Dissertation
submitted to the
Combined Faculties for the Natural Sciences and for Mathematics
of the Ruperto-Carola University of Heidelberg, Germany
for the degree of
Doctor of Natural Sciences

Put forward by

Dipl.-Phys. Björn Andres
Born in Mülheim an der Ruhr, Germany

Oral examination: July 20th, 2011, 10:30 a.m.

Automated Segmentation of Large 3D Images of Nervous Systems Using a Higher-order Graphical Model

Referees:	Prof. Dr. Fred A. Hamprecht
	Prof. Dr. Winfried Denk

Abstract

This thesis presents a new mathematical model for segmenting volume images (Chapter 3). The model is an energy function defined on the state space of all possibilities to remove or preserve splitting faces from an initial over-segmentation of the 3D image into supervoxels. It decomposes into potential functions that are learned automatically from a small amount of empirical training data. The learning is based on features of the distribution of gray values in the volume image and on features of the geometry and topology of the supervoxel segmentation.

To be able to extract these features from large 3D images that consist of several billion voxels, a new algorithm is presented in Chapter 4 that constructs a suitable representation of the geometry and topology of volume segmentations in a block-wise fashion, in log-linear runtime (in the number of voxels) and in parallel, using only a prescribed amount of memory.

At the core of this thesis is the optimization problem of finding, for a learned energy function, a segmentation with minimal energy. This optimization problem is difficult because the energy function consists of 3rd and 4th order potential functions that are not submodular. For sufficiently small problems with 10^4 degrees of freedom, it can be solved to global optimality using Mixed Integer Linear Programming. For larger models with 10^7 degrees of freedom, an approximate optimizer is proposed in Chapter 5 and compared to state-of-the-art alternatives.

Using these new techniques and a unified data structure for multi-variate data and functions (Chapter 6), a complete processing chain for segmenting large volume images, from the restoration of the raw volume image to the visualization of the final segmentation, has been implemented in C++. Results are shown for an application in neuroscience, namely the segmentation of a part of the inner plexiform layer of rabbit retina in a volume image of $2048 \times 1792 \times 2048$ voxels that was acquired by means of Serial Block Face Scanning Electron Microscopy (Denk and Horstmann, 2004) with a resolution of $22 \times 22 \times 30$ nm^3. The quality of the automated segmentation as well as the improvement over a simpler model that does not take geometric context into account, are confirmed by a quantitative comparison with the gold standard.

Zusammenfassung

Diese Arbeit stellt ein neues mathematisches Modell zur Segmentierung von Volumenbildern vor (Kapitel 3). Das Modell ist eine Energiefunktion im Zustandsraum aller Möglichkeiten, Trennflächen aus einer initialen Übersegmentierung des Volumenbildes in Supervoxel zu entfernen. Sie zerfällt in Potenzialfunktionen, die aus einer kleinen Menge empirischer Trainingsdaten maschinell gelernt werden. Das Lernen basiert auf Merkmalen der Grauwertverteilung des Volumenbildes sowie auf Merkmalen der Geometrie und Topologie der Supervoxel-Segmentierung.

Um diese Merkmale aus großen Volumenbildern mit mehreren Milliarden Bildpunkten extrahieren zu können, wird ein neuer Algorithmus vorgestellt (Kapitel 4), mit dessen Hilfe eine zweckmäßige Repräsentation der Geometrie und Topologie großer Volumen-Segmentierungen blockweise konstruiert werden kann, parallel und in logarithmisch-linearer Laufzeit, bei vorgegebenem, beschränktem Speicherbedarf.

Im Zentrum der Arbeit steht das Problem, für eine maschinell gelernte Energiefunktion eine Segmentierung mit minimaler Energie zu finden. Dieses Optimierungsproblem ist schwierig, da die Energiefunktion nicht submodulare Potenziale dritter und vierter Ordnung umfasst. Für hinreichend kleine Modelle mit 10^4 Freiheitsgraden lässt sich das Problem mittels Mixed Integer Linear Programming global optimal lösen. Für große Modelle mit 10^7 Freiheitsgraden wird ein approximatives Optimierungsverfahren vorgeschlagen und mit alternativen Verfahren verglichen (Kapitel 5).

Mit Hilfe der neuen Methoden und einer einheitlichen Datenstruktur für multivariate Daten und Funktionen (Kapitel 6), ist eine vollständige Verarbeitungskette zur Segmentierung großer Volumenbilder, von der Restauration der Rohdaten bis zur Visualisierung der finalen Segmentierung, in C++ implementiert worden. Gezeigt werden Ergebnisse für eine Anwendung in den Neurowissenschaften, die Segmentierung eines Ausschnitts der inneren plexiformen Schicht der Retina des Kaninchens in einem Volumenbild aus $2048 \times 1792 \times 2048$ Bildpunkten, das mittels Serial-Block-Face-Scanning-Electron-Microscopy (Denk and Horstmann, 2004) mit einer Auflösung von $22 \times 22 \times 30$ nm^3 aufgenommen wurde. Die Qualität der Segmentierung sowie die Verbesserung gegenüber einem einfacheren Modell, das keinen geometrischen Kontext einbezieht, werden quantitativ, durch Vergleich dem Gold-Standard, bestätigt.

Contents

Acknowledgments

The research condensed in this thesis was conducted at the Heidelberg Collaboratory for Image Processing (HCI) of the University of Heidelberg under the supervision of Fred A. Hamprecht (University of Heidelberg) and Winfried Denk (Max-Planck-Institute for Medical Research) who established a stimulating and challenging research environment. I am most grateful for their guidance and advice. The HCI offers excellent research opportunities and training in the theory and applications of image analysis. The SBFSEM data analyzed as part of this work was acquired by Kevin L. Briggman, Moritz Helmstaedter and Winfried Denk at the Max-Planck Institute for Medical Research. Working with this unique data is a pleasure.

Several ideas presented in this thesis have been inspired by discussions with my supervisors and with Ullrich Köthe, Thorben Kröger, Moritz Helmstaedter and Kevin L. Briggman (Chapters 3, 4 and 6), Jörg Kappes and Bogdan Savchynskyy (Chapter 5). The implementation of these ideas in general purpose software was made possible by support from the student researchers Thorsten Beier, Andreea Bonea, Diana Khoromskaia, Kevin Kunzmann, Guillaume Lapierre, Johannes Nordin, Oliver Petra and Lilian Villarin. Additional inspiration has come from workshops hosted by the Heidelberg Graduate School of Mathematical and Computational Methods for the Sciences and by the Research Training Group 1653 (Spatiotemporal Probabilistic Graphical Models and Applications in Image Analysis). I thank the directors, Michael Winckler and Christoph Schnörr, all investigators, assistants and my fellows for their efforts in running the graduate school and the research training group. In particular, I thank my present and former colleagues for maintaining a productive and enjoyable working atmosphere: Linus Görlitz, Michael Hanselmann, Michael Kelm, Frederik Kaster, Bernhard Kausler, Marc Kirchner, Anna Kreshuk, Thorben Kröger, Martin Lindner, Xinghua Lou, Björn Menze, Bernhard Y. Renard, Christoph Straehle and Barbara Werner.

I thank my parents Brigitte and Wulf-Dirk for their selfless indefatigable support. They brought me and my sisters Maren and Wiebke up in a spirit of morale and freedom of mind. I owe to them my curiosity. Research is thrilling: disappointing at times, stimulating and uplifting at others. All times have been lightened by the dear love of my wife, Birte.

1 Introduction

Until recently, experimental techniques in neuroscience have either provided detailed information on a small fraction of all neurons (cell recordings, imaging of stochastically stained cells), or information averaged over relatively large regions (fMRI, DTI, EEG). However, detailed knowledge of the complete connectivity pattern of all neurons, the so-called **connectome** (Sporns et al., 2005), would be of tremendous value for the understanding of neural computation (Briggman and Denk, 2006; Helmstaedter et al., 2008). Obtaining this knowledge has now become a realistic objective because serial block-face scanning electron microscopy (SBFSEM) (Briggman and Denk, 2006; Denk and Horstmann, 2004) makes it possible to acquire volume images of up to 1 mm^3 at high isotropic resolution ($\approx$ 25 nm), cf. Fig. 1.1.

SBFSEM collects image stacks by backscattering-contrast electron microscopy and serial sectioning (Denk and Horstmann, 2004). Every image in a stack is obtained by scanning the surface (block-face) of an embedded sample with an electron beam and detecting the backscattered electrons. A slice is then cut off from the sample with a specially designed diamond microtome to expose the next plane. Sections can be as thin as 25 nm, and a resolution of 10-20 nm is achievable in the lateral directions. Thus, the resolution of the combined volume image is sufficiently uniform to permit truly 3-dimensional image analysis. Moreover, alignment problems between consecutive slices (Kaynig et al., 2010a) are avoided from the outset since the sample remains stationary. SBFSEM volume images of 1 mm^3 will be as large as $40000^3 = 6.4 \cdot 10^{13}$ voxels.

As decisive as the acquisition of these volume images is their semi- or fully automated analysis because a manual reconstruction of neural circuits is too costly in terms of labor (Helmstaedter et al., 2008): Without the help of computers, the manual reconstruction of the 300 neurons of the nematode *Caenorhabditis elegans* from EM images took more than a decade (White et al., 1986). Even with the help of computers, appropriate visualization software and convenient user interfaces, a complete manual segmentation of SBFSEM volume images takes in the order of 10^5 person-years per cubic millimeter (Helmstaedter et al., 2008). Tracing only the center lines (skeletons) of neurons, their axons and dendrites, is faster but

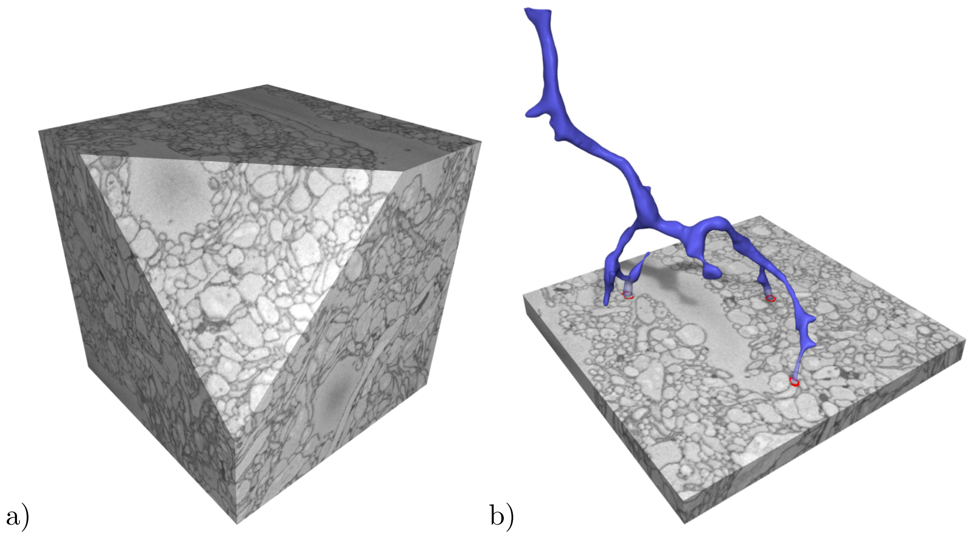


Figure 1.1: a) A subset of 512^3 voxels from a volume image acquired by serial block-face scanning electron microscopy (SBFSEM) (Briggman and Denk, 2006; Denk and Horstmann, 2004) at the nearly isotropic resolution of $22 \times 22 \times 30$ nm^3. b) One of several hundred neuronal processes reconstructed by means of the proposed method.

only by a factor of ten (Helmstaedter et al., 2008). An at least partially automated procedure is therefore indispensable.

This thesis presents a fully automated procedure for segmenting SBFSEM volume images of neural tissue, including dense neuropil. The problem is challenging because the intricate branching geometry of neurons poses a high risk of introducing both over-segmentation (false splits) and under-segmentation (false mergers). Furthermore, the volume image needs to be segmented completely, into hundreds of distinct cells, which is different from other biological and medical segmentation problems where the goal is to segment only one organ or body part versus the image background and which hampers the use of explicit shape models. Finally, the sheer size of SBFSEM volume images narrows the class of practically applicable algorithms to those whose runtime complexity is less than quadratic in the number of voxels, a limitation that complicates the extraction of geometric features.

Techniques from four research areas are used and developed further in this thesis to tackle these problems:

- **Machine Learning** is used extensively to learn from human experts how to transform the raw SBFSEM volume image into a proper segmentation.
- This learning is supported by geometric features that are extracted from an initial over-segmentation of the volume image into **supervoxels** using **computational geometry**.
- The final decision which supervoxel to merge in order to arrive at neurons, inclduing their axons and dendrites, is cast as a **combinatorial optimization problem** in a **graphical model** over supervoxel boundaries. This problem is solved exactly for small datasets and approximately for large datasets.
- **Efficient and parallel algorithms** and adequate data structures are developed so that the entire segmentation procedure can be applied in practice to volume images that extend well into the gigavoxel regime.

This thesis is organized as follows: Chapter 2 summarizes important theoretical foundations. Chapter 3 describes the entire segmentation procedure, from the restoration of the raw SBFSEM volume image and the collection of training data to the evaluation of the final result on an SBFSEM benchmark dataset (Helmstaedter et al., 2011) of $2048 \times 1792 \times 2048$ voxels that shows part of the inner plexiform layer of rabbit retina at a resolution of $22 \times 22 \times 30$ nm^3 (Fig. 1.1). New methods whose development was necessary for this procedure to be applicable in practice to large volume images are described in the subsequent chapters. These chapters constitute the methodological novelty of this thesis. Chapter 4 introduces a new computational geometry algorithm for extracting geometric features from large volume segmentations. Chapter 5 presents a new algorithm for combinatorial optimization in higher-order graphical models. Finally, Chapter 6 describes a versatile data structure for runtime-flexible multidimensional arrays that is used throughout the segmentation procedure, for computational geometry and combinatorial optimization.

2 Methodological Foundations

This chapter describes established methods and concepts from machine learning and computer vision that are fundamental to the novel approaches presented in the subsequent chapters. These methods, Random Forests, graphical models, and watershed segmentation, have attracted significant attention in recent years. The purpose of this chapter is to summarize relevant information contained in the cited research articles, to introduce a unified notation and to serve as a self-contained methodological introduction to this thesis.

Random Forests (Section 2.1.1) are classifiers and regressors that can be used to learn the relationship between high-dimensional observed data and unobserved labels or responses from a small amount of empirical training data. **Graphical models** (Section 2.1.2) can help to put this learned information into context.

Watershed segmentation (Section 2.2.1) has a long history in computer vision. A comprehensive overview is given by Roerdink and Meijster (2000). A little known fact is that the region growing algorithm by Meyer (1991) can be implemented such that it works in linear runtime (in the number of image points), provided that the elevation map can attain only finitely many values. This aspect is emphasized in Section 2.2.1 because it makes this algorithm suitable for segmenting large volume images.

2.1 Machine Learning

2.1.1 Random Forests

Random forests (Breiman, 2001) are ensembles of binary decision trees (Breiman et al., 1984). In supervised statistical learning, these decision trees are constructed from a training set S of observed or hand-labeled data. The purpose of learning is either **regression**, i.e. function approximation, or **classification**, i.e. the prediction of an object class based on a set of object features.

In the regression setting, each **observation** $(x^{(s)}, y^{(s)}) = s \in S$ consists of an **input vector** $x^{(s)} \in \mathbb{R}^m$ (with $m \in \mathbb{N}$) and a **response** $y^{(s)} \in \mathbb{R}$. Under the assumption that the observations are i.i.d. according to some (unknown) underlying distribution of the pair (X, Y) of random variables, a random forest is an estimator of the regression function $f : \mathbb{R}^m \to \mathbb{R}$ with

$$\forall x \in \mathbb{R}^m : \quad f(x) = E(Y|X = x)(x) \ . \tag{2.1}$$

In the classification setting, each **training sample** $(x^{(s)}, y^{(s)}) = s \in S$ consists of a **feature vector** $x^{(s)} \in \mathbb{R}^m$ (with $m \in \mathbb{N}$) and one **class label** $y^{(s)} \in C$ out of $n \in \mathbb{N}$ potential class labels $C = \{c_1, \ldots, c_n\}$. Under the assumption that the training samples are i.i.d. according to some (unknown) underlying distribution, a random forest is an estimator of the most likely assignment of feature vectors to class labels, i.e. of the classification function $c : \mathbb{R}^m \to C$ with

$$\forall x \in \mathbb{R}^m : \quad c(x) = \operatorname*{argmax}_{y \in C} P(Y|X = x)(y) \ . \tag{2.2}$$

2.1.1.1 Training

The construction of decision trees from a finite training set works similarly in both settings: First, the root vertex $r \in V$ of each decision tree (V, E) is associated with a subset $S_r \subseteq S$ of the training data. Popular choices are either the entire set S, or a **bootstrap sample** of S that consists of $|S|$ elements drawn randomly from S with replacement. The latter is called **bagging** (Breiman, 1996); it turns the random forest into a bag of classifiers (decision trees) each of which is trained on a different bootstrap sample of S. Breiman (1996) shows that bagging can give substantial gains in accuracy.

After the initialization of root vertices, the construction of each decision tree proceeds as follows: As long as there exists a leave $v \in V$ (initially, the root is a leaf) for which the associated subset S_v of training data does not fulfill a **purity predicate** (Section 2.1.1.2), a **split function** (Section 2.1.1.3) is invoked on S_v. This split function outputs a dimension $j_v \in \{1, \ldots, m\}$ and a value $\xi_v \in \mathbb{R}$ by which S_v is partitioned into

$$S'_v = \{s \in S_v | x^{(s)}_{j_v} \leq \xi_v\} \quad \text{and} \tag{2.3}$$

$$S''_v = \{s \in S_v | x^{(s)}_{j_v} > \xi_v\} \ . \tag{2.4}$$

The sets S'_v and S''_v are then associated with two new child vertices $v', v'' \in V$ which are connected to v via the new edges (v, v') and (v, v''). The construction of the tree ends when all leaves fulfill the purity predicate.

2.1.1.2 Purity Predicate

The purity predicate requires that no more than a fixed number $p \in \mathbb{N}$ of observations (in regression) or training samples (in classification) are contained in the set S_v or else, all observations (training samples) have the same response (label). The parameter p is one design parameter of the learning algorithm. The smaller p is, the smaller the bias of individual decision trees becomes. Lin and Jeon (2006) argue that p should be very small, usually 1, for high-dimensional input spaces whereas p should be optimized for low-dimensional input because the averaging over decision trees performed during prediction (Section 2.1.1.4) reduces the variance but not the bias of the random forest which suggests that the bias of individual decision trees should be small.

2.1.1.3 Split Function

Breiman (2001) proposes to draw a fixed number $m_{\text{try}} \in \{1, \dots, m\}$ of dimensions of the input vector at random, without replacement, and to search exhaustively over the selected dimensions and all values attained in these dimensions in the training sample S_v for a combination that **minimizes an objective function** over the partition of S_v into S'_v and S''_v.

In regression, this objective function is the **empirical variance** of all response variables of the training samples S'_v, plus the empirical variance of all response variables of the training samples S''_v. In classification, the objective function is the **Gini index** (2.12) of the set of labels of the training samples S'_v, weighted by $|S'_v|$, plus the Gini index of the set of labels of the training samples S'_v, weighted by $|S'_v|$. A rigorous motivation and unified perspective on these seemingly different approaches is given in Section 2.1.1.5.

Choosing $m_{\text{try}} = \sqrt{m}$ gives accurate results in practice (Breiman, 2001). The smaller m_{try} is chosen to be, the greater becomes the risk of drawing only dimensions for which no pronounced minimum of the objective function exists. The effect of this increased risk on the prediction accuracy depends on the purity predicate. In any case, it leads to deeper decision trees and thus to increased runtimes for computing predictions.

One alternative is to set $m_{\text{try}} = m$. This renders the construction of single decision trees deterministic and therefore only makes sense in conjunction with bagging. The effect on the prediction accuracy again depends on the purity predicate. The runtime spent on the exhaustive search for optimal splits during training increases linearly with m. The depth of decision trees is smaller for larger m_{try} but need not be minimal because the feature space is still partitioned in a greedy fashion. The construction of minimal decision trees is in general NP-hard (Hyafil and Rivest, 1976; Sieling, 2008).

A second alternative proposed by Lin and Jeon (2006) is to first draw one split value $\chi_j \in \mathbb{R}$ for each input dimension $j \in \{1, \ldots, m\}$ at random and to then draw a dimension j_v and associated split value $\xi_v := \chi_{j_v}$ at random from those that minimize the objective function for the partition of S_v into S'_v and S''_v. This approach is parameter-free. It specializes to random sampling if the objective function is equal for all splits. Compared to Breiman's approach, the chance of making informative splits is increased. However, the split values are no longer optimal but are distributed with non-zero variance around the optima.

2.1.1.4 Prediction

Prediction from a trained random forest works by passing the input $x \in \mathbb{R}^m$ down each tree, picking child nodes according to the inequalities in (2.3) and (2.4) that partition the input space. This way, the input vector ends up in a leaf node $v \in V$ in each tree.

In **regression**, the prediction from a single tree is the average response over all observations associated with the leaf node v. In consequence, the tree encodes the function $\hat{f}_{(V,E)} : \mathbb{R}^m \to \mathbb{R}$ with

$$\forall x \in \mathbb{R}^m : \quad \hat{f}_{(V,E)}(x) = \sum_{(x,y) \in S_v} \frac{y}{|S_v|} \quad . \tag{2.5}$$

The prediction of the entire random forest is the average prediction over all decision trees.

In **classification**, each decision tree predicts one class label that occurs most often in the training data associated with the leaf node. The tree thus represents a function $\hat{c}_{(V,E)} : \mathbb{R}^m \to C$ with

$$\forall x \in \mathbb{R}^m : \quad \hat{c}_{(V,E)}(x) = \underset{c \in C}{\operatorname{argmax}} \, |\{s \in S_v | \exists x \in \mathbb{R}^m : s = (x, c)\}| \quad . \tag{2.6}$$

The prediction of the entire random forest is a label for which the number of decision trees that predict this label is maximal.

The **relative frequency of occurrence** $\hat{r}_{(V,E),c} : \mathbb{R}^m \to [0, 1]$ of any label $c \in C$ can be obtained from a single decision tree by relating the absolute frequency of occurrence of this label to the total number of training samples in the leaf node:

$$\forall x \in \mathbb{R} : \quad \hat{r}_{(V,E),c}(x) = \frac{|\{s \in S_v | \exists x \in \mathbb{R}^m : s = (x, c)\}|}{|S_v|} \quad . \tag{2.7}$$

The relative frequency of occurrence in the entire random forest is the average relative frequency of occurrence over all decision trees.

2.1.1.5 Mean Distance and the Gini Index

Gini (1912) proposes to measure variability in a sequence of real numbers by taking the mean absolute difference over all pairs of its entries. A generalization of this statistic for arbitrary distance functions is introduced in the following and shown to specialize to the Gini index (2.12) for the L_0 distance, to the empirical variance (2.14) for the squared L_2 distance, and to a rank statistic (2.15) for the L_1 distance. This generalization provides a unified perspective on classification and regression with random forests.

Definition 1. For any sequence $(y_j)_{j\in\{1,\dots,n\}}$ of $n \in \mathbb{N}$ elements of an arbitrary set $C \neq \emptyset$ and any function $d : C \times C \to \mathbb{R}_0^+$ that is used to measure distance in C, the **mean distance** D over all pairs of entries of (y_j) is defined as

$$D = \frac{1}{n^2}\sum_{j=1}^{n}\sum_{k=1}^{n} d(y_j, y_k) \quad . \tag{2.8}$$

The mean distance is obviously invariant under all permutations of the sequence (y_j), and it is non-negative because d is assumed to be non-negative. There are no further restrictions on d in general. If d is positive definite, we have $D = 0$ if and only if all entries of (y_j) are equal. If d is symmetric, one could divide the r.h.s. of (2.8) by 2 because each value $d(y_j, y_k) = d(y_k, y_j)$ is added at least twice. However, these specializations are unnecessary. Important choices for d include the following symmetric positive definite functions of which (2.9) and (2.11) are metrics:

$$d(y_j, y_k) = \begin{cases} 0 & \text{if } y_j = y_k \\ 1 & \text{otherwise} \end{cases} \tag{2.9}$$

$$d(y_j, y_k) = (y_j - y_k)^2 \tag{2.10}$$

$$d(y_j, y_k) = |y_j - y_k| \quad . \tag{2.11}$$

In **classification**, one deals with sequences (y_j) in a set of $m \in \mathbb{N}$ distinct class labels $\{c_1, \dots, c_m\} = C$. A distance function that distinguishes only between similar and dissimilar labels is the L_0 distance (2.9). The mean distance w.r.t. this metric is called the **Gini index**. It can be expressed in terms of the frequencies of occurrence of the labels $c_1, \dots, c_m$ in the sequence (y_j), i.e. in terms of $n_1, \ldots, n_m \in \mathbb{N}$ with $\forall j \in \{1, \ldots, m\} : n_j = |\{k \in \{1, \ldots, n\} : y_k = c_j\}|$:

$$D = \frac{1}{n^2} \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} n_j n_k = \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} \frac{n_j}{n} \frac{n_k}{n} = 2 \sum_{j=1}^{n} \sum_{k=1}^{j-1} \frac{n_j}{n} \frac{n_k}{n} \quad . \tag{2.12}$$

The step from (2.8) to (2.12) is illustrated in Fig. 2.1.

In **regression**, one deals with sequences (y_j) of response variables in $\mathbb{R}$. In this case, the mean distance D w.r.t. the L_2 distance (2.10) equals twice the empirical variance of the sequence (y_n). This is shown in the following, using $\mathrm{E}_j^n(y_j)$ as a short hand notation for the empirical mean $\frac{1}{n} \sum_{j=1}^{n} y_j$, and the well-known relation

$$\begin{aligned}
& (\mathrm{E}_j^n(y_j))^2 + \mathrm{E}_j^n\left((y_j - \mathrm{E}_k^n(y_k))^2\right) = (\mathrm{E}_j^n(y_j))^2 + \frac{1}{n} \sum_{j=1}^{n} (y_j - \mathrm{E}_k^n(y_k))^2 \\
= \; & (\mathrm{E}_j^n(y_j))^2 + \frac{1}{n} \sum_{j=1}^{n} \left(y_j^2 - 2y_j \mathrm{E}_k^n(y_k) + (\mathrm{E}_k^n(y_k))^2\right) \\
= \; & (\mathrm{E}_j^n(y_j))^2 + \mathrm{E}_j^n(y_j^2) - 2\mathrm{E}_k^n(y_k) \frac{1}{n} \sum_{j=1}^{n} y_j + (\mathrm{E}_j^n(y_j))^2 \\
= \; & \mathrm{E}_j^n(y_j^2) \quad .
\end{aligned} \tag{2.13}$$

Starting with (2.8) and (2.10),

$$\begin{aligned}
D &= \frac{1}{n^2}\sum_{j=1}^{n}\sum_{k=1}^{n}(y_j-y_k)^2 = \frac{1}{n^2}\sum_{j=1}^{n}\sum_{k=1}^{n}(y_j^2-2y_jy_k+y_k^2) \\
&= \frac{1}{n^2}\sum_{j=1}^{n}\left(ny_j^2-2y_j\sum_{k=1}^{n}y_k+\sum_{k=1}^{n}y_k^2\right) \\
&= \frac{1}{n}\sum_{j=1}^{n}\left(y_j^2-2y_j\frac{1}{n}\sum_{k=1}^{n}y_k+\frac{1}{n}\sum_{k=1}^{n}y_k^2\right) \\
&= \frac{1}{n}\sum_{j=1}^{n}\left(y_j^2-2y_j\mathrm{E}_k^n(y_k)+\mathrm{E}_k^n(y_k^2)\right) \\
&= \frac{1}{n}\sum_{j=1}^{n}\left(y_j^2-2y_j\mathrm{E}_k^n(y_k)+(\mathrm{E}_k^n(y_k))^2+\frac{1}{n}\sum_{k=1}^{n}(y_k-\mathrm{E}_l^n(y_l))^2\right) \\
&= \frac{1}{n}\left(\sum_{j=1}^{n}(y_j-\mathrm{E}_k^n(y_k))^2+\sum_{k=1}^{n}(y_k-\mathrm{E}_l^n(y_l))^2\right) \\
&= \frac{2}{n}\sum_{j=1}^{n}(y_j-\mathrm{E}_k^n(y_k))^2 = 2\mathrm{E}_j^n\left((y_j-\mathrm{E}_k^n(y_k))^2\right) \quad . \qquad (2.14)
\end{aligned}$$

If the L_1 distance (2.11) is used instead and (y_j) is ordered increasingly,

$$\begin{aligned}
D &= \frac{1}{n^2}\sum_{j=1}^{n}\sum_{k=1}^{n}|x_j-x_k| \\
&= \frac{2}{n^2}\sum_{\substack{(j,k)\in\{1,\dots,n\}^2\\ k\le j}}(x_j-x_k) \\
&= \frac{2}{n^2}\left(\sum_{j=1}^{n}\sum_{k=1}^{j}x_j-\sum_{k=1}^{n}\sum_{j=k}^{n}x_k\right) \\
&= \frac{2}{n^2}\left(\sum_{j=1}^{n}jx_j-\sum_{k=1}^{n}(n-k+1)x_k\right) \\
&= \frac{2}{n^2}\sum_{j=1}^{n}(2j-n-1)x_j \quad . \qquad (2.15)
\end{aligned}$$

The sum (2.15) is a rank statistic with coefficients $-(n-1),\dots,(n-1)$.

(y_j)	D	D'
$(1,\dots,n)$	$\frac{n-1}{n}$	1
$(\underbrace{0,\dots,0}_{q},\underbrace{1,\dots,1}_{q})$	$\frac{1}{2}$	$\frac{n}{2(n-1)}$

Table 2.1: Difference between D and D' for the L_0 metric.

A final note concerns the difference between definition (2.8) and the mean distance over all pairs of **distinct** entries of (y_j). The latter is well-defined if $n \geq 2$:

$$D' = \frac{1}{n(n-1)} \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} d(y_j, y_k) \ . \tag{2.16}$$

If $d(y,y) = 0$ for all $y \in C$, the only difference between (2.8) and (2.16) is in the normalization, i.e.

$$D' = \frac{n}{n-1} D \ . \tag{2.17}$$

This distinction is particularly important in classification, i.e. w.r.t. the L_0 distance (2.9). When this metric is used, the upper bound on D,

$$D \ \leq \ \frac{1}{n^2} \sum_{j=1}^{n} \sum_{k=1}^{n} d(y_j, y_k) = \frac{n(n-1)}{n^2} = \frac{n-1}{n} \ , \tag{2.18}$$

is attained if and only if the entries in the sequence (y_j) are pairwise distinct. Moreover, this bound depends on the length n of the sequence, in contrast to the upper bound on D' which is 1, by (2.17) and (2.18). It can be a desired property that the upper bound be independent of n. If this is the case, D' should be used instead of D. Examples of the difference between D and D' for the L_0 metric are given in Tab. 2.1.

2.1.1.6 Consistency

Random forest are not just a heuristic. Lin and Jeon (2006) establish a close relationship between random forests and adaptive nearest neighbor classifiers. This relationship opens a fundamental and principled view on random forests and provides a starting point for analyzing their consistency (Biau et al., 2008).

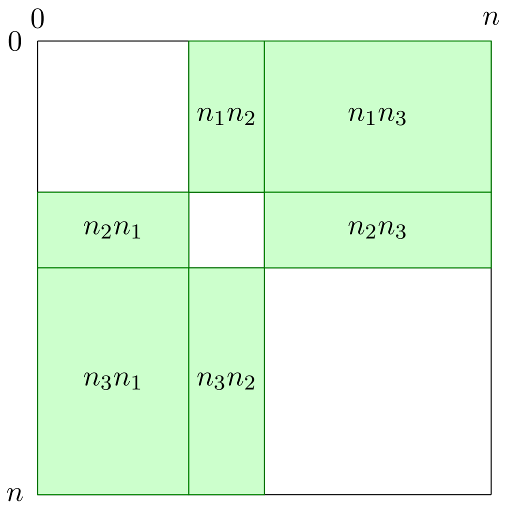


Figure 2.1: The L_0 distances of all pairs of entries in the sequence (y_j) form the matrix $A \in \{0,1\}^{n\times n}$ depicted above. If (y_j) consists of n_1 labels c_1, followed by n_2 labels c_2, followed by n_3 labels c_3, the colored blocks correspond to the entries which are one. All other entries are 0. The sum of entries equals (2.12) and (2.8).

2.1.2 Graphical Models

Graphical models (Cowell et al., 2007; Koller and Friedman, 2009; Lauritzen, 1996; Wainwright and Jordan, 2008) are structures that encode explicitly how a multi-variate function **decomposes** w.r.t. a given associative and commutative operation into functions that depend on subsets of all variables (Aji and McEliece, 2000). As an example, consider a function $\varphi : \{0,1\}^4 \to \mathbb{R}$ that decomposes w.r.t. addition into functions $\varphi_1, \varphi_2 : \{0,1\}^2 \to \mathbb{R}$, $\varphi_3 : \{0,1\}^3 \to \mathbb{R}$ and $\varphi_4 : \{0,1\} \to \mathbb{R}$ such that $\forall x_1, \ldots, x_4 \in \{0,1\}$:

$$\varphi(x_1, \ldots, x_4) = \varphi_1(x_1, x_2) + \varphi_2(x_1, x_3) + \varphi_3(x_2, x_3, x_4) + \varphi_4(x_4) \ . \tag{2.19}$$

In general, the variables can have different domains that can be finite, countably or uncountably infinite. The following discussion focuses on functions with finite domain which are the only class that is used in this work.

Graphical models describe decompositions in terms of the associative and commutative operation that leads to the decomposition, the operands of the decomposition, i.e. the functions $\varphi_1, \ldots, \varphi_4$ in the above example, and a graph (Fig. 2.2a) that encodes which operands depend on which variables in the form (2.19). A rigorous definition (Def. 3) is given below.

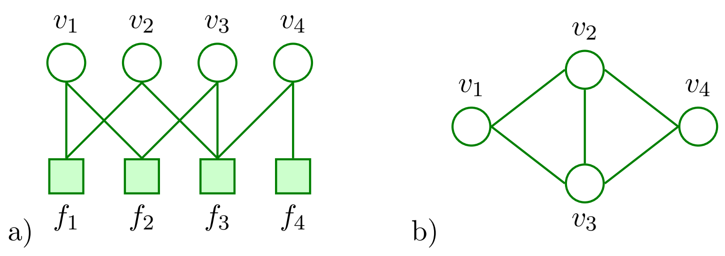


Figure 2.2: a) **Operand graph** of a graphical model representing the decomposition (2.19). This undirected bipartite graph consists of **variable vertices** $v_1, \ldots, v_4$ that are associated with the variables $x_1, \ldots, x_4$, and **operand vertices** $f_1, \ldots, f_4$ associated with the functions (operands) $\varphi_1, \ldots, \varphi_4$. A variable vertex is connected to an operand vertex if and only if the operand depends on that variable in the form (2.19). b) The **variable adjacency graph** connects any two variables for which there exists at least one operand that depends on at least these two variables.

Graphical models are an important concept for several reasons:

- Graphical models can be used to **model complex systems**. In Chapter 3, a system of volume image segmentations is considered that depends on 10^7 binary variables. A function that assigns an energy to each of the 2^{10^7} possible states of this system is defined in terms of a graphical model.
- The decomposition made explicit by a graphical model can be exploited for **optimization** (cf. Section 2.1.3 and Chapter 5), e.g. to find a volume image segmentation that has minimal energy (Chapter 3).
- Graphical models can be used in conjunction with **statistical learning** (cf. e.g. Franc and Savchynskyy, 2008). The energy function of volume image segmentations defined in Chapter 5 decomposes according to a graphical model into a sum of potential functions, and these potential functions are learned independently from empirical data by means of random forests (cf. Section 2.1.1 and Chapter 3).

One important application of graphical models are probability density functions where conditional independence relations on random variables lead to factorization (decomposition w.r.t. multiplication). Examples of probabilistic graphical models include Markov Random Fields (Kindermann and Snell, 1980; Lauritzen, 1996), Conditional Random Fields (Lafferty et al., 2001) and Bayesian Networks (Cowell et al., 2007; Minka, 2001; Pearl, 1988). Introductions to these closely related models are given e.g. by Bishop (2007); Koller and Friedman (2009); Wainwright and Jordan (2008). Graphical models are, however, not restricted to probabilistic modeling and have also been used more generally to represent decompositions of arbitrary multivariate functions (Aji and McEliece, 2000).

Different classes of graphs are used in graphical models, including but not limited to undirected graphs, directed acyclic graphs and factor graphs. Their relation is discussed e.g. by Bishop (2007); Kschischang et al. (2001); Wainwright and Jordan (2008). While undirected graphs and directed acyclic graphs contain one vertex for each variable and are suitable in particular for probabilistic models, factor graphs (Aji and McEliece, 2000; Kschischang et al., 2001) contain one vertex for each variable and one vertex for each operand and can express general decompositions. Factor graphs can be used with any associative and commutative operation and are therefore termed **operand graphs** in this thesis that deals with additive decompositions. The operand graph of a graphical model representing the decomposition (2.19) is depicted in Fig. 2.2a.

Definition 2. An **operand graph** is an undirected bipartite graph, i.e. a triple (V, F, E) with $E \subseteq V \times F$. The elements of V and F are termed **variable vertices** and **operand vertices**, respectively. The elements of E are termed edges. Two vertices $v \in V$ and $f \in F$ are said to be connected if and only if $(v, f) \in E$. Neighborhoods of vertices are defined as

$$\forall v \in V : \qquad N(v) = \{f \in F | (v, f) \in E\} \ , \tag{2.20}$$

$$\forall f \in F : \qquad N(f) = \{v \in V | (v, f) \in E\} \ . \tag{2.21}$$

Each operand graph induces an undirected **variable adjacency graph** (V', E') that connects any two variables for which there exists at least one operand to which both variables are connected, i.e.

$$(v, v') \in E' \quad :\Leftrightarrow \quad \exists f \in F : \ (v, f) \in E \wedge (v', f) \in E \ . \tag{2.22}$$

An example is depicted in Fig. 2.2b. Variable adjacency graphs are a more abstract representation of decompositions than operand graphs. Note, for example, that a decomposition $\varphi_5(x_1, x_2, x_3) + \varphi_6(x_2, x_3, x_4)$ of the function φ has a different operand graph than the decomposition (2.19) but the same variable adjacency graph (Fig. 2.2b).

Definition 3. A **graphical model** of a function $\varphi : X_1 \times \ldots \times X_n \to Y$ consists of (i) an associative and commutative operation $\otimes : Y \times Y \to Y$, (ii) an operand graph (V, F, E) with $V = \{1, \ldots, n\}$, and (iii) for each $f \in F$, a function $\varphi_f : X_{j_f(1)} \times \ldots \times X_{j_f(|N(f)|)} \to Y$ where $j_f(k)$ is the k-th smallest number in $N(f)$, such that $\forall (x_1, \ldots, x_n) \in X_1 \times, \ldots, X_n$:

$$\varphi(x_1, \ldots, x_n) = \bigotimes_{f \in F} \varphi_f(x_{j_f(1)}, \ldots, x_{j_f(|N(f)|)}) \ . \tag{2.23}$$

Graphical models define neither the order nor the hierarchy in which the operations on the r.h.s. of (2.23) are executed. This makes sense only because $\otimes$ is assumed to be associative and commutative. Otherwise, the r.h.s. of (2.23) would be ill-defined. Beyond the scope of this thesis, one can imagine a generalization of graphical models to operations that are non-associative, non-commutative, or both. Such generalizations would include explicit representations of the order and hierarchy in which the operations are performed. However, many algorithms that operate on graphical models exploit the invariance that arises from associativity and commutativity. This holds in particular for optimization algorithms.

2.1.3 Optimization of Graphical Model Functions

The optimization of functions that decompose according to a graphical model is important, e.g. to arrive at maximum-a-posteriori estimates for probabilistic models and to minimize energy functions as in Chapters 3. In several special cases, the optimization problem can be solved exactly and efficiently. In particular:

- If all operands in an additive graphical model are **submodular** w.r.t. a lattice in $X_1 \times \dots \times X_n$ and this lattice is induced by total orders in $X_1, \dots, X_n$, respectively, the global minimum can be found via a minimum s-t-cut in a graph that can be constructed efficiently from the graphical model (Boykov et al., 2001; Kolmogorov and Zabin, 2004). If total orders exist in $X_1, \dots, X_n$ that induce a lattice in $X_1 \times \dots \times X_n$ w.r.t. which the operands become submodular, these total orders can be found in polynomial time (Schlesinger, 2007). The global minimum can in this case be found via a graph cut w.r.t. the **permuted submodular** function. The functions defined in Chapter 3 are provably not permuted submodular which rules out optimization by graph cuts for this application (cf. Section 2.1.3.1).

- If the operand graph is a **tree**, the global optimum can be found by means of dynamic programming (Pearl, 1988). If the operand graph has loops, it is sometimes possible to group operands and variables such that the operand graph on the grouped vertices becomes a tree whose grouped operands are simple enough to be optimized exactly in affordable runtime, e.g. by variable elimination, exhaustive search or lazy flipping (Chapter 5). A more general and principled way of agglomerating factors and variables is by construction of a **junction tree** (Lauritzen, 1996; Lauritzen and Spiegelhalter, 1988) which requires a triangulation of the variable adjacency graph. The runtime for exact optimization is exponential in this case in the size of the largest clique of the triangulation. This size is bounded below by the **treewidth** of the variable adjacency graph which rules out this method for the large and highly connected graphical models defined in Chapter 3.

The optimization of non-submodular functions with loopy graphical models is NP-hard in the general case (Kolmogorov and Zabin, 2004). Some functions of this class can still be optimized exactly in affordable runtime using Mixed Integer Linear Programming (Schrijver, 1986, 2003) (Section 2.1.3.4). However, the runtime is exponential in the worst-case and the memory requirement eliminates the use of this method for large models, including those defined in Chapter 3.

