\vec{v}_{max/min}$ bestimmen die Richtung der Hauptachsen.

Die Eigenwerte $I_{max/min}$ lassen sich über die Nullstellen des charakteristischen Polynoms

$$\begin{vmatrix} I_{11} - z & I_{12} \\ I_{21} & I_{22} - z \end{vmatrix} = (I_{11} - z)(I_{22} - z) - I_{12}^2$$

berechnen. Es ergeben sich dann die beiden Werte:

$$I_{max/min} = \frac{1}{2} \cdot (I_{11} + I_{22} \pm \sqrt{(I_{11} - I_{22})^2 + 4\, I_{12}^2}\,)$$

Der **Drehwinkel** ϕ zwischen der x-Achse und der Achse mit dem kleinsten Trägheitsmoment ergibt sich nun zu:

$$\tan 2\phi = \frac{2\, I_{12}}{I_{11} - I_{22}}$$

In der Literatur werden noch weitere Größen vorgeschlagen, die die Exzentrizität einer Region ausdrücken sollen. Zunächst bietet sich die mathematische Definition der *numerischen Exzentrizität* einer Ellipse an:

$$\epsilon = \frac{\text{Abstand der Brennpunkte}}{\text{größte Achsenlänge}} = \frac{\sqrt{I_{max}^2 - I_{min}^2}}{I_{max}}$$

Bei einem Kreis gilt: $\epsilon = 0$. Im Extremfall kann ϵ unendlich werden. Doch die Berechnung dieser komplizierten Größe ist nicht unbedingt notwendig. *Ballard + Brown 82* schlagen eine *approximierte Exzentrizität* vor[1]:

$$e = \frac{(I_{11} - I_{22})^2 + 4\, I_{12}^2}{\text{size}} = \frac{(I_{max} - I_{min})^2}{\text{size}}$$

Winston + Horn 84 empfehlen die Berechnung der Größen:

$$\frac{I_{max} + I_{min}}{\text{size}^2} \quad \text{und} \quad \frac{I_{max} - I_{min}}{I_{max} + I_{min}}$$

Duda + Hart 73 vereinfachen noch weiter und schlagen vor, nur noch das Verhältnis von maximalem und minimalem Trägheitsmoment zu berechnen.

[1]Formel nach Berichtigung eines Druckfehlers

4.3 Implementierung einer Schnittstelle

Am Ende dieses Kapitels kommen wir auf die konkrete Aufgabe einer Schnittstellenprogrammierung zurück. Dem Programmierer stehen dafür folgende Pakete zur Verfügung:

KEY_TYPES: Hier sind alle Schlüsseltypen definiert. Zusätzlich gibt es noch Intervalltypen, die jeweils den kleinsten und den größten Schlüssel des Intervalls enthalten.

CHAIN_CODE_TYPES: Dieses Paket enthält alle Typen, die zur Benutzung der Kettencodes gebraucht werden.

SYMBOLIC_OBJECTS: Die Typen für die Punkte-, Kanten- und Bereichsentities sind in diesem Paket definiert. Die Records enthalten sowohl die Attribute als auch diejenigen Verweise, mit denen die Relationen implementiert sind. Als Beispiel sei hier ein Teil der Definition des Bereichs-Records angeführt:

```
type REGION_REC is
       record SIZE              : NATURAL;
              PERIMETER         : NATURAL;
                 ...            :   ...
              CLASS             : CLASS_KEY_TYPE;
              NEXT              : REGION_KEY_TYPE;
              ADJACENT_REGIONS  : INDEX_KEY_TYPE;
                 ...            :   ...
       end record;
```

CLASS_TYPES: Alle Klassen sind als varianter Record implementiert:

```
type CLASS_TYPE (KIND: KIND_OF_CLASSES) is
       record NUMBER_OF_ELEMENTS : NATURAL;
              NEAREST_CLASS      : CLASS_KEY_TYPE;
                 ...             :      ...
              case KIND is
                when REGION_CLASS
                  TYPICAL_REGION : REGION_KEY_TYPE;
                  FIRST_REGION   : REGION_KEY_TYPE;
```

```
                    ...          :       ...
              when   ...
            end case;
        end record;
```

GENERIC_LIST_PROCS: Dieses generische Paket enthält den in Abschnitt 4.1 angegebenen Listentyp. Einige Instanziierungen dieses Pakets werden noch benötigt:

```
with GENERIC_LIST_PROCS, KEY_TYPES, SYMBOLIC_OBJECTS;
  package REGION_KEYS is new package GENERIC_LIST_PROCS
                              (ELEMENT => REGION_KEY_TYPE);
  package REGIONS     is new package GENERIC_LIST_PROCS
                              (ELEMENT => REGION_REC);
```

STORAGE_LEVEL: Alle elementaren Anfrageprozeduren sind in diesem Paket zusammengefasst. Einige seien hier aufgezählt:

```
type DATABASE_TYPE          is limited private;

procedure OPEN_DATABASE     (DB      : in out DATABASE_TYPE);
procedure CLOSE_DATABASE    (DB      : in out DATABASE_TYPE);

procedure OPEN_IMAGE        (DB      : in  DATABASE_TYPE;
                             IMAGE   : in  IMAGE_KEY_TYPE);
procedure CLOSE_IMAGE       (IMAGE   : in  IMAGE_KEY_TYPE);

procedure GET_REGION        (DB      : in  DATABASE_TYPE;
                             KEY     : in  REGION_KEY_TYPE;
                             REG     : out REGION_REC);
function  GET_CLASS_RECORD  (DB      : in  DATABASE_TYPE;
                             KEY     : in  CLASS_KEY_TYPE)
                             return    CLASS_RECORD;
procedure GET_KEYLIST       (DB      : in  DATABASE_TYPE;
                             KEY     : in  INDEX_KEY_TYPE;
                             REGIONS : out REGION_KEYS.LIST);
```

Mit jeder Variablen vom Typ DATABASE_TYPE kann unabhängig von anderen Benutzern auf die Daten zugegriffen werden, wenn die Datenstruktur mit der Prozedur OPEN_DATABASE eröffnet wurde. Sollen mehrere Operationen auf den Daten eines Bildes ausgeführt werden, so empfiehlt es sich, das Bild mit der Prozedur OPEN_IMAGE zu puffern. Die Bedeutung der weiteren Prozeduren und Funktionen wird anhand des folgenden Beispiels erläutert.

Wir wollen nun eine kleine Beispiel-Schnittstelle entwerfen, die die Nachbarschafts- und die Klassen-Relation für Bereiche benutzt. Dazu gehört zunächst ein Record, der die gewünschten Attribute enthält. Hier genügt es einen sehr einfachen Typ zu betrachten:

```
type    SHORT_REGION_TYPE is record SIZE      : NATURAL;
                                    PERIMETER : NATURAL;
                                end record;
package SHORT_REGIONS is new package GENERIC_LIST_PROCS
                                   (ELEMENT => SHORT_REGION_TYPE);
```

Nun wollen wir eine Funktion anbieten, die eine Liste aller Nachbarn einer Region als Ergebnis zurückliefert:

```
function  GET_ADJACENT_REGIONS (KEY  : REGION_KEY_TYPE)
                                return SHORT_REGIONS.LIST;
```

Schließlich soll der Benutzer die Möglichkeit haben, alle Elemente einer Klasse von Bereichen sequentiell durchzuarbeiten:

```
type      CLASS_TYPE       is private;

procedure GET_CLASS (KEY   : in  CLASS_KEY_TYPE;
                     CLASS : out CLASS_TYPE);

procedure GET_FIRST_REGION_OF_CLASS (CLASS : in out CLASS_TYPE;
                                     REGION: out SHORT_REGION_TYPE);

procedure GET_NEXT_REGION_OF_CLASS  (CLASS : in out CLASS_TYPE;
                                     REGION: out SHORT_REGION_TYPE);
```

```
  function  END_OF_CLASS                (CLASS : in  CLASS_TYPE)
                                         return BOOLEAN;

  ------------------------------------------------------------


 private

  type      CLASS_TYPE is record FIRST_ELEMENT   : REGION_KEY_TYPE;
                                 NEXT_ELEMENT    : REGION_KEY_TYPE;
                           end record;
```

Der *body* einer solchen Schnittstelle könnte dann etwa folgendermaßen Aussehen:

```
with KEY_TYPES, SYMBOLIC_OBJECTS, REGION_KEYS, REGIONS, STORAGE_LEVEL;

package body EXAMPLE is

  use KEY_TYPES;

  NULL_CLASS_KEY = CONSTANT CLASS_KEY_TYPE:= UNDEFINED_CLASS;
  DB               : STORAGE_LEVEL.DATABASE_TYPE;

  function  GET_ADJACENT_REGIONS (KEY   : REGION_KEY_TYPE)
                                   return SHORT_REGIONS.LIST is
    REGION    : SYMBOLIC_OBJECTS.REGION_REC;
    SHORT_REG: SHORT_REGION_TYPE;
    REG_KEY   : REGION_KEY_TYPE;
    KEY_LIST  : REGION_KEYS.LIST;
    REG_LIST  : SHORT_REGIONS.LIST;
  begin
    STORAGE_LEVEL.GET_REGION (DB,KEY,REGION);
    STORAGE_LEVEL.GET_KEYLIST(DB,REGION.ADJACENT_REGIONS,KEY_LIST);

    REGION_KEYS.RESET (KEY_LIST);
    while not REGION_KEYS.NO_CURRENT_ELEMENT (KEY_LIST)
    loop
      REGION_KEYS.GET (KEY_LIST, REG_KEY);
```

```
    STORAGE_LEVEL.GET_REGION (DB, REG_KEY, REGION);
    SHORT_REG:= (SIZE      => REGION.SIZE,
                 PERIMETER => REGION.PERIMETER);
    SHORT_REGIONS.INSERT (REG_LIST, SHORT_REG);
    REGION_KEYS.NEXT (KEY_LIST);
  end loop;
  return REG_LIST;
end GET_ADJACENT_REGIONS;

procedure GET_CLASS (KEY   : in  CLASS_KEY_TYPE;
                     CLASS : out CLASS_TYPE) is
  FULL_CLASS: CLASS_TYPES.CLASS_TYPE
                    (KIND => CLASS_TYPES.REGION_CLASS);
begin
  FULL_CLASS:= STORAGE_LEVEL.GET_CLASS_RECORD (DB, KEY);
  CLASS     := (FIRST_ELEMENT   => FULL_CLASS.FIRST_REGION,
                NEXT_ELEMENT    => FULL_CLASS.FIRST_REGION);
end GET_CLASS;

procedure GET_FIRST_REGION_OF_CLASS
                     (CLASS : in out  CLASS_TYPE;
                      REGION: out SHORT_REGION_TYPE) is
  REG_RECORD: SYMBOLIC_OBJECTS.REGION_REC;
begin
  STORAGE_LEVEL.GET_REGION (DB, CLASS.FIRST_ELEMENT, REG_RECORD);
  CLASS.NEXT_ELEMENT:= REG_RECORD.NEXT;
  REGION:= (SIZE      => REG_RECORD.SIZE,
            PERIMETER => REG_RECORD.PERIMETER);
end GET_FIRST_REGION_OF_CLASS;

procedure GET_NEXT_REGION_OF_CLASS
                     (CLASS : in out CLASS_TYPE;
                      REGION: out SHORT_REGION_TYPE) is
  REG_RECORD: SYMBOLIC_OBJECTS.REGION_REC;
begin
  STORAGE_LEVEL.GET_REGION (DB, CLASS.NEXT_ELEMENT, REG_RECORD);
  CLASS.NEXT_ELEMENT:= REG_RECORD.NEXT;
  REGION:= (SIZE      => REG_RECORD.SIZE,
            PERIMETER => REG_RECORD.PERIMETER);
```

```
  end GET_NEXT_REGION_OF_CLASS;

  function  END_OF_CLASS (CLASS : in  CLASS_TYPE)
                          return BOOLEAN is
  begin
    return (CLASS.NEXT_ELEMENT = NULL_CLASS_KEY);
  end END_OF_CLASS;

begin
  OPEN_DATABASE (DB);
end EXAMPLE;
```

Kapitel 5

Transformation der Regionskarte

Der Regionsfinder liefert als Ergebnis eine sog. Regionskarte ab. Dies ist ein Array im Bildformat; hier sind aber Regionsnummern statt der Grauwerte eingetragen. Die Information (Randform, Größe, Flächenträgheitsmomente, Nachbarschaft, usw.), die in der Regionskarte implizit enthalten sind, müssen nun in expliziter Form in die Datenstruktur eingetragen werden. Dazu bekommt die Transformationsprozedur eine eigene Sicht auf die Datenstruktur, die ihrer Aufgabe möglichst gut angepaßt ist. In diesem Abschnitt wird diese Sicht und der Algorithmus erläutert.