Approximate solutions are therefore important. Several algorithms exist for this purpose, including loopy belief propagation (Kschischang et al., 2001; Pearl, 1988), tree-reweighted belief propagation (Kolmogorov, 2006; Wainwright and Jordan, 2008; Wainwright et al., 2005), dual decomposition Kappes et al. (2010); Komodakis et al. (2010), iterated conditional modes (Besag, 1986) and the Lazy Flipper (Chapter 5; Andres et al., 2010a). The performance of these algorithms on the optimization problem defined in Chapter 3 is analyzed in Chapter 5. Theoretical aspects and algorithms that are relevant for this optimization problem are discussed in the following.

2.1.3.1 Submodularity and Permuted Submodularity

Submodularity is defined w.r.t. a lattice whose definition is recalled here.

Definition 4. For any partially ordered set $(X, \leq)$ and any subset $Y \subseteq X$, an element $s \in X$ is termed a **least upper bound** or **supremum** of Y if and only if the following two conditions hold

$$\forall x \in Y : \qquad x \leq s \tag{2.24}$$

$$\forall s' \in X : \qquad \big(\forall x \in Y : \ x \leq s'\big) \Rightarrow s \leq s' \ . \tag{2.25}$$

The definition of a **greatest lower bound** or **infimum** is analogous.

A supremum need not exist, either because there is no upper bound or because the set of upper bounds has two or more elements of which none is a least element of that set. If Y has a supremum, it is unique and denoted by $\sup Y$. (If $s, s' \in X$ are both suprema of Y then $s \leq s'$ and $s' \leq s$ by (2.25) and thus, $s = s'$ by the antisymmetry of $\leq$).

Definition 5. A partially ordered set in which any two elements have a supremum and an infimum is called a **lattice**.

Here are two examples:

1. Let $n \in \mathbb{N}$ and let $(X_1, \preceq_1), \ldots, (X_n, \preceq_n)$ be finite sets, each equipped with a total order. In the Cartesian product $X_1 \times \cdots \times X_n =: X$, the relation "$\leq$" $\subseteq X \times X$ with $\forall x = (x_1, \ldots, x_n), x' = (x'_1, \ldots, x'_n) \in X$:

$$x \leq x' \quad \Leftrightarrow \quad \forall j \in \{1, \ldots, n\} : x_j \preceq x'_j \tag{2.26}$$

defines a partial order. For any two elements $x = (x_1, \dots, x_n), x' = (x'_1, \dots, x'_n) \in X$,

$$(\max\{x_1, x'_1\}, \dots, \max\{x_n, x'_n\}) \text{ and } (\min\{x_1, x'_1\}, \dots, \min\{x_n, x'_n\})$$

are well-defined w.r.t. the total orders $\preceq_1, \dots, \preceq_n$ and are the supremum and the infimum of x and x'. Thus, $(X, \leq)$ is in fact a lattice.

2. As a more specific example, let $n = 2$ and $(X_1, \preceq_1) = (X_2, \preceq_2) = (\{0, 1\}, \preceq)$ with the natural total order $(0 \preceq 1)$. From (2.26), it follows, for instance:

$$\sup\{(0,1), (1,0)\} = (1,1) \quad \text{and} \quad \inf\{(0,1), (1,0)\} = (0,0) \ .$$

Definition 6. Consider a lattice $(X, \leq)$. A function $f : X \to \mathbb{R}$ is called **submodular** if and only if

$$\forall x, y \in X : \quad f(\sup\{x, y\}) + f(\inf\{x, y\}) \leq f(x) + f(y) \ . \tag{2.27}$$

With respect to the lattice in example (2), a function $f : X_1 \times X_2 \to \mathbb{R}$ is submodular if and only if $f(0,0) + f(1,1) \leq f(0,1) + f(1,0)$. If one changes the order $\preceq$ in either X_1 or X_2 but not in both sets, e.g. such that $0 \preceq_1 1$ and $1 \preceq_2 0$, the submodularity condition becomes $f(0,1) + f(1,0) \leq f(0,0) + f(1,1)$.

Schlesinger (2007) shows that, given a function $f : X_1 \times \cdots X_n \to \mathbb{R}$ with finite domain, the question whether there exist total orders in $X_1, \dots, X_n$ such that f is submodular w.r.t. the partial order (2.26) in the Cartesian product $X_1 \times \cdots \times X_n$ can be answered in polynomial time, and that, in case of existence, such a sequence of orders can be constructed efficiently. Schlesinger (2007) terms this class of functions **permuted submodular**.

The exhaustive search over all combinations of orders shows that the potential functions defined in Chapter 3 are not permuted submodular.

A close connection exists between submodularity and convexity. The interested reader is referred to (Lovász, 1983). The optimization of submodular functions by means of graph cuts is described by Kolmogorov and Zabin (2004). Non-submodular functions can either be approximated by submodular functions or optimized approximately as discussed in the following sections and in Chapter 5.

2.1.3.2 Iterated Conditional Modes (ICM)

Iterated Conditional Modes (ICM) (Besag, 1986) is an algorithm for optimizing functions on discrete variables approximately. It starts from an initial assignment of values to the variables (the current best assignment) that can be chosen arbitrarily, e.g. at random. It then iterates over the variables in a prescribed (possibly random) order, considering a single variable in each iteration. If the value of the objective function can be improved by changing the value assigned to the selected variable and leaving the values assigned to all other variables fixed, the best such update becomes the current best assignment. ICM finishes when no change of the assignment that affects just one variable can further improve the function value.

By definition, ICM is a strictly convergent algorithm. It need not converge to the global optimum but the assignment attained after convergence is guaranteed to be optimal within a Hamming distance of 1. ICM is a greedy procedure and the output depends in general on the initialization. If the objective function decomposes according to a graphical model, the computation of function values is more efficient than in the general case because in each iteration, only those operands that depend on the updated variable need to be re-computed.

The strict convergence of ICM and the local optimality of the assignment found after convergence make this procedure interesting as a means to further improve approximations obtained by message passing algorithms (Section 2.1.3.3). For this purpose, it is beneficial to extend the search space of ICM to larger subsets of variables in order to arrive at assignments of values to the variables that are guaranteed to be optimal within greater Hamming distances. Chapter 5 presents a generalization of ICM (The Lazy Flipper) that does just that.

2.1.3.3 Message Passing Algorithms

One message passing algorithm for optimizing functions that decompose according to a graphical model is loopy belief propagation (Kschischang et al., 2001; Pearl, 1988) (BP). BP specializes to exact optimization by variable elimination if the graphical model is a tree and becomes an approximate optimizer (Yedidia et al., 2000) for loopy graphical models. BP approximations have been shown to be useful for decoding (McEliece et al., 1998) and are shown in Chapter 5 to outperform approximations obtained by other algorithms for the optimization problem defined in Chapter 3.

Different message passing schemes exist for graphical models as general as in Def. 3. Messages are functions that are associated with edges in the operand graph. In the synchronous message passing scheme that is suitable for models with loops, each message is associated with a variable node $v \in V$, an operand node $f \in F$, and the iteration $t \in \mathbb{N}$ of message passing. There are two types of messages: messages $m^{(t)}_{v \to f}$ that point from variable nodes to operand nodes and messages $m^{(t)}_{f \to v}$ that point from operand nodes to variable nodes. Every message is a unary function that maps from the domain X_v of the associated variable to the codomain Y of the function that decomposes according to the graphical model. The initialization of these messages for $t = 0$ depends on the application. For the purpose of optimization, identical constant functions are suitable.

During message passing, i.e. with increasing t, messages are updated according to the rules below. In these rules, j_f denotes the increasing sequence of all variable nodes in $N(f)$, and k_{fv} denotes the increasing sequence of all variables in $N(f) \setminus \{v\}$. Note that all possible assignments to the variables associated with $N(f) \setminus \{v\}$ are

$$(x_{k_{fv}(1)}, \ldots, x_{k_{fv}(|N(f)|-1)}) \in X_{k_{fv}(1)} \times \cdots \times X_{k_{fv}(|N(f)|-1)} \ . \tag{2.28}$$

Operand to variable update: $\forall f \in F \ \forall v \in N(f) \ \forall x_v \in X_v$:

$$m^{(t)}_{f \to v}(x_v) = \min_{(2.28)} \left(\varphi_f(x_{j_f(1)}, \ldots, x_{j_f(|N(f)|)}) \otimes \bigotimes_{v' \in N(f) \setminus \{v\}} m^{(t)}_{v' \to f}(x_{v'}) \right) \tag{2.29}$$

Variable to operand update: $\forall v \in V \ \forall f \in N(v) \ \forall x_v \in X_v$:

$$m^{(t+1)}_{v \to f}(x_v) \quad = \bigotimes_{f' \in N(v) \setminus \{f\}} m^{(t)}_{f' \to v}(x_v) \tag{2.30}$$

In each iteration $t \in \mathbb{N}$, **beliefs** are defined $\forall v \in V \ \forall x_v \in X_v$:

$$\mathrm{bel}^{(t)}_v(x_v) := \bigotimes_{f \in N(v)} m^{(t)}_{f \to v}(x_v) \ . \tag{2.31}$$

If the operand graph is a tree, all messages are guaranteed to converge (cf. Pearl, 1988). Each message $m^{(t)}_{f \to v}$ converges to the exact minimum of the function that corresponds to the subtree of the operand graph that is rooted at f and does not contain v (cf. Kschischang et al., 2001). If the graphical model has loops, BP is not guaranteed to converge. If BP converges for a loopy model, the assignment attained after convergence is an optimizer of a different function (Yedidia et al., 2000). It can be taken as an approximate solution of the original optimization problem.

BP approximations can sometimes be improved by means of **message damping** (Murphy et al., 1999). For an additive graphical model, message damping works by replacing (2.30) with the following weighted average in which $\alpha \in [0,1]$ is a damping parameter:

$$m_{v\to f}^{(t+2)}(x_v) \quad = \quad (1-\alpha)\sum_{f'\in N(v)\setminus\{f\}} m_{f'\to v}^{(t+1)}(x_v) + \alpha \sum_{f'\in N(v)\setminus\{f\}} m_{f'\to v}^{(t)}(x_v) \ . \tag{2.32}$$

Several generalizations of BP have been proposed, including tree-reweighted belief propagation (Kolmogorov, 2006; Wainwright and Jordan, 2008; Wainwright et al., 2005) which establishes a close connection between message passing and convex optimization, expectation propagation (Minka, 2001), and survey propagation (Braunstein et al., 2005).

2.1.3.4 Algorithms based on Convex Optimization

Consider the min-sum problem

$$\min_{(x_1,\ldots,x_n)\in X_1\times\cdots\times X_n} \sum_{f\in F} \varphi_f(x_{j_f(1)},\ldots,x_{j_f(|N(f)|)}) \tag{2.33}$$

and assume that $X_1,\ldots,X_n$ are finite sets. For every variable node $v\in V$, let $\mu_v : X_v \to \{0,1\}$, and for every operand node $f\in F$, let $\nu_f : X_{j_f(1)} \times \ldots \times X_{j_f(|N(f)|)} \to \{0,1\}$. The solution of (2.33) equals the solution of the **integer linear programming problem**

$$\begin{aligned}
\min_{\mu,\nu} \quad & \sum_{x\in X}\sum_{f\in F} \nu_f(x_{j_f(1)},\ldots,x_{j_f(|N(f)|)})\varphi_f(x_{j_f(1)},\ldots,x_{j_f(|N(f)|)}) \\
\text{subject to} \quad & \forall v\in V : \sum_{x\in X_v} \mu_v(x) = 1 \\
& \forall v\in V\ \forall \hat{x}\in X_v\ \forall f\in N(v) : \sum_{\{x\in X_{j_f(1)}\times\ldots\times X_{j_f(|N(f)|)} | x_{j_f^{-1}(v)}=\hat{x}\}} \nu_f(x) = \mu_v(\hat{x}) \ .
\end{aligned} \tag{2.34}$$

This follows from the fact that (i) every μ_v is an indicator function of the assignment $\mu_v^{-1}(1)$ to the variable v, i.e. $\mu_v(x)=1$ indicates that $x_v = x$, and (ii) every ν_f is an indicator function of the joint assignment $\nu_f^{-1}(1)$ to all variables in $N(f)$, i.e. $\nu_f(x)=1$ indicates that $(x_{j_f(1)},\ldots,x_{j_f(|N(f)|)}) = x$. The constraints ensure that μ and ν indicate precisely one consistent assignment of values to the variables $x_1,\ldots,x_n$.

If the problem (2.34) is small enough, it can sometimes be solved in affordable runtime by means of mixed integer linear programming (Schrijver, 1986, 2003) using branch-and-bound search (Dakin, 1965; Land and Doig, 1960). Otherwise, the corresponding **relaxed linear programming problem** where all μ_v and all ν_f map to $[0, 1]$ instead of $\{0, 1\}$ is the starting point for applying convex optimization algorithms, including tree-reweighted belief propagation (Kolmogorov, 2006; Wainwright and Jordan, 2008; Wainwright et al., 2005), and a dual decomposition ansatz using sub-gradient descent methods (Kappes et al., 2010; Komodakis et al., 2010). In Chapter 5, these algorithms are compared empirically as solvers for the optimization problem defined in Chapter 3.

2.2 Computer Vision

2.2.1 Watershed Segmentation in Linear Runtime

A variety of algorithms for image segmentation has been built on the **watershed transform** (Digabel and Lantuéjoul, 1978). A comprehensive overview is given by Roerdink and Meijster (2000). A little known fact is that the region growing algorithm by Meyer (1991) can be implemented such that it works in linear runtime (in the number of image points), provided that the elevation map can attain only finitely many values. This holds true independent of the dimension of the image and makes Meyer's algorithm a suitable candidate for segmenting large volume images. The basic idea is to replace the priority queue that is normally used for region growing by as many independent queues as there are elevation levels.

The algorithm described below differs slightly from that presented by Cousty et al. (2009). Here, a seeded region growing procedure is presented that operates on a voxel grid $G = \{1, \ldots, n_1\} \times \{1, \ldots, n_2\} \times \{1, \ldots, n_3\}$ whose 3-dimensional extent is given by $n_1, n_2, n_3 \in \mathbb{N}$. Voxels are connected in the so-called 6-neighborhood graph $(G, \sim)$ according to the following relation: $\forall v = (v_1, v_2, v_3), v' = (v'_1, v'_2, v'_3) \in G$:

$$v \sim v' \quad \Leftrightarrow \quad \sum_{j=1}^{3} |v_j - v'_j| = 1 \ . \tag{2.35}$$

Segmentation means a partitioning of G into connected components that are called segments. Algorithm 1 takes as input

- the extent $n_1, n_2, n_3 \in \mathbb{N}$ of the voxel grid,

- An **elevation map** $\varphi : G \to \{0, \dots, 255\}$,
- a **seed map** $\sigma : G \to \mathbb{N}_0$ that assigns labels to voxels such that, for each label $s \in \mathbb{N}$, the set of voxels $\sigma^{-1}(s)$ forms a connected component in $(G, \sim)$ that is termed a **seed**.

On the computer, the mappings φ and σ are stored as 3-dimensional arrays of unsigned integers, using 8 bits for each entry of φ and 32 or 64 bits for each entry of σ (32 bits resulting in $2^{32}-1$ non-zero labels identifying the same number of distinct segments are sufficient for the application in Chapter 3). Algorithm 1 labels all voxels in G by incrementally growing seeds until they touch. Ultimately, each voxel belongs to a segment and no voxel explicitly represents a boundary between segments. In fact, each voxel is assigned the label of a seed whose **min-max distance** to the given voxel is minimal (cf. Cousty et al., 2009; Nguyen et al., 2003; Turaga et al., 2009). The min-max distance between two points $v, v' \in G$ is the highest point w.r.t. the elevation map φ on the lowest path $C(v, v')$ in $(G, \sim)$ from v to v':

$$d(v, v') = \min_{C(v,v')} \max_{w \in C(v,v')} \varphi(w) \ . \tag{2.36}$$

The region growing is performed in-place on the array σ. Voxels at the border of growing regions are stored in 256 queues, one for each possible elevation level. This makes a search for candidate voxels unnecessary and thus allows the algorithm to work in linear runtime in the number of voxels. The memory overhead for the queues is bounded by the number of voxels in the surfaces of growing segments and thus by the total number of voxels. These are important assets that facilitate the segmentation of large volume images.

Algorithm 1: Seeded Region growing

```
Input: $\varphi : G \to \{0, \dots, 255\}$ (elevation map), $\sigma : G \to \mathbb{N}$ (seed
       map)
Output: $\sigma$ (segmentation)
queue $q_0, \dots, q_{255} \leftarrow \emptyset$ ;
foreach $r \in G$ do
    if isSeedBorder($\varphi$, $r$) then
        $q_{\varphi(r)}$.push($r$) ;
    end
end
for $j = 0$ to 255 do
    while $q_j \neq \emptyset$ do
        $r \leftarrow q_j$.pop() ;
        for $d = 1$ to 3 do
            if $r_d > 1$ then
                $r' \leftarrow r$;  $r'_d \leftarrow r'_d - 1$ ;
                if $\sigma(r') = 0$ then
                    $\sigma(r') \leftarrow \sigma(r)$ ;
                    $k \leftarrow \max\{\varphi(r'), j\}$ ;
                    $q_k$.push($r'$) ;
                end
            end
        end
        for $d = 1$ to 3 do
            if $r_d < n_d$ then
                $r' \leftarrow r$;  $r'_d \leftarrow r'_d + 1$ ;
                if $\sigma(r') = 0$ then
                    $\sigma(r') \leftarrow \sigma(r)$ ;
                    $k \leftarrow \max\{\varphi(r'), j\}$ ;
                    $q_k$.push($r'$) ;
                end
            end
        end
    end
end
```

Function isSeedBorder(φ, r)

```
if $\varphi(r) = 0$ then
    return false;
else
    for $d = 1$ to $3$ do
        if $r_d \neq 0$ then
            $r' \leftarrow r$ ;
            $r'_d \leftarrow r'_d - 1$ ;
            if $\varphi(r') = 0$ then
                return true;
            end
        end
    end
    for $d = 1$ to $3$ do
        if $r_d < n_d$ then
            $r' \leftarrow r$ ;
            $r'_d \leftarrow r'_d + 1$ ;
            if $\varphi(r') = 0$ then
                return true;
            end
        end
    end
    return false;
end
```

3 Automated Segmentation of SBFSEM Volume Images of Neuropil

3.1 Synopsis

The segmentation of large volume images of neuropil acquired by serial sectioning electron microscopy is an important step towards the reconstruction of neural circuits (Helmstaedter et al., 2008). This chapter presents an automated procedure that partitions the volume image into supervoxels and selectively merges supervoxels based on features derived from these. The problem which supervoxels to merge is posed as a combinatorial optimization problem in a joint graphical model over supervoxel boundaries. This joint model supersedes a previous approach in which all pairs of adjacent supervoxels are considered separately. Higher-order potentials in the new graphical model incorporate geometric information into the optimization problem. This information improves segmentations of neuropil in the inner plexiform layer of rabbit retina in a benchmark dataset of 2000^3 voxels. The potentials are learned from training data collected by one expert in three days. Runtime measurements confirm that the new procedure scales well into the gigavoxel regime. C++ code is provided.

3.2 Introduction

Overall, the automated segmentation procedure consists of eight steps:

1. From the SBFSEM volume image, a set of rotation-invariant nonlinear features is extracted, describing the 3-dimensional neighborhood of each voxel (cf. Section 3.4.1 and Fig. 3.2–3.6).

2. A classifier C_{voxel}, trained to distinguish intra-cellular from extra-cellular tissue based on these features, predicts the probability of each voxel to belong to either class (cf. Section 3.4.1 and Fig. 3.1b).

3. Based on these probabilities, the volume image is over-segmented into supervoxels (Armstrong et al., 2007; Ren and Malik, 2003) using the marker-based watershed algorithm described in Section 2.2.1 (cf. Fig. 3.1c).

4. The faces between supervoxels and the curves between these faces are represented as lists of topological coordinates (Brice and Fennema, 1970), and the neighborhood system of these objects is stored in a cellular complex (cf. Klette, 2000; Kovalevsky, 1989, 1993) using the algorithm described in Chapter 4.

5. From this explicit representation of the geometry and topology of the volume segmentation, non-local features are extracted that describe faces between supervoxels and the distribution of angles between these faces (cf. Section 3.4.3).

6. For every supervoxel face, a second classifier C_{face} predicts the probability that this face should be preserved, given its features. Moreover, for every curve in which three adjacent faces meet, a third classifier C_{curve} predicts the probabilities of the eight possible decisions to preserve or remove these faces jointly, given the angles between them (cf. Section 3.4.4).

7. The predicted probabilities from C_{face} and C_{curve} are combined as first and third order potentials in a graphical model, i.e. an energy function that depends on as many binary variables as there are supervoxel faces, indicating whether these faces should be preserved or removed (cf. Sections 2.1.2 and 3.4.4).

8. A joint optimal decision to preserve or remove supervoxel faces is found by minimizing this energy function approximately using loopy belief propagation (Kschischang et al., 2001; Pearl, 1988) with message damping (Murphy et al., 1999), and lazy flipping (Chapter 5). Results are shown in Fig. 3.1d and Section 3.5.

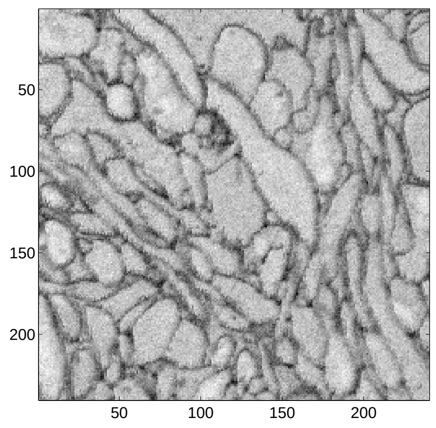
This procedure is applied to the HRP-stained SBFSEM benchmark dataset (Helmstaedter et al., 2011) that consists of $2048 \times 1792 \times 2048$ voxels and shows part of the inner plexiform layer of rabbit retina at a resolution of $22 \times 22 \times 30$ nm^3. A subset is depicted in Fig. 1.1. The homogeneous intra-cellular space makes up more than 90% of this volume image and contrasts the stained extra-cellular space that forms thin membranous faces (cf. Fig. 3.1a). Compared to previous results (Andres et al., 2008) that were obtained using only C_{voxel} and C_{face}, the number of false mergers is reduced by more than 46% while the number of false splits is reduced by 28% at the same time (Section 3.5). Although the thinnest neuronal processes are still falsely split and the 3D reconstruction of complete neural circuits remains an ambitious goal, the accuracy achieved by the new method constitutes substantial progress (cf. Fig. 3.28) over (Andres et al., 2008), and the resulting segmentations (Fig. 3.28) provide a basis for applying stitching procedures.

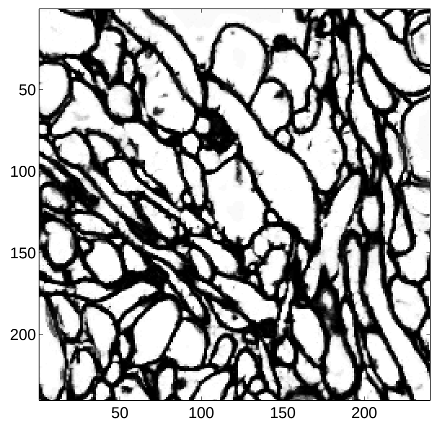
3.3 Related Work

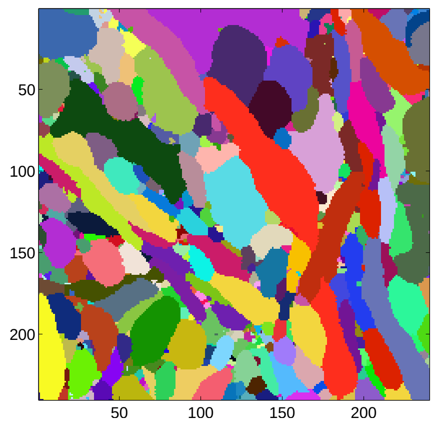
On the methodological side, the classification of voxels in Steps 1 and 2 shares aspects with edge detection in 2-dimensional images. Various statistical methods have been used to learn from manually segmented images which brightness, color and texture features identify edges: Generalized linear models by Levner and Zhang (2007); Martin et al. (2004), support vector machines by Martin et al. (2004), (boosted) classification trees by Dollar et al. (2006); Martin et al. (2004), likelihood ratio tests by Konishi et al. (2003), combinatorial search by Alpert et al. (2010), fuzzy classification by Derivaux et al. (2007), and k-means by Martin et al. (2004). Consistent improvements over traditional edge detectors are reported. The use of learned edge probabilities as elevation maps for the watershed algorithm was proposed by Derivaux et al. (2007); Levner and Zhang (2007).

Over-segmentations that capture all boundaries between objects in an image at the cost of introducing excessive splits (Step 3) have been studied by Ren and Malik (2003) who introduce the term superpixel and demonstrate the advantage of superpixel segmentations as intermediate structures that support the extraction of non-local features. A similar concept is used for volume images by Armstrong et al. (2007) who extract features from segments and the faces between segments. Steps 4 and 5 of the segmentation procedure go beyond this approach by extracting features also from curves between these faces, using the explicit representation of the geometry and topology of the volume segmentation introduced in Chapter 4.

The removal of excessive supervoxel faces in Steps 6–8 builds on existing work on hierarchical segmentation, including multi-scale watershed segmentation (Vanhamel et al., 2003) and the more closely related superpixel and supervoxel methods (Armstrong et al., 2007; Ren and Malik, 2003). While scale-space approaches are compromised by the undesirable averaging of gray values across object boundaries, region merging on the basis of statistics learned from superpixels and supervoxels can lead to significant improvements of the initial segmentation (Armstrong et al., 2007; Ren and Malik, 2003). We extend the existing methods in which boundaries between segments are classified separately by considering sets of boundaries jointly and combining the decisions to remove or preserve these boundaries

a) 

b) 

c) 

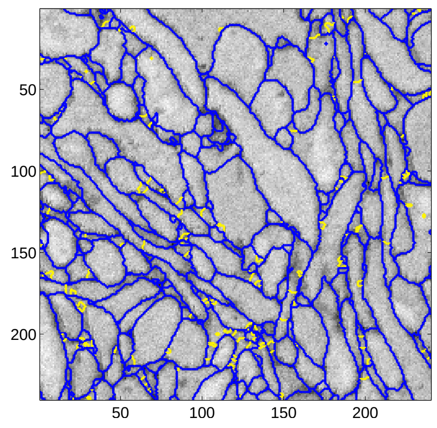
d) 

Figure 3.1: a) A yz-slice of 242^2 voxels from the HRP-stained SBFSEM benchmark volume image (Helmstaedter et al., 2011) that consists of $2048 \times 1792 \times 2048$ voxels and shows part of the inner plexiform layer of rabbit retina at a resolution of $22 \times 22 \times 30$ nm^3. b) Restoration by means of the Random Forest classifier C_{voxel}. c) Initial supervoxel segmentation. d) Faces between supervoxels classified as either essential (blue) or excessive (yellow) by the approximate solution of the combinatorial optimization problem (3.3).

in a global combinatorial optimization problem whose objective function decomposes according to a graphical model (cf. Section 2.1.2).

Graphical models have been used successfully for image analysis, e.g. by Bergtholdt et al. (2009); Besag (1986); Boykov et al. (2001); Geman and Geman (1984); Glocker et al. (2008). They describe how a multi-variate function decomposes w.r.t. an associative and commutative operation into functions that depend on subsets of all variables. The majority of graphical models used for image segmentation are Markov Random Fields (MRFs) (Geman and Geman, 1984) in which each pixel holds one variable to which a segment label is assigned. Energy functions that depend on one and two variables respectively relate these variables to an observed image and penalize label transitions. Neighboring pixels that are assigned the same label at minimal energy are grouped into segments. MRFs of this class have also been used with superpixels instead of pixels (He et al., 2006) which is preferable if superpixels are distinguishable by texture and each superpixel belongs to a specific class, e.g. foreground and background or sky, street, building, car, etc.. Labels can in this case be identified with classes.

In the problem at hand, supervoxels are indistinguishable by texture. If we were to use an MRFs of the class just described, there would be no preference of assigning one particular label to a given supervoxel. Any assignment of labels to supervoxels that leads to the same segmentation would have the same energy. The number of labels would have to be chosen large enough to encode all possible aggregations of supervoxels into segments. The objective function of the graphical model would be invariant under all changes of the labeling that lead to the same segmentation. As the number of labels would have to be large, this degeneracy would complicate the optimization.

One attempt to break this degeneracy is to assign a priori costs to the labels themselves (Delong et al., 2010). Problems remain, however, because even if the segmentation was known, finding an optimal assignment of labels to (super-)pixels is a graph coloring problem that is NP-hard for the problem at hand.

A different approach is therefore proposed here: Segmentation problems are encoded not as labeling problems on the supervoxels but as decision problems on the faces between supervoxels. Faces between supervoxels can either be removed or preserved, and segments are defined as the connected components of the supervoxels that are connected because faces inbetween have been removed.

A similar model is defined by Zhang and Ji (2010) who connect the decision and the labeling problem. We extend their work by dropping the labeling problem altogether. The graphical model built in Step 7 of the segmentation procedure contains only one binary variable for each face between adjacent superpixels, indicating whether this face is to be removed or preserved. While He et al. (2006); Zhang and Ji (2010) focus on 2-dimensional superpixel segmentations, we apply our model to a 3-dimensional supervoxel segmentation.

Energy functions that decompose according to a graphical model can be minimized efficiently in several special cases (cf. Section 2.1.2) but none of the conditions is fulfilled in this particular application. The minimization of general energy functions is an NP-hard problem (Kolmogorov and Zabin, 2004). For small enough problems, exact solutions can sometimes be found by means of integer linear programming (Schrijver, 2003) using branch-and-bound search (Dakin, 1965; Land and Doig, 1960) but the problem at hand is too large to apply this method. Among the algorithms that compute approximations in affordable runtime, loopy belief propagation (Kschischang et al., 2001; Pearl, 1988) outperforms tree-reweighted belief propagation (Wainwright and Jordan, 2008) and a dual decomposition ansatz using sub-gradient descent methods (Kappes et al., 2010; Komodakis et al., 2010) for the given problem, as shown in Chapter 5. Approximate solutions obtained by belief propagation are further improved by means of lazy flipping (Andres et al., 2010a), a generalization of iterated conditional modes (ICM) (Besag, 1986) described in Chapter 5.

On the application side, the classification of intra-cellular vs. extra-cellular space is closely related to the segmentation of SBFSEM volume images by means of convolutional neural networks (CNNs) (Jain et al., 2007; Turaga et al., 2010). These approaches complement semi-automated procedures that propagate contours of neuronal processes to subsequent slices (Macke et al., 2008) and track axons (Jurrus et al., 2009) in SBF-SEM volume images using explicit shape models. Neuron segmentation in serial section transmission EM volume images in which intra-cellular structures are stained is addressed by Chklovskii et al. (2010); Jurrus et al. (2010); Kaynig et al. (2010b). The identification of specific intra-cellular structure is addressed by Kumar et al. (2010); Lucchi et al. (2010). A semi-automated approach to reconstruct neuronal processes from fluorescent light microscopic images is presented by Lu et al. (2009).

Table 3.1: Local structure in the SBFSEM volume image is restored by predicting the probability of each voxel to belong to the intra-cellular space, based on 28 rotation-invariant non-linear features, including the gradient magnitude, eigenvalues of the Structure Tensor (Bigun, 2005) and the Hessian matrix. These features that are depicted in Fig. 3.2–3.6 are computed for the volume image as well as from the output of a 4-fold iterated bilateral filter for which the functions w_{σ_s} and w_{σ_v} with scale parameters σ_s and σ_v are used in the spatial and the intensity domain. Derivatives are computed by Derivative-of-Gaussian filters at scales σ_m, σ_i and σ_h. Entries of the Structure Tensor are averaged with a Gaussian filter at scale σ_o.

Index	Feature	Scale
1	SBFSEM volume image	
2	Gradient magnitude	$\sigma_m = 1$
3-8	Structure Tensor eigenvalues	$\sigma_i \in \{1, 1.5\}, \sigma_o = 2\sigma_i$
9-14	Hessian matrix eigenvalues	$\sigma_h \in \{1, 1.5\}$
15	Iterated bilateral filter (Barash, 2002) $w_{\sigma_s}(r) = \frac{1}{\sigma_s(2\pi)^{3/2}} \exp\left(-\frac{r^2}{2\sigma_s^2}\right)$ $w_{\sigma_v}(v) = \frac{1}{1+\frac{v^2}{\sigma_v^2}}$	$\sigma_s = 1, \sigma_v = 3$
16-28	Analogous to 2-14 but on 15 instead of 1	

3.4 Automated Segmentation

3.4.1 Restoration of Local Structure by Voxel Classification

Steps 1 and 2 of the segmentation procedure outlined in the introduction serve to enhance the contrast between intra-cellular and extra-cellular space and to close local gaps in the extra-cellular space. Towards this goal, 28 rotation-invariant non-linear features (Tab. 3.1, Fig. 3.2–3.6) are extracted from the volume image, describing the distribution of gray values in a $31 \times 31 \times 31$ neighborhood around each voxel. The 28-dimensional feature vector of each voxel is mapped to an estimated probability of this voxel to belong to the intra-cellular space. This mapping is not hard-coded into the algorithm but learned automatically from hand-labeled data (Section 3.5) by means of a Random Forest classifier (C_{voxel}).

Random Forests (Breiman, 2001) are ensembles of decision trees. Their construction during learning is described in Section 2.1.1. Each decision tree partitions the feature space into subsets, classifying each subset as either intra-cellular or extra-cellular. The fraction of decision trees in the ensemble that classify a given feature vector as intra-cellular is an estimate of the probability of the corresponding voxel to belong to this class. The predicted probabilities of all voxels form a 3-dimensional **probability map** that is essentially a restored and contrast-enhanced version of the SBFSEM volume image (Fig. 3.7).

Feature extraction and Random Forest prediction are operations whose runtime complexity is linear in the number of voxels. The construction of decision trees during learning has log-linear runtime in the number of training samples which is negligible in this application, compared to the prediction time. Absolute runtimes are summarized in Section 3.5. The C++ code of the Random Forest and the volume image features are available as part of the image processing library Vigra[1] (Köthe, 2000).

3.4.2 Supervoxel Segmentation

To permit the extraction of non-local and geometric features, the restored volume image is over-segmented into supervoxels (Armstrong et al., 2007; Ren and Malik, 2003) such that as few neurons as possible are incorrectly merged (cf. Section 3.5), at the cost of introducing excessive splitting faces that do not correspond to cellular membranes. These excessive faces are removed later, in Step 7 (Section 3.4.4), based on non-local features.