5.1 Spezifikation der Eingabe

An die Regionskarte werden folgende Bedingungen gestellt:

- Alle Pixel einer Region müssen über einen Pfad miteinander verbunden werden können, dessen Pixel alle in dieser Region liegen. Dabei sollen zwei aufeinanderfolgende Pixel des Pfades über eine gemeinsame Elementarkante direkt benachbart sein. Üblicherweise wird diese Bedingung Vierer-Zusammenhang genannt.

- Die Regionen sind durchnummeriert. Die kleinste mögliche Nummer ist 1. Die größte Regionsnummer muß bekannt sein. Die Reihenfolge der Nummern in der Karte ist völlig beliebig.

5.2 Schnittstelle zur Datenstruktur

Folgende Eigenschaften sollen berechnet werden (siehe Seite 13):

- Größe, d.h. die Anzahl der Pixel einer Region.
- Zeilen- bzw. Spaltensumme, d.h. $\sum_{(x,y)\in\Re} y$ und $\sum_{(x,y)\in\Re} x$.
- Quadratische Zeilen-, Spalten- und Gemischt-Summe, d.h. $\sum_{(x,y)\in\Re} x^2$, $\sum_{(x,y)\in\Re} y^2$ und $\sum_{(x,y)\in\Re} xy$.
- Umfang, d.h. die Anzahl der Elementarkanten zu Nachbarbereichen.
- Umschreibendes Rechteck, d.h. die minimale und maximale Zeilen- und Spalten-Nummer.

Außerdem sollen die folgenden Beziehungen an die Datenstruktur übergeben werden:

- Zu jedem Bereich eine Liste seiner Nachbarbereiche und der Grenze mit ihnen.
- Der Zusammenhang der Grenzen eines Bereichs. Auch muß zumindest eine der Außengrenzen jedes Bereichs gekennzeichnet sein. Mit diesen Listen wird dann die **Teil-von**-Relation berechnet. Dazu siehe den Abschnitt 5.5.

Schließlich sollen die Bereichsgrenzen folgende Informationen enthalten:

- Eine Liste von Anfangs- und Endpunkten, sowie der Kettencode. Weil bei der Berechnung die Länge des Kettencodes noch nicht bekannt ist, werden Arrays mit einer konstanten Anzahl von Richtungselementen verkettet. Somit läßt sich die Länge relativ einfach vergrößern. Die einzelnen Richtungen sind so kodiert, daß eine Drehung um 180 Grad durch eine einfache NOT-Operation durchgeführt werden kann.

5.3 Hilfsstrukturen

Sowohl bei der Transformation der Regionskarte, als auch bei der Transformation von Punkten und Kanten, ist es wichtig, über Regionsnummern auf die Schlüssel von Regionen und Grenzen zugreifen zu können. Dazu wird während der Regionstransformation folgende Abbildungen aufgebaut:

Regions-Nummer $\longrightarrow$ Regions-Schlüssel

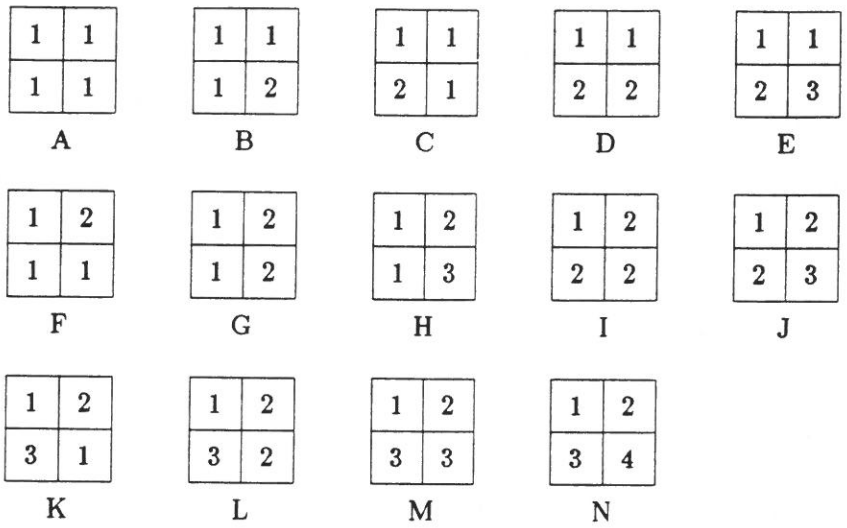


Abbildung 5.1: Auflistung aller möglichen Fälle einer lokalen Umgebung

$$\text{Regions-Nummer}^2 \longrightarrow \text{Segment-Schlüssel}$$

Eine Grenze wird also immer über die Nummern der Regionen eindeutig identifiziert, die sie begrenzt. Dies hat leider zur Folge, daß eine Grenze aus mehreren nicht zusammenhängenden Teilen bestehen kann.

Die Repräsentation der Abbildungen ist in einem Paket verborgen (siehe auch die Übersichtsabbildung 1.1; dort ist dieses Paket unter dem Namen *Abbildungen* eingezeichnet). Bisher wird die Abbildung durch einfache Arrays implementiert.

Eine spätere Überlegung zeigt, daß die Anzahl der Grenzen nur linear mit der Anzahl der Regionen wächst. Daher ist eine Reduzierung des Speicherplatzbedarfs möglich, wenn die Abbildung anders repräsentiert wird. Bisher reichte der Speicherplatz auch für die einfache Implementierung aus.

5.4 Entwicklung des Algorithmus

Die Transformation der Darstellung soll in möglichst nur einem zeilenweisen Durchlauf durch die Regionskarte erfolgen. Dabei muß der Algorithmus eine lokale Umgebung von 4 Pixeln betrachten. Bei den genannten Beschränkungen der Regionskarte lassen sich 14 verschiedene solcher Gruppen von Pixeln unterscheiden. In Abbildung 5.1 sind alle Fälle systematisch

aufgelistet. Die Zahlen stehen für beliebige Regionsnummern. Das Pixel unten-rechts wird im folgenden als *aktuelles* Pixel bezeichnet. Im folgenden wird über den Buchstaben unter den Bildern auf den entsprechenden Fall verwiesen.