[1] `http://hci.iwr.uni-heidelberg.de/vigra`

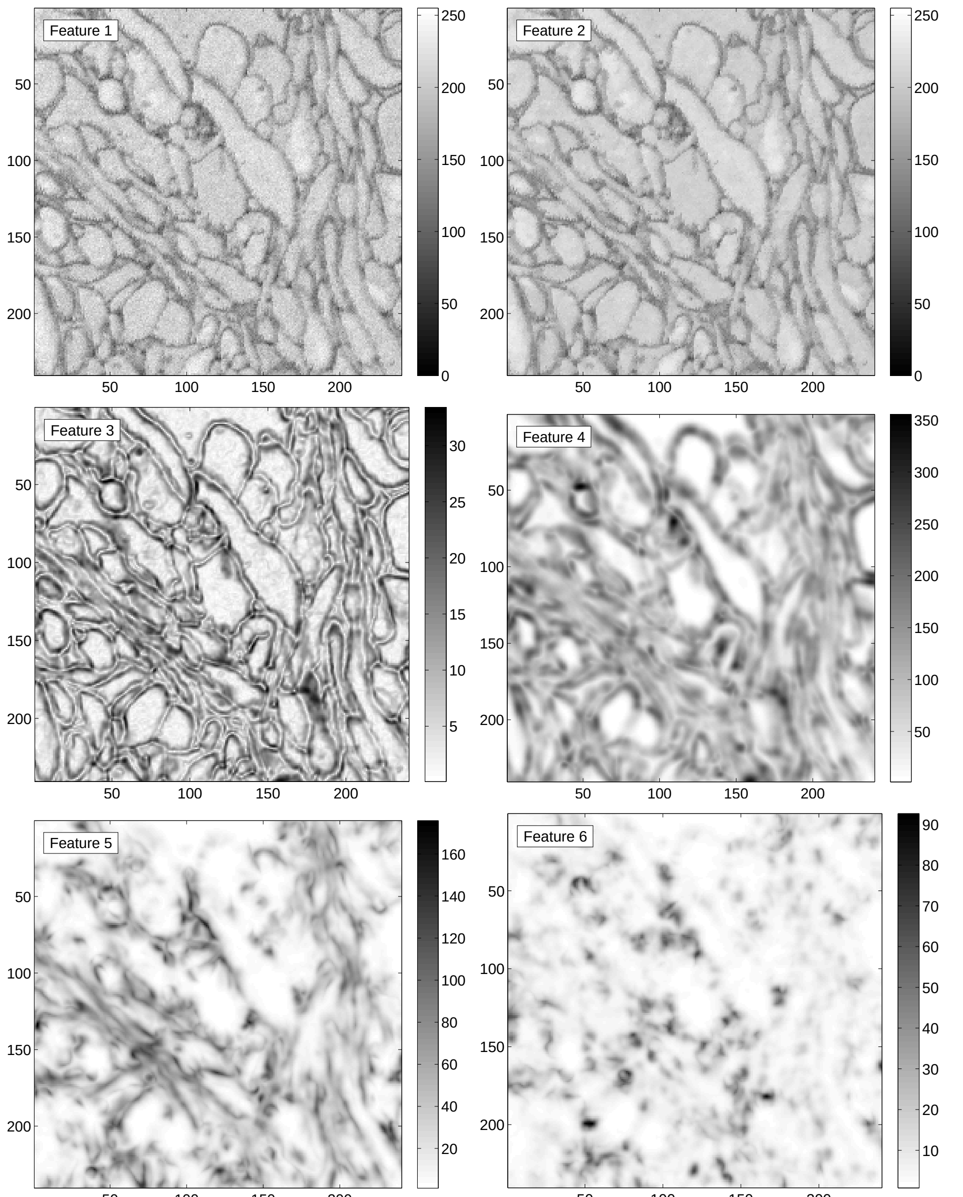


Figure 3.2: Voxel neighborhood features 1–6

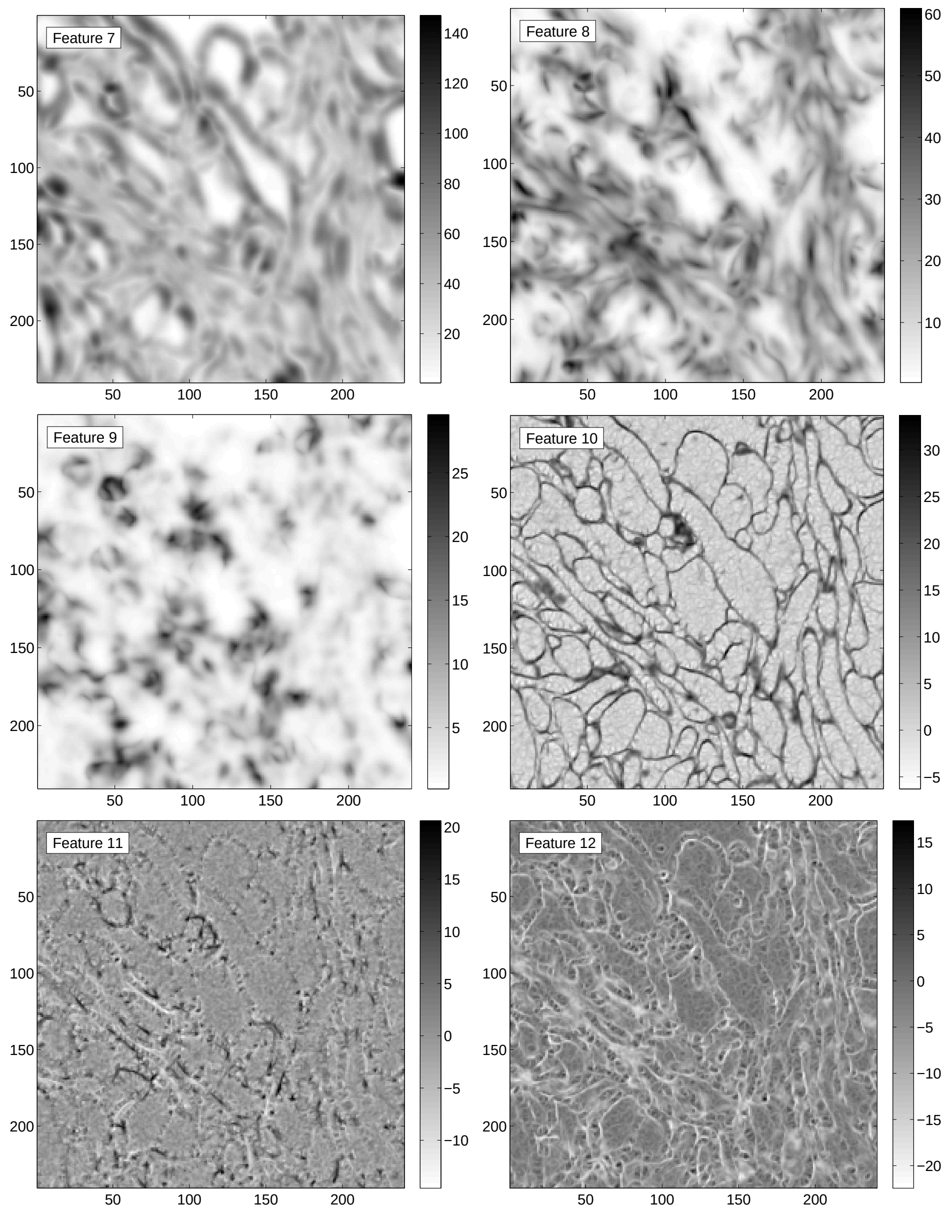


Figure 3.3: Voxel neighborhood features 7–12

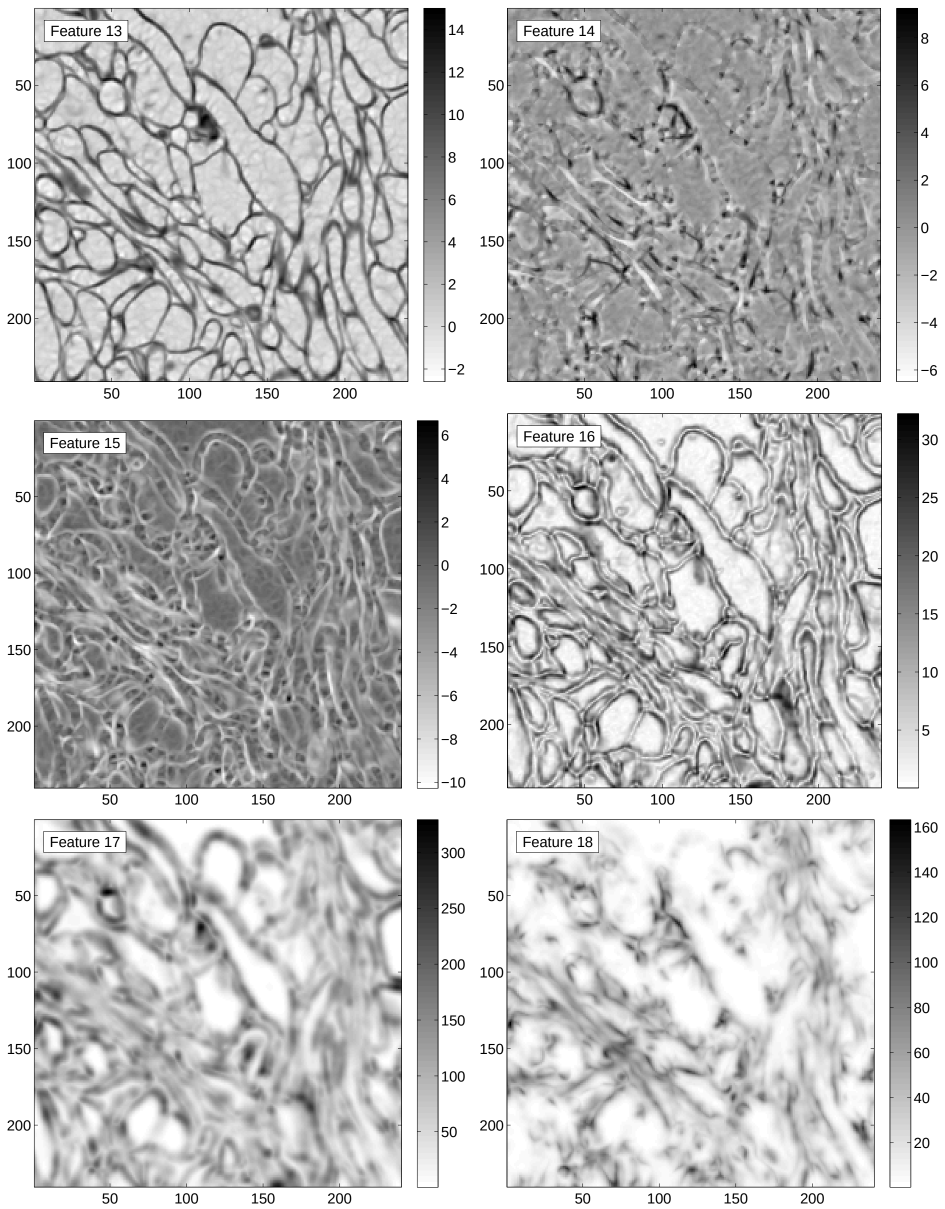


Figure 3.4: Voxel neighborhood features 13–18

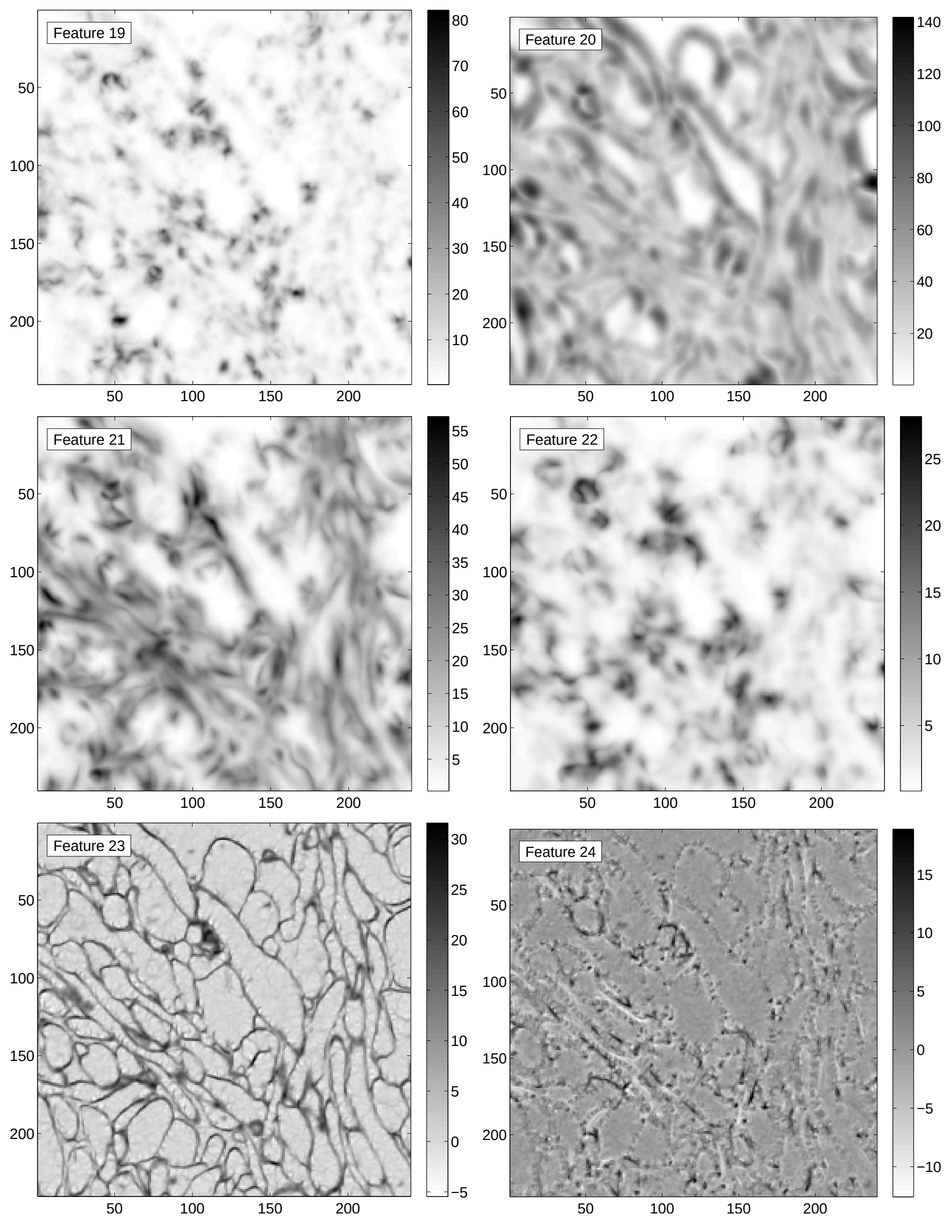


Figure 3.5: Voxel neighborhood features 19–24

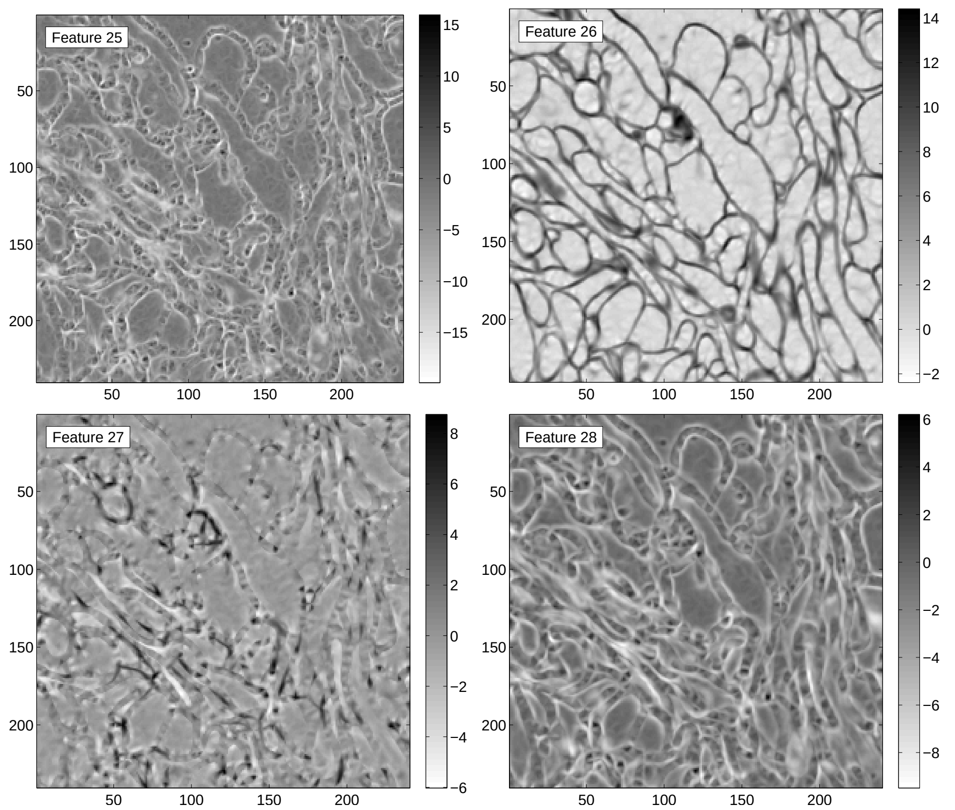


Figure 3.6: Voxel neighborhood features 25–28

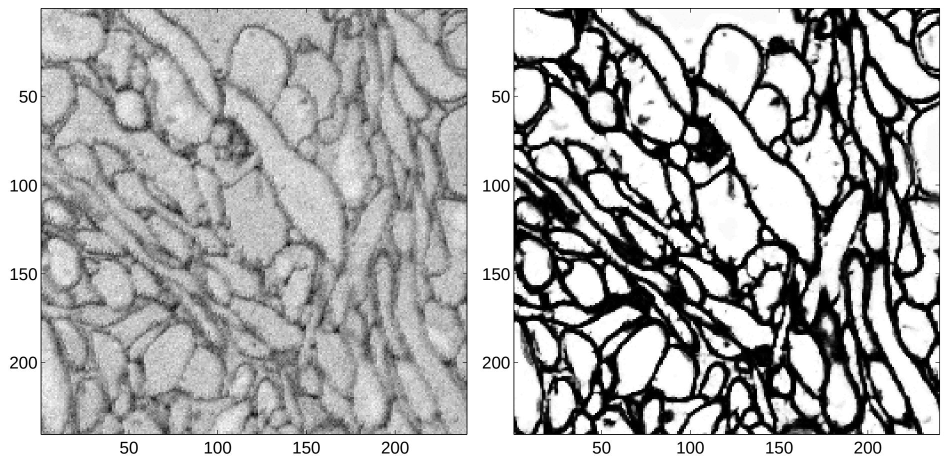


Figure 3.7: SBFSEM raw data and restoration predicted by C_{voxel}.

The initial supervoxel segmentation is found by means of the marker-based watershed algorithm described in Section 2.2.1. Markers are defined as the connected components of those voxels that are classified as intra-cellular space by **all** decision trees of the Random Forest C_{voxel}, i.e. those voxels whose predicted probability to represent intra-cellular space is 1 (Fig. 3.8a). These markers serve as supervoxel seeds. They are grown until the entire volume is occupied such that each voxel is assigned the label of a seed whose **min-max distance** to the given voxel is minimal (cf. Section 2.2.1).

The runtime of this algorithm is linear in the number of voxels. Its output is a 3-dimensional **segment label map** (Fig. 3.8b) that assigns a segment label to each voxel. The segment label map encodes faces between supervoxels and the curves between these faces only implicitly, as neighboring voxels whose segment labels differ. Moreover, it does not store explicitly which segments are adjacent, separated by which faces and in which curves adjacent faces meet. Inspired by the work of Armstrong et al. (2007), we have developed algorithms and data structures (Chapter 4) that encode every face between two supervoxels and every curve between supervoxel faces as a list of topological coordinates (Brice and Fennema, 1970) and store the adjacency of these objects in a cellular complex (Klette, 2000; Kovalevsky, 1989, 1993). This representation is constructed in a runtime that is linear in the number of voxels and log-linear in the number of faces and curves (cf. Chapter 4). It facilitates the extraction of non-local features from the supervoxel segmentation based on which excessive faces are identified and removed.

3.4.3 Extraction of Contextual Features

In order to learn from hand-labeled training data (Section 3.5) which gray value structures distinguish essential from excessive supervoxel faces and which configurations of adjacent supervoxel faces support a cell boundary, contextual features are extracted from the supervoxel segmentation (Step 5), 31 features of supervoxel faces (Tab. 3.2) and 21 features of curves between adjacent faces that describe the distribution of angles between the faces along the curve.

The estimation of angles between supervoxel faces is described in the following and illustrated in a video contained in the supplementary material. For each point P_j on a given curve $(P_1, \dots, P_n)$ between three adjacent faces, all points on these faces that lie within a radius of 7 voxels from P_j are averaged, leading to average points A_j, B_j and C_j on the re-

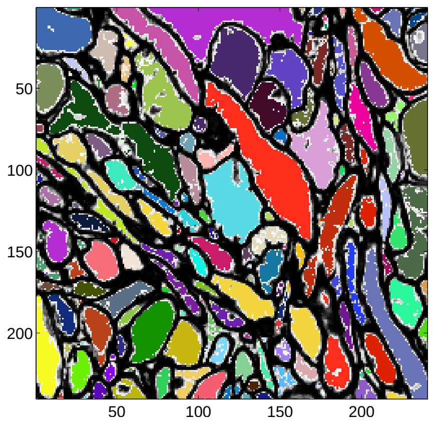
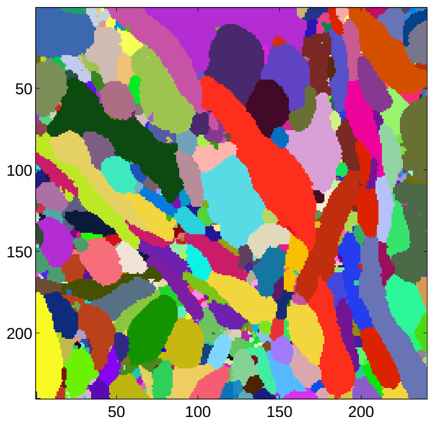
a)  b) 

Figure 3.8: The initial supervoxel segmentation is found by means of the marker-based watershed algorithm described in Section 2.2.1. Markers (a) are defined as the connected components of those voxels that are classified as intra-cellular space by **all** decision trees of the Random Forest C_{voxel}, i.e. those voxels whose predicted probability to represent intra-cellular space is 1. These markers serve as supervoxel seeds. They are grown until the entire volume is occupied such that each voxel is assigned the label of a seed whose **min-max distance** to the given voxel is minimal (b).

Table 3.2: From every supervoxel face and the adjacent supervoxels, 31 statistical features are extracted. The size of a supervoxel face is estimated by the number of topological coordinates, the size of a supervoxel by the numbers of voxels. The statistics $\mathcal{S}$ include the mean, standard deviation, minimum, 0.25-quantile, median, 0.75-quantile, and maximum over all voxels that are adjacent to the supervoxel face.

Index	Feature	Details
1	Size of the face	
2–3	Sizes v_1 and v_2 of adjacent supervoxels	$(v_1+v_2)^{1/3}$, $\lvert v_1-v_2\rvert^{1/3}$
4–10	Bilateral filter output	$\mathcal{S}$
11–17	Gradient magnitude of bilateral filter	$\mathcal{S}$
18–24	Hessian matrix of bilateral filter	greatest eigenvalue, $\mathcal{S}$
25–31	Probability predicted by C_{voxel}	$\mathcal{S}$

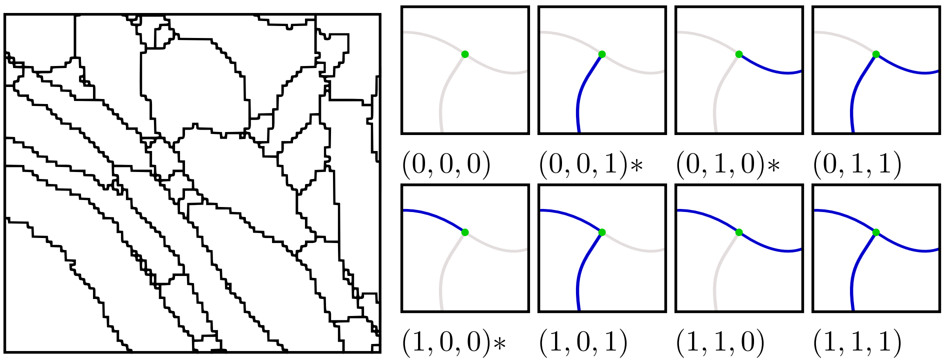


Figure 3.9: Humans can often distinguish excessive from essential supervoxel faces without seeing the SBFSEM volume image at all, merely from the configuration or gestalt of supervoxel faces (left). There are 8 possibilities (right) to preserve or remove three supervoxel faces (blue) that meet in one curve (green). In the biological problem studied, sharp edges, $(0, 1, 1)$ and $(1, 0, 1)$, are unlikely, and smooth continuations, $(1, 1, 0)$, are more probable than junctions, $(1, 1, 1)$. Isolated edges (*) do not occur because all surfaces in the SBFSEM volume image are closed. The probabilities of all other configurations are learned from hand-labeled data by the Random Forest C_{curve}.

spective face and to angles $\alpha_j = \angle(A_j, P_j, B_j)$, $\beta_j = \angle(B_j, P_j, C_j)$ and $\gamma_j = \angle(C_j, P_j, A_j)$ between the faces at P_j. For each point P_j on the curve, a separate triple $(\alpha_j, \beta_j, \gamma_j)$ of angles is obtained. The mean, the standard deviation, the minimum, the 0.25-quantile, the median, the 0.75-quantile and the maximum of the sequences (α_j), (β_j), and (γ_j) make a total of 21 features that are associated with the curve.

The motivation behind the extraction of angles is that humans can often distinguish excessive from essential supervoxel faces without seeing the SBFSEM volume image at all, merely from the configuration or gestalt of supervoxel faces. It is, for instance, obvious to humans that smooth continuations of supervoxel faces are more probable in neural tissue than sharp edges, the latter indicating possible over-segmentation (Fig. 3.9). The improvement achieved by learning and incorporating this information is shown in Section 3.5.

3.4.4 Removal of Over-Segmentation

The last step towards the construction of the final segmentation is to remove remaining over-segmentation, i.e. to identify and remove excessive supervoxel faces based on the features just described. Three approaches are considered in the following and compared experimentally in Section 3.5.

In all approaches, unique indices are assigned to each of the $n_f \in \mathbb{N}$ supervoxel faces and to each of the $n_c \in \mathbb{N}$ curves between faces. Every supervoxel face j is associated with a binary variable $x_j \in \{0, 1\}$ that indicates whether this face is to be removed ($x_j = 0$) or preserved ($x_j = 1$). Due to the discrete structure of the voxel grid, curves have either three or, rarely, four adjacent faces (cf. Chapter 4). The sets $N_3 \subseteq \{1, \ldots, n_c\}$ and $N_4 \subseteq \{1, \ldots, n_c\}$ contain the indices of the respective curves. The indices of the adjacent faces are stored in the rows of matrices $A \in \mathbb{N}^{|N_3| \times 3}$ and $B \in \mathbb{N}^{|N_4| \times 4}$.

In **Approach A**, a Random Forest C_{face} is trained on hand-labeled data (Section 3.5) to distinguish essential from excessive faces based exclusively on the features in Tab. 3.2, exactly as proposed in (Andres et al., 2008). The distribution of angles between faces is ignored in this approach. For every face j, C_{face} predicts a probability $p_j^{(1)}(0)$ that this face should be removed and the corresponding probability $p_j^{(1)}(1) = 1 - p_j^{(1)}(0)$ that this face should be preserved, i.e. one probability mass function $p_j^{(1)} : \{0, 1\} \to [0, 1]$ for every face j. The final segmentation is obtained by removing all faces for which $p_j^{(1)}(1)$ is below a **confidence level** $\beta \in [0, 1]$. This confidence level establishes a trade-off between false removals (under-segmentation) and false preservations (over-segmentation) (Section 3.5).

In **Approach B**, the decision is based on a crude representation of the gestalt of the over-segmentation, namely the 21 features that describe the angles between supervoxel faces. All features in Tab. 3.2 are ignored in this approach. A Random Forest C_{curve} is trained on hand-labeled data (Section 3.5) to predict for every curve $k \in N_3$ between three adjacent faces a_{k1}, a_{k2}, a_{k3} the probability of each of the eight possibilities to remove or preserve these faces (Fig. 3.9). One probability mass function $p_k^{(3)} : \{0,1\}^3 \to [0,1]$ is thus obtained for every curve k (Tab. 3.3). It translates into an energy function $E_k^{(3)} : \{0,1\}^3 \to \bar{\mathbb{R}}$ that is minimal where the probability $p_k^{(3)}$ is maximal (Tab. 3.3). Curves at which only one adjacent face is to be preserved (Fig. 3.9) do not occur in the SBFSEM volume image that contains only closed surfaces. The energy of these configurations is therefore set to infinity. The same argument applies to curves $k \in N_4$ with four adjacent faces $b_{k1}, \ldots, b_{k4}$ which motivates the introduction of additional 4th order energy functions that suppress the 4 out of the 16 possible configurations in which only one face is preserved[2]:

$$E_k^{(4)} : \{0,1\}^4 \to \bar{\mathbb{R}} \quad \text{such that} \quad \forall x_1, \ldots, x_4 \in \{0,1\} :$$
$$E_k^{(4)}(x_1, \ldots, x_4) = \begin{cases} \infty & \text{if } x_1 + \ldots + x_4 = 1 \\ 0 & \text{otherwise} \ . \end{cases} \tag{3.1}$$

It happens in practice that predictions from different curves that delimit the same face lead to conflicting recommendations as to preserve or remove the given face. Moreover, the decision to preserve or remove one face has implications on the decisions for the neighboring faces, and these implications propagate across the adjacency graph of faces. The final segmentation can thus no longer be obtained by a separate classification of supervoxel faces but requires the joint minimization of the total energy comprising the 3rd order potentials of Tab. 3.3 and the 4th order potentials (3.1), i.e. the solution of the following combinatorial optimization problem:

$$\min_{x \in \{0,1\}^{n_f}} \left(\sum_{k \in N_3} E_k^{(3)}(x_{a_{k1}}, x_{a_{k2}}, x_{a_{k3}}) + \sum_{k \in N_4} E_k^{(4)}(x_{b_{k1}}, x_{b_{k2}}, x_{b_{k3}}, x_{b_{k4}}) \right) \tag{3.2}$$

[2]Learning to distinguish the remaining 12 configurations based on angles, similar as in Tab. 3.3, would require substantially more training data and increase the labeling time several times over, an effort that is not justified given that most curves have only three adjacent faces. Thus, deterministic potentials (3.1) are used.

Table 3.3: Probabilities $p_k^{(3)}$ predicted by C_{curve} for configurations of three adjacent supervoxel faces, and corresponding energies $E_k^{(3)} = -\log p_k^{(3)}$.

$x_{a_{k1}}$	$x_{a_{k2}}$	$x_{a_{k3}}$	$p_k^{(3)}(x_{a_{k1}}, x_{a_{k2}}, x_{a_{k3}})$	$E_k^{(3)}(x_{a_{k1}}, x_{a_{k2}}, x_{a_{k3}})$
0	0	0	$p_k(0,0,0)$	$-\log p_k(0,0,0)$
0	0	1	0	∞
0	1	0	0	∞
0	1	1	$p_k(0,1,1)$	$-\log p_k(0,1,1)$
1	0	0	0	∞
1	0	1	$p_k(1,0,1)$	$-\log p_k(1,0,1)$
1	1	0	$p_k(1,1,0)$	$-\log p_k(1,1,0)$
1	1	1	$p_k(1,1,1)$	$-\log p_k(1,1,1)$

Approach C is the combination of A and B. For every supervoxel face j, the function $p_j^{(1)}$ obtained from C_{face} translates into a unary potential $E_j^{(1)} = -\log p_j^{(1)}$ that assigns a higher energy to the less likely decision. These unary potentials are added to the objective function in (3.2), leading to the following joint optimization problem that specializes to Approach A for $\alpha = 0$ and to Approach B for $\alpha = 1$. The **mixture parameter** $\alpha \in [0,1]$ is an adjustable parameter of the processing chain; it is optimized on training data (Section 3.5).

$$\min_{x \in \{0,1\}^{n_f}} \left((1-\alpha) \sum_{j=1}^{n_f} E_j^{(1)}(x_j) + \alpha \sum_{k \in N_3} E_k^{(3)}(x_{a_{k1}}, x_{a_{k2}}, x_{a_{k3}}) \right. \\ \left. + \alpha \sum_{k \in N_4} E_k^{(4)}(x_{b_{k1}}, x_{b_{k2}}, x_{b_{k3}}, x_{b_{k4}}) \right) . \qquad (3.3)$$

Finding solutions of (3.2) and (3.3), even approximately, is a formidable problem. The objective functions are not submodular (cf. Section 2.1.3.1). In particular, the projections $E_k^{(3)}(1,\cdot,\cdot)$ are supermodular whereas the projections $E_k^{(3)}(0,\cdot,\cdot)$ are submodular (cf. Tab. 3.3). An exhaustive search over all permutations of labels shows that these functions are not permuted submodular either which rules out polynomial-time optimization by graph cuts. Furthermore, a typical current dataset contains in the order of one million faces and ten million curves. For small problems obtained from subsets of 150^3 voxels (less than 1/2000) of the dense interior region of the SBFSEM benchmark dataset, global optima of (3.2) and (3.3) can be found by means of mixed integer linear programming (MILP) using branch-and-bound search, as shown in Chapter 5. Less than 8 GB of RAM and a few minutes of runtime are sufficient in this case. However, MILP is unsuitable for large models. More than 512 GB of RAM would be required for models obtained from 1000^3 voxels. State-of-the-art approximate solvers are therefore assessed in Chapter 5 on ten subsets of 150^3 voxels where the energies can be compared to the global optimum found by MILP. The results shown in Chapter 5 indicate that loopy belief propagation (Kschischang et al., 2001; Pearl, 1988) with message damping (Murphy et al., 1999), performs exceptionally well, significantly outperforming both tree-reweighted belief propagation (BP) (Wainwright and Jordan, 2008) and a dual decomposition ansatz using sub-gradient descent methods (Kappes et al., 2010; Komodakis et al., 2010) on this problem. Approximate solutions found by BP deviate by only 0.4% on average from the global optimum and are improved further (to 0.1%) by means of lazy flipping (Chapter 5). Here, we therefore use a combination of BP and lazy flipping to solve (3.2) and (3.3) approximately.

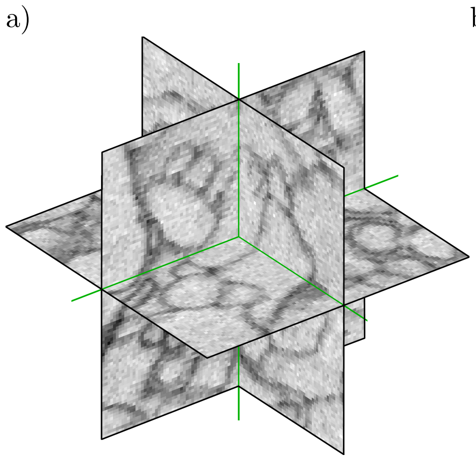


Figure 3.10: The individual steps and the final result of the segmentation procedure are evaluated on the gold standard validation dataset, a sub-block of 100^3 voxels (a) in which all intra-cellular voxels have been labeled manually (b). The labels have been provided by Moritz Helmstaedter and Winfried Denk.

3.5 Application to SBFSEM Volume Data

We apply the segmentation procedure to the HRP-stained SBFSEM benchmark dataset (Helmstaedter et al., 2011), a volume image of $2048 \times 1792 \times 2048 \approx 7.5 \cdot 10^9$ voxels that shows part of the inner plexiform layer (IPL) of rabbit retina at a resolution of $22 \times 22 \times 30$ nm^3 (Fig. 1.1). The performance is evaluated quantitatively and compared to the previous approach (Andres et al., 2008) on the gold standard validation subset of 100^3 voxels in which all intra-cellular voxels have been labeled by hand (Fig. 3.10). In addition, for a qualitative impression, reconstructions of neurons from the entire volume image are compared to reconstructions of the same neurons found by means of (Andres et al., 2008).

The Random Forests C_{voxel}, C_{face} and C_{curve} are learned from training data collected in two blocks of 150^3 voxels, one from the dense inside of the IPL where neuronal processes intertwine, the other from the border of the IPL. These blocks have no overlap with the validation set and make up less than 0.1% of the volume image. For C_{voxel}, 3200 voxels are labeled as either intra-cellular or extra-cellular using the MATLAB tool contained in the supplementary material, starting with 500 voxels per class that are placed at least 5 voxels away from each other. The classifier is then trained, and the predicted probability maps for the training volumes are loaded in the labeling tool. Another 500 voxels per class are labeled where the probability maps need improvement. The procedure is repeated another two times, labeling 300 voxels per class. Incremental labeling according to this protocol takes an expert approximately one day. All Random Forests in the segmentation procedure are constructed with 255 decision trees, a number that is large enough for the learning curve to converge and small enough to store the prediction in 8 bits. The contrast enhancement achieved by the trained Random Forest C_{voxel} on validation data is depicted in Fig. 3.1. On a maximal random stratified subset of the validation set that contains as many intra-cellular voxels as extra-cellular voxels, 91.8% of all voxels are classified in agreement with the manual tracing. 8.2% are classified in disagreement with the manual tracing, 5.1% as extra-cellular space and 3.1% as intra-cellular space.

While a hard thresholding of these predictions fails to give accurate segmentations, the topography of the probability map can still be exploited to obtain a proper over-segmentation by means of the watershed transform. It is crucial that as little under-segmentation as possible is introduced at this stage because false mergers could not be corrected in subsequent steps of the procedure as presented here. The following indicator of under-segmentation corroborates this assumption: For every connected component j of voxels labeled as intra-cellular in the validation set and every segment k in the watershed segmentation of the same volume, Q_{jk} denotes the number of voxels that belong at the same time to the connected component (true segment) j and to the watershed segment k. Let R be obtained from the overlap matrix Q by column normalization. R_{jk} is then the fraction of the watershed segment k overlapped by the true segment j. In an over-segmentation, all except very small segments have no relevant overlap with more than one true segment. This is quantified by the under-segmentation index, the second largest entry of the k-th column of R. In this application, **all** watershed segments that are larger than 100 voxels exhibit an under-segmentation index of less than 10%.

Faces between supervoxels and the curves between these faces are extracted from the segmentation as lists of topological coordinates, along with the adjacency of these objects. Features of individual faces and the angles between adjacent faces are computed. The absolute runtime, memory consumption and parallelization of these and all other processing steps are summarized in Tab. 3.6.

To train C_{face} and C_{curve}, 5000 supervoxel faces are labeled by hand in the watershed segmentations of the training volumes using an interactive tool

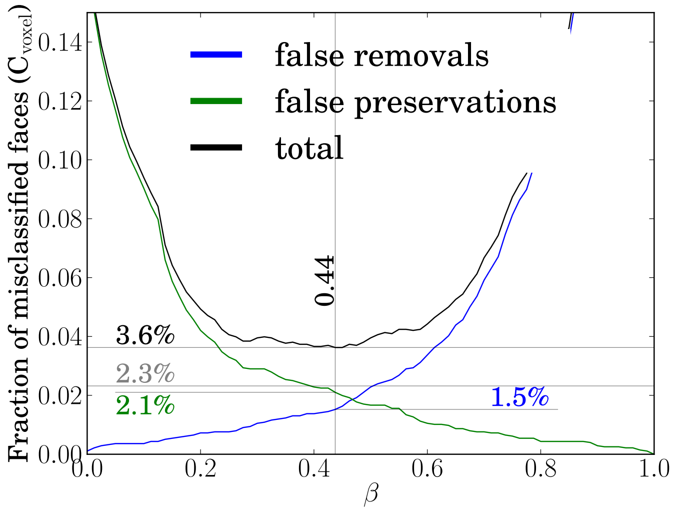
Table 3.4: Quality of automatically computed segmentations in terms of the fraction of falsely preserved and falsely removed faces in the initial over-segmentation of the gold standard validation dataset (Fig. 3.10). F/C means false/correct. P/R means preservation/removal. All values are given in percent.

	FR	**FP**	**CR**	**CP**	**C**
Approach A ($\beta = 0.45$)	1.5	2.1	17.2	79.2	96.4
Approach B	6.1	5.3	13.9	74.7	88.6
Approach C ($\alpha = 0.17$)	**0.8**	**1.5**	**17.7**	**80.0**	**97.7**

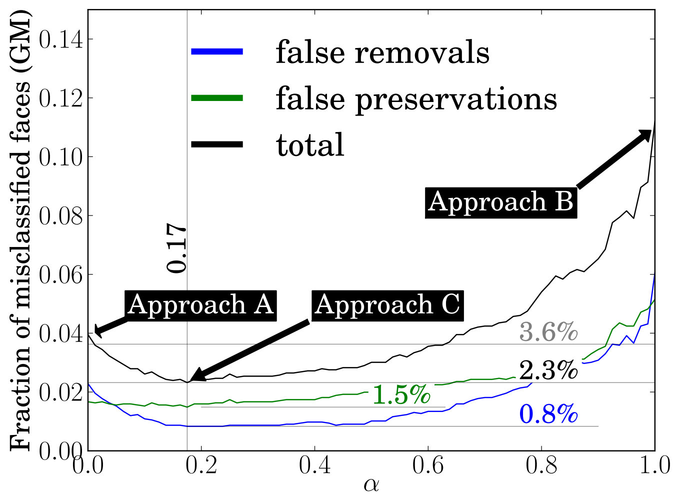
(Kröger, 2010) based on the Visualization Toolkit (`www.vtk.org`). Faces that are easy to label are labeled first because less obvious decisions become clear once the surrounding faces are labeled. It takes an expert roughly two days to collect a training set of the given size. The labeled faces and the features of these faces make up a training set for C_{face}. Triples of adjacent labeled faces and the angles between these faces are used to train C_{curve}. The training set is extended by adding all permutations of adjacent supervoxel faces. As an example, a triple where the first face is labeled as excessive and the last two faces are labeled as essential $(0, 1, 1)$ makes up three items labeled $(0, 1, 1)$, $(1, 0, 1)$ and $(1, 1, 0)$ in the training set for which the angle features are permuted accordingly.

Over-segmentation is removed by identifying and removing excessive supervoxel faces. Approaches A, B and C described in Section 3.4.4 are compared. Optimization is performed by loopy belief propagation (75 steps) with a message damping of 0.3, followed by lazy flipping (Chapter 5), using a maximum subgraph size of 3. The quality of the final segmentation is measured on the validation set in terms of the fraction of falsely removed and falsely preserved supervoxel faces.

For Approach A, the trade-off between false removals and false preservations w.r.t. the confidence level β is depicted in Fig. 3.11a. At the optimal $\beta = 0.45$, 96.4% of all faces are classified correctly (Tab. 3.4). Approach B performs worse, classifying 88.6% of all faces correctly (Tab. 3.4). The reason can be seen in the confusion matrix of C_{curve} (Tab. 3.5): The configurations $(0, 1, 1)$, $(1, 0, 1)$ and $(1, 1, 0)$ are separated well by the angle features which demonstrates their predictive power. However, it is not possible to distinguish these configurations from $(1, 1, 1)$ based on the angles alone. This motivates the combination of the predictions from C_{face} and C_{curve} in Approach C. Approach C yields the overall best results for all



a)



b)

Figure 3.11: The quality of automatically computed segmentations is evaluated in terms of the fraction of falsely preserved and falsely removed faces in the initial over-segmentation of the gold standard validation dataset (Fig. 3.10). Falsely preserved and falsely removed faces respectively lead to over-segmentation (false splits) and under-segmentation (false mergers). a) Segmentations computed using Approach A with different confidence levels β, b) Segmentations computed using the full graphical model (Approach C) with different mixture parameters α. Approach B corresponds to the setting $\alpha = 1$. It can be seen from these figures that the full graphical model (Approach C) outperforms Approaches A and B, reducing both the number of false mergers and false splits (cf. Tab. 3.4).

Table 3.5: At every curve in which three supervoxel faces meet, the classifier C_{curve} predicts a probability for each of the eight possibilities to remove (0) or preserve (1) these faces (cf. Fig. 3.9). The three configurations $(0, 0, 1)$, $(0, 1, 0)$ and $(1, 0, 0)$ in which only one face is preserved do not occur in the SBFSEM volume image that contains only closed surfaces. The table below shows the confusion of the classifier C_{curve} for the remaining five configurations. Columns correspond to the predictions, rows to the truth. It can be seen from this table that the configurations $(0, 1, 1)$, $(1, 0, 1)$ and $(1, 1, 0)$ are well separated whereas it is not possible to distinguish these configurations from $(1, 1, 1)$ by C_{curve} alone. This motivates the combination of C_{face} and C_{curve} in the graphical model (3.3).

C_{curve}:	$(0,0,0)$	$(1,1,0)$	$(1,0,1)$	$(0,1,1)$	$(1,1,1)$
$(0,0,0)$	**1.9%**	0.7%	0.9%	0.5%	0.4%
$(1,1,0)$	1.7%	**6.9%**	0.9%	0.7%	2.9%
$(1,0,1)$	2.6%	0.9%	**6.7%**	1.1%	3.2%
$(0,1,1)$	2.3%	0.7%	1.6%	**5.3%**	2.9%
$(1,1,1)$	4.6%	5.0%	6.8%	5.2%	**33.8%**

mixture parameters α between 0.05 and 0.6 (Fig. 3.11b). At the optimal $\alpha = 0.17$, 97.7% of all faces are classified correctly (Tab. 3.4). Compared to Approach A, the fraction of falsely removed faces is reduced by more than 46% to 0.8%. At the same time, the fraction of false preservations is reduced by 28% to 1.5% (cf. Tab. 3.4). For a visual comparison of the three approaches, cf. Fig. 3.12–3.15.

Reconstructions of neurons are possible using Approaches A and C. Approach B produces severe under-segmentation because the false removal rate is too high. This rate strongly affects the quality of large scale reconstructions because neurons in the SBFSEM volume image have between 1000 and 10000 supervoxel faces all of which need to be correctly preserved. In practice, a false removal rate below 1% is indispensable to prevent under-segmentation in large scale reconstructions. This can be achieved with Approach A by setting $\beta = 0.2$ and by Approach C with $\alpha = 0.17$. It can be seen from Fig. 3.24 that the full-featured Approach C affords better reconstructions; less boundaries are falsely preserved and thus, larger parts of neuronal processes are correctly merged.

Reconstructions of neural processes are shown in Fig. 3.25–3.28. These reconstructions are not perfect: Over-segmentation still occurs at points where neuronal processes shrink in diameter to one voxel in the SBFSEM

Table 3.6: Absolute runtime, memory consumption and parallelization of the segmentation procedure on an Intel 4×Quad Xeon equipped with 128 GB RAM, running at 2.4 GHz. *) adjustable by selecting a block. **) per CPU. size.

Computation	**Runtime**	**CPUs**	**RAM****	**HDD**
Voxel features	1d 11h 59m	10	< 2 GB*	842 GB
Training of C_{voxel}	< 10m	1	< 2 GB	< 1 GB
Voxel classification (C_{voxel})	11h 39m	10	< 2 GB*	8 GB
Supervoxel segmentation	2h 13m	1	72 GB	61 GB
Geometry extraction	19h 22m	10	< 2 GB*	273 GB
Supervoxel face features	2d 14h 28m	10	< 2 GB	3 GB
Training of C_{face}	< 10m	1	< 2 GB	< 1 GB
Face classification (C_{face})	< 10m	10	< 2 GB	< 1 GB
Curve angle features	1h 25m	10	< 2 GB	2 GB
Training of C_{curve}	< 10m	1	< 2 GB	< 1 GB
Curve classification (C_{curve})	< 10m	10	< 2 GB	< 1 GB
Lazy Flipper ($d = 3$)	2d 9h 38m	1	89 GB	< 1 GB

volume image. However, making all supervoxel face preservation/removal decisions **jointly** as in Approach C constitutes substantial progress over the predictions from Approach A.

3.6 Conclusion

An automated procedure for segmenting SBFSEM volume images of neuropil is presented. It starts by over-segmenting the volume image into supervoxels and selectively merges supervoxels based on non-local features of the over-segmentation. The coupling of the individual decisions in a joint optimization problem allows clear evidence for the existence or absence of a cell boundary to propagate over long distances to more dubious regions. The representation used and the training set provided allow the inference procedure to learn by itself some gestalt laws that are most salient for the problem at hand. Compared to a previous approach where merging decisions are made separately, the fraction of false mergers is reduced by more than 46% to 0.8%. The fraction of false splits is reduced by 28% to 1.5% at the same time. The thinnest neuronal processes are still falsely split, but advances in tissue preparation and imaging may alleviate this problem in the future. In general, the method should carry over to other biological or medical images that bear resemblance with the data studied here.