Je nachdem, was für ein Fall gerade vorliegt, müssen folgende Operationen durchgeführt werden:

1. **Wechsel der Bereichsnummer in horizontaler Richtung:**

   Fälle laut Abbildung: B,C,E,G,H,J,K,L,N

   Falls ein Wechsel der Bereichsnummer des aktuellen Pixels stattfindet, müssen die Eigenschaften Größe, Zeilen- bzw. Spaltensumme sowie die Summen zweiter Ordnung und das umschreibende Rechteck aktualisiert werden. Dabei sind die Formeln

   $$\sum_{i=1}^{n} i = \frac{n\,(n+1)}{2}$$

   und

   $$\sum_{i=1}^{n} i^2 = \frac{n\,(n+1)\,(2n+1)}{6} = \frac{2n+1}{3} \sum_{i=1}^{n} i$$

   nützlich.

   Bei neu auftauchender Regionsnummer oder neuer Kombination zweier Regionsnummern zu einer Grenze, müssen neue Schlüssel erzeugt und die Hilfsstruktur aktualisiert werden. Die Grenze, bei der eine Region neu auftritt, ist Außengrenze dieser Region und muß entsprechend gekennzeichnet werden.

   Die aktuelle Region wird gekennzeichnet, um später erkennen zu können, welche Bereiche in der letzten Zeile noch aufgetaucht sind.

2. **Eintragen eines senkrechten bzw. waagerechten Kantenelements:**

   Fälle laut Abbildung: waagerecht: B,D,E,F,H,J,K,M,N
   senkrecht : B,C,E,G,H,J,K,L,N

   Es wird der Kettencode-Teil der entsprechenden Grenze gesucht, in der das Kantenelement angefügt werden kann. Dabei ist zu beachten,

Fälle	Richtung	Anknüpfungs-Punkt	neue Richtung	Anfang/Ende neu	Verschmelzen möglich
	*down*	Anfang	*up*	$(col, row + 1)$	nein
	*down*	Ende	*down*	$(col, row + 1)$	nein
	*down*	Anfang	*down*	(col, row)	nein
	*down*	Ende	*up*	(col, row)	nein
	*right*	Anfang	*left*	$(col + 1, row)$	ja
	*right*	Ende	*right*	$(col + 1, row)$	ja
	*right*	Anfang	*right*	(col, row)	ja
	*right*	Ende	*left*	(col, row)	ja

In der ersten Spalte der Tabelle ist die Situation symbolisch gezeigt. Der dicke Strich zeigt das einzutragende Kantenelement. Die Pfeile deuten Kettencode-Teile an, an die das Kantenelement angehängt werden könnte. Die zweite Spalte zeigt die Richtung der Elementarkante und die dritte Spalte schließlich den Punkt, an dem die Elementarkante in einen Kettencode eingehängt werden könnte. Die weiteren Spalten beschreiben, was nun zu tun ist: Eventuell muß die Kantenrichtung beim Einhängen umgedreht werden. Der Anfangs- bzw. Endpunkt des Kettencodes muß geändert werden. Schließlich besteht eventuell die Möglichkeit, daß das elementare Kantenstück zwei bisher getrennte Kettencode-Teile der gleichen Grenze miteinander verbindet.

Tabelle 5.1: Auflistung aller möglichen Fälle einer Kantenumgebung

daß eine waagerechte Kante an beiden Enden an ein schon vorhandenes Kantenstück angrenzen kann. Die Richtung des neuen Kantenelements hängt davon ab, an welchem Ende des Kantenstücks es angefügt wird. Schließlich kann das Kantenelement zwei Kantenstücke zu einem zusammenhängenden neuen Kantenstück verschmelzen. Auch die Anfangs- bzw. Endkoordinaten des entsprechenden Kantenstücks müssen geändert werden. Alle diese Möglichkeiten sind in Tabelle 5.1 zusammengefasst.

3. **Verbindung zwischen verschiedenen Kantenstücken:**

   Fälle laut Abbildung: E,H,L,M,N

   Um später die Teil-von-Relation berechnen zu können, muß bekannt sein, welche Grenzen einer Region miteinander in Verbindung stehen. Damit zwei verschiedene Grenzen aufeinandertreffen können, müssen mindestens drei verschiedene Regionen in der lokalen Umgebung vorhanden sein. Dies ist allerdings keine hinreichende Bedingung, wie die Fälle J und K aus Abbildung 5.1 zeigen. In diesen Fällen kann man allerdings auch geteilter Meinung sein. Denn umrundet man z.B. im Fall J die Regionen 1 und 3, so sind die Grenzen zwischen 1 und 2 sowie 2 und 3 nicht verbunden. Umrundet man jedoch Region 2, so sind diese Grenzen doch miteinander verbunden.

   Diese Merkwürdigkeit kommt von der inneren Widersprüchlichkeit des Vierer-Zusammenhangs. Diagonal gegenüberliegende Pixel sind nicht zusammenhängend, obwohl sie nicht durch einen Pixelpfad (der dem Vierer-Zusammenhang genügt) getrennt werden können, denn die zweite Diagonale ist ebenfalls nicht zusammenhängend.

   Unsere Lösung des Konflikts, nämlich die Grenzen als nicht-verbunden zu betrachten, entspricht dem Zusammenhang der Diagonalpixel mit gleicher Regionsnummer. Dies erscheint am sinnvollsten, wenn man bedenkt, daß die Pixel einer Region ähnlich sein sollten. Die Konsequenz daraus ist, daß Region 2 (im Fall J) mehrere Ränder, also ein Loch, hat. Entweder liegt Region 1 oder Region 3 innerhalb von Region 2.

Abbildung 5.2 zeigt die Einteilung der Fälle von Abbildung 5.1 nochmals als Mengendiagramm.

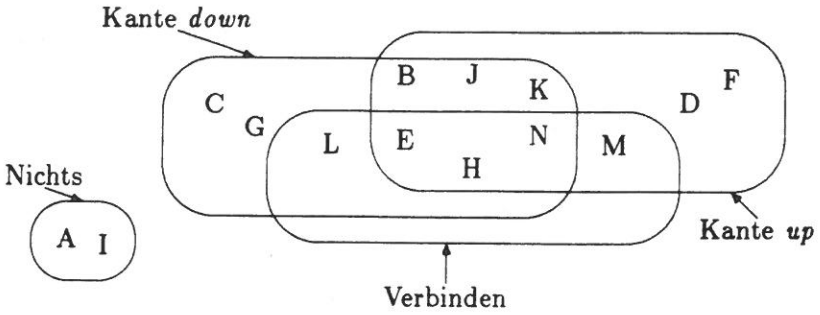


Abbildung 5.2: Mengendiagramm der Einteilung der Fälle einer lokalen Umgebung

Es ergibt sich im ganzen folgendes Bild der Prozedur: Die Regionskarte wird zeilenweise gelesen. Für jedes Pixel werden diejenigen der aufgezählten Operationen ausgeführt, die der lokalen Umbegung entsprechen. Ist eine Zeile beendet, so werden diejenigen Regionen und Grenzen ausgesondert, die bereits fertig berechnet sind. Dies kann dadurch festgestellt werden, daß die Region in der letzten Zeile nicht mehr vorgekommen ist. Bei Grenzen muß eine der angrenzenden Regionen beendet worden sein, bevor sie ausgesondert werden kann.[1]

Die Schnittstelle zur Datenstruktur übernimmt es dann, die Beziehungen in geeigneter Form in die Datenstruktur einzutragen.

5.5 Berechnung der Teil-von-Relation

Es muß noch angedeutet werden, wie aus den berechneten Daten die Teil-von-Relation erhalten werden kann. Dazu sehen wir uns zunächst noch einmal die Definition von Seite 15 an:

Es sei $Pf_{\mathcal{A}}$ die Menge der Pfade, die vom Rand des Bildes bis zur Region $\mathcal{A}$ laufen. Ein Pfad P besteht aus einem Tupel von Pixeln, wobei aufeinanderfolgende Pixel jeweils zusammenhängen. Es gilt:

$$\text{Innerhalb}(\mathcal{A},\mathcal{B}) \overset{\text{def}}{\Longleftrightarrow} \bigwedge_{P \in Pf_{\mathcal{A}}} \bigvee_{p_i \in P} : p_i \in \mathcal{B}$$

[1] Es reicht nicht aus, daß kein Kantenelement mehr angefügt wurde, denn Grenzen müssen ja nicht zusammenhängend sein.

und

$$\text{Teil-von}(\mathcal{A}, \mathcal{B}) \stackrel{\text{def}}{\Longleftrightarrow} \text{Innerhalb}(\mathcal{A}, \mathcal{B}) \wedge \text{Benachbart}(\mathcal{A}, \mathcal{B})$$

Wir müssen also nur die Nachbarn einer Region $\mathcal{B}$ untersuchen, um alle Elemente der Relation Teil-von $(x, \mathcal{B})$ zu erhalten. Falls die Region $\mathcal{B}$ eine Region $\mathcal{A}$ enthält, muß sie mindestens zwei Ränder haben. Dies folgt sofort aus der Definition der Innerhalb-Relation. Einer dieser Ränder ist der Außenrand. Ihn erkennt man daran, daß man vom Rand des Bildes kommend immer zunächst auf ihn trifft. Die Eigenschaft einer Grenze, außen zu liegen, vererbt sich nun auf alle Grenzen dieser Region, die mit jener Grenze verbunden sind. Alle Regionen, die dann durch eine Grenze mit der betrachteten Region verbunden sind, die nicht Außengrenze ist, erfüllen die Teil-von-Relation mit Region $\mathcal{B}$.

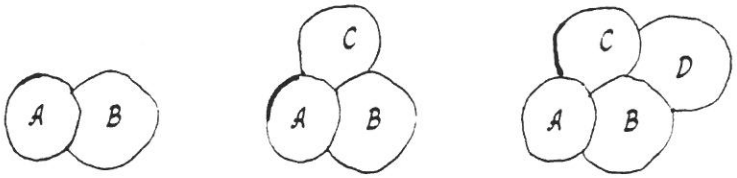
Man vergleiche diesen Algorithmus mit der Methode von [*Kraasch + Zach 78*].

5.6 Zur Wahl der Puffergrößen

Soll ein neues Bild in die Datenstruktur eingetragen werden, so muß die maximale Anzahl der Entities jeden Typs (Bereiche, Kanten, Punkte) angegeben werden. Die Datenstruktur puffert diese Entities solange, bis das Bild fertig eingetragen ist, weil Zugriffe auf diese Entities jederzeit effizient möglich sein müssen.

Hier tritt nun ein Problem auf. Wir wissen zwar die maximale Anzahl der Regionen (durch die maximale Regionsnummer), aber die Anzahl der Grenzen zwischen ihnen ist unbekannt. Da eine Grenze durch zwei verschiedene Regionsnummern eindeutig bestimmt ist, kann man schnell die folgende obere Abschätzung gewinnen. Sei n die Anzahl der Regionen im Bild (ohne die Außenregion außerhalb des Bildes), dann können höchstens $\frac{n(n+1)}{2} = O(n^2)$ Grenzen zwischen diesen Regionen oder mit der Außenregion auftreten.

An Beispielen kann man aber sofort erkennen, daß diese obere Schranke für große n bei weitem nicht erreicht wird. Dies muß auch nicht verwundern, denn wir haben die Tatsache, daß die Regionskarte planar ist, noch nicht ausgenutzt. Konstruieren wir also eine Karte so, daß möglichst viele Grenzen entstehen (siehe Abbildung 5.3). Wir beginnen mit zwei benachbarten Regionen $\mathcal{A}$ und $\mathcal{B}$. Es entstehen 3 Grenzen (2 Grenzen mit der Außenregion). Zur Konstruktion von Region $\mathcal{C}$ verbinden wir einen Punkt auf der Außengrenze von $\mathcal{A}$ mit einem Punkt auf der Außengrenze von $\mathcal{B}$.