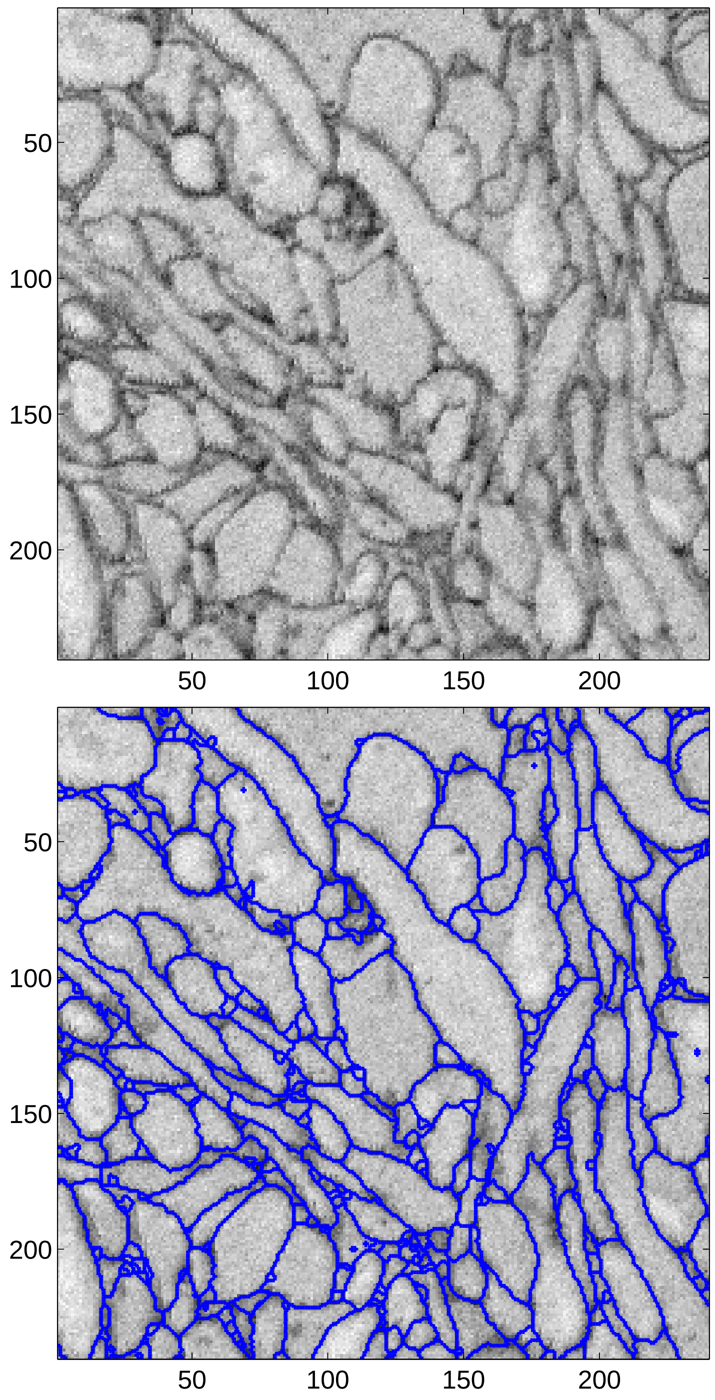


Figure 3.12: In a 2-dimensional slice of the SBFSEM volume image (top), faces between adjacent supervoxels appear as inter-pixel curves (bottom).

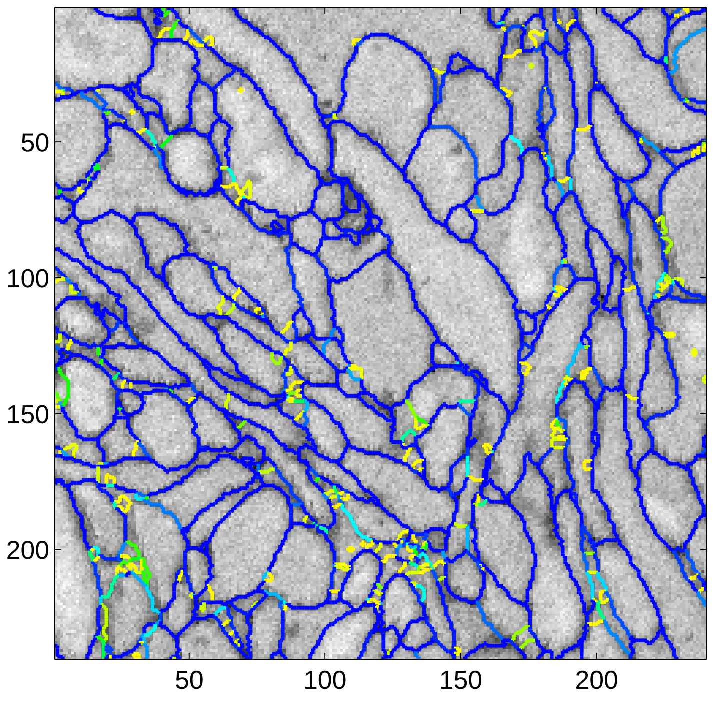


Figure 3.13: Faces between adjacent supervoxels (cf. Fig. 3.12) colored according to the probability to preserve the respective face predicted by C_{face}.

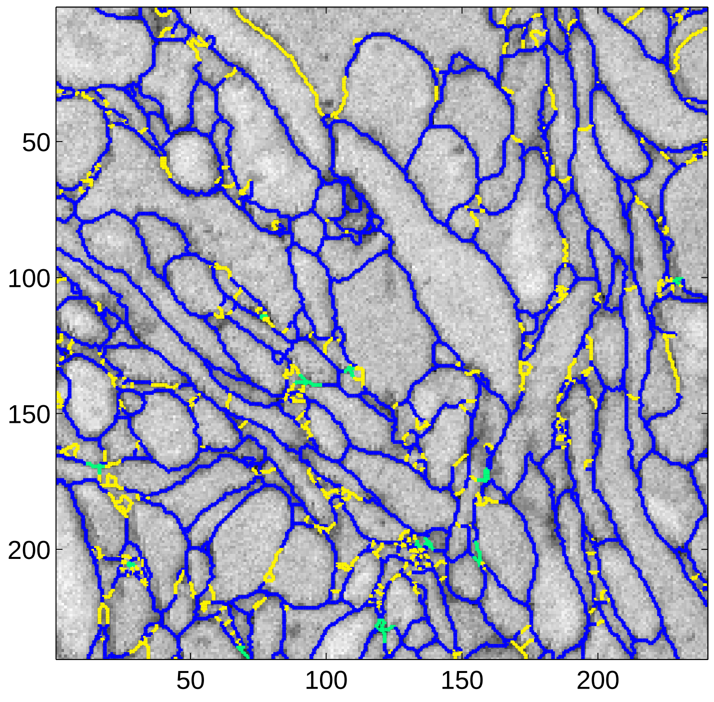


Figure 3.14: For each curve in 3D space in which three supervoxel faces meet, a probability is predicted by C_{curve} for all 8 possibilities (5 consistent possibilities) to preserve or remove these faces. In the figure above, configurations with maximum probability are color coded. Blue and yellow indicate preservation and removal, green indicates equal probability of all configurations. Different predictions are usually obtained from several curves that bound the same face. Thus, each point on each faces is colored according to the prediction from the nearest curve. This figure can provide only a limited insight to the geometry in 3D space and the reader is referred to Fig. 3.11 to see the quantitative effect of incorporating this evidence into the graphical model.

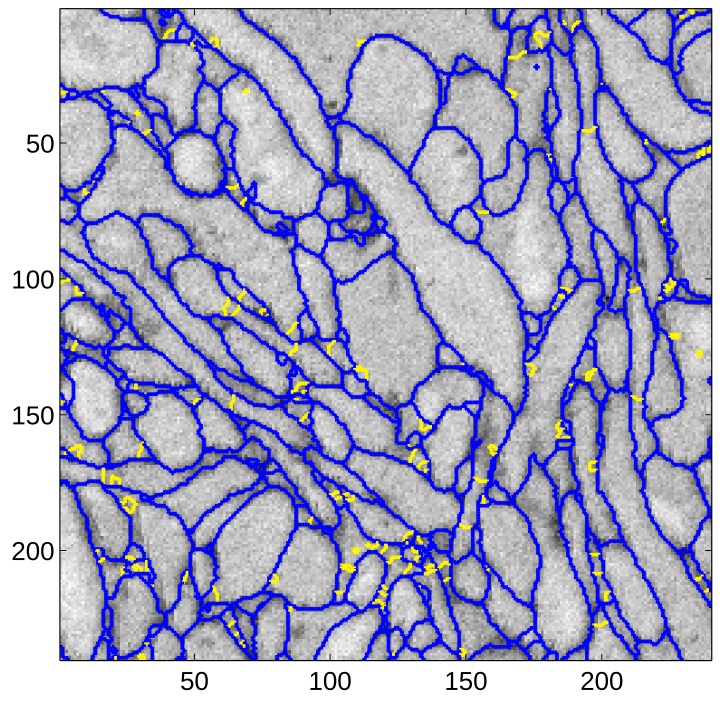


Figure 3.15: Approximate optimal decision to remove or preserve supervoxel faces obtained by means of loopy belief propagation and lazy flipping from the full graphical model (Approach C).

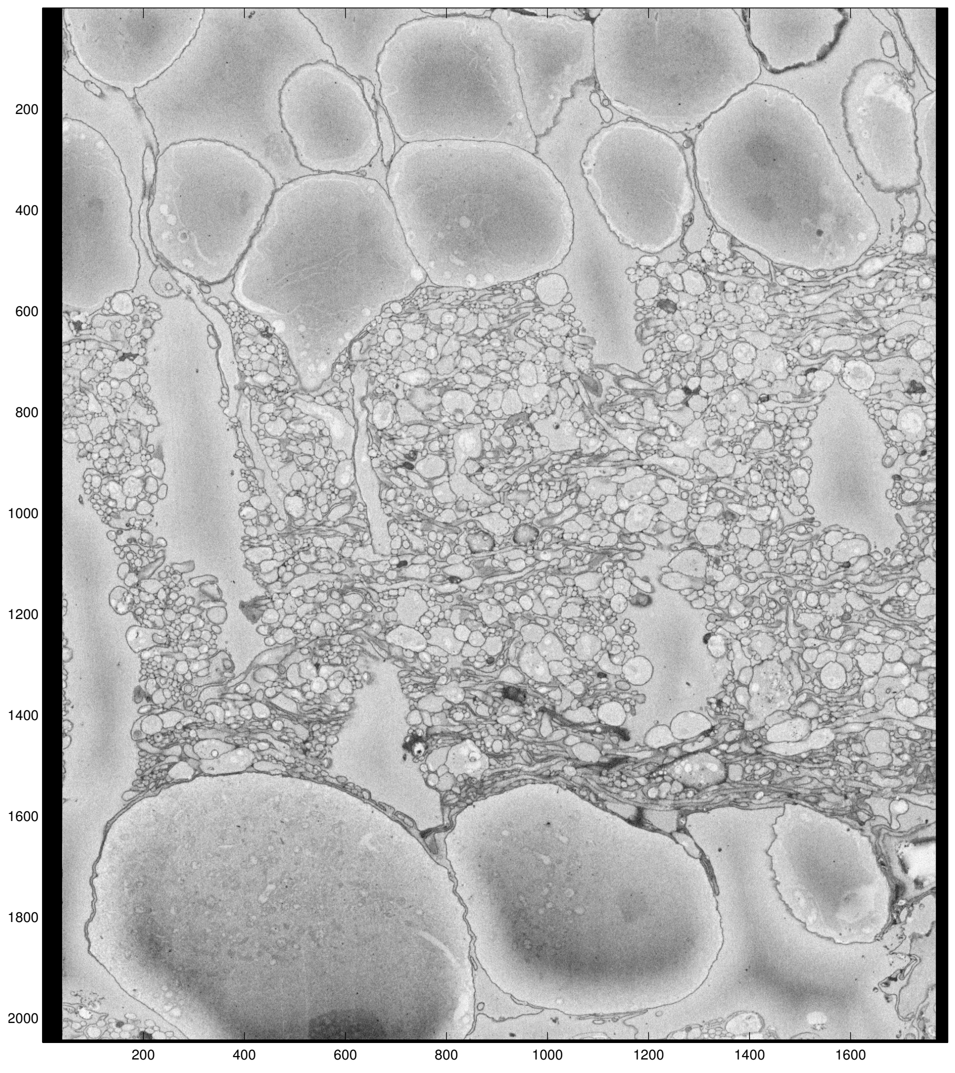


Figure 3.16: A slice (at $z = 120$) of the HRP-stained SBFSEM benchmark dataset (Helmstaedter et al., 2011), a volume image of $2048 \times 1792 \times 2048 \approx 7.5 \cdot 10^9$ voxels that shows part of the inner plexiform layer (IPL) of rabbit retina at a resolution of $22 \times 22 \times 30$ nm^3.

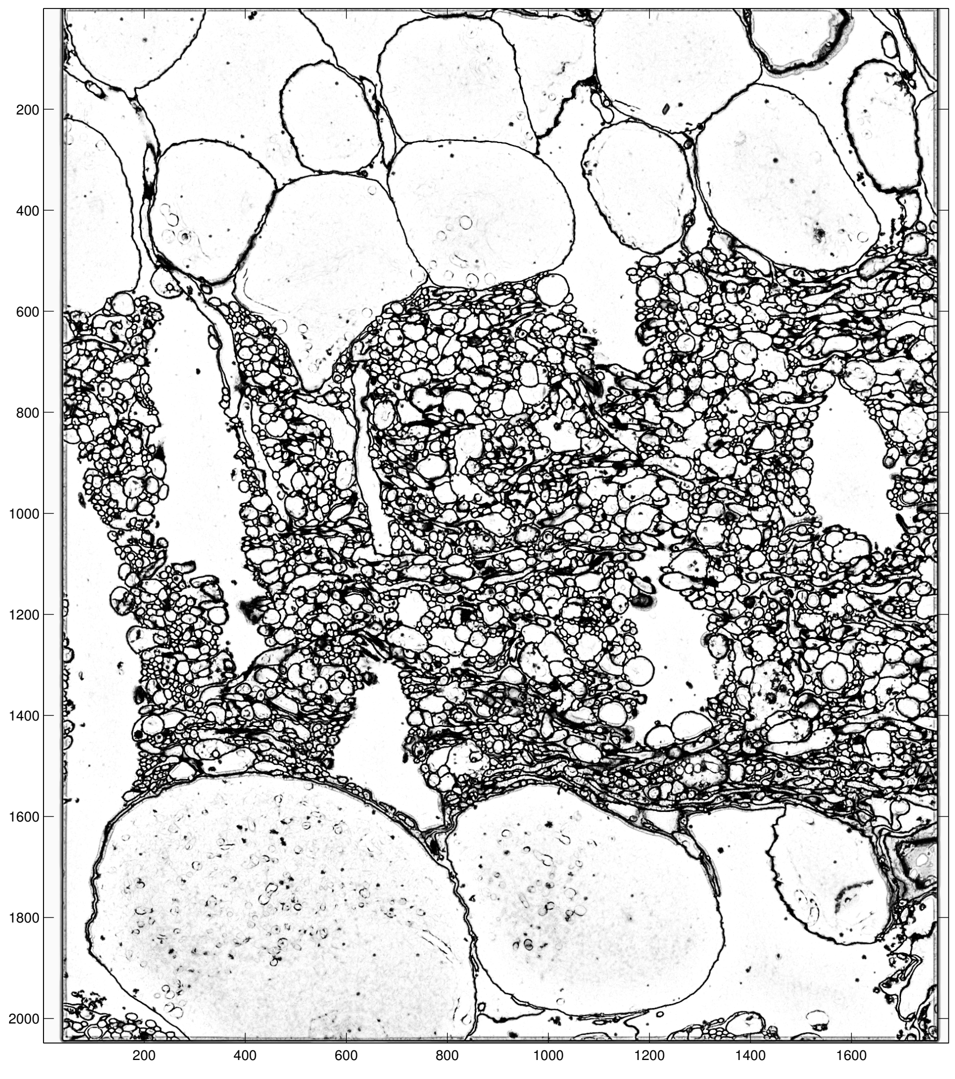


Figure 3.17: A slice (at $z = 120$) of the 3-dimensional probability map that corresponds to the slice of the SBFSEM benchmark dataset depicted in Fig. 3.16.

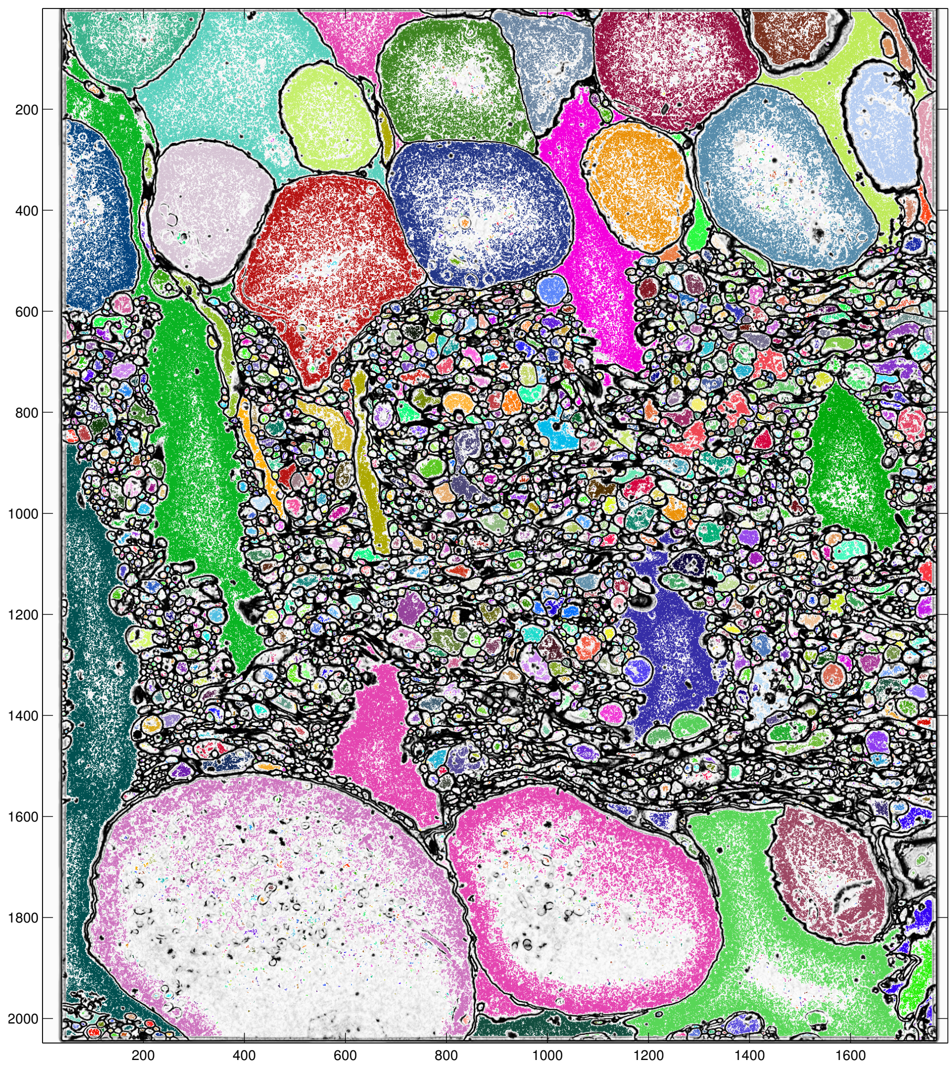


Figure 3.18: A slice (at $z = 120$) of the 3-dimensional supervoxel seed map. Seeds are connected components (in 3D space) for which the estimated probability (Fig. 3.17) to represent a membrane is zero.

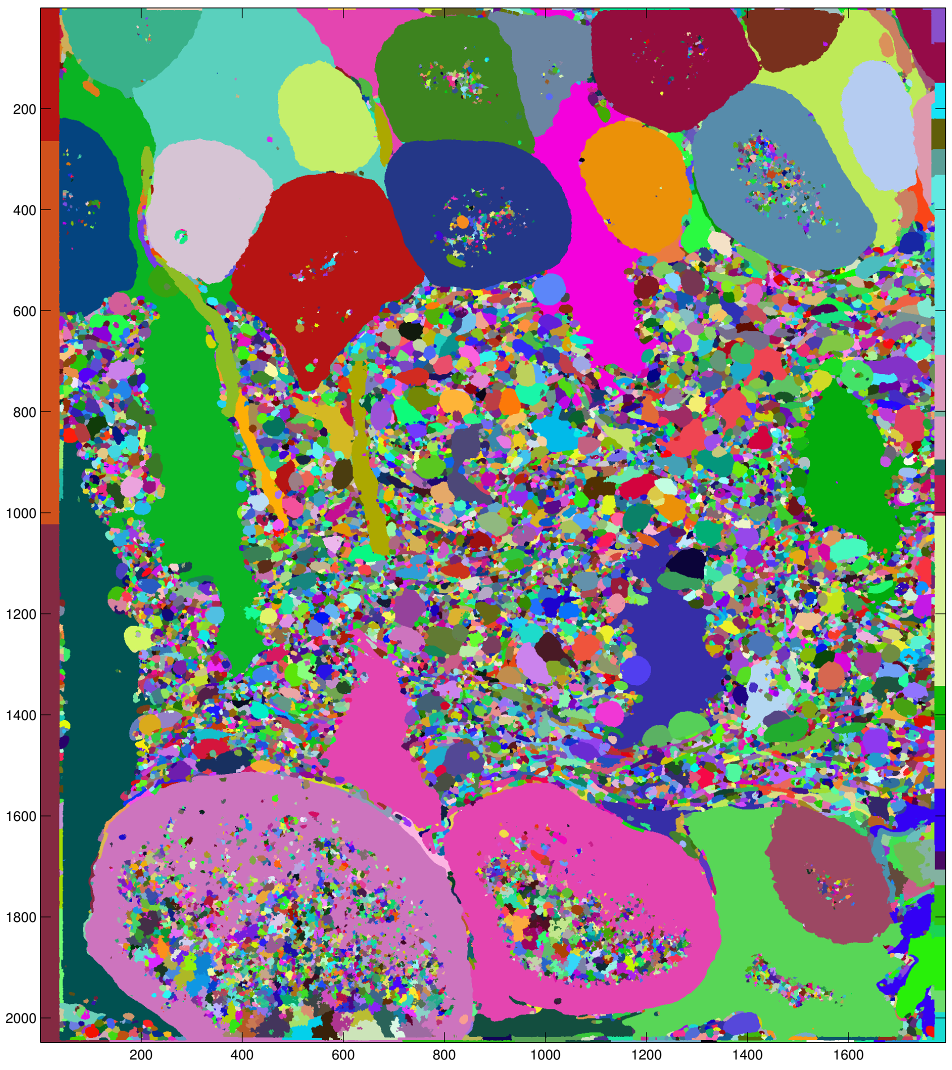


Figure 3.19: A slice (at $z = 120$) of the 3-dimensional supervoxel segmentation of the SBFSEM benchmark dataset (Fig. 3.17) computed by means of marker-based watershed segmentation using the probabilities in Fig. 3.17 as an elevation map and the seeds in Fig. 3.18 as markers.

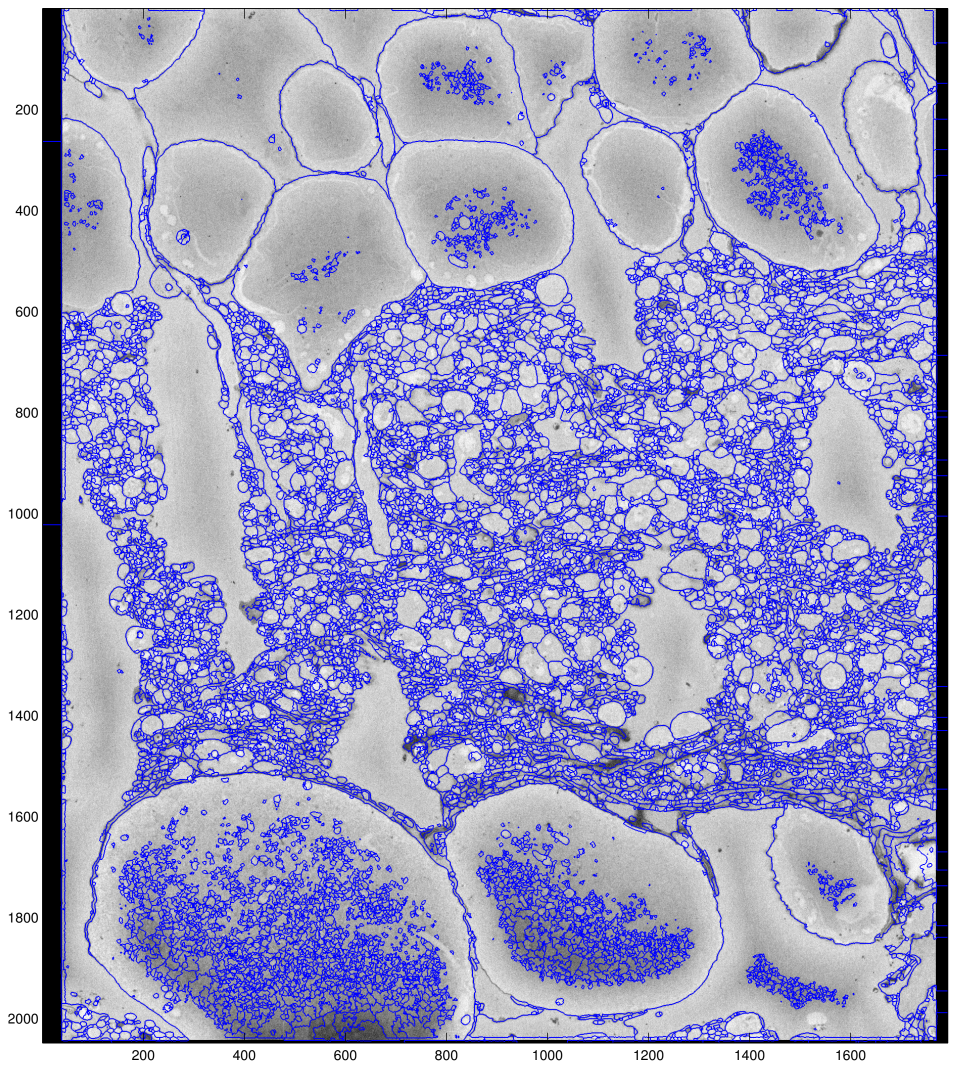


Figure 3.20: Faces between supervoxels in a slice (at $z = 120$) of the 3-dimensional supervoxel segmentation (Fig. 3.19) of the SBFSEM benchmark dataset (Fig. 3.17).

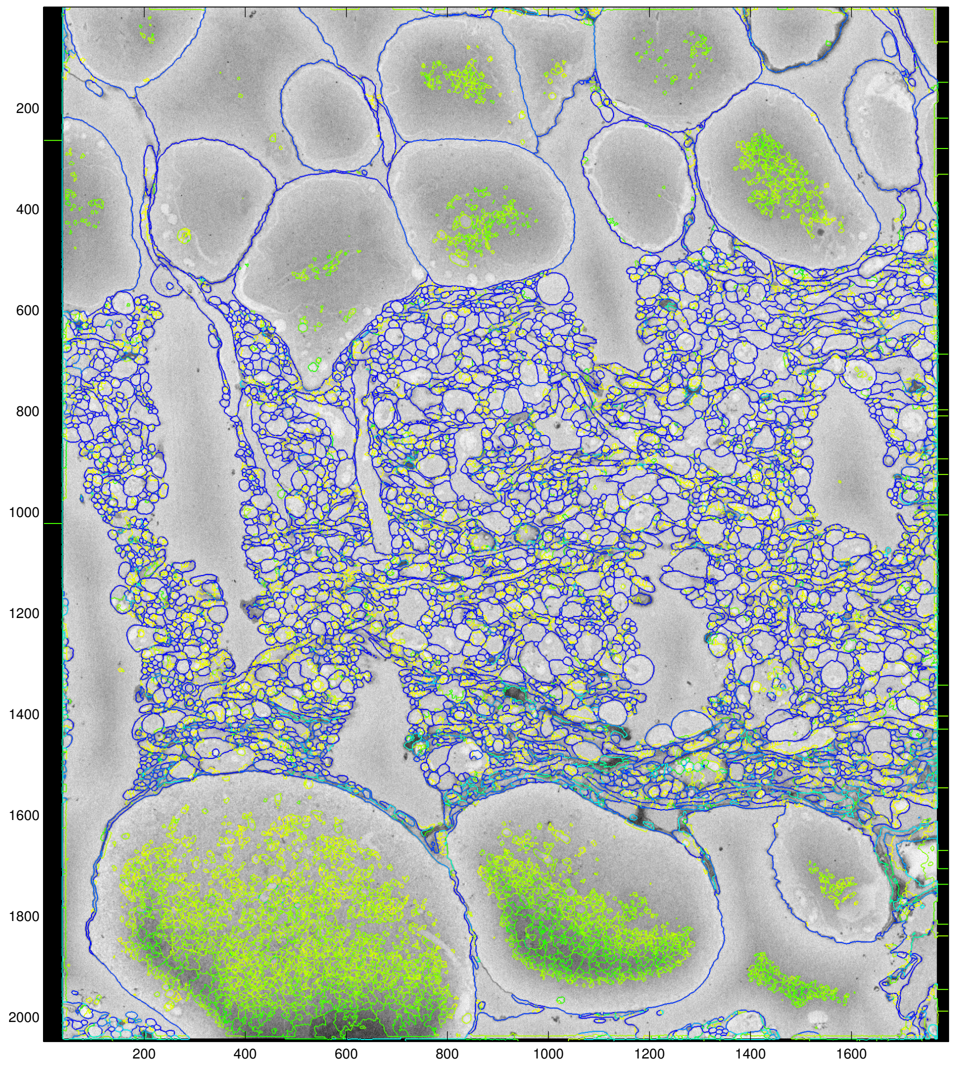


Figure 3.21: Supervoxel faces colored according to the probability to preserve the respective face predicted by C_{face}.

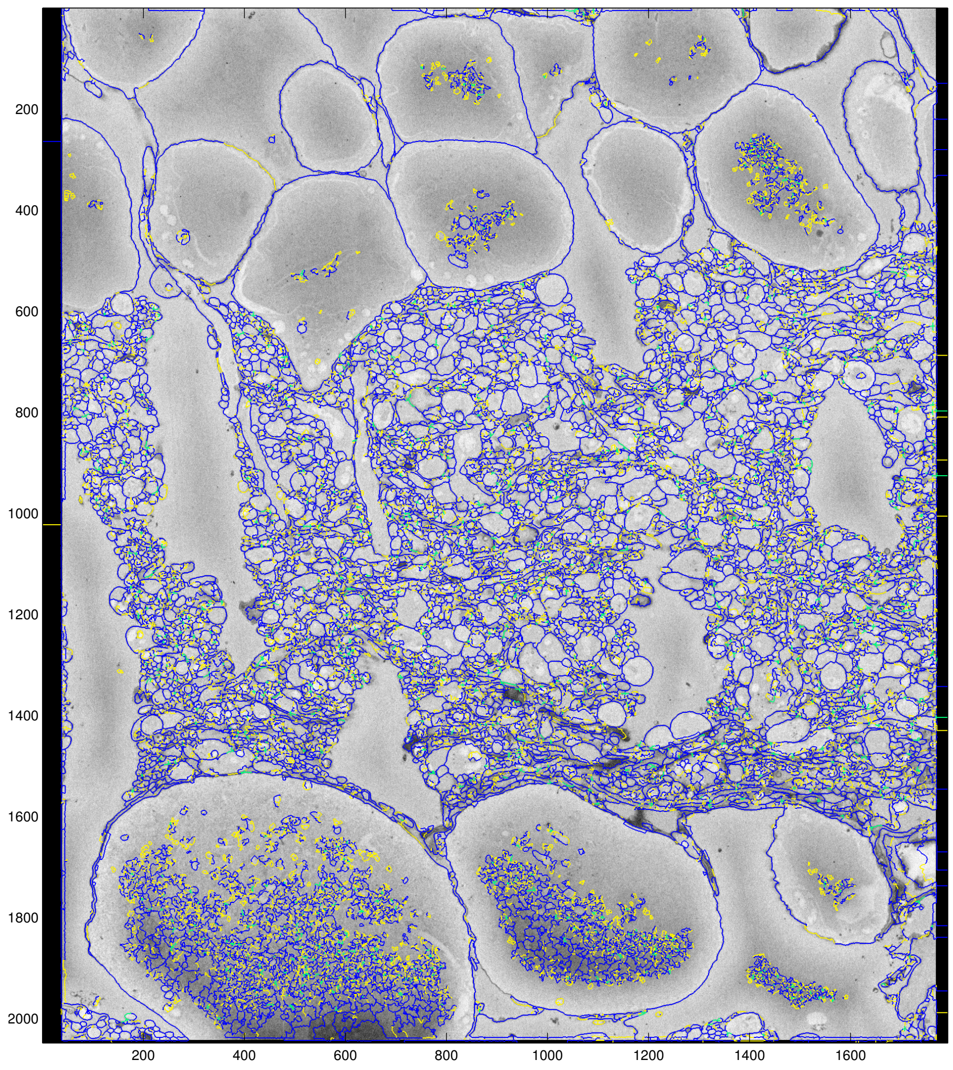


Figure 3.22: For each curve in 3D space in which three supervoxel faces meet, a probability is predicted by C_{curve} for all 8 possibilities (5 consistent possibilities) to preserve or remove these faces. In the figure above, configurations with maximum probability are color coded. Blue and yellow indicate preservation and removal, green indicates equal probability of all configurations. Different predictions are usually obtained from several curves that bound the same face. Thus, each point on each faces is colored according to the prediction from the nearest curve. This figure can provide only a limited insight to the geometry in 3D space and the reader is referred to Fig. 3.11 to see the quantitative effect of incorporating this evidence into the graphical model.

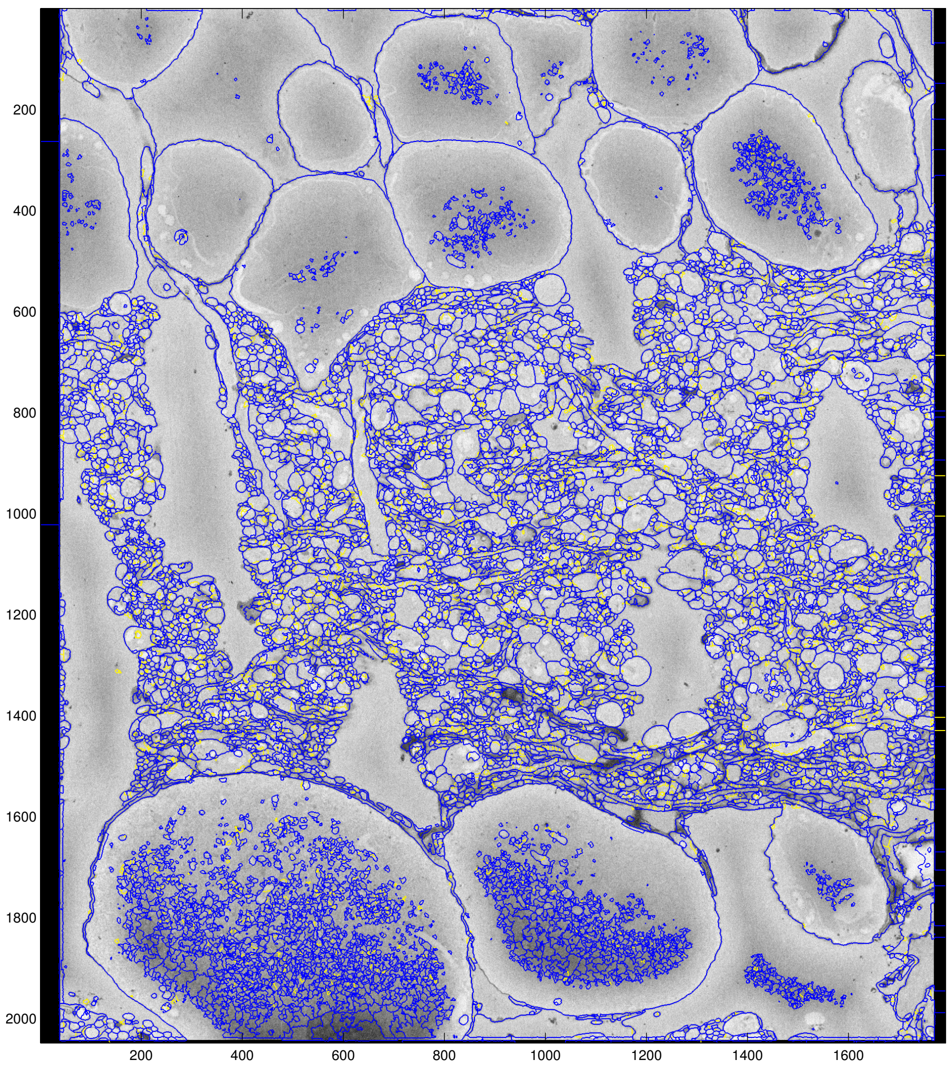


Figure 3.23: Approximate optimal decision to remove or preserve supervoxel faces obtained by means of loopy belief propagation and lazy flipping from the full graphical model (Approach C).

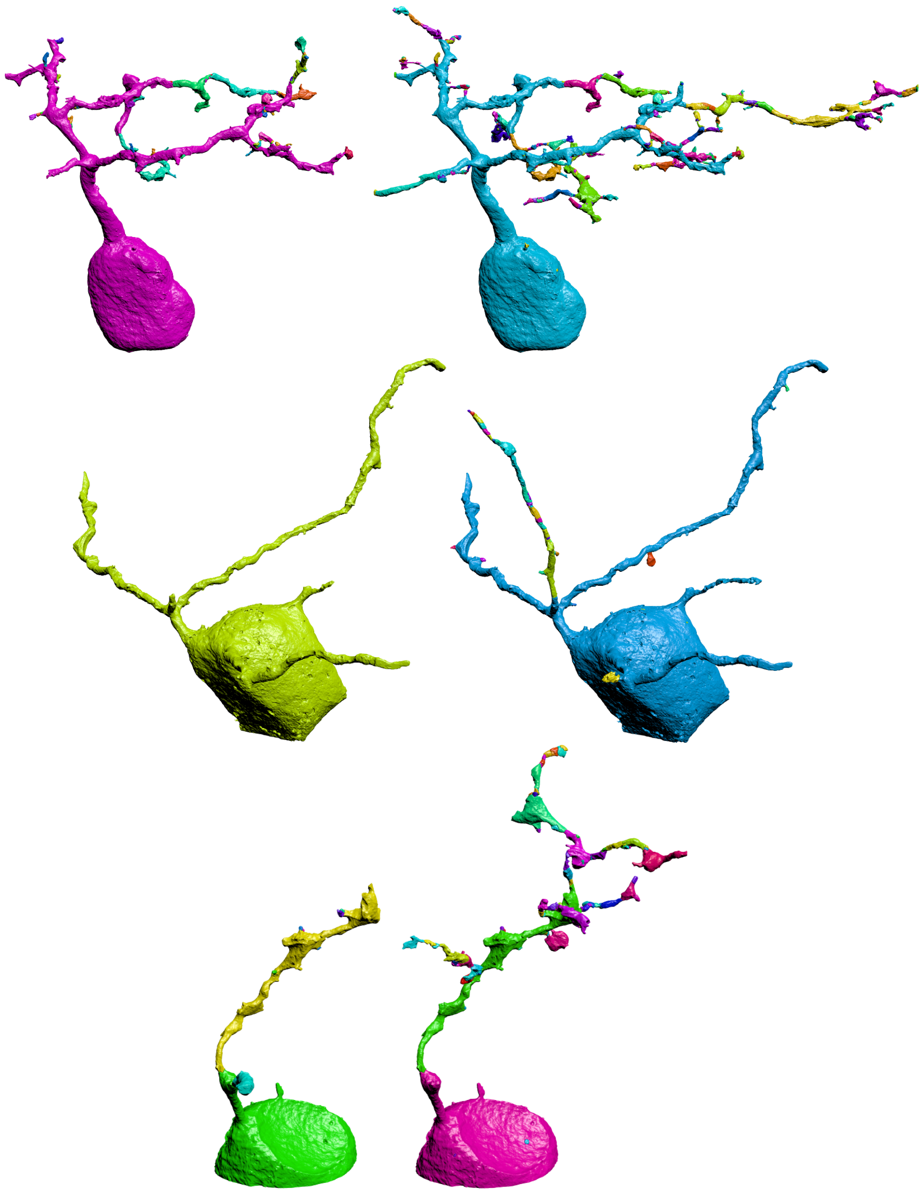


Figure 3.24: Comparison of automatic segmentations by Approach A with $\beta = 0.2$ (left) and the full graphical model (Approach C) with $\alpha = 0.17$ (right). In this example, Approach C reduces over-segmentation without introducing under-segmentation which is in accordance with the quantitative results in Tab. 3.4.

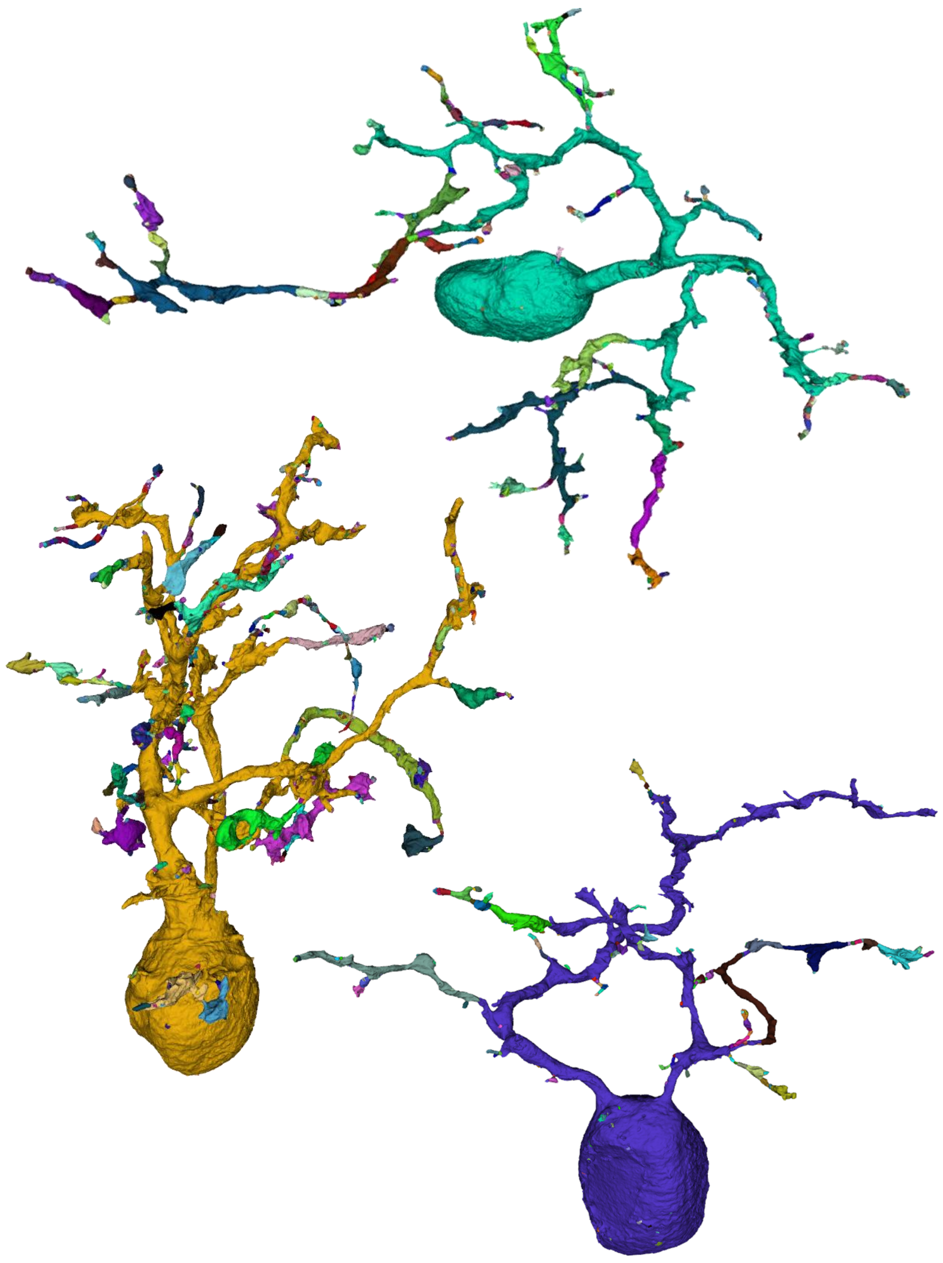


Figure 3.25: Neurons reconstructed by merging supervoxels according to the approximate optimizer of the full graphical model (Approach C).

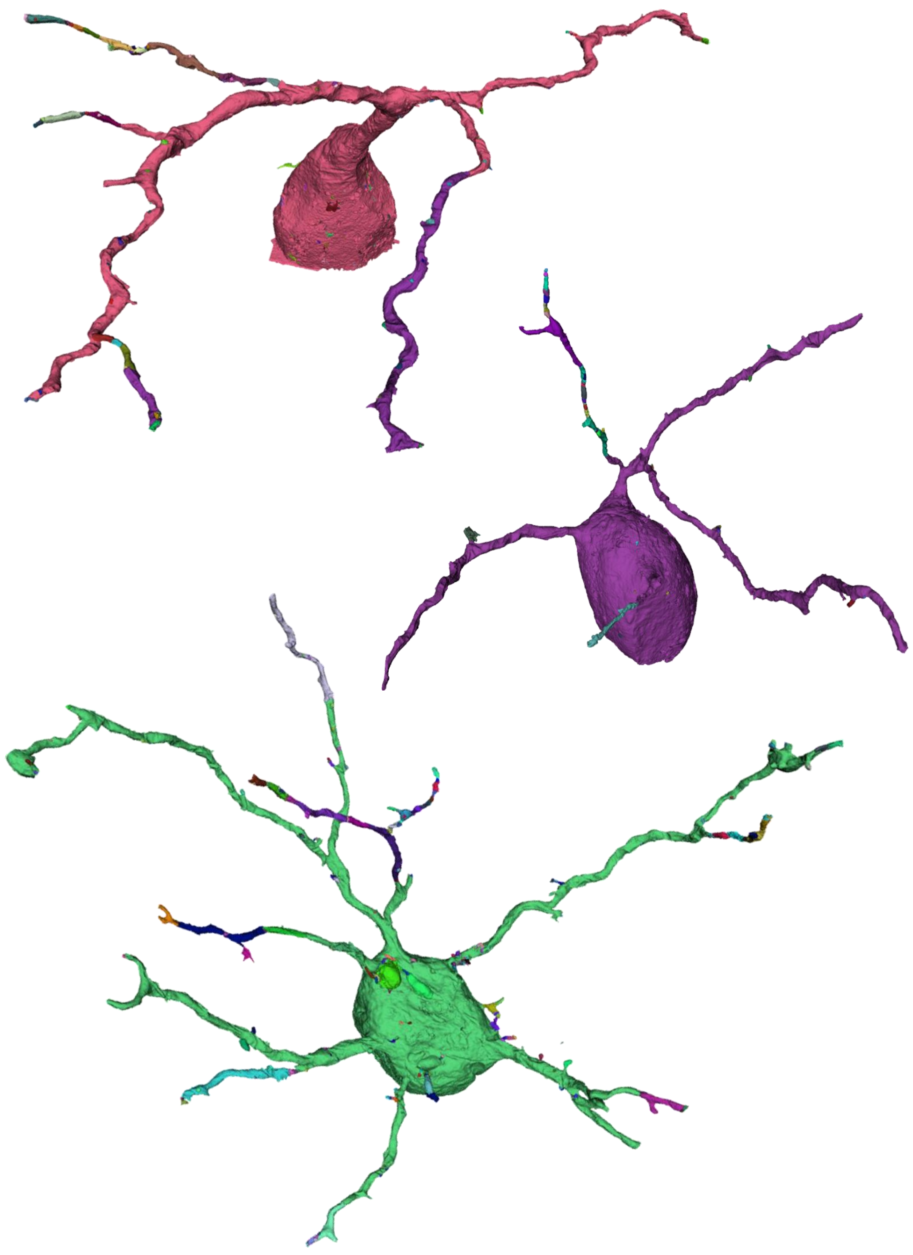


Figure 3.26: (Contd.) Neurons reconstructed by merging supervoxels according to the approximate optimizer of the full graphical model (Approach C).

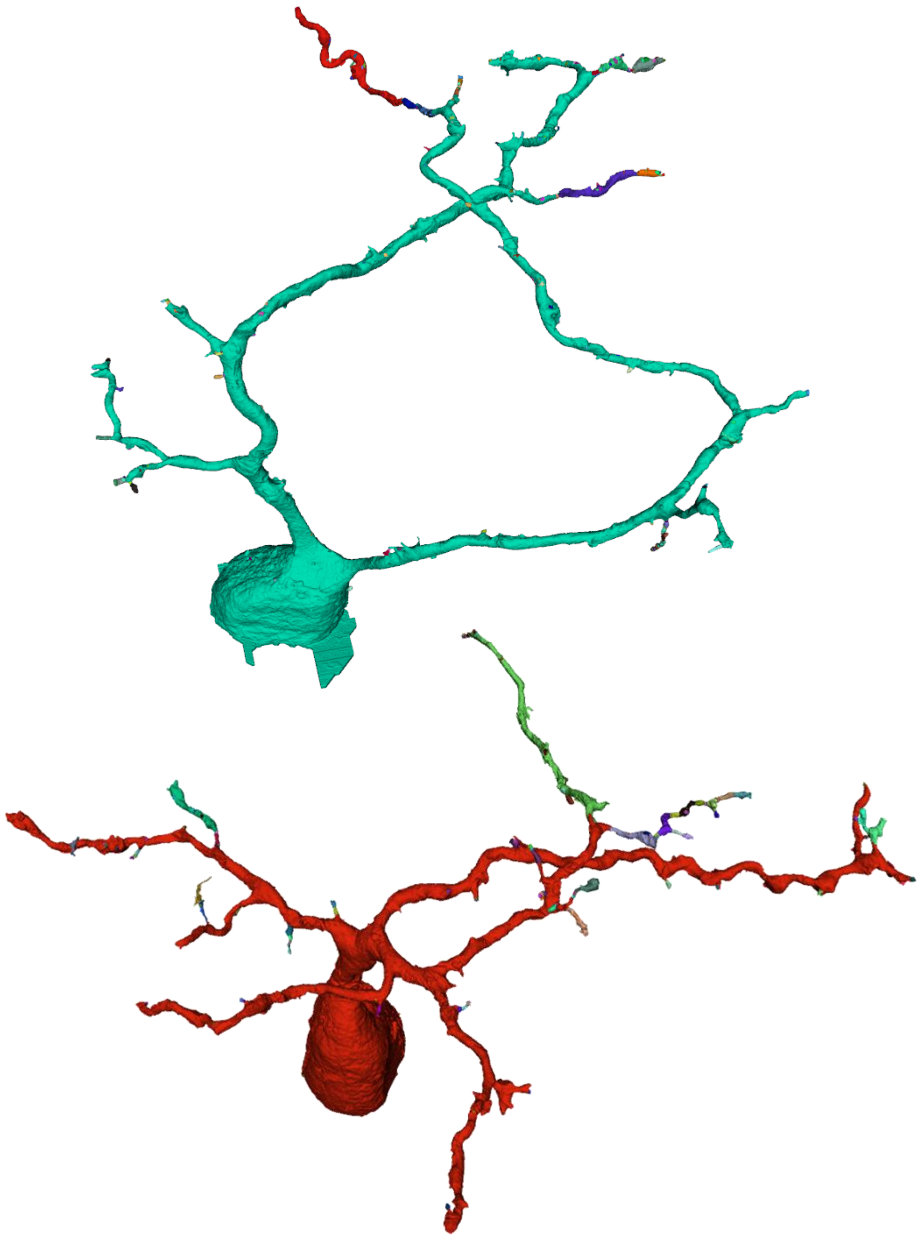


Figure 3.27: (Contd.) Neurons reconstructed by merging supervoxels according to the approximate optimizer of the full graphical model (Approach C).

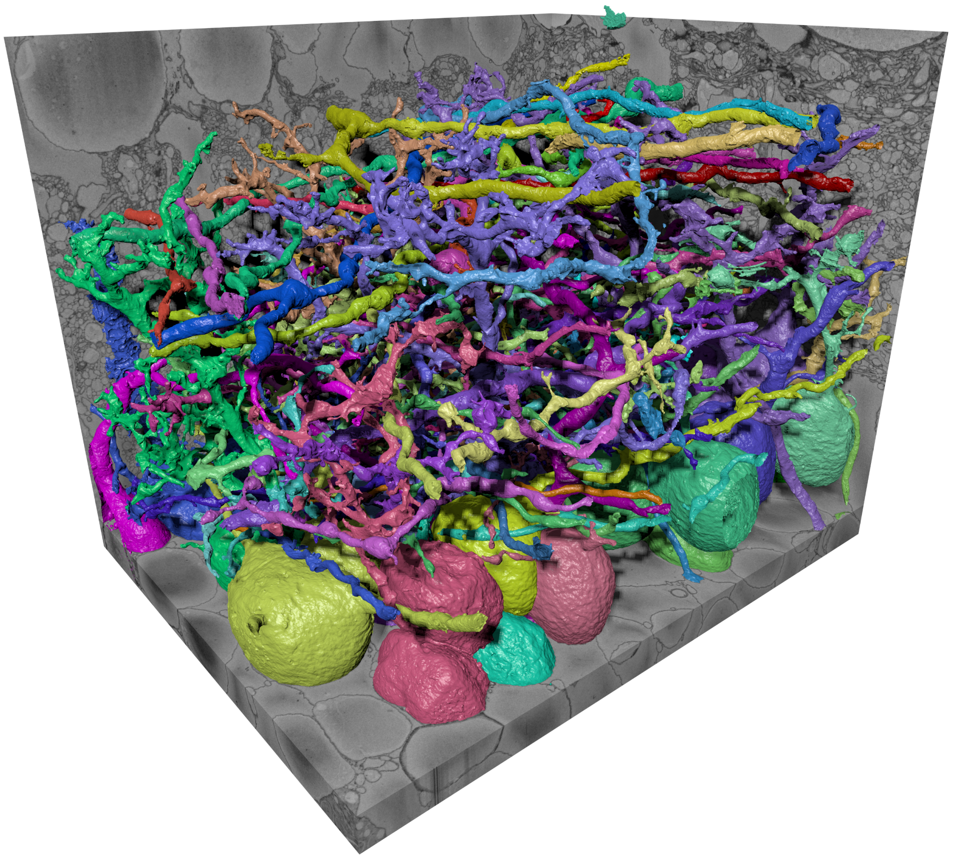


Figure 3.28: Automated reconstruction of 100 neurons in an SBFSEM volume image of the inner plexiform layer of rabbit retina using the full graphical model (Approach C).

4 Geometry and Topology Extraction from Large Volume Segmentations

4.1 Synopsis

The reconstruction of neurons from 3-dimensional images as introduced in Chapter 3 relies on features of the geometry and topology of an initial over-segmentation into supervoxels. In order to extract such features, an efficient algorithm and data structure are needed to encode segments, faces between segments, curves in which several faces meet–as well as the topology of these objects. Existing algorithms encode this information in designated data structures, but require that these data structures fit entirely in Random Access Memory (RAM). Today, 3D images with several billion voxels are acquired (cf. Chapter 3). Since these large volumes can no longer be processed with existing methods, a new algorithm is presented in this chapter that performs geometry and topology extraction with a runtime linear in the number of voxels and log-linear in the number of faces and curves. The parallelizable algorithm proceeds in a block-wise fashion and constructs a consistent representation of the entire volume image on the hard drive, making the structure of very large volume segmentations accessible to image analysis.

4.2 Introduction

Segmentations of volume images partition the volume into different subsets: **Segments**, **faces** between segments, **curves** in which several faces meet, as well as the **points** between these curves, (Fig. 4.1). Features that describe these subsets are essential in many analyses. To be able to extract such features, a data structure is needed that provides fast access to

- the **geometry** of these subsets, i.e. for every segment, face, curve and point, a list of coordinates that constitutes the respective set.
- the **topology** of the segmentation, i.e. the neighborhood system of its subsets.

However, volume segmentations are usually stored simply as **volume labelings**, i.e. as 3-dimensional arrays in which each entry is a label that uniquely identifies the segment to which the voxel belongs. This form of storage does not represent geometry and topology explicitly. Instead, faces between segments and the curves between these faces are encoded only implicitly, by adjacent voxels whose segment labels differ. All that can be obtained from an array in constant computation time is the segment label at a given voxel. Neither is the set of voxels that belong to the same segment readily available, nor are the faces between adjacent segments or the curves in which several of these faces meet. It is not stored explicitly which segments are adjacent, separated by which faces, and in which curves adjacent faces meet.

The new algorithm presented in this chapter takes a volume labeling as input and extracts the geometry and topology of all subsets in a block-wise fashion, in a runtime that is linear in the number of voxels and log-linear in the number of faces and curves. Blocks of the volume labeling can be processed either sequentially, on a single computer that might have only a few hundred megabytes of RAM, or in parallel, on several computers, which facilitates geometry and topology extraction from datasets that consist of more than 10^9 voxels. In both cases, a consistent representation of the entire volume segmentation is constructed on the hard drive, in a data structure from which all subsets and their adjacency can be obtained in constant computation time. The new algorithm makes the geometry and topology of large volume segmentations accessible to image analysis.

This chapter is organized as follows: Related work is discussed in Section 4.3. In Section 4.4, the data structure that captures the geometry and topology of a volume segmentation is introduced. The algorithm for its construction is defined in Section 4.5 and extended in Section 4.6 to work with limited RAM and in parallel. The correctness of the algorithm is proved and its complexity analyzed. Section 4.7 describes the efficient storage of the data structure on the hard drive, and Section 4.8 concludes the chapter.

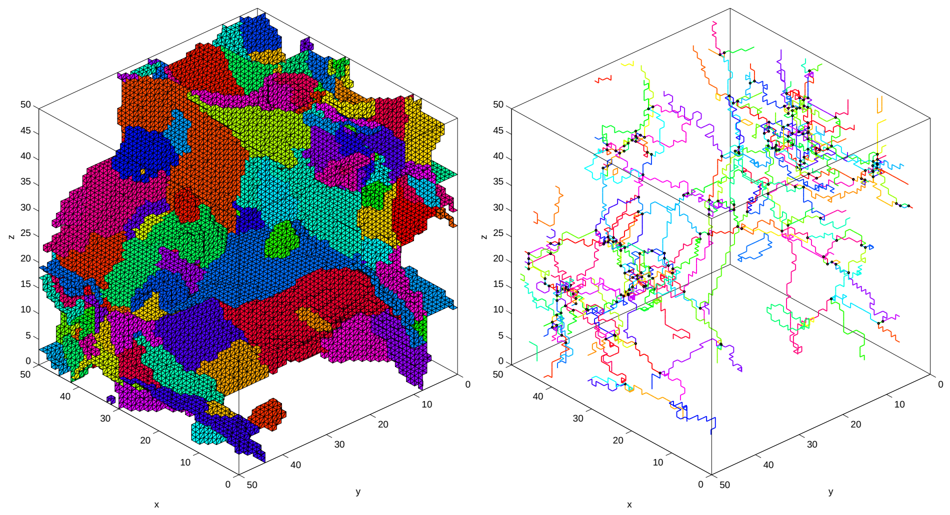


Figure 4.1: A volume segmentation consists of segments, faces between adjacent segments (left), the curves in which several of these faces meet, as well as the points between these curves (right).

4.3 Related Work

An explicit representation of all subsets of an image segmentation was proposed already by Brice and Fennema (1970); it encodes segments as sets of pixels, curves between segments as sets of inter-pixel edges, and the end points of these curves as pixel corners (cf. Fig. 4.2). Naive attempts to represent the different subsets all as sets of pixels on the pixel grid of the underlying image are topologically inconsistent, as shown by Pavlidis (1977) and proven generally and rigorously by Kovalevsky (1989, 1993). To overcome this inconsistency, Khalimsky et al. (1990) introduced the **topological grid** whose points correspond to pixels, inter-pixel edges, and pixel corners. The concept of a 3-dimensional topological grid is depicted in Fig. 4.2.

Data structures that store, for every subset of a segmentation, all points of the topological grid that constitute this subset were proposed and implemented by Meine and Köthe (2005) for segmentations of images and envisioned by Damiand (2008) for segmentations of 3-dimensional volume images. However, a storage concept that is suitable for large volume segmentations has so far been missing.

Along with representations that capture the subsets of a segmentation, at least three different structures have been used to encode the neighborhood system: **Region Adjacency Graphs** (RAGs) (Pavlidis, 1977) encode the adjacency of segments. RAGs do not capture the topology of a segmentation completely because several disconnected faces that separate the same two segments correspond to the same edge in the RAG. Kropatsch (1995) introduces multiple edges and self-loops in the RAG which results in a multi-graph whose dual graph represents faces as vertices and the adjacency of faces as edges. This concept can be implemented as a data structure. However, both the graph and its dual have to be stored and maintained which is algorithmically challenging.

Combinatorial maps were introduced in image analysis by Braquelaire and Guitton (1991) and are used as data structures, e.g. by Meine and Köthe (2005) and in some algorithms of the Computational Geometry Algorithms Library (CGAL)[1]. The extension of combinatorial maps to higher dimensions is involved but possible (Lienhardt, 1989, 1991) and has facilitated the development of the 3-dimensional topological map (Bertrand et al., 2000; Damiand and Resch, 2003). This map captures not only the topology of a segmentation but also its embedding into the segmented space, i.e. containment relations and orders of objects (Damiand, 2008; Lienhardt, 1989). It is therefore more expensive to construct and manipulate than a data structure that encodes only the topology.

A simple structure that encodes only the topology is a finite **cellular complex**, cf. Hatcher (2002); Munkres (1995) also known as a cell complex or CW-complex (Klette, 2000) where CW stands for the two axioms closure-finiteness and weak topology, cf. Hatcher (2002). Cellular complexes were first used in image processing by Kovalevsky (1989, 1993). Their application in 3D is simple and intuitive.

The main focus of previous efforts to extract and encode the geometry and topology of segmentations has not been on large volume segmentations but on the efficient processing of the merging and splitting of segments. These operations are required within the context of inter-active segmentation. Damiand (2008); Meine et al. (2004) construct representations of the geometry and topology incrementally, using random access to already constructed parts of the data structure. In order for these algorithms to work efficiently, the underlying data structures need to be kept entirely in RAM. To extract the geometry and topology of a volume segmentation of $2,000^3$ voxels, $2,000^3 \cdot 2^3 \cdot 4$ bytes ≈ 238 GB of RAM are required for the labeling of the topological grid, an amount that is not available on present

[1]http://www.cgal.org

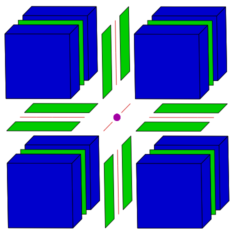


Figure 4.2: A topological grid T is used to represent the geometry of a volume segmentation explicitly. Its elements are called cells. A cell $(t_1, t_2, t_3) \in T$ with j odd entries is called a j-cell. 3-cells, 2-cells, 1-cells, and 0-cells respectively represent voxels (blue), faces between voxels (green), lines between faces (red), and points between lines (purple).

day desktop computers. Beyond $3,500^3$ voxels, even the 1 TB of RAM of a large server are insufficient. The method presented in this chapter overcomes this limitation by means of block-wise processing. It makes geometry and topology extraction from large volume segmentations possible.

4.4 From Voxels to Geometry and Topology

The starting point for geometry and topology extraction is a volume image on a **voxel grid** $G = \{1, \ldots, n_1\} \times \{1, \ldots, n_2\} \times \{1, \ldots, n_3\}$ whose extent in each of the three dimensions is given by $n_1, n_2, n_3 \in \mathbb{N}$. Two voxels $v, w \in G$ are said to be **connected** if $\sum_{j=1}^{3} |v_j - w_j| = 1$. Each voxel is thus connected to 6 other voxels unless it is at the boundary of the grid. For every voxel, the connected voxels are called its **6-neighbors**. A set of voxels $U \subseteq G$ is called connected if and only if any two distinct voxels $v, w \in U$ are linked by a path in U, i.e. by a sequence of voxels in U that starts with v and ends with w, in which each voxel is connected to its predecessor.

A volume segmentation partitions the voxel grid G into connected components called segments. A **volume labeling**

$$\sigma : G \to \mathbb{N} \tag{4.1}$$

assigns to each voxel a label that identifies the segment to which the voxel belongs. Since each voxel belongs to a segment, the faces between segments, the curves between these faces and the points between these curves (Fig. 4.1) cannot be represented on the voxel grid. The structure that is used for this purpose is a **topological grid**,

$$T = \{1, \ldots, 2n_1 - 1\} \times \{1, \ldots, 2n_2 - 1\} \times \{1, \ldots, 2n_3 - 1\} \ . \tag{4.2}$$

This grid has about eight times the size of the voxel grid. Its elements are called **cells**. Cells with j odd entries are called **j-cells**, cf. Fig. 4.2.

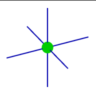
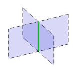
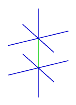
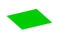
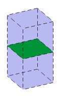
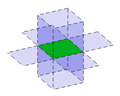
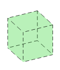
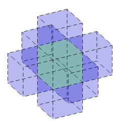
n	n-cell	Γ	$\leftrightarrow$
0			$\emptyset$
1			
2			
3		$\emptyset$	

Figure 4.3: Two relations are crucial to the definition of connected components of cells. (i) The Γ-neighborhood of a j-cell consists of all its 6-neighbors on the topological grid T that are $(j+1)$-cells. (ii) The connectivity relation "$\leftrightarrow$" connects two cells $t_1, t_2 \in T$ if and only if there exists a third cell $t \in T$ such that both t_1 and t_2 are Γ-neighbors of t.

Segments, faces between segments, curves between faces, and points between curves correspond to connected components of cells. Two relations are crucial to the definition of these connected components. The first relation the Γ-neighborhood of cells. It is depicted in the third column of Fig. 4.3.

Definition 7. The **Γ-neighborhood** is the mapping $\Gamma : T \to \mathcal{P}(T)$ such that, for each $j \in \{0, \ldots, 3\}$ and any j-cell $t \in T$, $\Gamma(t)$ consists of all 6-neighbors of t on the topological grid T that are $(j+1)$-cells.

Any 2-cell, for instance, has two Γ-neighbors that correspond to two voxels.

The second important relation is the **connectivity** of cells; it is depicted in the last column of Fig. 4.3.

Definition 8. The connectivity relation "$\leftrightarrow$" $\subseteq T \times T$ connects any two cells $t_1, t_2 \in T$ (denoted $t_1 \leftrightarrow t_2$) if and only if there exists a $t \in T$ such that both t_1 and t_2 are Γ-neighbors of t.

Segments, faces, curves and points can now be defined recursively as connected components of 3-cells, 2-cells, 1-cells, and 0-cells which are called j-components. In the following definition, a distinction is made between active and inactive cells.

Definition 9. Any 3-cell is said to be **active**. A set of all (active) 3-cells that belong to the same segment is called a **3-component**.

For $j \in \{2, 1, 0\}$ and any j-cell $t \in T$, let $\{t_1, \ldots, t_{6-2j}\} = \Gamma(t)$ be its Γ-neighbors. For each $k \in \{1, \ldots, 6-2j\}$, let τ_k be the connected component of t_k if t_k is active, and let $\tau_k = \emptyset$ otherwise. Define $\theta(t)$ to be the set of connected components that occur precisely once in $(\tau_1, \ldots, \tau_{6-2j})$. These connected components are said to be **bounded** by t. Moreover, t is called **active** if $\theta(t) \neq \emptyset$.

For each $j \in \{0, 1, 2\}$, a j**-component** is a maximal set $U \subseteq T$ with the following properties:

(i) Any $t \in U$ is an active j-cell.

(ii) All $t \in U$ bound the same connected components of $(j+1)$-cells, i.e. there exists a set Θ such that $\theta(t) = \Theta$, for all $t \in U$.

(iii) For any $t_1, t_2 \in U$, there exists a path in U from t_1 to t_2 in which each cell is connected via $\leftrightarrow$ to its predecessor.

This definition captures not only the geometry but also the topology a a volume segmentation. Given, for instance, a face between two segments (i.e. a 2-component) and any of its 2-cells, t, $\theta(t)$ identifies the two segments (3-components) that are bounded by the face. In practice, $\theta(t)$ can be stored for each j-component. In theory, this corresponds to a cellular complex representation that is isomorphic to the topology of the volume segmentation Kovalevsky (1989). Cellular complexes are defined as follows.

Definition 10. A **cellular complex** is a triple $(C, <, \dim)$ in which "$<$" is a strict partial order in C and $\dim : C \to \mathbb{N}_0$ maps elements of C to non-negative integers such that $\forall c, c' \in C : c < c' \Rightarrow \dim(c) < \dim(c')$. The elements of C are called **cells**, "$<$" the **bounding relation** and dim the **dimension function** of the cellular complex.

As an example, consider the cellular complex that contains as cells all points of the topological grid T (these points have already been referred to as cells), and as a bounding relation the transitive closure of the Γ-neighborhood, i.e. the strict partial order that relates any $t_1, t_2 \in T$ precisely if there exist an $n \in \mathbb{N}$ and a sequence of n cells $p : \{1, \ldots, n\} \to T$ such that $p(1) = t_1$, $p(n) = t_2$, and $\forall j \in \{2, \ldots, n\} : p(j) \in \Gamma(p(j-1))$. The dimension function simply maps each cell to its order, either 0, 1, 2 or 3. This cellular complex corresponds to the topology of the finest possible segmentation in which each voxel is a separate segment.

A coarser cellular complex contains as cells the j-components (Def. 9). Its bounding relation is the transitive closure of the bounding relation θ of Def. 9. Its dimension function maps each connected component to the order of its cells. This cellular complex captures the topology of the volume segmentation. It would make sense to refer to its elements again as cells. However, to avoid confusion, the term j-components is used throughout this chapter.

An important property of Def. 9 is that it is constructive and thus motivates an algorithm for the labeling of j-components.

4.5 Extraction of Segmentation Geometry and Topology

The geometry of a volume segmentation is made explicit by labeling not only the segments but also the faces between segments, the curves between faces and the points between curves, i.e. the j-components of the segmentation, on the topological grid T. The resulting **topological label map**

$$\tau : T \to \mathbb{N}_0 \tag{4.3}$$

assigns a positive integer, representative of a j-component, to each active cell, and zero to all inactive cells. On the computer, τ is stored as a 3-dimensional array.

The first step towards this labeling is to copy all segment labels from the volume labeling σ to the topological grid labeling τ by means of Algorithm 3. Subsequently, 2-components and 1-components are identified and labeled by means of Algorithm 4 that performs a depth-first-search[2]. Finally, active 0-cells are identified and labeled by means of Algorithm 5. Besides labeling the topological grid, these algorithms construct the bounding relation θ of j-components (Def. 9) and thus, a cell complex representation of the topology of the segmentation.

Overall, this connected component labeling is an exact implementation of Def. 9 and is thus known to be correct. Its runtime is linear in the number of voxels and so is its space complexity. The memory dynamically allocated for the stack is in addition bounded by the number of cells in the largest face.

The absolute memory requirement nevertheless renders the procedure impractical for large volume segmentation. As shown in the introduction, the topological label map τ of a volume segmentation that consists of $2{,}000^3$ voxels is too large to fit in the RAM of a desktop computer. Storing τ on the hard drive and loading blocks into RAM on demand as a sub-routine of Algorithm 4 does not solve the problem because any caching of blocks becomes inefficient if segments and faces extend unsystematically across large parts of the volume which is often the case, in particular in connectomics

[2]The auxiliary function **once** used in this algorithm takes an input sequence of integers and returns an ordered sequence of the same length that contains those positive integers that occur precisely once in the input sequence. Additional entries in the output sequence are filled with zeros.

datasets (Chapter 3). Fortunately, the labeling itself can be constrained to small blocks of the volume that can be chosen systematically and processed independently, with very limited RAM and in parallel.

Algorithm 2: Labeling of 3-cells

```
Input: $\sigma : G \to \mathbb{N}$ (segment label map)
Output: $\tau : T \to \mathbb{N}$ (topological label map, preliminary), $n \in \mathbb{N}$
        (maximum segment label)
1 $n \leftarrow 0$;
2 foreach $r \in G$ do
3     $\tau(2r - 1) \leftarrow \sigma(r)$;
4     if $\sigma(r) > n$ then
5         $n \leftarrow \sigma(r)$;
6     end
7 end
```

4.6 Block-wise Processing of Large Segmentations

In order to extract the geometry and topology from large volume segmentations efficiently with limited RAM, the labeling of components is constrained to sufficiently small blocks of the topological grid. Each block is processed independently using Algorithms 3, 4 and 5. The independent results are stored on the hard drive and subsequently combined into a consistent labeling of the entire topological grid. More precisely, the procedure works as follows.

Step 1 (connected component labeling). The topological grid is subdivided into blocks such that each block begins and ends in each direction with a layer that contains 3-cells. Adjacent blocks are chosen to overlap each other by one cell in each direction as is depicted in Fig. 4.4. In consequence, each 1-cell and each 2-cell within a region of overlap belongs to two different blocks (Fig. 4.4b). Each block is then labeled independently using Algorithms 3, 4, and 5, and the respective labelings are stored on the hard drive. In consequence, the labeling of connected components starts in each block with the label 1.

Step 2 (label disambiguation). The processed blocks are put in an arbitrary but fixed order. If the j-th block in this order contains m_2 2-components, the **offset** m_2 is stored along with block $j+1$ where it is added on demand to all non-zero 2-cell labels, similarly for 1-cells and 0-cells,

Algorithm 3: Labeling of 2-cells and 1-cells

```
Input: $\tau : T \to \mathbb{N}$ (topological label map, preliminary), $c \in \{1, 2\}$
       (cell order)
Output: $\tau$ (modified), $n \in \mathbb{N}$ (number of $c$-components),
        $\alpha \in \mathbb{N}^{n\times(6-2c)}$ (neighborhood relation of $c$-cells)
$n \leftarrow 0$;
Stack $s \leftarrow \emptyset$;
foreach c-cell $t \in T$ do
    if $\tau(t) = 0$ then
        $p \leftarrow (6 - 2c)$;
        $(t_1, \dots, t_p) \leftarrow \Gamma(t)$;
        $(x_1, \dots, x_p) \leftarrow (\tau(t_1), \dots, \tau(t_p))$;
        $(y_1, \dots, y_p) \leftarrow \text{once}(x_1, \dots, x_p)$;
        if $y_1 \neq 0$ then
            $n \leftarrow n + 1$;
            for $j = 1$ to $p$ do
                $\alpha(n, j) \leftarrow y_j$;
            end
            $s$.push($t$);
            while $s \neq \emptyset$ do
                $u \leftarrow s$.pop();
                $\tau(u) \leftarrow n$;
                foreach $v \sim u$ do
                    if $\tau(v) = 0$ then
                        $(v_1, \dots, v_p) \leftarrow \Gamma(v)$;
                        $(x'_1, \dots, x'_p) \leftarrow (\tau(v_1), \dots, \tau(v_p))$;
                        $(y'_1, \dots, y'_p) \leftarrow \text{once}(x'_1, \dots, x'_p)$;
                        if $(y'_1, \dots, y'_p) = (y_1, \dots, y_p)$ then
                            $s$.push($v$);
                        end
                    end
                end
            end
        end
    end
end
```

```
Algorithm 4: Labeling of active 0-cells
  Input: $\tau : T \to \mathbb{N}$ (topological label map, preliminary)
  Output: $\tau$ (modified), $n \in \mathbb{N}$ (number of active 0-cells), $\alpha \in \mathbb{N}^{n \times 6}$
          (neighborhood relation of 0-cells)
1  $n \leftarrow 0$;
2  foreach 0-cell $t \in T$ do
3      $(t_1, \ldots, t_6) \leftarrow \Gamma(t)$;
4      $(x_1, \ldots, x_6) \leftarrow (\tau(t_1), \ldots, \tau(t_6))$;
5      $(y_1, \ldots, y_6) \leftarrow \text{once}(x_1, \ldots, x_6)$;
6      if $y_1 \neq 0$ then
7          $n \leftarrow n + 1$;
8          for $j = 1$ to 6 do
9              $\alpha(n, j) \leftarrow y_j$;
10         end
11         $\tau(u) \leftarrow n$;
12     end
13 end
```

arriving at maximal labels M_0, M_1, M_2 of 0-, 1-, and 2-cells, respectively, for the entire volume.

Step 3 (label reconciliation). Whenever connected components of cells extend across block boundaries, their labels in the respective blocks (with offsets added) need to be reconciled. Two disjoint set data structures equipped with the operations **union** and **find** (Cormen et al., 2009) are used for this purpose, one for 1-cells and one for 2-cells, the former with M_1, the latter with M_2 initially distinct sets, each set containing one label. First, union(l_1, l_2) is called for the pair (l_1, l_2) of distinct labels assigned to any active 1-cells and 2-cells within a region of overlap. Second, each label l is replaced by the representative find(l) of the union to which it belongs.

Step 4 (curve merging). As is elucidated in the correctness analysis of this algorithm in Section 4.6.1, 1-components can still be falsely split and 0-cells falsely labeled as active at this stage. Thus, in a last step, each 0-cell t_0 and any pair (t_1, t'_1) of 1-components bounded by t_0 is considered. The labels of t_1 and t'_1 are reconciled if t_1 and t'_1 bound the same connected components of 2-cells. If at least one reconciliation has taken place, the activity of t_0 is re-computed.

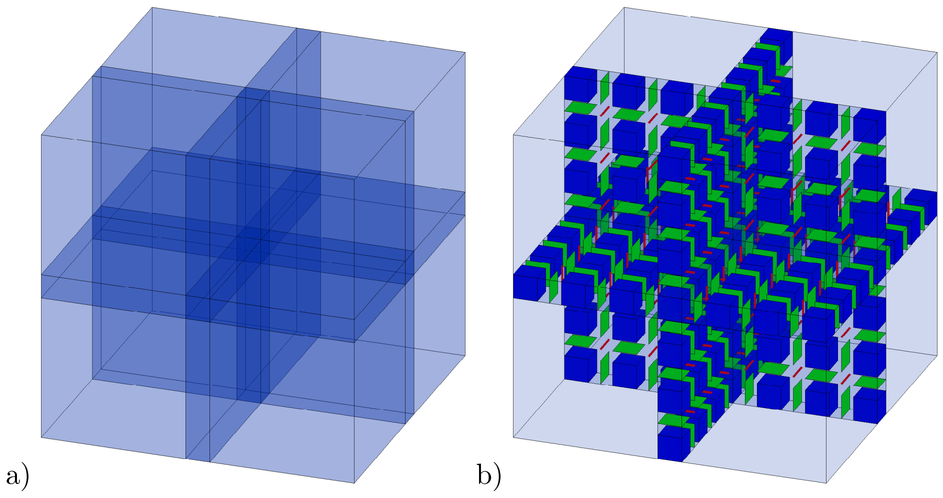


Figure 4.4: The topological grid is subdivided into blocks, leaving an overlap of one cell in each direction (a). Cells inside regions of overlap (b) are assigned two different labels during the independent processing of the blocks. These labels are subsequently reconciled.

4.6.1 Correctness of the Algorithm

In order to prove that the block-wise processing is correct, the segment label map σ as well as the decomposition of the topological grid into blocks are assumed to be arbitrary but fixed. The labeling τ' output by the block-wise method is compared to the labeling τ obtained from the application of Algorithms 3, 4 and 5 to the entire segment label map. While the latter is known to be correct, the former is correct if the two labelings are isomorphic:

Definition 11. Two labelings $\tau, \tau' : T \to \mathbb{N}$ of the topological grid T are isomorphic w.r.t. a subset $U \subseteq T$ if and only if the following conditions hold:

$$\forall u \in U : \quad \tau(u) = 0 \Leftrightarrow \tau'(u) = 0 \ , \tag{4.4}$$
$$\forall u, v \in U : \quad \tau(u) = \tau(v) \Leftrightarrow \tau'(u) = \tau'(v) \ . \tag{4.5}$$

If τ and τ' are isomorphic w.r.t. the entire domain T, they are simply called isomorphic.

Proposition 1. τ and τ' are isomorphic w.r.t. all 3-cells of T.

Proof. 3-cell labels are copied from the segment label map σ to τ and

τ', respectively by both algorithms. The labelings τ and τ' are therefore identical and thus isomorphic w.r.t. all 3-cells of T. □

Proposition 2. *τ and τ' are isomorphic w.r.t. all 2-cells of T.*

Proof. During block-wise processing, the decision whether or not a 2-cell obtains a non-zero label depends exclusively on the labeling of 3-cells w.r.t. which τ and τ' are identical. Thus, (4.4) holds for all 2-cells of T.

Let $u, v \in T$ be any 2-cells. If $\tau(u) \neq \tau(v)$, u and v bound different pairs of segments and hence obtain different labels during block-wise processing. Such labels are not reconciled. Thus, $\tau'(u) \neq \tau'(v)$.

If $\tau(u) = \tau(v)$, there exists a path of 2-cells between u and v on which all cells separate the same pair of segments. If this path is contained in one single block, all its 2-cells obtain the same label during the independent processing of that block. If the path crosses the boundaries of blocks, the labels along the path are reconciled. Thus, $\tau'(u) = \tau'(v)$.

Hence, (4.5) hold for all pairs of 2-cells. In conclusion, τ and τ' are isomorphic w.r.t. all 2-cells of T. □

Proposition 3. *For each block $U \subseteq T$, any 1-cell $t_1 \in U$, and any 2-cell $t_2 \in \Gamma(t_1)$, the label assigned to t_2 is unique among the labels assigned to all 2-cells in $\Gamma(t_1)$ before label reconciliation if and only if it is unique afterwards.*

Proof. ($\Rightarrow$) Suppose t_2 had a unique label among the elements of $\Gamma(t_1)$ before label reconciliation and the same label as another $t_2' \in \Gamma(t_1)$ afterwards, i.e. in τ'. Then, t_2 and t_2' separated the same pair of segments because τ' is isomorphic to the correct labeling τ w.r.t the 2-cells. Moreover, we know by definition that $t_2 \leftrightarrow t_2'$, so t_2 and t_2' would have obtained the same label before label reconciliation. A contradiction. ($\Leftarrow$) Trivial. □

Proposition 4. *For all 1-cells $t_1 \in T$ holds $\tau(t_1) = 0 \Leftrightarrow \tau'(t_1) = 0$.*

Proof. During the independent processing of each block, any 1-cell $t_1 \in T$ obtains a non-zero label if and only if at least one 2-cell label is unique among the labels assigned to all 2-cells of $\Gamma(t_1)$. In this step, the 2-cell labels before label reconciliation are considered. However, by Prop. 3, this is no different than considering the 2-cell labels of τ'. Moreover, τ' and τ are isomorphic w.r.t. all 2-cell and thus, 1-cells obtain a non-zero label in τ' precisely if they are labeled non-zero in τ, i.e. $\tau(t_1) = 0 \Leftrightarrow \tau'(t_1) = 0$. □

Proposition 5. *For all 1-cells $u, v \in T$ holds $\tau(u) = \tau(v) \Leftarrow \tau'(u) = \tau'(v)$.*

Proof. If $\tau'(u) = 0$, the conjecture holds by virtue of Prop. 4. If $\tau'(u) \neq 0$, it follows that $\tau'(v) \neq 0$ (by assertion) as well as $\tau(u) \neq 0$ and $\tau(v) \neq 0$ (by

Prop. 4). Moreover, $\tau'(u) = \tau'(v)$ requires by construction of τ' that u and v are connected by a path of 1-cells all of which bound the same connected components of 2-cells (in τ'). As τ and τ' are isomorphic w.r.t. the 2-cells and as 1-cells are labeled correctly in τ, it follows that $\tau(u) = \tau(v)$. □

Proposition 6. For all 0-cells $t \in T$ holds $\tau(t) \neq 0 \Rightarrow \tau'(t) \neq 0$.