$n = 2$ 3 Bögen | $n = 3$ 6 Bögen | $n = 4$ 9 Bögen

Abbildung 5.3: Erzeugung einer Regionskarte mit vielen Grenzen

Es entstehen 3 neue Grenzen, nämlich zwischen A und C, B und C sowie C und der Außenregion. Um eine neue Region hinzuzufügen, muß jeweils ein neuer Bogen gezogen werden. Ein Bogen ist dabei als Linie von einem Punkt zu einem weiteren Punkt anzusehen. Ein Punkt soll ein Ort sein, an dem mehrere Bögen zusammentreffen. Die Bögen entsprechen also nicht unseren Grenzen, aber es gibt immer mindestens soviele Bögen wie Grenzen. Wird ein neuer Bogen gezeichnet, so entstehen höchstens 3 Bögen mehr. Nämlich der eine neu gezeichnete sowie vier andere, durch die Zerlegung der beiden Bögen, welche von dem neuen Bogen berührt werden (die zerlegten Bögen existieren dann nicht mehr). Wir erhalten also nach $(n-2)$-maligem Zeichnens eines Bogens genau n Regionen, sowie $3+3(n-2) = 3(n-1)$ Bögen. Die Anzahl der Punkte, die in etwa dem Auftreten der Verbindungs-Prozedur entspricht, beträgt übrigens $2 + 2(n-2) = 2(n-1)$.

Als neue obere Schranke für die Anzahl der Grenzen erhalten wir also die lineare Funktion: $3(n-1) = O(n)$. Dies entspricht einer wesentlichen Reduzierung von Speicherplatz. Die bisherigen Ergebnisse zeigen, daß die obere Schranke zu etwa 90% erreicht wird. Die Gründe, daß sie nicht voll erreicht wird, sind die folgenden 3 Situationen:

1. mehr als 3 Grenzen stoßen an einem Punkt zusammen (Fall N in Abbildung 5.1)

2. eine Region ist in einer anderen Region enthalten und

3. es gibt eine Grenze, die aus mehreren Teilstücken besteht.

Diese obere Schranke ist auch aus der Graphentheorie bekannt, siehe z.B. [*Busacker + Saaty 68*], an deren Darstellung die obige Herleitung angelehnt ist.

Kapitel 6

Relationen zwischen gleichartigen Merkmalen

Die Datenstruktur erhält ihre Daten von mehreren unabhängig voneinander arbeitenden Merkmalsfindern. Die Ergebnisse dieser verschiedenen Algorithmen müssen nun in konsistenter Weise miteinander in Beziehung gesetzt werden. Eine Querverbindung einfacher Art ist die Relation **Liegt-in** (siehe Seite 15). Diese Beziehung ist hierarchisch und kann durch einen einfachen Zugriff auf die Regionskarte und anschließender Umwandlung der Regionsnummer in einen Regionsschlüssel sehr einfach hergestellt werden.

Anders sieht es mit den Beziehungen zwischen gleichartigen Merkmalen der verschiedenen Segmentationsalgorithmen aus. Hierzu gehören die Relationen **Punkteentsprechung** und **Kantenentsprechung** (siehe Seite 16). Im Gegensatz zur Liegt-in-Relation sind diese Relationen letztlich nur durch inhaltliche Interpretation zu erhalten. In dieser frühen Phase der Verarbeitung können nur einfache Heuristiken benutzt werden, um dieses Problem zu lösen. Spätere Algorithmen sollten sich der Tatsache bewußt sein, daß diese Relationen erhebliche Unsicherheiten enthalten. Man kann deshalb die Relationen zunächst auch über die Algorithmen definieren, die sie erzeugen. Jedes weitere Verarbeitungsprogramm kann dann selber die Aussagekraft dieser Beziehungen beurteilen und Evidenz für eine inhaltliche Interpretation daraus ableiten.

In diesem Kapitel werden nun die Heuristiken vorgestellt, mit denen wir diese Relationen herstellen wollen.

6.1 Relationen zwischen Punkten

Es gibt zwei Arten von Punkten: markante Punkte und Schnitt-Punkte. Eine Beziehung zwischen diesen Punkten wird über den euklidschen Abstand hergestellt. Die Relation Punkteentsprechung ordnet einfach jedem Schnittpunkt den nächstliegenden markanten Punkt zu. Die Annahme, die hier gemacht wird ist, daß Punkte innerhalb eines Bildes auch von verschiedenen Verfahren an denselben Stellen gefunden werden sollten.

Der Ablauf beim Eintragen in die Datenstruktur ist nun der folgende:

- Einlesen der markanten Punkte. Dabei Aufbau einer Tabelle, die den effizienten Zugriff auf einen Punkt über Ortskoordinaten erlaubt.
- Einlesen der Schnittpunkte. Zu jedem Punkt wird in der Tabelle nach dem nächstliegenden markanten Punkt innerhalb einer bestimmten Entfernung gesucht. Übergabe dieses Punktes an die Kantensicht der Datenstruktur.

Das skizzierte Verfahren zeigt eine gewisse Unsymmetrie. Zu jedem Schnitt-Punkt wird höchstens ein markanter Punkt gefunden, aber es wird nicht verhindert, daß ein markanter Punkt mehreren Schnitt-Punkten entspricht. Dies hat Rückwirkungen auf die Darstellung der entsprechenden Relation.

Im folgenden wird der Aufbau der Punkte-Tabelle und der darauf arbeitende Suchalgorithmus genauer erläutert.

Aufbau der Punkte-Tabelle

Die Punkte-Tabelle, wie sie in Abbildung 1.1 eingezeichnet ist, stellt im wesentlichen zwei Operationen zur Verfügung:

- Eintragen eines Punktes mit Schlüssel und Koordinate
- Suchen des, zu einer angegebenen Koordinate, innerhalb eines Abstandes von n Pixeln, nächsliegenden eingetragenen Punkts. Wird innerhalb des Abstandes kein Punkt gefunden, so wird ein Nullwert zurückgeliefert.

Das Bild wird dazu in quadratische Blöcke zu je n^2 Pixeln aufgeteilt. Diese Blöcke werden von links nach rechts und oben nach unten durchnummeriert. Jeder einzutragende Punkt wird unter seiner Blocknummer abgelegt.

Wird der nächste eingetragene Punkt zur Koordinate (row, col) gesucht, so wird zunächst im entsprechenden Block nachgesehen. Findet sich dort

ein Punkt P_1 mit dem Abstand r zur angegebenen Koordinate, so müssen noch all diejenigen Blöcke durchgesehen werden, die von dem Kreis mit Mittelpunkt (row, col) und Radius r überstrichen werden. Wird dort ein weiterer Punkt gefunden, so ersetzt er P_1, wenn er einen kleineren Abstand zur Koordinate (row, col) hat. Wurde kein Punkt gefunden, so müssen natürlich alle 8 umgebenden Blöcke durchsucht werden.