Proof. If $\tau(t) \neq 0$, at least one 1-cell label is non-zero and unique among the labels assigned to all 1-cells in $\Gamma(t)$, i.e.

$$\exists u \in \Gamma(t) : \quad \tau(u) \neq 0 \wedge \forall v \in \Gamma(t) \setminus \{u\} : \tau(u) \neq \tau(v) \ .$$

For any such u follows by Prop. 4 and 5

$$\tau'(u) \neq 0 \wedge \forall v \in \Gamma(t) \setminus \{u\} : \tau'(u) \neq \tau'(v)$$

and thus, by construction of τ', the conjecture. □

In order to prove that τ and τ' are isomorphic, it remains to be shown that the inverse implications of Prop. 5 and 6 also hold. Unlike the above propositions which hold by construction of τ' in Steps 1–3 of the block algorithm, the two missing implications are enforced explicitly, by Step 4.

As the following example shows, 1-components can indeed be falsely split and 0-cells falsely labeled active if Step 4 is omitted. In Fig. 4.5a, a segment label map on a grid of $3 \times 3 \times 2$ voxels is shown. Six segments are identified by the integers 1 through 6. The correct corresponding topological label map is depicted in Fig. 4.5b, and the connected components are plotted in Fig. 4.6a and 4.6b. The 1-cell labels in Fig. 4.5b are colored in accordance with the graphical visualization in Fig. 4.6b.

Assume that the segment label map in Fig. 4.5a is processed block-wise, with blocks of $2 \times 2 \times 2$ voxels. Note that this block-size includes an overlap of one voxel in each direction. The bold font in Fig. 4.5a indicates one of these blocks. The topological label map that is constructed when this block is processed independently is depicted in Fig. 4.5c. While the 1-cells labeled 1 and 2 are merged into one connected component during label reconciliation after all blocks have been processed, the 1-cells labeled 3 and 4 are merged only in Step 4 of the algorithm. If Step 4 were omitted, the incorrect labeling shown in Fig. 4.6c would be computed.

Theorem 1. τ and τ' are isomorphic.

Proof. In addition to the implications proven above, Step 4 of the block-wise processing enforces:

(1) For all 1-cells $u, v \in T$ holds $\tau(u) = \tau(v) \Leftarrow \tau'(u) = \tau'(v)$.

(2) For all 0-cells $t \in T$ holds $\tau(t) \neq 0 \Leftarrow \tau'(t) \neq 0$. □

a)

z = 1		
1	1	1
1	**2**	**1**
1	**1**	**3**

z = 2		
4	4	4
4	**5**	**4**
4	**4**	**6**

b)

z = 1				
1	0	1	0	1
0	0	1	0	0
1	1	2	1	1
0	0	1	0	2
1	0	1	2	3

z = 2				
3	0	3	0	3
0	0	**1**	0	0
3	**1**	4	**1**	3
0	0	**1**	0	**2**
3	0	3	**2**	5

z = 3				
4	0	4	0	4
0	0	6	0	0
4	6	5	6	4
0	0	6	0	7
4	0	4	7	6

c)

z = 1		
2	1	1
1	0	2
1	2	3

z = 2		
3	**2**	5
1	**1**	**4**
4	**3**	6

z = 3		
5	7	4
7	0	8
4	8	6

Figure 4.5: a) A segment label map on a grid of $3 \times 3 \times 2$ voxels. b) The correct corresponding topological label map. Colors are in accordance with the 1-cells shown in Fig. 4.6b. c) The topological label map of the block depicted in bold font in (a). 1-cells are colored in accordance with Fig. 4.6c. The 1-cells labeled 1 and 2 are merged during label reconciliation while the 1-cells labeled 3 and 4 are merged in Step 4 of the block-wise processing.

By virtue of this theorem, the block-wise processing is correct.

4.6.2 Complexity

The runtime overhead introduced by the merging of labels is $O((N + M_1 + M_2)\log(M_1 + M_2))$ where N is the number of cells within regions of overlap. Time $O(N \log(M_1 + M_2))$ is used for the $O(N)$ calls of **union**, whereas time $O((M_1 + M_2)\log(M_1 + M_2))$ is required for the $M_1 + M_2$ **find**-operations. In practice, this overhead is negligible compared to the runtime of the connected component labeling.

4.6.3 Parallelization

The algorithm can be used in two different settings: Blocks can be processed consecutively in order to extract the geometry of a large volume

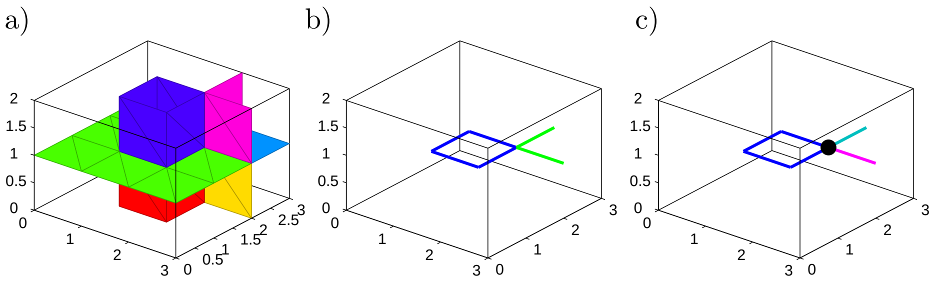


Figure 4.6: Connected components of 2-cells (a) and 1-cells (b) defined by the topological label map in Fig. 4.5b. c) 1-cells and the 0-cell from a block-wise labeling where Step 4 of the algorithm is omitted.

segmentation in limited RAM. Perhaps more interestingly, blocks can be processed in parallel, possibly on several machines, with virtually no process synchronization or inter-process communication.

Indeed, if it were not for parallelization, the block-wise connected component labeling could have been implemented simpler, starting with only the 2-cells, followed by the disambiguation and reconciliation of their labels across all blocks even before any 1-cell or 0-cell is labeled. This would render Step 4 of the block-wise processing unnecessary. However, the program would have to wait until the 2-cells of **all** blocks have been labeled before it could label the first 1-cell. In contrast, the proposed algorithm starts labeling the 1-cells within a block as soon as the labeling of 2-cells within that block is finished, regardless of the progress on other blocks.

4.7 Redundant Storage for Constant Time Access

The algorithm proposed in the last section labels segments, faces between segments, the curves between faces and the points between curves on the topological grid. The output is a topological label map that provides constant time access to the label at any topological coordinate. It allows to determine in constant time whether there is a face, curve, or point at a given location and if so, to determine its label. This is important, e.g. for the visualization of 2-dimensional slices of a segmentation that show not only segments but also faces, curves, and points. The algorithm makes explicit the bounding relations between the geometric objects.

However, not only the labels of individual cells and the adjacency of geometric objects are important but also the set of all cells that belong to the same component. For image analysis, it can for instance be useful to compute the mean gray value over a face between two segments. Yet, a list of all 2-cells of the face cannot be obtained in constant time from the topological grid labeling. Thus, a redundant representation of geometry is constructed that contains, for each j-component, a list of all its j-cells.

This redundant representation as well as the topological grid labeling are stored on the hard drive using the Hierarchical Data Format (HDF5). HDF5 was originally developed by the National Center for Supercomputing Applications (NCSA) and is now maintained by the non-profit HDF5-Group[3]. It is widely used, especially in the life sciences (Dougherty et al., 2009). An HDF5 file contains two principal types of objects, groups and datasets. Datasets represent the actual storage containers and are multi-dimensional arrays of a unique type, while groups represent an organizational concept analogous to a directory that enables the user to hierarchically structure the data within the file. Furthermore, attributes may be assigned to any dataset or group and contain meta information pertaining to the data stored within these objects.

Two HDF5 files are used here. The first file is associated with the labeling algorithm of Section 4.6. Its structure is depicted in Fig. 4.7a. For each block and its index j in the order of blocks, a sub-group named j is created in the group **blocks**. The sub-group j contains the dataset **topological-grid**, a 3-dimensional array that stores the topological label map of the block. Furthermore, it contains datasets for the neighborhood relations of connected components as well as the label offsets of the block which are computed during label disambiguation. During label reconciliation, the datasets **relabeling-**k and **neighborhood-**k are created in the main file to store the labeling and the neighborhood relations of k-components of the entire topological grid.

Using this file, constant time access to the label of a given cell works as follows. After identifying a block k to which the j-cell of interest belongs, the label l is read off from the dataset **topological-grid** of that block. Except for 3-cells and inactive cells, the offset m associated with cell order j and block k is loaded and the dataset **relabeling-**j accessed at location $l+m$ for the globally consistent label of the cell. In practice, all data except the topological label map can be kept in RAM.

The second HDF5 file stores one coordinate list for each 1-, 2-, and 3-component as well as the coordinates of all active 0-cells. Initially, the most straight forward group hierarchy was chosen for this file: Three groups associated with segments, faces, and curves, each containing one extendible

[3]`http://www.hdf5group.org`

a)

t	d	Name
D	1	segmentation-shape
D	1	block-shape
G		blocks
G		$\langle b \rangle$
D	3	topological-grid
D	1	max-labels
D	1	label-offsets
D	2	neighborhood-0
D	2	neighborhood-1
D	2	neighborhood-2
D	1	max-labels
D	1	relabeling-1
D	1	relabeling-2
D	2	neighborhood-0
D	2	neighborhood-1
D	2	neighborhood-2

b)

t	d	Name
A	1	number-of-bins
D	1	segmentation-shape
D	1	max-labels
D	2	0-cells
G		1-components
G		bin-$\langle b \rangle$
D	2	$\langle q \rangle$-$\langle p \rangle$
G		2-components
G		bin-$\langle b \rangle$
D	2	$\langle q \rangle$-$\langle p \rangle$
G		3-components
G		bin-$\langle b \rangle$
D	2	$\langle q \rangle$-$\langle p \rangle$
D	1	parts-counters-1
D	1	parts-counters-2
D	1	parts-counters-3

Figure 4.7: Two HDF5 files store the information extracted during block-wise processing. Along with each data item, its type t (a group G, a dataset D, or an attribute A) and dimension d are shown. a) The 1st file provides constant-time access to the label of any cell as well as to the neighborhoods of connected components of cells. b) The 2nd file provides constant-time access to the coordinate lists of entire segments, faces and curves.

dataset for each component. In addition, the coordinates of all 0-cells were stored in one 2-dimensional array. This group hierarchy turned out to be problematic when more than 10^6 datasets were created per group, using version 1.8.4 of the HDF5 library. To overcome this problem, the more complex group hierarchy shown in Fig. 4.7b is used instead. In each of the groups **1-components**, **2-components**, and **3-components**, a fixed number of sub-groups is created into which datasets containing coordinates lists are distributed. Also for performance reasons, the use of extendible datasets was dropped, which means that due to the block-wise nature of the algorithm, one complete j-component may be associated with several datasets representing its fragments from different blocks. The HDF5 file contains the datasets **parts-counters-**j for the number of datasets a single j-connected is split into.

4.7.1 Alternatives

A more obvious way to store the coordinate lists is to create one binary file for each list. The use of many files offers the advantage that the coordinate lists may easily be extended by appending to files, an important asset for block-wise processing. However, bearing in mind that large segmentations easily contains in excess of 10^6 connected components, the vast amount of files places a heavy burden on the file system, making simple operations such as copying the data extremely time consuming. In contrast, HDF5 was designed to organize large numbers of binary datasets.

A good alternative to HDF5 is a relational database. Experiments with a PostgreSQL database showed promising performance. This approach was nevertheless abandoned in favor of HDF5 because the necessity to install and configure a database might deter potential users from trying out the software.

4.8 Conclusion

In this chapter, a new algorithm for geometry and topology extraction from large volume segmentations is proposed. In contrast to previous methods, this algorithms processes volume segmentations in a block-wise fashion. This facilitates geometry and topology extraction from large volume segmentations with limited RAM and in parallel. The geometry is stored in HDF5 files that provide constant time access to the labels of segments, faces between segments, curves between faces and points between curves at any location as well as to lists of coordinates that constitute these objects. This representation makes a geometric analysis of large volume segmentations practical.

4.9 Compiling and Installing the Software

The CGP software is provided as C++ source code with a CMake 2.6 build system for the command line tools **cgpx** and **cgpr** as well as for MATLAB mex files. CGP depends on the HDF5 Library (version 1.8.4 or higher[4] and can optionally make use of the Message Passing Interface[5] (MPI), and the Visualization Toolkit[6] vtk, . An example segmentation of $50 \times 50 \times 50$ voxels is included, along with the according outputs of **cgpx** and **cgpr**. The following paragraphs describe how CGP can be compiled on a system that has HDF5 installed.

4.9.1 Linux/UNIX and GNU C++

Unpack the source archive and create a build directory. Execute CMake in this directory, providing the path to the source as the last parameter:

```
unzip cgp.zip
mkdir build-cgp && cd build-cgp
CMake ../cgp
make
```

If HDF5, MATLAB, or vtk are installed in a non-standard way, CMake will not find them automatically. In this case, paths to include files and libraries need to be set manually in the above call, e.g. for HDF5:

```
CMake -DHDF5_INCLUDE_DIR=$HOME/inc \
  -DHDF5_LIBRARY=$HOME/lib/libhdf5.so \
  ../cgp
```

4.9.2 Microsoft Windows and VisualStudio

Unpack the source archive, create a build directory, and use the CMake GUI to configure. If CMake does not find HDF5, MATLAB, or vtk although they are installed, set the include paths and library paths for these packages manually. CMake will generate a VisualStudio solution file. Open this file, build the target ALL_BUILD in release mode, and install the binaries by building the target INSTALL.

[4]http://www.hdfgroup.org/HDF5
[5]http://www.mcs.anl.gov/research/projects/mpi
[6]http://vtk.org

4.10 Using the Software

4.10.1 From the Command Line

The command line tools **cgpx** and **cgpr** compute a representation of the geometry and topology of a volume segmentation. The segmentation needs to be stored as a 3-dimensional array of 32-bit unsigned integers in one dataset in the root group of an HDF5 file. The command line tools are then used as follows:

cgpx ⟨*input file*⟩ ⟨*dataset*⟩ ⟨*b*1⟩ ⟨*b*2⟩ ⟨*b*3⟩ ⟨*output file*⟩
cgpr ⟨*input file*⟩ ⟨*output file*⟩.

The first tool constructs a labeled topological grid in a block-wise fashion using the block shape $b1 \times b2 \times b3$. The second tool writes a list of topological coordinates for each geometric object. Suppose, as an example, that a segmentation of $2{,}000^3$ voxels is stored as a 3-dimensional array in the dataset **seg** of the HDF5 file **segmentation.h5**. On a desktop computer equipped with 2 GB of RAM, a block-size of 200^3 voxels is reasonable, so a representation of the geometry and topology of the segmentation can be obtained like this:

```
cgpx segmentation.h5 seg 200 200 200 grid.h5
cgpr grid.h5 objects.h5
```

In order to process several blocks in parallel, invoke the command line tool **cgpx** via **mpiexec**, e.g.

```
mpiexec -n 2 cgpx segmentation.h5 seg
  200 200 200 grid.h5
```

4.10.2 From MATLAB

In MATLAB, segmentations are conveniently stored as 3-dimensional arrays whose entries are 32-bit unsigned integers that correspond to segment labels. In order to extract the geometry and topology from a segmentation, the array has to be written to an HDF5 file by means of the function **cgp_save**. The following call of **cgp_save** writes the array S as the dataset **seg** into the HDF5 file **seg.h5**:

```
  cgp_save('seg.h5', '/seg', S);
```

Geometry and topology extraction can now be performed either from the command line, using the tools **cgpx** and **cgpr** as described in Section 4.10.1, or directly from MATLAB, using the according mex-functions:

```
cgpx('seg.h5', 'seg', uint32([b1 b2 b3]),
  'grid.h5');
cgpr('grid.h5', 'objects.h5');
```

where $b1 \times b2 \times b3$ specifies the block shape.

A number of functions named with the prefix **cgp** can be used to selectively load data from the geometry file. In the following example, one curve and its adjacent faces are plotted.

```
desc = cgp_open('objects.h5');
curve_id = 100;
hold on;
tcl = cgp_load_object(desc, 1, curve_id);
cgp_plot_1cells(tcl);
neighbors = desc.neighborhoods{2}(curve_id,:);
for j = 1:length(neighbors)
  if neighbors(j) == 0
      break;
  end
  tcl = cgp_load_object(desc,2,neighbors(j));
  tri = cgp_triangulate(tcl);
  cgp_plot_triangulation(tri, rand(1,3),0.7);
end
hold off;
cgp_close(desc);
```

A 0-cell and its adjacent curves are plotted as follows:

```
desc = cgp_open('objects.h5');
point_id = 100;
hold on;
tcl = cgp_load_object(desc, 0, point_id);
cgp_plot_0cells(tcl);
neighbors = desc.neighborhoods{1}(point_id,:);
for j = 1:length(neighbors)
  if neighbors(j) == 0
      break;
  end
  tcl = cgp_load_object(desc,1,neighbors(j));
  cgp_plot_1cells(tcl, rand(1,3));
end
hold off;
cgp_close(desc);
```

4.10.3 From C++

The C++ API for parallelized geometry and topology extraction is defined in the header files **cgp_hdf5.hxx**, **CgpxMaster.hxx**, and **CgpxWorker.hxx**. The construction of the topological grid is invoked by classes

```
template<class T, class C> class CgpxMaster;
template<class T, class C> class CgpxWorker;
```

that implement a master-worker-scheme using MPI. The type T is used for labels of geometric objects; unsigned integers of at least 32 bits should be used. The type C is used for coordinates to navigate in arrays. 16-bit integers are sufficient if the segmentation is smaller than 32769 voxels in each dimension.

The coordinate lists of all geometric objects can be constructed from the topological grid by means of the function

```
template<class T, class C>
void geometry3blockwise(
  const hid_t&, // input HDF5 file
  const hid_t&  // output HDF5 file
);
```

The source files of the command line tools,

```
src/cmd/cgpx.cxx
src/cmd/cgpr.cxx
```

show the interested reader how the classes and function are used.

5 Energy Minimization in Higher-order Graphical Models by Lazy Flipping

5.1 Synopsis

In Chapter 3, the problem of segmenting volume images of nervous systems is cast into an NP-hard problem of optimizing functions of binary variables that decompose according to a graphical model. Chapter 4 describes the representation of the volume image on which the graphical model is built. This chapter presents a new search algorithm for the approximate solution of the optimization problem raised in Chapter 3. The scope of this algorithm goes beyond the application in Chapter 3. It can in fact be applied to binary-valued models of any order and structure. The main novelty is a technique to constrain the search space based on the structure of the graph. When pursued to the full search depth, the algorithm is guaranteed to converge to a global optimum, passing through a series of monotonously improving local optima that are guaranteed to be optimal within a given and increasing Hamming distance. For a search depth of 1, the algorithm specializes to Iterated Conditional Modes (ICM). Between these extremes, a useful tradeoff between approximation quality and runtime is established. Experiments on models derived from both illustrative and real problems show that approximations found with limited search depth match or improve those obtained by state-of-the-art methods based on message passing and linear programming.

5.2 Introduction

Energy functions that depend on thousands of binary variables and decompose according to a graphical model (Section 2.1.2) into potential functions that depend on subsets of all variables have been used successfully for pattern analysis, e.g. in the seminal works by Besag (1986); Boykov et al. (2001); Geman and Geman (1984); McEliece et al. (1998). An important problem is the minimization of the sum of potentials, i.e. the search for an assignment of zeros and ones to the variables that minimizes the energy. This problem can be solved efficiently by dynamic programming if the graph is acyclic (Pearl, 1988) or its treewidth is small enough (Lauritzen, 1996), and by finding a minimum s-t-cut (Boykov et al., 2001) if the energy function is (permuted) submodular (Kolmogorov and Zabin, 2004; Schlesinger, 2007). In general, the problem is NP-hard (Kolmogorov and Zabin, 2004). For moderate problem sizes, exact optimization is sometimes tractable by means of Mixed Integer Linear Programming (MILP) (Schrijver, 1986, 2003). Contrary to popular belief, some practical computer vision problems can indeed be solved to optimality by modern MILP solvers (cf. Section 5.6). However, all such solvers are eventually overburdened when the problem becomes too large. In cases where exact optimization is intractable, one has to settle for approximations. While substantial progress has been made in this direction, a deterministic non-redundant search algorithm that constrains the search space based on the topology of the graphical model has not been proposed before. This article presents a depth-limited exhaustive search algorithm, the Lazy Flipper, that does just that.

The Lazy Flipper starts from an arbitrary initial assignment of zeros and ones to the variables that can be chosen, for instance, to minimize the sum of only the first order potentials of the graphical model. Starting from this initial configuration, it searches for flips of variables that reduce the energy. As soon as such a flip is found, the current configuration is updated accordingly, i.e. in a greedy fashion. In the beginning, only single variables are flipped. Once a configuration is found whose energy can no longer be reduced by flipping of **single** variables, all those subsets of two and successively more variables that are connected via potentials in the graphical model are considered. When a subset of more than one variable is flipped, all smaller subsets that are affected by the flip are revisited. This allows the Lazy Flipper to perform an exhaustive search over all subsets of variables whose flip potentially reduces the energy. Two special data structures described in Section 5.4 are used to represent each subset of connected variables precisely once and to exclude subsets from the search whose flip cannot reduce the energy due to the topology of the graphical model and the history of unsuccessful flips.

Overall, the new algorithm has four favorable properties: (i) It is strictly convergent. While a global minimum is found when searching through all subgraphs (typically not tractable), approximate solutions with a guaranteed quality certificate (Section 5.5) are found if the search space is restricted to subgraphs of a given maximum size. The larger the subgraphs are allowed to be, the tighter the upper bound on the minimum energy becomes. This allows for a favorable trade-off between runtime and approximation quality. (ii) Unlike in brute force search, the runtime of lazy flipping depends on the topology of the graphical model. It is exponential in the worst case but can be shorter compared to brute force search by an amount that is exponential in the number of variables. It is approximately linear in the size of the model for a fixed maximum search depth. (iii) The Lazy Flipper can be applied to graphical models of any order and topology, including but not limited to the more standard grid graphs. Directed Bayesian Networks and undirected Markov Random Fields are processed in the exact same manner; they are converted to factor graph models (Kschischang et al., 2001) before lazy flipping. (iv) Only trivial operations are performed on the graphical model, namely graph traversal and evaluations of potential functions. These operations are cheap compared, for instance, to the summation and minimization of potential functions performed by message passing algorithms, and require only an implicit specification of potential functions in terms of program code that computes the function value for any given assignment of values to the variables.

Experiments on simulated and real-world problems, submodular and non-submodular functions, grids and irregular graphs (Section 5.6) assess the quality of Lazy Flipper approximations, their convergence as well as the dependence of the runtime of the algorithm on the size of the model and the search depth. The results are put into perspective by a comparison with Iterated Conditional Modes (ICM) (Besag, 1986), Belief Propagation (BP) (Kschischang et al., 2001; Pearl, 1988), Tree-reweighted BP (Wainwright and Jordan, 2008; Wainwright et al., 2005) and a Dual Decomposition ansatz using sub-gradient descent methods (Kappes et al., 2010; Komodakis et al., 2010).

5.3 Related Work

The Lazy Flipper is related in at least four ways to existing work.

First, it generalizes Iterated Conditional Modes (ICM) for binary variables (Besag, 1986). While ICM leaves all variables except one fixed in each step, the Lazy Flipper can optimize over larger (for small models: all) connected subgraphs of a graphical model. Furthermore, it extends Block-ICM (Frey and Jojic, 2005) that optimizes over specific subsets of variables in grid graphs to irregular and higher-order graphical models. Naive attempts to generalize ICM and Block-ICM to optimize over subgraphs of size k would consider all sequences of k connected variables and ignore the fact that many of these sequences represent the same set. This causes substantial problems because the redundancy is large, as we show in Section 5.4. The Lazy Flipper avoids this redundancy, at the cost of storing one unique representative for each subset. Compared to randomized algorithms that sample from the set of subgraphs (Jung et al., 2009; Swendsen and Wang, 1987; Wolff, 1989), this is a memory intensive approach. Up to 8 GB of RAM are required for the optimizations shown in Section 5.6. Now that servers with much larger RAM are available, it has become a practical option.

Second, the Lazy Flipper is a deterministic alternative to the randomized search for tighter bounds proposed and analyzed in 2009 by Jung et al. (2009). Exactly as in (Jung et al., 2009), sets of variables that are connected via potentials in the graphical model are considered and variables flipped if these flips lead to a smaller upper bound on the sum of potentials. In contrast to (Jung et al., 2009), unique representatives of these sets are visited in a deterministic order. Both algorithms maintain a current best assignment of values to the variables and are thus related to the Swendsen-Wang algorithm (Barbu and Zhu, 2003; Swendsen and Wang, 1987) and Wolff algorithm (Wolff, 1989).

Third, lazy flipping with a limited search depth as a means of approximate optimization competes with message passing algorithms (Globerson and Jaakkola, 2007; Kschischang et al., 2001; Minka, 2001; Wainwright and Jordan, 2008) and with algorithms based on convex programming relaxations of the optimization problem (Globerson and Jaakkola, 2007; Kohli et al., 2008; Kumar et al., 2009; Werner, 2007), in particular with Tree-reweighted Belief Propagation (TRBP) (Kolmogorov, 2006; Wainwright and Jordan, 2008; Wainwright et al., 2005) and sub-gradient descent (Kappes et al., 2010; Komodakis et al., 2010).

Fourth, the Lazy Flipper guarantees that the best approximation found with a search depth $n_{\max}$ is optimal within a Hamming distance $n_{\max}$. A similar guarantee known as the Single Loop Tree (SLT) neighborhood (Weiss and Freeman, 2001) is given by BP in case of convergence. The SLT condition states that in any alteration of an assignment of values to the variables that leads to a lower energy, the altered variables form a subgraph in the graphical model that has at least two loops. The fact that Hamming optimality and SLT optimality differ can be exploited in practice. We show in one experiment in Section 5.6 that BP approximations can be further improved by means of lazy flipping.

5.4 The Lazy Flipper Data Structures

Two special data structures are crucial to the Lazy Flipper. The first data structure that we call a **connected subgraph tree (CS-tree)** ensures that only **connected** subsets of variables are considered, i.e. sets of variables which are connected via potentials in the graphical model. Moreover, it ensures that every such subset is represented precisely once (and not repeatedly) by an ordered sequence of its variables, (cf. Moerkotte and Neumann, 2006). The rationale behind this concept is the following: If the flip of one variable and the flip of another variable not connected to the first one do not reduce the energy then it is pointless to try a simultaneous flip of both variables because the (energy increasing) contributions from both flips would sum up. Furthermore, if the flip of a disconnected set of variables reduces the energy then the same and possibly better reductions can be obtained by flipping connected subsets of this set consecutively, in any order. All disconnected subsets of variables can therefore be excluded from the search if the connected subsets are searched ordered by their size.

Finding a unique representative for each connected subset of variables is important. The alternative would be to consider all sequences of pairwise distinct variables in which each variable is connected to at least one of its predecessors and to ignore the fact that many of these sequences represent the same set. Sampling algorithms that select and grow connected subsets in a randomized fashion do exactly this. However, the redundancy is large. As an example, consider a connected subset of six variables of a 2-dimensional grid graph as depicted in Fig. 5.1a. Although there is only one connected set that contains all six variables, 208 out of the $6! = 720$ possible sequences of these variables meet the requirement that each variable is connected to at least one of its predecessors. This 208-fold redundancy hampers the exploration of the search space by means of randomized algorithms; it is avoided in lazy flipping at the cost of storing one unique representative for every connected subgraph in the CS-tree.

The second data structure is a **tag list** that prevents the repeated assessment of unsuccessful flips. The idea is the following: If some variables have been flipped in one iteration (and the current best configuration has been updated accordingly), it suffices to revisit only those sets of variables that are connected to at least one variable that has been flipped. All other sets of variables are excluded from the search because the potentials that depend on these variables are unaffected by the flip and have been assessed in their current state before.

The tag list and the connected subgraph tree are essential to the Lazy Flipper and are described in the following sections, 5.4.1 and 5.4.2. For a quick overview, the reader can however skip these sections, take for granted that it is possible to efficiently enumerate all connected subgraphs of a graphical model, ordered by their size, and refer directly to the main al-

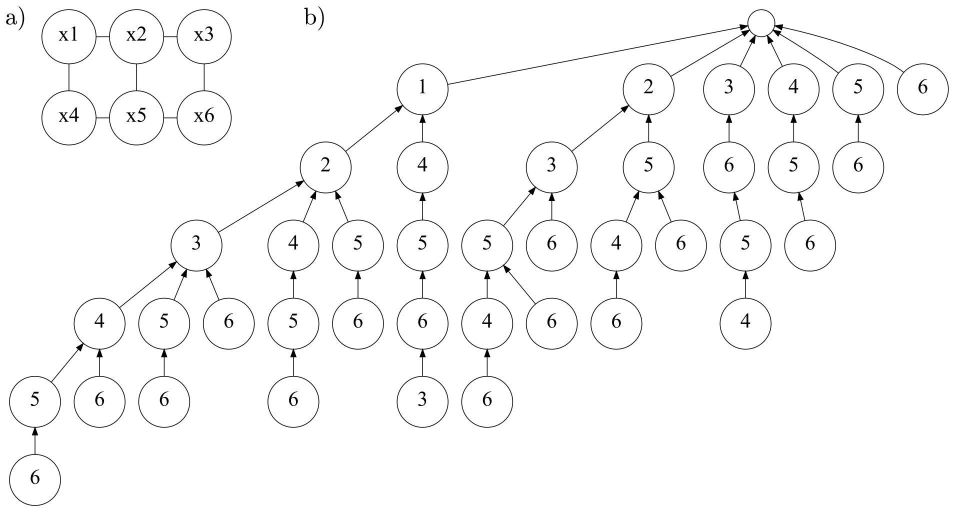


Figure 5.1: All connected subgraphs of a graphical model (a) can be represented uniquely in a connected subgraph tree (CS-tree) (b). Every path from a node in the CS-tree to the root node corresponds to a connected subgraph in the graphical model. While there are $2^6 = 64$ subsets of variables in total in this example, only 40 of these subsets are connected.

gorithm (Section 5.5 and Alg. 5). All non-trivial sub-functions used in the main algorithm are related to tag lists and the CS-tree and are described in detail now.

5.4.1 Connected Subgraph Tree (CS-tree)

The CS-tree represents subsets of connected variables uniquely. Every node in the CS-tree except the special root node is labeled with the integer index of one variable in the graphical model. The same variable index is assigned to several nodes in the CS-tree unless the graphical model is completely disconnected. The CS-tree is constructed such that every connected subset of variables in the graphical model corresponds to precisely one path in the CS-tree from a node to the root node, the node labels along the path indicating precisely the variables in the subset, and vice versa, there exists precisely one connected subset of variables in the graphical model for each path in the CS-tree from a node to the root node.

In order to guarantee by construction of the CS-tree that each subset of connected variables is represented precisely once, the variable indices of each subset are put in a special order, namely the lexicographically smallest order in which each variable is connected to at least one of its predecessors. The following definition of these sequences of variable indices is recursive and therefore motivates an algorithm for the construction of the CS-tree for the Lazy Flipper. A small grid model and its complete CS-tree are depicted in Fig. 5.1.

Definition 12 (CSR-Sequence). Given an undirected graph $G = (V, E)$ whose $m \in \mathbb{N}$ vertices $V = \{1, \dots, m\}$ are integer indices, every sequence that consists of only one index is called **connected subset representing (CSR)**. Given $n \in \mathbb{N}$ and a CSR-sequence $(v_1, \dots, v_n)$, a sequence $(v_1, \dots, v_n, v_{n+1})$ of $n + 1$ indices is called a **CSR-sequence** precisely if the following conditions hold:

(i) v_{n+1} is not among its predecessors, i.e. $\forall j \in \{1, \dots, n\} : v_j \neq v_{n+1}$.

(ii) v_{n+1} is connected to at least one of its predecessors, i.e. $\exists j \in \{1, \dots, n\} : \{v_j, v_{n+1}\} \in E$.

(iii) $v_{n+1} > v_1$.

(iv) If $n \geq 2$ and v_{n+1} could have been added at an earlier position $j \in \{2, \dots, n\}$ to the sequence, fulfilling (i)–(iii), all subsequent vertices $v_j, \dots, v_n$ are smaller than v_{n+1}, i.e.

$$\forall j \in \{2, \dots, n\} \left(\{v_{j-1}, v_{n+1}\} \in E \Rightarrow (\forall k \in \{j, \dots, n\} : v_k < v_{n+1}) \right) \ . \tag{5.1}$$

Based on this definition, three functions are sufficient to recursively build the CS-tree T of a graphical model G, starting from the root node. The function $q =$ **growSubset**(T, G, p) appends to a node p in the CS-tree the smallest variable index that is not yet among the children of p and fulfills (i)–(iv) for the CSR-sequence of variable indices on the path from p to the root node. It returns the appended node or the empty set if no suitable variable index exists. The function $q =$ **firstSubsetOfSize**(T, G, n) traverses the CS-tree on the current deepest level $n - 1$, calling the function **growSubset** for each leaf until a node can be appended and thus, the first subset of size n has been found. Finally, the function $q =$ **nextSubset-OfSameSize**(T, G, p) starts from a node p, finds its parent and traverses from there in level order, calling **growSubset** for each node to find the length-lexicographic successor of the CSR-sequence associated with the node p, i.e. the representative of the next subset of the same size. These functions are used by the Lazy Flipper (Alg. 5) to **construct** the CS-tree.

In contrast, the **traversal** of already constructed parts of the CS-tree (when revisiting subsets of variables after successful flips) is performed by functions associated with tag lists which are defined the following section.

5.4.2 Tag Lists

Tag lists are used to tag variables that are affected by flips. A variable is affected by a flip either because it has been flipped itself or because it is connected (via a potential) to a flipped variable. The tag list data structure comprises a Boolean vector in which each entry corresponds to a variable, indicating whether or not this variable is affected by recent flips. As the total number of variables can be large (10^6 is not exceptional) and possibly only a few variables are affected by flips, a list of all affected variables is maintained in addition to the vector. This list allows the algorithm to untag all tagged variables without re-initializing the entire Boolean vector. The two fundamental operations on a tag list L are **tag**(L, x) which tags the variable with the index x, and **untagAll**(L).

For the Lazy Flipper, three special functions are used in addition: Given a tag list L, a (possibly incomplete) CS-tree T, the graphical model G, and a node $s \in T$, *tagConnectedVariables*(L, T, G, s) tags all variables on the path from s to the root node in T, as well as all nodes that are connected (via a potential in G) to at least one of these nodes. The function $s =$ *firstTaggedSubset*(L, T) traverses the first level of T and returns the first node s whose variable is tagged (or the empty set if all variables are untagged). Finally, the function $t =$ *nextTaggedSubset*(L, T, s) traverses T in level order, starting with the successor of s, and returns the first node t for which the path to the root contains at least one tagged variable. These functions, together with those of the CS-tree, are sufficient for the Lazy Flipper, Alg. 5.

5.5 The Lazy Flipper Algorithm

In the main loop of the Lazy Flipper (lines 2–26 in Alg. 5), the size n of subsets is incremented until the limit $n_{\max}$ is reached (line 24). Inside this main loop, the algorithm falls into two parts, the **exploration part** (lines 3–11) and the **revisiting part** (lines 12–23). In the exploration part, flips of previously unseen subsets of n variables are assessed. The current best configuration c is updated in a greedy fashion, i.e. whenever a flip yields a lower energy. At the same time, the CS-tree is grown, using the functions defined in Section 5.4.1. In the revisiting part, all subsets of sizes 1 through n that are affected by recent flips are assessed iteratively until no flip of any of these subsets reduces the energy (line 14). The indices of affected variables are stored in the tag lists L_1 and L_2 (cf. Section 5.4.2). In practice, the Lazy Flipper can be stopped at any point, e.g. when a time limit is exceeded, and the current best configuration c taken as the output. It eventually reaches configurations for which it is guaranteed that no flip of n or less variables can yield a lower energy because all such flips that could potentially lower the energy have been assessed (line 14). Such configurations are therefore guaranteed to be optimal within a Hamming radius of n:

Definition 13 (Hamming-n bound). Given a function $E : \{0,1\}^m \to \mathbb{R}$, a configuration $c \in \{0,1\}^m$, and $n \in \mathbb{N}$, $E(c)$ is called a **Hamming-n upper bound** on the minimum of E precisely if $\forall c' \in \{0,1\}^m (|c' - c|_1 \leq n \Rightarrow E(c) \leq E(c'))$.

5.6 Experiments

For a comparative assessment of the Lazy Flipper, four optimization problems of different complexity are considered, two simulated problems and two problems based on real-world data. For the sake of reproducibility, the simulations are described in detail and the models constructed from real data are provided as supplementary material.

The first problem is a ferromagnetic Ising model that is widely used in computer vision for foreground vs. background segmentation (Boykov et al., 2001). Energy functions of this model consist of first and second order potentials that are submodular. The global minimum can therefore be found via a graph cut. We simulate random instances of this model in order to measure how the runtime of lazy flipping depends on the size of the model and the coupling strength, and to compare Lazy Flipper approximations to the global optimum (Section 5.6.1).

The second problem is a problem of finding optimal subgraphs on a grid. Energy functions of this model consist of first and fourth order potentials, of which the latter are not permuted submodular. We simulate difficult instances of this problem that cannot be solved to optimality, even when allowing several days of runtime. In this challenging setting, Lazy Flipper approximations and their convergence are compared to those of BP, TRBP and DD as well as to the lower bounds on local polytope relaxations obtained by DD (Section 5.6.2).

The third problem is the graphical model for removing excessive boundaries from image over-segmentations introduced in Chapter 3 but applied in 2D, to the 100 natural test images of the Berkeley Segmentation Database (BSD) (Martin et al., 2001). Energy functions of this model consist of first, third and fourth order potentials. In contrast to the grid graphs of the Ising

Algorithm 5: Lazy Flipper

```
Input: $G$: graphical model with $m \in \mathbb{N}$ binary variables,
       $c \in \{0,1\}^m$: initial configuration, $n_{\max} \in \mathbb{N}$: maximum size
       of subgraphs to be searched
Output: $c \in \{0,1\}^m$ (modified): configuration corresponding to
        the smallest upper bound found ($c$ is optimal within a
        Hamming radius of $n_{\max}$).
1  $n \leftarrow 1$; CS-Tree $T \leftarrow$ {root}; TagList $L_1 \leftarrow \emptyset$, $L_2 \leftarrow \emptyset$;
2  repeat
3      $s \leftarrow$ firstSubsetOfSize($T, G, n$);
4      if $s = \emptyset$ then break;
5      while $s \neq \emptyset$ do
6          if energyAfterFlip($G, c, s$) < energy($G, c$) then
7              $c \leftarrow$ flip($c, s$);
8              tagConnectedVariables($L_1, T, G, s$);
9          end
10         $s \leftarrow$ nextSubsetOfSameSize($T, G, s$);
11     end
12     repeat
13         $s_2 \leftarrow$ firstTaggedSubset($L_1, T$);
14         if $s_2 = \emptyset$ then break;
15         while $s_2 \neq \emptyset$ do
16             if energyAfterFlip($G, c, s_2$) < energy($G, c$) then
17                 $c \leftarrow$ flip($c, s_2$);
18                 tagConnectedVariables($L_2, T, G, s_2$);
19             end
20             $s_2 \leftarrow$ nextTaggedSubset($L_1, T, s_2$);
21         end
22         untagAll($L_1$); swap($L_1, L_2$);
23     end
24     if $n = n_{\max}$ then break;
25     $n \leftarrow n + 1$;
26 end
```

model and the optimal subgraph model, the corresponding factor graphs are irregular but still planar. The higher-order potentials are not permuted submodular but the global optimum can be found by means of MILP in approximately 10 seconds per model using one of the fastest commercial solvers (IBM ILOG CPLEX, version 12.1). Since CPLEX is closed-source software, the algorithm is not known in detail and we use it as a black box. The general method used by CPLEX for MILP is a branch-and-bound algorithm (Dakin, 1965; Land and Doig, 1960). The 100 instances of this model obtained from the test images are used to compare the Lazy Flipper to algorithms based on message passing and linear programming in a real-world setting where the global optimum is accessible (Section 5.6.3).

The fourth problem is identical to the third, except that instances are obtained from the 3-dimensional volume images of neural tissue acquired by means of Serial Block Face Scanning Electron Microscopy (SBFSEM) (Denk and Horstmann, 2004), cf. Chapter 3. Unlike in the 2-dimensional case, the factor graphs are no longer planar. Whether exact optimization by means of MILP is practical depends on the size of the model. The full SBFSEM datasets considered in Chapter 3 consist of more than 2000^3 voxels. To be able to compare approximations to the **global** optimum, we consider 16 models obtained from 16 SBFSEM volume sub-images of only 150^3 voxels for which the global optimum can be found by means of MILP within a few minutes (Section 5.6.4). The application of the best approximate solver (BP followed by lazy flipping) to the full dataset is shown in Chapter 3.

5.6.1 Ferromagnetic Ising model

The ferromagnetic Ising model consists of $m \in \mathbb{N}$ binary variables $x_1, \ldots, x_m \in \{0, 1\}$ that are associated with points on a 2-dimensional square grid and connected via second order potentials $E_{jk}(x_j, x_k) = 1 - \delta_{x_j, x_k}$ (δ: Kronecker delta) to their nearest neighbors. First order potentials $E_j(x_j)$ relate the variables to observed evidence in underlying data. The total energy of this model is the following sum in which $\alpha \in \mathbb{R}_0^+$ is a weight on the second order potentials, and $j \sim k$ indicates that the variables x_j and x_k are adjacent on the grid:

$$\forall x \in \{0,1\}^m : \quad E(x) = \sum_{j=1}^{m} E_j(x_j) + \alpha \sum_{j=1}^{m} \sum_{\substack{k=j+1 \\ k \sim j}}^{m} E_{jk}(x_j, x_k) \ . \tag{5.2}$$

For each $\alpha \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, an ensemble of ten simulated Ising models of $50 \cdot 50 = 2500$ variables is considered. The first order potentials E_j are initialized randomly by drawing $E_j(0)$ uniformly from the interval $[0, 1]$ and setting $E_j(1) := 1 - E_j(0)$. The exact global minimum of the total energy is found via a graph cut.

For each model, the Lazy Flipper is initialized with a configuration that minimizes the sum of the first order potentials. Upper bounds on the minimum energy found by means of lazy flipping converge towards the global optimum as depicted in Fig. 5.2. Color scales and gray scales in this figure respectively indicate the maximum size and the total number of distinct subsets that have been searched, averaged over all models in the ensemble. It can be seen from this figure that upper bounds on the minimum energy are tightened significantly by searching larger subsets of variables, independent of the coupling strength α. It takes the Lazy Flipper less than 100 seconds (on a single CPU of an Intel Quad Xeon E7220 at 2.93GHz) to exhaustively search all connected subsets of 6 variables. The amount of RAM required for the CS-tree (in bytes) is 24 times as high as the number of subsets (approximately 50 MB in this case) because each subset is stored in the CS-tree as a node consisting of three 64-bit integers: a variable index, the index of the parent node and the index of the level order successor (Section 5.4.1)[1].

For $n_{\max} \in \{1, 6\}$, configurations corresponding to the upper bounds on the minimum energy are depicted in Fig. 5.3. It can be seen from this figure that all connected subsets of falsely set variables are larger than $n_{\max}$. For a fixed maximum subgraph size $n_{\max}$, the runtime of lazy flipping scales approximately linearly with the number of variables in the Ising model (cf. Fig.5.4).

5.6.2 Optimal Subgraph Model

The optimal subgraph model consists of $m \in \mathbb{N}$ binary variables $x_1, \ldots, x_m \in \{0, 1\}$ that are associated with the edges of a 2-dimensional grid graph. A subgraph is defined by those edges whose associated variables attain the value 1. Energy functions of this model consist of first order potentials, one for each edge, and fourth order potentials, one for each node $v \in V$ in which four edges $(j, k, l, m) = \mathcal{N}(v)$ meet:

$$\forall x \in \{0,1\}^m : \quad E(x) = \sum_{j=1}^{m} E_j(x_j) \; + \sum_{(j,k,l,m) \in \mathcal{N}(V)} E_{jklm}(x_j, x_k, x_l, x_m) \ . \quad (5.3)$$

[1]The size of the CS-tree becomes limiting for very large problems. However, for regular graphs, implicit representations can be envisaged that overcome this limitation.

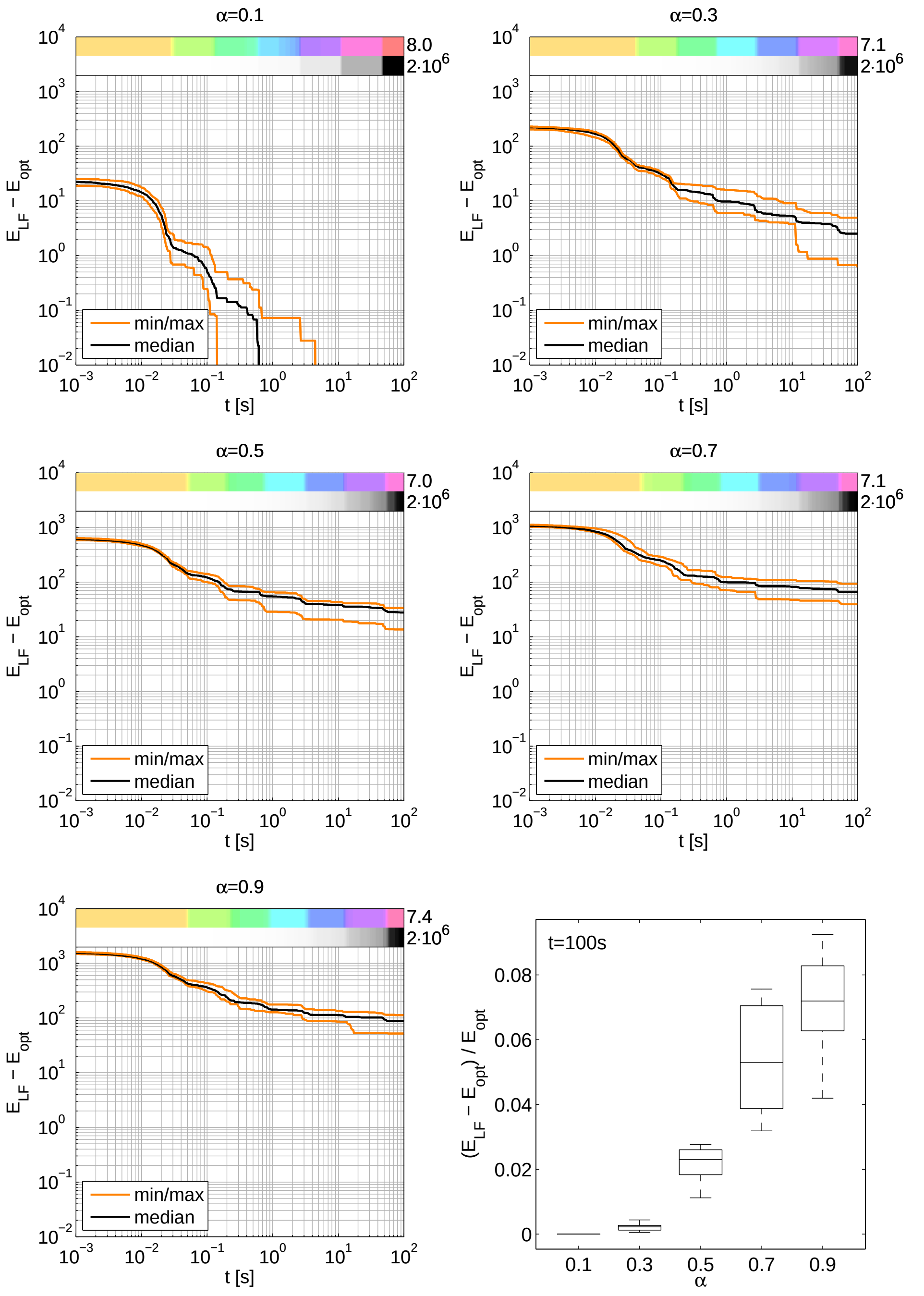


Figure 5.2: Upper bounds on the minimum energy of a graphical model can be found by flipping subsets of variables. The deviation of these upper bounds from the minimum energy is shown above for ensembles of ten random Ising models (Section 5.6.1). Compared to optimization by ICM where only one variable is flipped at a time, the Lazy Flipper finds significantly tighter bounds by flipping also larger subsets. The deviations increase with the coupling strength α. Color scales and gray scales indicate the size and the total number of searched subsets.

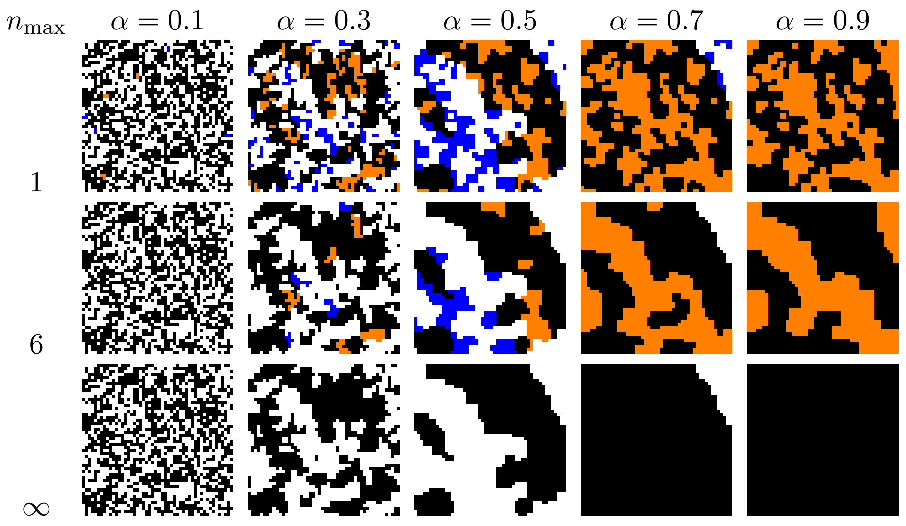


Figure 5.3: Configurations found by lazy flipping converge to a global optimum as the search depth $n_{\max}$ increases. For Ising models with different coupling strengths α (columns), deviations from the global optimum ($n_{\max} = \infty$) are depicted in blue (false 0) and orange (false 1), for $n_{\max} \in \{1, 6\}$. As the Lazy Flipper is greedy, these approximate solutions depend on the initialization and on the order in which subsets are visited.

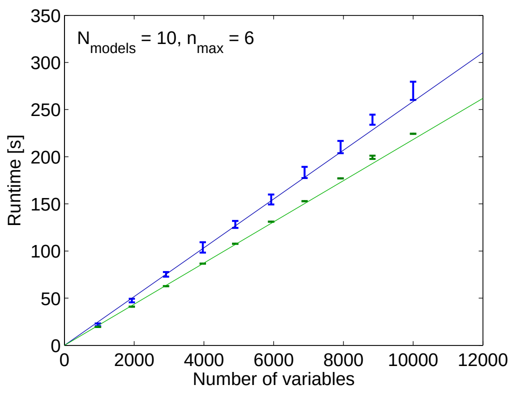


Figure 5.4: For a fixed maximum subgraph size ($n_{\max} = 6$), the runtime of lazy flipping scales only slightly more than linearly with the number of variables in the Ising model. It is measured for coupling strengths $\alpha = 0.25$ (upper curve) and $\alpha = 0.75$ (lower curve). Error bars indicate the standard deviation over 10 random models. Lines are fitted by least squares. Lazy flipping takes longer (0.0259 seconds per variable) for $\alpha = 0.25$ than for $\alpha = 0.75$ (0.0218 s/var) because more successful flips initiate revisiting.

All fourth order potentials are equal, penalizing dead ends and branches of paths in the selected subgraph:

$$E_{jklm}(x_j, x_k, x_l, x_m) = \begin{cases} 0.0 & \text{if } s = 0 \\ 100.0 & \text{if } s = 1 \\ 0.6 & \text{if } s = 2 \\ 1.2 & \text{if } s = 3 \\ 2.4 & \text{if } s = 4 \end{cases} \quad \text{with} \quad s = x_j + x_k + x_l + x_m \ . \tag{5.4}$$

An ensemble of 16 such models is constructed by drawing the unary potentials at random, exactly as for the Ising models. Each model has 19800 variables, the same number of first order potentials, and 9801 fourth order potentials. Approximate optimal subgraphs are found by Min-Sum Belief Propagation (BP) with parallel message passing (Kschischang et al., 2001; Pearl, 1988) and message damping (Murphy et al., 1999), by Tree-reweighted Belief Propagation (TRBP) (Wainwright and Jordan, 2008), by Dual Decomposition (DD) (Kappes et al., 2010; Komodakis et al., 2010) and by lazy flipping (LF). DD affords also lower bounds on the minimum energy. Details on the parameters of the algorithms and the decomposition of the models are given in Appendix 5.7.

Bounds on the minimum energy converge with increasing runtime, as depicted in Fig. 5.5. It can be seen from this figure that Lazy Flipper approximations converge fast, reaching a smaller energy after 3 seconds than the other approximations after 10000 seconds. Subgraphs of up to 7 variables are searched, using approximately 2.2 GB of RAM for the CS-tree. A gap remains between the energies of all approximations and the lower bound on the minimum energy obtained by DD. Thus, there is no guarantee that any of the problems has been solved to optimality. However, the gaps are upper bounds on the deviation from the global optimum. They are compared at $t = 10000$ s in Fig. 5.5. For any model in the ensemble, the energy of the Lazy Flipper approximation is less than 4% away from the global optimum, a substantial improvement over the other algorithms for this particular model.

5.6.3 Pruning of 2D Over-Segmentations

The graphical model for removing excessive boundaries from image over-segmentations contains one binary variable for each boundary between segments, indicating whether this boundary is to be removed (0) or preserved (1). First order potentials relate these variables to the image content, and non-submodular third and fourth order potentials connect adjacent boundaries, supporting the closedness and smooth continuation of preserved boundaries. The energy function is a sum of these potentials: $\forall x \in \{0,1\}^m$

$$E(x) = \sum_{j=1}^{m} E_j(x_j) + \sum_{(j,k,l)\in J} E_{jkl}(x_j, x_k, x_l) + \sum_{(j,k,l,p)\in K} E_{jklp}(x_j, x_k, x_l, x_p) \ . \quad (5.5)$$

We consider an ensemble of 100 such models obtained from the 100 BSD test images (Martin et al., 2001). On average, a model has 8845 ± 670 binary variables, the same number of unary potentials, 5715 ± 430 third order potentials and 98 ± 18 fourth order potentials. Each variable is connected via potentials to at most six other variables, a sparse structure that is favorable for the Lazy Flipper.

BP, TRBP, DD and the Lazy Flipper solve these problems approximately, thus providing upper bounds on the minimum energy. The differences between these bounds and the global optimum found by means of MILP are depicted in Fig. 5.6. It can be seen from this figure that, after 200 seconds, Lazy Flipper approximations provide a tighter upper bound on the global

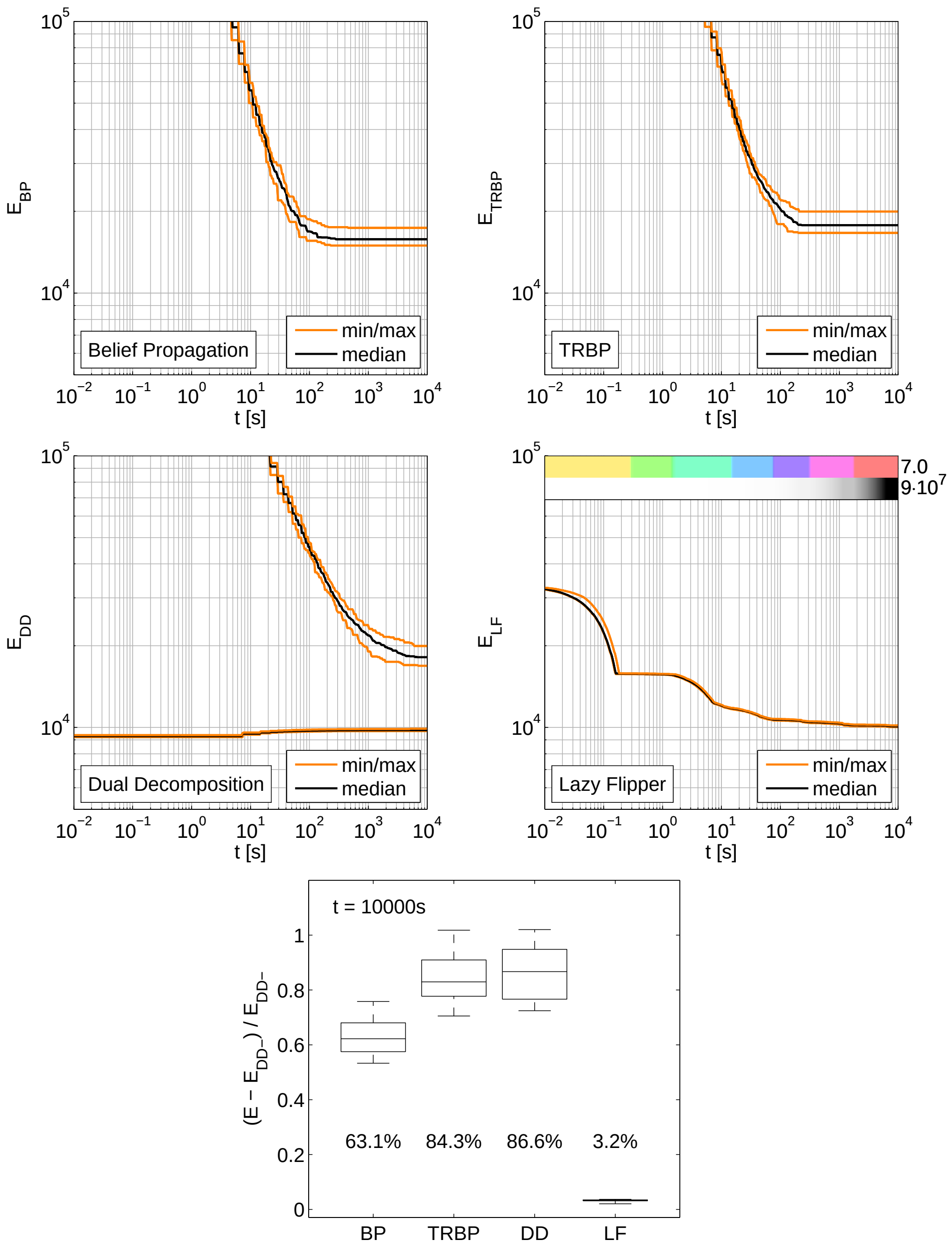


Figure 5.5: Approximate solutions to the optimal subgraph problem (Section 5.6.2) are found by BP, TRBP, DD and the Lazy Flipper (LF). Depicted are the median, minimum and maximum (over 16 models) of the corresponding energies. DD affords also lower bounds on the minimum energy. The mean search depth of LF ranges from 1 (yellow) to 7 (red). At $t = 10^4$ s, the energies of LF approximations come close to the lower bounds obtained by DD and thus, to the global optimum.

minimum in the median than those of the other three algorithms. BP and DD have a better peak performance, solving one problem to optimality. The Lazy Flipper reaches a search depth of 9 after around 1000 seconds for these sparse graphical models using roughly 720 MB of RAM for the CS-tree. At $t = 5000$ s and on average over all models, its approximations deviate by only 2.6% from the global optimum.

5.6.4 Pruning of 3D Over-Segmentations

The model described in the previous section is now applied in 3D to remove excessive boundaries from the over-segmentation of a volume image. In an ensemble of 16 such models obtained from 16 SBFSEM volume images, models have on average 16748 ± 1521 binary variables (and first order potentials), 26379 ± 2502 potentials of order 3, and 5081 ± 482 potentials of order 4.

For BP, TRBP, DD and Lazy Flipper approximations, deviations from the global optimum are shown in Fig. 5.7. It can be seen from this figure that BP performs exceptionally well on these problems, providing approximations whose energies deviate by only 0.4% on average from the global optimum. One reason is that most variables influence many (up to 60) potential functions, and BP can propagate local evidence from all these potentials. Variables are connected via these potentials to as many as 100 neighboring variables which hampers the exploration of the search space by the Lazy Flipper that reaches only of search depth of 5 after 10000 seconds, using 4.8 GB of RAM for the CS-tree, yielding worse approximations than BP, TRBP and DD for these models.

In practical applications where volume images and the according models are several hundred times larger and can no longer be optimized exactly, it matters whether one can further improve upon the BP approximations. Dashed lines in the first plot in Fig. 5.7 show the result obtained when initializing the Lazy Flipper with the BP approximation at $t = 100$s. This reduces the deviation from the global optimum at $t = 50000$ s from 0.4% on average over all models to 0.1%.

5.7 Parameters and Model Decomposition

In all experiments, the damping parameters for BP and TRBP are chosen optimally from the set $\{0, 0.1, 0.2, \ldots, 0.9\}$. The step size of the sub-

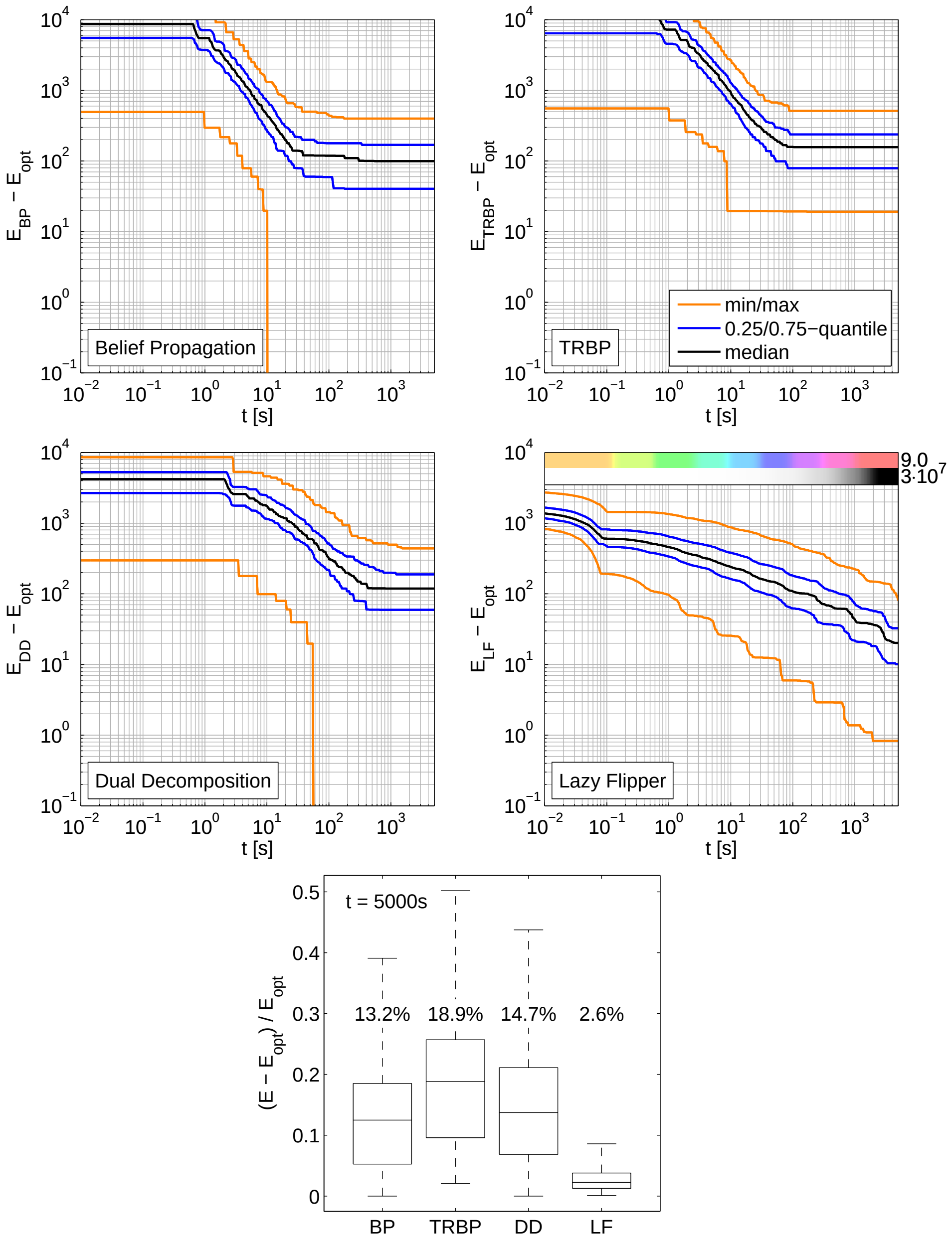


Figure 5.6: Approximate solutions to the problem of removing excessive boundaries from over-segmentations of natural images. The search depth of the Lazy Flipper, averaged over all models in the ensemble, ranges from 1 (orange) to 9 (red). At $t = 5000$ s, $3 \cdot 10^7$ subsets are stored in the CS-tree.

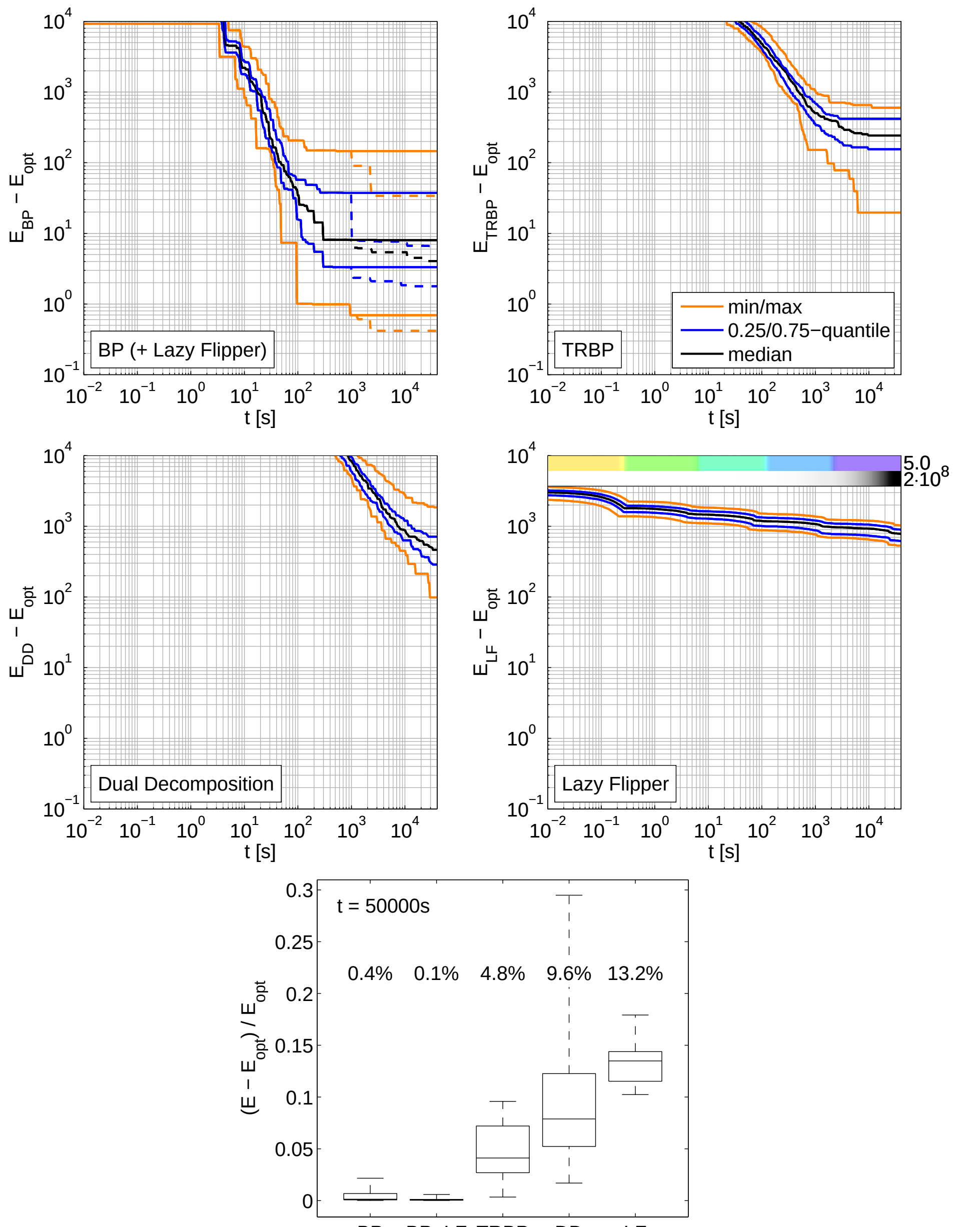


Figure 5.7: Approximate solutions to the problem of removing excessive boundaries from over-segmentations of 3-dimensional volume images. The search depth of the Lazy Flipper, averaged over all models in the ensemble, ranges from 1 (yellow) to 5 (purple). After 50000 s, $2 \cdot 10^8$ subsets are stored in the CS-tree. Dashed lines in the first plot show the result obtained when initializing the Lazy Flipper with the BP approximation at $t = 100$s. This reduces the deviation from the global optimum at $t = 50000$ s from 0.4% on average over all models to 0.1%.

gradient descent is chosen according to

$$\tau_t = \alpha \frac{1}{1 + \beta t} \tag{5.6}$$

where $\beta = 0.01$ and α is chosen optimally from $\{0.01, 0.025, 0.05, 0.1, 0.25, 0.5\}$. The sequence of step sizes, in particular the function (5.6) and β could be tuned further. Moreover, (Komodakis et al., 2010) consider the primal-dual gap and (Kappes et al., 2010) smooth the sub-gradient over iterations in order to suppress oscillations. These measures can have substantial impact on the convergence.

The upper bounds obtained by BP, TRBP and DD do not decrease monotonously. After each iteration of these algorithms, we therefore consider the elapsed runtime and the current best bound, i.e. the best bound of the current and all preceding iterations. All five algorithms are implemented in C++, using the same optimized data structures for the graphical model and a visitor design pattern that allows us to measure runtime without significantly affecting performance.

The same decomposition of each graphical model into tree models is used for TRBP and DD. Tree models are constructed in a greedy fashion, each comprising as many potential functions as possible. The procedure is generally applicable to irregular models with higher-order potentials: Initially, all potentials of the graphical model are put on a **white list** that contains those potentials that have not been added to any tree model. A **black list** of already added potentials and a **gray list** of recently added potentials are initially empty. As long as there are potentials on the white list, new tree models are constructed. For each newly constructed tree model, the procedure iterates over the white list, adding potentials to the tree model if they do not introduce loops. Added potentials are moved from the white list to the gray list. After all potentials from the white list have been processed, potentials from the black list that do not introduce loops are added to the tree model. The gray list is then appended to the black list and cleared. The procedure finishes when the white list is empty. As recently shown by Kappes et al. (2010), decompositions into cyclic subproblems can lead to significantly tighter relaxations and better integer solutions.

5.8 Conclusion

The optimum of a function of binary variables that decomposes according to a graphical model can be found by an exhaustive search over only the connected subgraphs of the model. We implemented this search, using a CS-tree to efficiently and uniquely enumerate the subgraphs. Our algorithm is guaranteed to converge to a global minimum when searching through all subgraphs which is typically intractable. With limited runtime, approximations can be found by restricting the search to subgraphs of a given maximum size. Simulated and real-world problems exist for which these approximations compare favorably to those obtained by message passing and sub-gradient descent. For large scale problems, the applicability of the Lazy Flipper is limited by the memory required for the CS-tree. However, for regular graphs, this limit can be overcome by an implicit representation of the CS-tree that is subject of future research.

6 Representation of Multi-dimensional Functions and Data

6.1 Synopsis

Multi-dimensional arrays are the fundamental data structure used in Chapter 4 to store labelings of the voxel grid and the topological grid and in Chapter 5 to store the value tables of multi-variate potential functions. In C++, excellent template libraries exist for arrays whose dimension is fixed at runtime. Arrays whose dimension can change at runtime have been implemented in C. However, a generic object-oriented C++ implementation of runtime-flexible arrays has so far been missing. In this chapter, a new implementation called Marray is presented, a package of class templates that fills this gap. Marray is based on views as an underlying concept which brings some of the flexibility known from high-level languages such as R and MATLAB® to C++. It is used throughout the implementation of the algorithms of Chapters 4 and 5.

6.2 Introduction and Related Work

A d-dimensional array is a data structure in which each data item can be addressed by a d-tuple of non-negative integers called coordinates. Addressing data by coordinates is useful in many practical applications. As an example, consider a digital image of 1920x1080 pixels. In this image, each pixel can either be identified by the memory address where the associated color is stored or, more intuitively, by a pair of coordinates $(y, x) \in \{0, \ldots, 1919\} \times \{0, \ldots, 1079\}$. Closely related to multi-dimensional arrays are multi-dimensional views. While arrays are the storage containers for multi-dimensional data, views are interfaces that allow the programmer to access data as if it was stored in an array. In the above example, views can be used to treat any sub-image as if it was stored in a separate array. Scientific programming environments such as R[1] (Ligges, 2008) and MATLAB®[2] exploit the versatility of views.

[1]http://www.r-project.org
[2]http://www.mathworks.com/products/matlab

Some of the best implementations of arrays whose dimension is fixed at runtime are written in C++, among these are the Boost Multidimensional Array Library (Garcia and Lumsdaine, 2005), Blitz++ (Veldhuizen, 1998), and MultiArray of the image processing library Vigra[3] (Köthe, 2000). All three packages implement a common interface for views and arrays. Boost in addition allows the programmer to treat arrays as a hierarchy of nested containers. In a hierarchy of nested containers, an $(n+1)$-dimensional array is a container for n-dimensional arrays that have the same size. In this hierarchy, 1-dimensional arrays differ from all other arrays in that they are containers of array **entries** that need not be arrays themselves. This distinction is realized in all three implementations by means of template specialization with respect to the dimension of an array, an approach that achieves great runtime performance and compatibility with the simple multi-dimensional arrays that are native to C. However, template specialization also means that the data type of an array depends on its dimension. Thus, the hierarchy of nested containers does not generalize well in C++ to arrays whose dimension is known only at runtime. This chapter therefore presents an implementation that is based exclusively on views. Little is lost because the hierarchy of nested containers can still be implemented as a cascade of views.

Many practical applications do not require runtime-flexibility because the dimensions of all arrays are either known to the programmer or restricted to a small number of possibilities that can be dealt with explicitly. However, the range of applications where the dimension of arrays is not known a priori and can change at runtime, possibly depending on the user input, is significant. In particular, these are applications that deal with multi-variate data (Chapters 3 and 4) and/or multi-variate functions of discrete variables (Chapter 5). It is no surprise that the runtime-flexible arrays of R and MATLAB® have proven useful in these settings.

In this chapter, Section 6.3 summarizes the mathematics of runtime-flexible multi-dimensional views and arrays. It is a concise compilation of existing ideas from excellent research articles (Garcia and Lumsdaine, 2005; Veldhuizen, 1998) and text books, e.g. (Brass, 2008). Section 6.4 deals with the C++ implementation of the mathematical concepts and provides some examples that show how the classes can be used in practice. Readers who prefer a practical introduction are encouraged to read Section 6.4 first. Section 6.4.4 discusses already implemented extensions based on the C++0x standard proposal (Stroustrup, 2006). Section 6.5 concludes the chapter.

[3] http://hci.iwr.uni-heidelberg.de/vigra

6.3 The Mathematics of Views and Arrays

6.3.1 Views

Views provide an interface to access data as if it was stored in an array. A few definitions are sufficient to describe the properties (syntax), function (semantics), and transformation of views. These definitions are dealt with in this section. As they are implemented one-to-one in the Marray classes, this section also explains in detail how these classes work internally.

Definition 14 (View). A non-degenerate multi-dimensional view is a quadruple $(d, s, t, p_0) \in \mathbb{N} \times \mathbb{N}^d \times \mathbb{N}^d \times \mathbb{N}$ in which d is called the **dimension**, s the **shape**, t the **strides**, and p_0 the **offset** of the view. A tuple $(0, \emptyset, \emptyset, p_0)$ is called a **degenerate/scalar/0-dimensional** view.

Views allow the programmer to address data by tuples of d positive integers called coordinates. These coordinates are taken from ranges of values that are determined by the view's shape:

Definition 15 (Coordinates). Given a view $V = (d, s, t, p_0)$,

$$C_V := \begin{cases} \{0, \dots, s_0 - 1\} \times \dots \times \{0, \dots, s_{d-1} - 1\} & \text{if } d \neq 0 \\ \emptyset \text{ otherwise} \end{cases} \tag{6.1}$$

is called the set of coordinate tuples of V.

According to this definition, coordinates start from 0 as is the standard for C++, and not from 1 as in many high-level languages. Which data item is addressed by a coordinate tuple $(c_0, \dots, c_{d-1}) \in C_V$ is determined by the addressing function of the view. This function is parameterized by the view's strides and offset:

Definition 16 (Addressing Function). Given a view $V = (d, s, t, p_0)$ with $d \neq 0$, the function $a_V : C_V \to \mathbb{N}_0$ with

$$\forall c \in C_V : \quad a_V(c) = p_0 + \sum_{j=0}^{d-1} t_j c_j \tag{6.2}$$

is called the **addressing function** of V.

Semantically, a coordinate tuple $c = (c_0, \dots, c_{d-1})$ identifies the data item that is stored at the address $a_V(c)$ in memory. Here are some examples: Assume that the integers $1, \dots, 6$ are stored consecutively in memory at the addresses $100, \dots, 105$. The six views in Tab. 6.1 address this memory and are written down next to the table in matrix notation, i.e. as tables in which the entry at row j and column k corresponds to the integer addressed by the coordinate (j, k):

Table 6.1: Multi-dimensional views on the same data can differ in dimension, shape, strides, and offset.

View	Dim d	Shape s	Strides t	Offset p_0
V_1	2	$(3,2)$	$(1,3)$	100
V_2	2	$(3,2)$	$(2,1)$	100
V_3	2	$(2,3)$	$(1,2)$	100
V_4	2	$(2,3)$	$(3,1)$	100
V_5	2	$(2,2)$	$(3,1)$	101
V_6	1	(3)	(2)	101

$$V_1 : \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} \qquad V_2 : \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad V_3 : \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

$$V_4 : \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \qquad V_5 : \begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix} \qquad V_6 : (2,4,6)$$

The views $V_1, \ldots, V_4$ address the same set of integers but in a different shape and with different addressing functions. Perhaps more interestingly, V_5 is a sub-view of V_4 that has the same dimension but a different shape, and V_6 is a sub-view of V_3 whose dimension has been reduced. In general, sub-views can be defined as follows:

Definition 17 (Sub-View). Given a view $V = (d, s, t, p_0)$ with $d \neq 0$, a start coordinate $c \in C_V$, and a shape $s' \in \mathbb{N}^d$ such that $\forall j \in \{0, \ldots, d-1\} : c_j + s'_j \leq s_j$,

$$\text{sub-view}(V, c, s') := (d, s', t, p_0 + a_V(c)) \tag{6.3}$$

is called the **sub-view of V with the shape s', starting at the coordinate c.**

The convenient access to sub-views is one of the main reasons why multi-dimensional views are useful in practice.

As important as the construction of sub-views is the binding of coordinates. If one coordinate in a d-dimensional view is bound to a value, the result is a $(d-1)$-dimensional view. In the above example, V_6 arises from V_3 by binding coordinate 0 to the value 1. In general, coordinate binding works as follows:

Definition 18 (Coordinate Binding). Given a view $V = (d, s, t, p_0)$ with $d \neq 0$, a dimension $j \in \{0, \dots, d-1\}$ and a value $x \in \{0, \dots, s_j - 1\}$,

$$\mathrm{bind}(V, j, x) := (d-1, s', t', p'_0) \tag{6.4}$$

with $s' = (s_0, \dots, s_{j-1}, s_{j+1}, \dots, s_{d-1})$, $t' = (t_0, \dots, t_{j-1}, t_{j+1}, \dots, t_{d-1})$ and $p'_0 = a_V(c)$ with $c \in C_V$ such that $\forall k \in \{0, \dots, d-1\} : c_k = x\delta_{jk}$ is said to arise from V by **binding** coordinate j to the value x.

By Def. 17, sub-view(V, c, s') has the same dimension as V. However, the shape of the sub-view may be equal to one in some dimensions, i.e. $s'_j = 1$ for some j. Since 0 is the only admissible coordinate in these singleton dimensions, it makes sense to bind such coordinates to 0. Binding the coordinates in all singleton dimensions to 0 is called **squeezing**.

An operation that preserves both the dimension and the memory addressed by a view is permutation. Permuting a view permutes the view's shape and strides, respectively:

Definition 19 (Permutation). The **permutation** of a non-degenerate view $V = (d, s, t, p_0)$ w.r.t. a bijection $\sigma : \{0, \dots, d-1\} \to \{0, \dots, d-1\}$ is the view

$$\mathrm{permute}(V, \sigma) := (d, s', t', p_0) \tag{6.5}$$

where $s', t' \in \mathbb{N}^d$ and $\forall j \in \{0, \dots, d-1\} : s'_j = s_{\sigma(j)} \wedge t'_j = t_{\sigma(j)}$.

Two special cases of permutations are **transpositions** and **cyclic shifts**. Transpositions exchange the shape and strides in only two dimensions. In the above example, V_1 and V_4 are transposes of each other, and so are V_2 and V_3. Cyclic shifts permute a view in a cyclic fashion. As an example, consider a 3-dimensional view whose shape is $(2, 3, 7)$. If this view is shifted by 1, the resulting view has the shape $(7, 2, 3)$, and a shift by -1 yields a view having the shape $(3, 7, 2)$. In general, cyclic shifts can be defined and computed as follows:

Definition 20 (Cyclic Shift). The cyclic shift of a non-degenerate view $V = (d, s, t, p_0)$ w.r.t. $z \in \mathbb{Z}$ is the view

$$\mathrm{shift}(V, z) := \begin{cases} \mathrm{shift}(V, z \bmod d) & \text{if } d \leq |z| \\ \mathrm{shift}(V, z - d) & \text{if } 0 < z < d \\ (d, s', t', p_0) & \text{otherwise} \end{cases} \tag{6.6}$$

with $s', t' \in \mathbb{N}^d$ and $\forall j \in \{0, \dots, d-1\} : s'_j = s_{(j-z) \bmod d} \wedge t'_j = t_{(j-z) \bmod d}$.

6.3.2 Scalar Indexing and Iterators

The coordinate tuples of a view can be put in some order. Imposing such an order allows the programmer to access any data item under the view by a single index, namely the index of the associated coordinate tuple in the given order. This is useful in practice because it in turn allows us to handle sub-views as if they were single-indexed containers holding a subset of data. Moreover, it facilitates the definition of iterators (Austern, 1998) on views.

Among all possible orders that can be imposed on coordinate tuples, two are most commonly used[4]. In the First Coordinate Major Order (FCMO), the first coordinate is used as the strongest ordering criterion, meaning that one tuple is greater than all tuples whose first coordinate is smaller. Coordinates at higher dimensions are used for ordering only if all coordinates at lower dimensions are equal. In the Last Coordinate Major Order (LCMO), the last coordinate is the strongest ordering criterion. In the special case of 2-dimensional views, FCMO and LCMO are called **row-major order** and **column-major order**, respectively. These terms refer to the matrix notation of data under 2-dimensional views. FCMO is used in native C arrays whereas LCMO is used in Fortran and MATLAB. Both orders are defined implicitly by a function that maps coordinate tuples to unique integer indices. One coordinate is smaller than another precisely if the associated index is smaller.