In der Praxis erweist es sich als einfacher, statt der von einem Kreis überstrichenen Blöcke, alle diejenigen zu durchsuchen, die von einem Quadrat mit r als Seitenlänge überstrichen werden. Dazu wird zunächst festgestellt, ob das Quadrat oben, unten, rechts oder links über den Rand des Blocks hinausragt. Dabei sollte Fließkomma-Arithmetik vermieden werden. Darum rechnen wir nur mit den quadratischen Abständen, die ja alle ganzzahlig sind. Die Bedingung, daß die obere Grenze des Blocks überschritten wird, kann dann folgendermaßen umgeformt werden (dabei wird $r > 0$ implizit verwendet):

$$\begin{array}{lccc} & \mathit{Blocknr}(row - r, col) & < & \mathit{Blocknr}(row, col) \\ \Longleftrightarrow & (row - r) \text{ div } n & < & row \text{ div } n \\ \Longleftrightarrow & (row \bmod n) - r & < & 0 \\ \Longleftrightarrow & r & > & (row \bmod n) \\ \Longleftrightarrow & r^2 & > & (row \bmod n)^2 \end{array}$$

Die restlichen Bedingungen können ebenso umgeformt werden, man erhält dann insgesamt:

$$\begin{array}{lccl} \text{oben}: & r^2 & > & (row \bmod n)^2 \\ \text{unten}: & r^2 & \geq & (n - (row \bmod n))^2 \\ \text{links}: & r^2 & > & (col \bmod n)^2 \\ \text{rechts}: & r^2 & \geq & (n - (col \bmod n))^2 \end{array}$$

Nun läßt sich leicht feststellen, ob ein Block von dem Quadrat erfaßt wird.

6.2 Relationen zwischen Kanten

Es gibt zwei Arten von linienartigen Objekten. Zum einen sind dies die Grenzen zwischen den Regionen, die als eigenständige Objekte in der Datenstruktur abgelegt sind. Zum anderen gibt es, von einem speziellen Operator gesuchte Kanten, die die Unstetigkeiten im Bild angeben. Es sollte nun Kanten geben, die den Grenzen der Regionen entsprechen.

Hanson + Riseman 78 definieren einen k-Durchschnitt, um Kanten und Regionsgrenzen miteinander in Beziehung zu setzen. Eine Kante und eine Grenze sind identisch, wenn Kante und Grenze sich höchstens um k Pixel voneinander entfernen. In dieser Arbeit wird ein ähnliches Vorgehen vorgeschlagen:

Zu jeder Kante werden diejenigen Pixel jeder Seite betrachtet, die in einer Umgebung von k Pixeln liegen. Für jede Seite wird nun ein Histogramm über die vorkommenden Regionen aufgebaut. Das Maximum des Histogramms gibt dann die am häufigsten auf der entsprechenden Seite vorkommende Region an. Die Maxima der beiden Seiten definieren nun eventuell eine Grenze. Falls diese Grenze existiert, wird sie mit der ursprünglichen Kante in Beziehung gesetzt.

Hier tritt das gleiche Problem wie bei den Punkten auf. Zu jeder Kante gibt es höchstens eine entsprechende Grenze, aber umgekehrt kann es zu jeder Grenze mehrere Kanten geben. Dies muß bei der Implementation der Relation berücksichtigt werden.

Dem obigen Vorgehen liegt die Annahme zugrunde, daß eine Kante nur einer Grenze entsprechen sollte. Man kann sich aber auch Kanten vorstellen, die entlang mehrerer Grenzen hintereinander laufen. Falls sich hier ein Problem ergibt, muß über eine Erweiterung nachgedacht werden.

Kapitel 7

Zusammenfassung und Ausblick

Das vorgestellte Konzept einer Datenstruktur für Bildfolgen ist im wesentlichen implementiert. Die Schnittstellen zum Regions- und Punktefinder sind getestet. Offen ist hier nur noch die Frage, welche Attribute an die Datenstruktur übergeben werden sollen. Dies hängt wesentlich von den Anforderungen ab, die die nachfolgenden Verarbeitungsalgorithmen an die Datenstruktur haben.

Ziel der Arbeit war es, eine Schnittstelle zwischen den Merkmalsfindern und den weiterverarbeitenden bildverstehenden Algorithmen zu schaffen. Folgende Punkte wurden mit der Implementation der Datenstruktur erreicht:

- Sie bietet einfache Schnittstellen zu den Merkmalsfindern.
- Es können große Datenmengen abgelegt werden.
- Paralleler Zugriff auf die Daten ist möglich.
- Die Implementation von verschiedenen Sichten auf die abgelegten Daten wird unterstützt.

Der erste Punkt beinhaltet die Aufbereitung der Daten. So muß sich der Bereichsfinder nicht mit topologischen Problemen auseinandersetzen. Das zweite Ziel wurde durch Abspeicherung der Daten auf Platte erreicht. Das dritte Ziel wird dadurch berücksichtigt, daß Zugriffe auf die Daten von der Datenverwaltung synchronisiert werden.

Der letzte Punkt wird dadurch unterstüzt, daß dem Benutzer bei der Implementierung einer Datensicht folgende Probleme abgenommen werden:

- Speicherung variabel langer Attribute (Kettencode).
- Anlegen und Verwalten von Dateien.
- Pufferung von Daten für schnellen Zugriff.
- Realisierung von parallelem Zugriff.

Experimente müssen nun zeigen, ob die in der Einleitung gemachten Voraussetzungen auch eingehalten werden können. Hier sei nur an die Voraussetzung des 2-Phasenablaufs (Seite 4) erinnert. Wenn bei der Korrespondenzanalyse Schwierigkeiten auftreten, könnte es sinnvoll sein, ein Bild zu resegmentieren, d.h. in Bildteilen nach weiteren Merkmalen mit veränderten Parametern zu suchen. In einem solchen Fall sind größere Änderungen in der Datenstruktur vorzunehmen. So könnte man z.B. das resegmentierte Bild zusammen mit einem Hinweis auf die alte Segmentierung speichern.

Erprobt wurde die Datenstruktur mit speziellen Testprogrammen sowie Anwendungen wie *Färben der Regionskarte* und *Menügesteuertes Anfrageprogramm mit graphischer Ausgabe*. Im Moment wird an dem Entwurf weiterer externer Schnittstellen gearbeitet. Dies geschieht parallel zur Implementation von Wissensquellen zur Lösung des Korrespondenzproblems.

Ich danke meiner Betreuerin Dr. Leonie Dreschler-Fischer für die Hilfe bei der Erstellung dieser Arbeit. Bedanken möchte ich mich auch bei Dr. Wolfgang Benn, der die Zeit hatte, mir meine Fragen zu seiner Arbeit zu beantworten. Den Mitgliedern der SISSY-Projektgruppe danke ich für die gute Zusammenarbeit und Carsten Schröder für die kritische Durchsicht des Textes.

Literatur

Ballard + Brown 82 : *Computer Vision,* D.A. Ballard und C.M. Brown, Englewood Cliffs, New Jersey, Prentice Hall, 1982.

Barrow + Popplestone 71 : *Relational description in picture processing,* H.G. Barrow und R.J. Popplestone, Machine Intelligence **6**, University Press Edinburgh, 1971, 377–396.

Bartsch u.a. 86 : *Merkmalsdetektion in Farbbildern als Grundlage der Korrespondenzanalyse in Stereo-Bildfolgen,* Thomas Bartsch, Leonie Dreschler-Fischer und Carsten Schröder, 9. DAGM-Symposium "Mustererkennung", Paderborn, 30.9. - 2.10.1986; G. Hartmann (Hrsg.); Informatik-Fachberichte Bd. 125; Springer-Verlag, Berlin, Heidelberg, New York, 1986, 94–98.

Benn 86a : *Query-by-Structure-Example: Objektorientierter Datenbankzugriff für bildbeschreibende Strukturen,* Wolfgang Benn, 9. DAGM-Symposium "Mustererkennung", Paderborn, 30.9. - 2.10.1986; G. Hartmann (Hrsg.); Informatik-Fachberichte Bd. 125; Springer-Verlag, Berlin, Heidelberg, New York, 1986, 154–158.

Benn 86b : *Dynamische nicht-normalisierte Relationen und symbolische Bildbeschreibung,* Wolfgang Benn, Informatik-Fachberichte Bd. 128; Springer-Verlag, Berlin, Heidelberg, New York, 1986.

Benn + Radig 82 : *Entwurf einer relationalen Datenbank zur Unterstützung der Analyse von Bildfolgen,* Wolfgang Benn und Bernd Radig, Mitteilung IfI-HH-M-116/82, Fachbereich Informatik, Universität Hamburg, 1982.

Bertelsmeier + Radig 78 : *Context-Guided Analysis of Scenes with Moving Objects,* R. Bertelsmeier und B. Radig, Bericht IfI-HH-B-41/78, Fachbereich Informatik, Universität Hamburg, 1978.

Busacker + Saaty 68 : *Endliche Graphen und Netzwerke, Eine Einführung mit Anwendungen,* Robert G. Busacker und Thomas L. Saaty, R. Oldenbourg, München, Wien, 1968.

Dreschler-Fischer 86 : *A Blackboard System for Dynamic Stereo Matching,* Leonie S. Dreschler-Fischer, International Conference on Autonomous Systems, Amsterdam, Dezember 8–10, 1986, im Druck.

Duda + Hart 73 : *Pattern Classification and Scene Analysis,* Richard O. Duda und Peter E. Hart, John Wiley & Sons, New York, London, Sydney, Toronto, 1973.

Goos + Wulf 81 : *DIANA Reference Manual,* G. Goos und W.A. Wulf (Ed.), Universität Karlsuhe, Institut für Informatik II, Bericht 1/81, 3/81.

Habermann + Perry 83 : *Ada for Experienced Programmers,* A. Nico Habermann und Dewayne E. Perry, Addison-Wesley Reading, Massachusetts, 1983.

Hanson + Riseman 78 : *Segmentation of Natural Scenes,* Allen R. Hanson und Edward M. Riseman, in: A.R. Hanson, E.M. Riseman (Hrsg.): Computer Vision Systems; Academic Press, New York, 1978, 129–163.

Kraasch + Zach 78 : *SERF - Eine Untersuchung zur Segmentation und symbolischen Beschreibung von Fernsehbildern,* Reinhard Kraasch und Willie Zach, Mitteilung IfI-HH-M-59/78, Fachbereich Informatik, Universität Hamburg, 1978.

Marr + Hildreth 80 : *A Theory of Edge Detection,* David Marr und Ellen C. Hildreth, *Proc. Roy. Soc. of London* **B207**(1980), 187–217.

Mühle + Radig 81 : *Entwurf eines Datenbanksystems zur Untersuchung der Analyse von Bildfolgen,* K. Mühle und B. Radig, 4. DAGM-Symposium "Modelle und Strukturen", Hamburg, Okt. 1981; B. Radig (Hrsg.); Informatik-Fachberichte Bd. 49; Springer-Verlag, Berlin, Heidelberg, New York, 1981, 144–150.

Radig 82 : *Symbolische Beschreibung von Bildfolgen I: Relationengebilde und Morphismen,* Bernd Radig, Mitteilung IfI-HH-M-90/82, Fachbereich Informatik, Universität Hamburg, 1982.

Radig 84 : *Image Sequence Analysis using Relational Structures,* Bernd Radig, *Pattern Recognition* **17**(1984), 161–167.

Schlageter + Stucky 83 : *Datenbanksysteme: Konzepte und Modelle,* Gunter Schlageter und Wolffried Stucky, Teubner Studienbücher Informatik Band 37, Stuttgart, 1983.

Sties u.a. 76 : *Organization of Object Data for an Image Information System,* M. Sties, B. Sanyal und K. Leist, 3rd. Intern. Joint Conf. on Pattern Recogn. 1976, Coronado, CA, 863–869.

Tamura + Yokoya 83 : *Image Database Systems: A Survey,* Hideyuki Tamura, Naokazu Yokoya, *Pattern Recognition* **17**(1984), 29–43.

Winston + Horn 84 : *Lisp, second edition,* Patrick Henry Winston und Berthold Klaus Paul Horn, Addison-Wesley Reading, Massachusetts, 1984.

Wirth 79 : *Algorithmen und Datenstrukturen,* Niklaus Wirth, Teubner Studienbücher Informatik Band 31, Stuttgart, 1979.