Definition 21 (Indexing). Given a view $V = (d, s, t, p_0)$ with $d \neq 0$ and a coordinate $c = (c_0, \ldots, c_{d-1}) \in C_V$,

$$\mathrm{fcmo}(c) := \sum_{j=0}^{d-1} u_j c_j \quad \text{with} \quad u_j = \prod_{k=j+1}^{d-1} s_k \ , \tag{6.7}$$

$$\mathrm{lcmo}(c) := \sum_{j=0}^{d-1} u_j c_j \quad \text{with} \quad u_j = \prod_{k=0}^{j-1} s_k \ . \tag{6.8}$$

are called the FCMO- and LCMO-**index** of c, respectively. Given that either FCMO or LCMO is used, $u_0, \ldots, u_{d-1}$ are called the **shape strides** of V.

As an example, consider a 3-dimensional view $V = (d, s, t, p_0)$. Herein, the indices that correspond to a given coordinate $c \in C_V$ are computed

[4]Note, however, that more complex orders can be obtained by defining views with specific strides.

according to

$$\begin{aligned}\text{fcmo}(c) &= s_1 s_2 c_0 + s_2 c_1 + c_2 \ , \\ \text{lcmo}(c) &= c_0 + s_0 c_1 + s_0 s_1 c_2 \ .\end{aligned}$$

The index that corresponds to a coordinate tuple can be computed according to Def. 21. Conversely, the coordinates that correspond to a given FCMO- or LCMO-index are computed by means of Alg. 7. Given that either FCMO or LCMO is used, it can happen that the strides are equal to the shape strides of a view. Such views are called **unstrided**. In an unstrided view $V = (d, s, t, p_0)$, the address that corresponds to an index $x \in \mathbb{N}_0$ is simply $x + p_0$, whereas in a strided view, one needs to compute first the coordinate c that corresponds to the index x (Alg. 7) and then the address $a_V(c)$ (Def. 16).

Algorithm 6: IndexToCoordinates

```
Input: $x \in \mathbb{N}_0$ (index), $(u_0, \dots, u_{d-1}) \in \mathbb{N}^d$ (shape strides)
Output: $(c_0, \dots, c_{d-1}) \in \mathbb{N}^d$ (coordinates)
1  if $u_0 = 1$ then
2      // LCMO
3      for j = d-1 to 0 do
4          $c_j \leftarrow \lfloor x/u_j \rfloor$;
5          $x \leftarrow x \bmod u_j$;
6      end
7  else
8      // FCMO
9      for j = 0 to d-1 do
10         $c_j \leftarrow \lfloor x/u_j \rfloor$;
11         $x \leftarrow x \bmod u_j$;
12     end
13 end
```

In summary, we have seen that views are powerful interfaces to address data either by coordinates or by single indices. It is simple to obtain subviews and to bind and permute coordinates.

6.3.3 Arrays

A multi-dimensional array is a data structure whose interface is a view. While views only reference data via their addressing function, arrays contain data. In the following definition, the memory is modeled as a function μ that maps addresses to memory content.

Definition 22 (Array). A d-dimensional array is a tuple (V, q, μ) such that $V = (d, s, t, p_0)$ is a view, $q \in \{\text{FCMO, LCMO}\}$, V is unstrided w.r.t. q, and μ is a function

$$\mu : \left\{ p_0, \ \ldots, \ p_0 + \left(\prod_{j=0}^{d-1} s_j \right) - 1 \right\} \to \mathbb{N} \ . \tag{6.9}$$

For each $c \in C_V$, $\mu(a_V(c))$ is called the **entry** of the array at position c. Moreover, $|C_V|$ is termed the array's **size**.

Two transformations are defined on arrays, namely reshaping and resizing. Reshaping can change the dimension and shape of an array while preserving its size and entries.

Definition 23 (Reshaping). Given an array $A = ((d, s, t, p_0), q, \mu)$ as well as $d' \in \mathbb{N}$, and $s' = (s_0, \ldots, s_{d'-1})$ such that $\prod_{j=0}^{d'-1} s'_j = \prod_{j=0}^{d-1} s_j$, the **reshaping** of A w.r.t. s' is the array

$$\text{reshape}(A, s') := ((d, s', t', p_0), q, \mu) \tag{6.10}$$

in which (d, s', t', p_0) is a view that is unstrided w.r.t. q.

In fact, reshaping can not only be defined for arrays but also, more generally, for unstrided views.

In contrast to reshaping, resizing can change the size and hence the interval of memory of an array:

Definition 24 (Resizing). Given an array $A = (V, q, \mu)$, a new dimension $d' \in \mathbb{N}$ and a new shape $s' = (s_0, \ldots, s_{d'-1})$, an array (V', q, μ') is called a resizing of A w.r.t. s', denoted resize(A, s'), if and only if the following conditions hold:

(i) $V' = (d', s', t', p'_0)$ is a view that is unstrided w.r.t. q. (Note that the offset p'_0 of the new array can differ from that of V due to a possible re-allocation of memory).

(ii) entries of A are preserved according to the following rule:

$$\forall (c, c') \in D : \quad \mu(a_V(c)) = \mu'(a_{V'}(c')) \tag{6.11}$$

with

$$\begin{aligned} D = \{(c, c') \in C_V \times C'_V \mid \forall j \in \{0, \ldots, \min(d, d') - 1\} : c_j = c'_j \\ \wedge \forall j \in \{\min(d, d'), \ldots, d - 1\} : c_j = 0 \\ \wedge \forall j \in \{\min(d, d'), \ldots, d' - 1\} : c'_j = 0)\} \end{aligned}$$

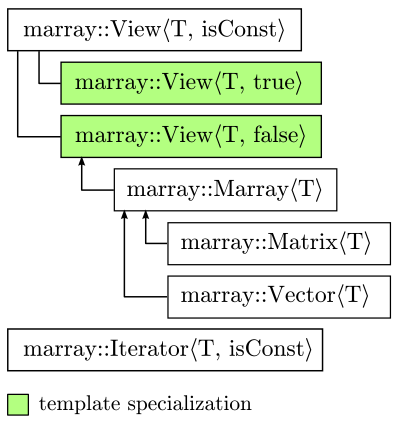


Figure 6.1: Class template hierarchy of the Marray package. The five major class templates are View, Marray, Matrix, Vector, and Iterator. The Boolean template parameter **isConst** is used to determine whether the data addressed by views and iterators is constant or mutable.

Finally, all transformations of views can be used similarly with arrays.

6.4 Implementation

The definitions introduced above are implemented in C++ in the Marray package (cf. Chapter 7). Marray depends only on the C++ Standard Template Library (STL) (Austern, 1998). The single header file `marray.hxx` is sufficient to use the package. This header file contains the source code as well as reference documentation in the doxygen format[5]. In addition to this file, we provide unit tests (Hamill, 2004) in the file `tests.cxx` as well as the reference documentation in HTML.

Five major class templates are defined in the namespace `marray`. These are `View`, `Marray`, `Matrix`, `Vector`, and `Iterator`. Their organization is depicted in Fig. 6.1. The Boolean template parameter `isConst` is used to determine whether the data addressed by views and iterators is constant or mutable. This facilitates a unified implementation for both cases without any redundancy in the code (cf. Meyers, 2005). The class templates `Marray`, `Matrix`, and `Vector` inherit the interface from `View<T, false>`.

[5]`http://www.doxygen.org/`

6.4.1 Using Arrays

The simplest means to construct an array is a pair of iterators that point to the beginning and the end of a sequence that determines the array's shape:

```
size_t shape[] = {3, 2, 4};
marray::Marray<float> a(shape, shape+3);
```

Constructing matrices and vectors is even simpler and works as most programmers will expect, namely by providing the size and the number of rows and columns, respectively:

```
marray::Vector<float> v(42);
marray::Matrix<float> m(7, 8);
```

In addition to the shape, one can specify an initial value for all array entries as well as the order in which entries are stored, e.g.

```
marray::Marray<float> b(shape, shape+3, 1.0f,
   marray::FirstMajorOrder);
```

By default, all entries of a `marray::Marray<T>` are initialized with `T()` and are stored in Last Coordinate Major Order, cf. Section 6.3. Depending on the application, the initialization of array entries is sometimes unnecessary and can thus be skipped to improve performance. Initialization skipping works as follows:

```
size_t shape[] = {3, 2, 4};
marray::Marray<float> a(marray::SkipInitialization,
   shape, shape+3);
marray::Vector<float> v(marray::SkipInitialization, 42);
marray::Matrix<float> m(marray::SkipInitialization, 7, 8);
```

After construction, the dimension, size, shape, and storage order of an array can be obtained as follows.

```
unsigned short dimension = a.dimension();
size_t size = a.size();
bool firstMajorOrder = a.firstMajorOrder();
marray::Vector<size_t> shape(dimension);
for(size_t j=0; j<dimension; ++j)
   shape[j] = a.shape(j);
```

The entries of an array can be accessed in three different ways: by coordinates, by single indices, and by means of STL compliant random access iterators (cf. Austern, 1998). In fact, the following four assignments have the same effect on the array `a`.

```
// 1.
a(1, 0, 2) = 4.2f;
// 2.
size_t pos[] = {1, 0, 2};
a(pos) = 4.2f;
// 3.
a(13) = 4.2f;
// 4.
marray::Marray<float>::iterator it = a.begin();
it[13] = 4.2f;
```

It can sometimes be useful to print the entries of an array to `std::cout`. This can be done using

```
std::cout << a.asString(marray::TableStyle);
std::cout << a.asString(marray::MatrixStyle);
std::cout << a.asString(); // MatrixStyle is the default
```

In table style output, each printed row consists of a coordinate tuple and the corresponding array entry. In the more compact matrix notation only the entries of the array are printed.

Both the shape and the size of an array can be changed at runtime. Reshaping modifies an array's shape and dimension while preserving its size. Resizing can in addition cause the amount of memory allocated by the array to grow or shrink.

```
size_t newShape[] = {2, 2, 3, 2};
a.reshape(newShape, newShape+4);
newShape[0] = 4;
a.resize(newShape, newShape+4);
```

The function `resize` can alternatively be called with a third parameter that specifies the initial value for newly allocated entries. For matrices and vectors, reshaping and resizing works as follows:

```
v.resize(56);
m.reshape(8, 7);
m.resize(2, 4);
```

It can sometimes be useful to permute the dimensions of an array, e.g. to transpose a matrix. Three functions, `permute`, `transpose`, and `shift` serve this purpose. While `permute` deals with the most general case of permuting dimensions in any desired way, `transpose` with two parameters swaps any two dimensions, `transpose` with no parameters reverses the order of dimensions, and `shift` shifts them in a cyclic fashion. No matter which function is used, only the array's interface is adjusted; no data is moved or copied.

```
size_t shape[] = {3, 2, 4};
marray::Marray<float> c(shape, shape+3);
size_t permutation[] = {1, 0, 2};
c.permute(permutation); // (2, 3, 4)
c.transpose(0, 2); // (4, 3, 2)
c.shift(-1); // (3, 2, 4)
c.shift(2); // (2, 4, 3)
c.transpose(); // (3, 4, 2)
```

Finally, the arithmetic operators `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=` are defined. They operate on an array and its entry data type (in any order), as well as on pairs of arrays that have the same shape. In the latter case, the operation is performed on each pair of entries, for every coordinate. In summary, this allows the programmer to use arithmetic expressions like these:

```
marray::Marray<float> d;
d = -a + 0.5f*a - 0.25f*a*a;
d = 1.0f / (1.0f + a*a);
d = (a /= 2.0f);
--a;
```

6.4.2 Using Views

Arrays, including matrices and vectors, are containers. Views are interfaces that allow the programmer to access data as if it was stored in an array. A view can be constructed either as a sub-view of another view or array, or directly on an interval of memory. In the following example, a 2-dimensional sub-view is constructed that ranges from position $(3, 2, 4)$ to position $(7, 2, 8)$ in a 3-dimensional array.

```
size_t shape[] = {20, 20, 20};
marray::Marray<float> d(shape, shape+3);
```

```
size_t base[] = {3, 2, 4};
size_t subShape[] = {5, 1, 5};
marray::View<float> v = d.view(base, subShape);
v.squeeze(); // collapse singleton dimension
```

Each view defines an internal order of coordinates, either First or Last Coordinate Major Order. This order determines how an iterator traverses the view as well as how single indices are mapped to coordinates, e.g. which entry of `d` in the above example is referenced by, say, `v(7)`. The coordinate order of a sub-view need not be the same as the coordinate order of the view based on which it is constructed, although this is the default. Instead, it is possible to specify the coordinate order of sub-views explicitly, e.g.

```
marray::View<float> v =
   d.view(base, subShape, marray::FirstMajorOrder);
```

This facilitates the construction of sub-views that behave exactly like the views or arrays on which they are based, except that the coordinate order is reverted:

```
marray::Vector<size_t> base(d.dimension());
marray::Vector<size_t> subShape(d.dimension());
for(size_t j=0; j<d.dimension(); ++j)
   subShape(j) = d.shape(j);
marray::View<float> v = d.view(base.begin(), subShape.begin(),
   marray::FirstMajorOrder);
```

Views can be constructed directly on an interval of memory. If all data in this interval is to be referenced by the view, i.e. if the view is to be unstrided (cf. Section 6.3), it is sufficient to provide the view's shape and a pointer to the beginning of the data.

```
float data[24];
size_t shape[] = {3, 2, 4};
marray::View<float> w(shape, shape+3, data);
```

The same constructor can be used with two additional parameters,

```
marray::View<float> w(shape, shape+3, data,
   marray::LastMajorOrder, marray::FirstMajorOrder);
```

These parameters specify the external coordinate order based on which the strides of the view are computed as well as the internal coordinate order that is used for indexing and iterators. By default, Last Coordinate Major Order is used for both. Views on constant data are constructed similar to views on mutable data, e.g.

```
marray::View<float, marray::Const> w(shape, shape+3, data);
```

Constructing unstrided views is only the simplest case. In general, the strides as well as the offset of a view (cf. Section 6.3) can be set explicitly, e.g.

```
size_t shape[] = {3, 2, 4};
size_t strides[] = {2, 1, 6};
size_t offset = 0;
marray::View<float> w(shape, shape+3, strides, data, offset,
   marray::FirstMajorOrder);
```

The data under a view is accessed similar to the entries of arrays, i.e. by coordinates, by single indices, or by means of iterators. Coordinate permutation works on views exactly the same way it works on arrays. A subview where one coordinates is bound to a certain value can be obtained as follows:

```
marray::View<float> x = w.boundView(2, 1);
// binds dimension 2 to coordinate 1
```

The member functions `reshape`, `permute`, `transpose`, `shift`, and `squeeze` transform the view for which they are called. They are complemented by member functions called `reshapedView`, `permutedView`, etc. that leave the view for which they are called unchanged and return a new view that is transformed in the desired way. The latter functions are first of all convenient but they also resemble the way transformations are implemented in Boost for views whose dimension is fixed at runtime. In fact, all operations that change the dimension of a view need to be implemented in this way if the dimension of the view is a template parameter because the data type changes together with the dimension.

All arithmetic operators are defined on views. Assigning a view `x` to a view on mutable data `y` via `y = x` copies the data under `x` to the memory addressed by `y`, provided that `x` and `y` have the same shape. The copy is performed per coordinate, not per scalar index or iterator. Potential memory overlaps between the two views `x` and `y` are taken care of. Data is copied if necessary, in an assignment `y = x`, as well as in in-place operations such as `x += y`. Assigning a view `x` to a view on constant data `z` copies the view, not the data. This is useful to recycle the memory allocated for a view on constant data.

In summary, the views, arrays, matrices, and vectors provided in the Marray package behave exactly like STL containers (Austern, 1998) in terms of their fundamental interface. Additional functions going beyond the interface of STL containers allow the programmer to adjust the dimension, shape, strides, as well as the storage order at runtime.

6.4.3 Invariants

For the sake of runtime performance, some redundancy is built into the view classes. In particular, the size and the shape strides of views are stored explicitly as attributes although they could be computed on demand from the shape and the internal order of coordinates. An additional Boolean flag indicates whether a view is unstrided and has a zero offset. This flag supports the fast copying of data via `memcpy`, provided that views do not overlap. In case of overlap, the necessary temporary copy is created internally.

The redundant attributes need to be kept consistent under all possible transformations of views and arrays. The private member functions

```
testInvariant()
```

check for consistency. They are called after any transformation in debug mode. The reader is encouraged to look these functions up in the source code. Since views and arrays are fundamental data structures that should work at peak performance in released code, it is important that all tests can be removed. A function proposed by Stroustrup (2000) is used to meet this requirement.

```
template<class A> inline void Assert(A assertion) {
   if(!assertion)
      throw std::runtime_error("Assertion failed.");
}
```

Along with this function, the Boolean constants `NO_DEBUG` and `NO_ARG_TEST` are defined in the namespace `marray`. Invariant testing and the testing of function arguments is conditioned on these variables, e.g.

```
Assert(NO_DEBUG || this->dimension_ > 0);
Assert(NO_ARG_TEST || std::distance(begin, end) != 0);
```

In consequence, compilers will remove the respective tests if NO_DEBUG and NO_ARG_TEST are set to true. By default, both variables are set in accordance with NDEBUG.

6.4.4 C++0x Extensions

Features of the C++0x standard proposal (Stroustrup, 2006) facilitate three highly desirable extensions whose implementation in C++98 would have drawbacks. The C++0x code is part of the Marray package. However, since C++0x is not yet approved, these extensions are considered experimental and have to be enabled explicitly by defining the variables

```
HAVE_CPP0X_TEMPLATE_TYPEDEFS
HAVE_CPP0X_VARIADIC_TEMPLATES
HAVE_CPP0X_INITIALIZER_LISTS
```

6.4.4.1 Template Aliases

Views are declared as class templates in the namespace marray:

```
template<class T, bool isConst = false> class View;
```

To support the writing of self-explanatory code, the constants Const = true and Mutable = false are defined. Still, having to write

```
marray::View<float, marray::Const> v;
```

to declare a view on constant data is perhaps not what a programmer would guess. We could have implemented a class template ConstView separately. However, even with inheritance, this would have led to excessive redundancy in the code that would have made the implementation error prone and hard to maintain (Sutter and Alexandrescu, 2004). C++0x provides an elegant solution, namely the definition of the template alias (Reis and Stroustrup, 2007)

```
template<class T> using ConstView = View<T, true>;
```

This alias allows the programmer to construct a view on constant data in a straightforward way:

```
marray::ConstView<float> v;
```

6.4.4.2 Variadic Templates

The entries of views and arrays can be accessed by coordinates. For the sake of convenience, it should be possible for the programmer to use `operator()` with any number of parameters.

C++98 has inherited from C a syntax for functions whose number of parameters is unspecified at compile time. However, this mechanism is not type safe (Stroustrup, 2000) and its use is therefore discouraged. In the C++98 compatible part of the code, we thus make a compromise and implement the operator in a type safe manner for up to four parameters. A runtime error is issued if the wrong instance is used. Beyond four dimensions, `operator()` can be used with one argument, an iterator to a coordinate sequence.

C++0x defines variadic templates (Gregor et al., 2006, 2007) that allow us to recursively define `operator()` in a type safe manner for any number of parameters. We quote here the main recursive declaration and refer to the source code for details.

```
template<typename... Args>
   reference_type operator()(const size_t &&,
      const Args && ...);
reference_type operator()(const size_t &&);
```

6.4.4.3 Initializer Lists

Constructors and member functions of Marray classes take iterators into coordinate sequences as input. One iterator that points to the beginning of the sequence is sufficient if the length of the sequence can be derived, e.g. in the member function `permute` of `View`. Iterator pairs are required otherwise, e.g. in the member function `resize` of `Marray`. Iterators are used extensively in the STL, so most programmers will find them familiar. However, the use of iterators and iterator pairs is cumbersome if sequences are known at compile time. In fact, neither of the following alternatives is really convenient:

```
size_t shape[] = {4, 2, 3};
marray::Marray<float> a(shape, shape+3);

std::vector<size_t> shape(3);
shape[0] = 4;
shape[1] = 2;
```

```
shape[2] = 3;
marray::Marray<float> a(shape.begin(), shape.end());
```

C++0x defines initializer lists (Stroustrup and Reis, 2007) that allow us to overload functions such that the programmer can simply write

```
marray::Marray<float> a({4, 2, 3});
```

6.5 Conclusion

C++ class templates are provided for multi-dimensional views and arrays whose dimension, shape, and size can change at runtime. The C++98 interface of these templates is as convenient as in the best implementations of arrays with fixed dimensions. Usability is further improved by C++0x extensions.

7 Conclusion

This thesis has introduced a new graphical model for segmenting volume images based on an initial over-segmentation of the volume into supervoxels (Chapter 3). The problem which supervoxels to merge in order to arrive at the correct segmentation has been formulated as an energy minimization problem on binary variables that are associated with faces between supervoxels and indicate whether these faces should be preserved or removed. This new formulation constitutes substantial progress over models that cast the segmentation problem into a supervoxel labeling problem (Section 3.3) because it has the same expressiveness but an exponentially smaller and less degenerate state space. Non-binary variables and the arbitrariness as to which label to assign to which segment are avoided altogether.

Chapter 4 has presented a new algorithm that encodes the geometry and topology of volume segmentations explicitly, in a designated data structure. This algorithm has facilitated the extraction of features from supervoxel segmentations of volume images based on which the potential functions of the graphical model could be learned from a small amount of empirical training data. The new algorithm operates on the volume image in a block-wise fashion and in parallel, thus limiting memory consumption to a prescribed amount while keeping runtime linear in the number of voxels and log-linear in the number of faces and curves. This is of fundamental importance for the analysis of large volume images because it makes computational geometry applicable in practice in the gigavoxel regime (Tab. 3.6).

The problem of finding a segmentation with minimal energy for a learned graphical model has been solved to optimality for problems with up to 10^5 variables (Chapter 5). This result is important because superpixel segmentation of 2-dimensional natural images have exactly this complexity (Section 5.6.3). For models that are too large to be optimized exactly (the full model in Chapter 3 has 10^7 variables, and 5 GB are required to store only the values of its potential functions), tight approximate solutions have been found by means of a new constrained search algorithm (the Lazy Flipper) introduced in Chapter 5. The Lazy Flipper is the first generalization of Iterated Conditional Modes (ICM) for binary variables that handles a search depth greater than one efficiently in graphical models with arbitrary and possibly irregular structure.

A practical application of the segmentation model, the geometry extraction and the optimization algorithm to the problem of segmenting volume images of neural tissue acquired by means of Serial Block Face Scanning Electron Microscopy (SBFSEM) (Denk and Horstmann, 2004) for the purpose of neural circuit reconstruction (connectomics) has been shown in Chapter 3. Compared to an automated segmentation approach that does not take the geometry of supervoxels into account (Andres et al., 2008), the reconstruction quality has improved substantially (Fig. 3.11), as measured on a benchmark dataset of $2048 \times 1792 \times 2048$ voxels that shows a part of the inner plexiform layer of rabbit retina, by comparing the final segmentation to the gold standard. The thinnest neuronal processes are still falsely split, but advances in tissue preparation and imaging may alleviate this problem in the future.

The new algorithms and data structures introduced in this work have been put into generally applicable C++ source code and software and are available for download[1]. These are:

- **CGP**, the C++ package for geometry and topology extraction from very large volume segmentations described in Chapter 4
- **OpenGM**, a library for optimization in higher order graphical models with discrete variables (Andres et al., 2010b) that includes an implementation of the Lazy Flipper (Chapter 5) and other state-of-the-art optimization algorithms
- **Marray**, the fundamental data structure for multi-variate functions and data (Chapter 6).

The most important result from this work is that boundary-based segmentation models and computational geometry on supervoxel segmentations are applicable in practice in the gigavoxel regime. Given the rapid development of volume image acquisition techniques, these concepts will become more important and see applications and adaptations beyond the scope of this thesis.

[1] `http://hci.iwr.uni-heidelberg.de/Software/`

Bibliography

S. M. Aji and R. J. McEliece. The generalized distributive law. **Transactions on Information Theory**, 46(2):325–343, 2000. DOI http://dx.doi.org/10.1109/18.825794.

S. Alpert, M. Galun, B. Nadler, and R. Basri. Detecting faint curved edges in noisy images. In K. Daniilidis, P. Maragos, and N. Paragios, editors, **Proceedings of the European Conference on Computer Vision 2010**, volume 6314 of **LNCS**, pages 750–763. Springer, 2010.

B. Andres, U. Köthe, M. Helmstaedter, W. Denk, and F. A. Hamprecht. Segmentation of SBFSEM volume data of neural tissue by hierarchical classification. In G. Rigoll, editor, **Pattern Recognition**, volume 5096 of **LNCS**, pages 142–152. Springer, 2008. DOI http://dx.doi.org/10.1007/978-3-540-69321-5_15.

B. Andres, J. H. Kappes, U. Koethe, and F. A. Hamprecht. The Lazy Flipper: MAP inference in higher-order graphical models by depth-limited exhaustive search. **ArXiv e-prints**, 2010a. URL http://arxiv.org/abs/1009.4102.

B. Andres, J. H. Kappes, U. Koethe, C. Schnörr, and F. A. Hamprecht. An empirical comparison of inference algorithms for graphical models with higher order factors using OpenGM. In **DAGM 2010**, pages 363–372, 2010b.

C. J. Armstrong, B. L. Price, and W. A. Barrett. Interactive segmentation of image volumes with live surface. **Computer Graphics**, 31(2):212–229, 2007. DOI http://dx.doi.org/10.1016/j.cag.2006.11.015.

M. H. Austern. **Generic Programming and the STL: Using and Extending the C++ Standard Template Library**. Addison-Wesley, 1998.

D. Barash. A fundamental relationship between bilateral filtering, adaptive smoothing, and the nonlinear diffusion equation. **Transactions on Pattern Analysis and Machine Intelligence**, 24(6):844–847, 2002. DOI http://dx.doi.org/10.1109/TPAMI.2002.1008390.

A. Barbu and S.-C. Zhu. Graph partition by Swendsen-Wang cuts. In **Proceedings of the 9th International Conference on Computer Vision**, page 320, Washington, DC, USA, 2003. IEEE Computer Society.

M. Bergtholdt, J. Kappes, S. Schmidt, and C. Schnörr. A study of parts-based object class detection using complete graphs. **International Journal of Computer Vision**, 2009. DOI http://dx.doi.org/10.1007/s11263-009-0209-1.

Y. Bertrand, G. Damiand, and C. Fiorio. Topological encoding of 3D segmented images. In G. Borgefors, I. Nyström, and G. Sanniti di Baja, editors, **Proceedings of the DGCI 2000**, LNCS 1953, pages 311–324. Springer, 2000. DOI http://dx.doi.org/10.1007/3-540-44438-6.

J. Besag. On the statisical analysis of dirty pictures. **Journal of the Royal Statistical Society B**, 48:259–302, 1986.

G. Biau, L. Devroye, and G. Lugosi. Consistency of random forests and other averaging classifiers. **Journal of Machiche Learning Research**, 9:2015–2033, 2008. URL jmlr.csail.mit.edu/papers/volume9/biau08a/biau08a.pdf.

J. Bigun. **Vision with Direction: A Systematic Introduction to Image Processing and Computer Vision**. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

C. M. Bishop. **Pattern Recognition and Machine Learning**, chapter 8. Springer, 2nd edition, 2007.

Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. **Transactions on Pattern Analysis and Machine Intelligence**, 23(11):1222–1239, 2001. DOI http://dx.doi.org/10.1109/34.969114.

J.-P. Braquelaire and P. Guitton. 2 1/2d scene update by insertion of contour. **Computers and Graphics**, 15(1):41–48, 1991. DOI http://dx.doi.org/10.1016/0097-8493(91)90029-H.

P. Brass. **Advanced Data Structures**. Cambridge University Press, 2008.

A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. **Random Structures and Algorithms**, 27(2):201–226, 2005. DOI http://dx.doi.org/10.1002/rsa.v27:2.

L. Breiman. Bagging predictors. **Machine Learning**, 24(2):123–140, 1996. DOI http://dx.doi.org/10.1023/A:1018054314350.

L. Breiman. Random forests. **Machine Learning**, 45(1):5–32, 2001. DOI http://dx.doi.org/10.1023/A:1010933404324.

L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. **Classification and Regression Trees**. Chapman and Hall/CRC, 1984.

C. R. Brice and C. L. Fennema. Scene analysis using regions. **Artificial Intelligence**, 1:205–226, 1970. DOI http://dx.doi.org/10.1016/0004-3702(70)90008-1.

K. L. Briggman and W. Denk. Towards neural circuit reconstruction with volume electron microscopy techniques. **Current Opinion in Neurobiology**, 16(5):562–570, 2006. DOI http://dx.doi.org/10.1016/j.conb.2006.08.010.

D. B. Chklovskii, S. Vitaladevuni, and L. K. Scheffer. Semi-automated reconstruction of neural circuits using electron microscopy. **Current Opinion in Neurobiology**, 2010. DOI http://dx.doi.org/10.1016/j.conb.2010.08.002.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. **Introduction to Algorithms**, chapter 21. Data structures for disjoint sets, pages 498–524. MIT Press, 3rd edition, 2009.

J. Cousty, G. Bertrand, L. Najman, and M. Couprie. Watershed cuts: Minimum spanning forests and the drop of water principle. **Transactions on Pattern Analysis and Machine Intelligence**, 31(8):1362–1374, 2009. DOI http://dx.doi.org/10.1109/TPAMI.2008.173.

R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. **Probabilistic Networks and Expert Systems: Exact Computational Methods for Bayesian Networks**. Springer, 2007.

R. J. Dakin. A tree-search algorithm for mixed integer programming problems. **Computer Journal**, 8:250–255, 1965.

G. Damiand. Topological model for 3D image representation: Definition and incremental extraction algorithm. **Computer Vision and Image Understanding**, 109(3):260–289, 2008. DOI http://dx.doi.org/10.1016/j.cviu.2007.09.007.

G. Damiand and P. Resch. Split and merge algorithms defined on topological maps for 3D image segmentation. **Graphical Models**, 65(1-3):149–167, 2003. DOI http://dx.doi.org/10.1016/S1524-0703(03)00009-2.

A. Delong, A. Osokin, H. Isack, and Y. Boykov. Fast approximate energy minimization with label costs. **International Journal of Computer Vision**, 2010.

W. Denk and H. Horstmann. Serial block-face scanning electron microscopy to reconstruct three-dimensional tissue nanostructure. **PLoS Biology**, 2(11):e329, 2004. DOI http://dx.doi.org/10.1371/journal.pbio.0020329.

S. Derivaux, S. Lefevre, C. Wemmert, and J. Korczak. On machine learning in watershed segmentation. In **IEEE Workshop Machine Learning for Signal Processing**, pages 187–192, 2007.

H. Digabel and C. Lantuéjoul. Iterative algorithms. In J. L. Chermant, editor, **2nd European Symposium on Qualitative Analysis of Microstructures in Material Science, Biology and Medicine**, pages 85–99, Stuttgart, 1978. Riederer Verlag.

P. Dollar, Z. Tu, and S. Belongie. Supervised learning of edges and object boundaries. In **Proceedings of the Conference on Computer Vision and Pattern Recognition**, pages 1964–1971. IEEE Computer Society, 2006. DOI http://dx.doi.org/10.1109/CVPR.2006.298.

M. T. Dougherty, M. J. Folk, E. Zadok, H. J. Bernstein, F. C. Bernstein, K. W. Eliceiri, W. Benger, and C. Best. Unifying biological image formats with hdf5. **Communications of the ACM**, 52(10):42–47, 2009. DOI http://dx.doi.org/10.1145/1562764.1562781.

V. Franc and B. Savchynskyy. Discriminative learning of max-sum classifiers. **Journal of Machine Learning Research**, pages 67–104, 2008. URL http://www.jmlr.org/papers/volume9/franc08a/franc08a.pdf.

B. J. Frey and N. Jojic. A comparison of algorithms for inference and learning in probabilistic graphical models. **Transactions on Pattern Analysis and Machine Intelligence**, 27:1392–1416, 2005. DOI http://dx.doi.org/10.1109/TPAMI.2005.169.

R. Garcia and A. Lumsdaine. MultiArray: a C++ library for generic programming with arrays. **Software: Practice and Experience**, 35: 159–188, 2005. DOI http://dx.doi.org/10.1002/spe.630.

S. Geman and D. Geman. Stochastic relaxation, Gibbs distribution and the Bayesian restoration of images. **Transactions on Pattern Analysis and Machine Intelligence**, 6:721–741, 1984. DOI http://dx.doi.org/10.1109/TPAMI.1984.4767596.

C. Gini. Variabilita e mutabilita. (reprinted in: E. Pizetti and T. Salvemini. Memorie di metodologica statistica. Libreria Eredi Virgilio Veschi. Rome 1955), 1912.

A. Globerson and T. Jaakkola. Fixing max-product: Convergent message passing algorithms for MAP LP-relaxations. In **Advances in Neural Information Processing Systems**, pages 553–560, Cambridge, MA, USA, 2007. MIT Press.

B. Glocker, N. Komodakis, G. Tziritas, N. Navab, and N. Paragios. Dense image registration through MRFs and efficient linear programming. **Medical Image Analysis**, 12(6):731–741, 2008.

D. Gregor, J. Järvi, and G. Powell. Variadic templates (revision 3). Technical Report N2080=06-0150, ANSI/ISO C++ Standard Committee, 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2080.pdf.

D. Gregor, J. J. ans J. Maurer, and J. Merrill. Proposed wording for variadic templates. Technical Report N2152=07-0012, ANSI/ISO C++ Standard Committee, 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2152.pdf.

P. Hamill. **Unit Test Frameworks**. O'Reilly, 2004.

A. Hatcher. **Algebraic topology**. Cambridge University Press, 2002.

X. He, R. S. Zemel, and D. Ray. Learning and incorporating top-down cues in image segmentation. **Proceedings of the European Conference on Computer Vision**, pages 338–351, 2006.

M. Helmstaedter, K. L. Briggman, and W. Denk. 3D structural imaging of the brain with photons and electrons. **Current Opinion in Neurobiology**, 18(6):633–641, 2008. DOI http://dx.doi.org/10.1016/j.conb.2009.03.005.

M. Helmstaedter, K. L. Briggman, and W. Denk. High-precision neurite tracing for high-throughput neuroanatomy. 2011. (forthcoming).

L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is NP-complete. **Information Processing Letters**, 5(1):15–17, 1976. DOI http://dx.doi.org/10.1016/0020-0190(76)90095-8.

V. Jain, J. F. Murray, F. Roth, S. Turaga, V. Zhigulin, K. L. Briggman, M. N. Helmstaedter, W. Denk, and H. S. Seung. Supervised learning of image restoration with convolutional networks. In **Proceedings of the International Conference on Computer Vision**, pages 1–8, Los Alamitos, CA, USA, 2007. IEEE Computer Society. DOI http://dx.doi.org/10.1109/ICCV.2007.4408909.

K. Jung, P. Kohli, and D. Shah. Local rules for global MAP: When do they work? In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, **Advances in Neural Information Processing Systems 22**, pages 871–879. 2009.

E. Jurrus, M. Hardy, T. Tasdizen, P. T. Fletcher, P. Koshevoy, C.-B. Chien, W. Denk, and R. Whitaker. Axon tracking in serial block-face scanning electron microscopy. **Medical Image Analysis**, 13(1):180–188, 2009. DOI http://dx.doi.org/10.1016/j.media.2008.05.002.

E. Jurrus, A. R. Paiva, S. Watanabe, J. R. Anderson, B. W. Jones, R. T. Whitaker, E. M. Jorgensen, R. E. Marc, and T. Tasdizen. Detection of neuron membranes in electron microscopy images using a serial neural network architecture. **Medical Image Analysis**, 14(6):770–783, 2010. DOI http://dx.doi.org/10.1016/j.media.2010.06.002.

J. H. Kappes, S. Schmidt, and C. Schnörr. MRF inference by k-fan decomposition and tight Lagrangian relaxation. In K. Daniilidis, P. Maragos, and N. Paragios, editors, **Proceedings of the European Conference on Computer Vision 2010**, pages 735–747. Springer, 2010.

V. Kaynig, B. Fischer, E. Müller, and J. M. Buhmann. Fully automatic stitching and distortion correction of transmission electron microscope images. **Journal of Structural Biology**, 171(2):163–173, 2010a. DOI http://dx.doi.org/10.1016/j.jsb.2010.04.012.

V. Kaynig, T. Fuchs, and J. M. Buhmann. Neuron geometry extraction by perceptual grouping in ssTEM images. In **Proceedings of the Conference on Computer Vision and Pattern Recognition**, 2010b.

E. Khalimsky, R. Kopperman, and P. Meyer. Computer graphics and connected topologies on finite ordered sets. **Topology and its Applications**, 36:1–17, 1990. DOI http://dx.doi.org/10.1016/0166-8641(90)90031-V.

R. Kindermann and J. L. Snell. **Markov Random Fields and Their Applications**. AMS, 1980. URL http://www.ams.org/publications/online-books/conm1-index.

R. Klette. Cell complexes through time. In L. J. Latecki, D. M. Mount, and A. Y. Wu, editors, **Vision Geometry IX**, volume 4117, pages 134–145. Society of Photo-Optical Instrumentation Engineers (SPIE), 2000.

P. Kohli, A. Shekhovtsov, C. Rother, V. Kolmogorov, and P. Torr. On partial optimality in multi-label MRFs. In **Proceedings of the 25th International Conference on Machine Learning**, 2008. DOI http://dx.doi.org/10.1145/1390156.1390217.

D. Koller and N. Friedman. **Probabilistic Graphical Models**. MIT Press, 2009.

V. Kolmogorov. Convergent tree-reweighted message passing for energy minimization. **Transactions on Pattern Analysis and Machine Intelligence**, 28(10):1568–1583, 2006. DOI http://dx.doi.org/10.1109/TPAMI.2006.200.

V. Kolmogorov and R. Zabin. What energy functions can be minimized via graph cuts? **Transactions on Pattern Analysis and Machine Intelligence**, 26(2):147–159, 2004. DOI http://dx.doi.org/http://dx.doi.org/10.1109/TPAMI.2004.1262177.

N. Komodakis, N. Paragios, and G. Tziritas. MRF energy minimization and beyond via dual decomposition. **Transactions on Pattern Analysis and Machine Intelligence**, 99(PrePrints), 2010. DOI http://dx.doi.org/10.1109/TPAMI.2010.108.

S. Konishi, A. Yuille, J. Coughlan, and S. Zhu. Statistical edge detection: Learning and evaluating edge cues. **Transactions on Pattern Analysis and Machine Intelligence**, 25(1):57–74, 2003. DOI http://dx.doi.org/10.1109/TPAMI.2003.1159946.

U. Köthe. **Generische Programmierung für die Bildverarbeitung**. PhD thesis, University of Hamburg, 2000.

V. A. Kovalevsky. Finite topology as applied to image analysis. **Computer Vision, Graphics, and Image Processing**, 46(2):141–161, 1989.

V. A. Kovalevsky. Digital geometry based on the topology of abstract cell complexes. In **Proceedings of the DGCI 1993**, pages 259–284, 1993.

T. Kröger. Exploratory and computational analysis of huge volume images for connectomics. Master's thesis, University of Heidelberg, Germany, 2010.

W. G. Kropatsch. Building irregulars pyramids by dual graph contraction. **Proceedings of the IEEE Conference on Vision, Image and Signal Processing**, 142(6):366–374, 1995.

F. R. Kschischang, B. J. Frey, and H. Loeliger. Factor graphs and the sum-product algorithm. **Transactions on Information Theory**, 47: 498–519, 2001. DOI http://dx.doi.org/10.1109/18.910572.

M. P. Kumar, V. Kolmogorov, and P. H. S. Torr. An analysis of convex relaxations for MAP estimation of discrete MRFs. **Journal of Machine Learning Research**, 10:71–106, 2009. URL http://jmlr.csail.mit.edu/papers/volume10/kumar09a/kumar09a.pdf.

R. Kumar, A. V. Reina, and H. Pfister. Radon-like features and their application to connectomics. In **IEEE Computer Society Workshop on Mathematical Methods in Biomedical Image Analysis (MMBIA)**, 2010.

J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In **Proceedings of the 18th International Conference on Machine Learning**, pages 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. **Econometrica**, 28(3):497–520, 1960.

S. L. Lauritzen. **Graphical Models**. Statistical Science. Oxford, 1996.

S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. **Journal of the Royal Statistical Society, Series B**, 50(2):157–224, 1988. DOI http://dx.doi.org/10.2307/2345762.

I. Levner and H. Zhang. Classification-driven watershed segmentation. **Transactions on Image Processing**, 16(5):1437–1445, 2007. DOI http://dx.doi.org/10.1109/TIP.2007.894239.

P. Lienhardt. Subdivisions of n-dimensional spaces and n-dimensional generalized maps. In **Proceedings of the 5th Annual Symposium on Computational Geometry**, pages 228–236, New York, NY, USA, 1989. ACM. DOI http://dx.doi.org/10.1145/73833.73859.

P. Lienhardt. Topological models for boundary representation: a comparison with n-dimensional generalized maps. **Computer Aided Design**, 23(1):59–82, 1991. DOI http://dx.doi.org/10.1016/0010-4485(91)90082-8.

U. Ligges. **Programmieren mit R**. Springer, 3rd edition, 2008.

Y. Lin and Y. Jeon. Random forests and adaptive nearest neighbors. **Journal of the American Statistical Association**, 101(474):578–590, 2006. DOI http://dx.doi.org/10.1198/016214505000001230.

L. Lovász. Submodular functions and convexity. **Mathematical programming - State of the Art**, pages 235–257, 1983.

J. Lu, J. C. Fiala, and J. W. Lichtman. Semi-automated reconstruction of neural processes from large numbers of fluorescence images. **PLoS ONE**, 4(5):e5655, 2009. DOI http://dx.doi.org/10.1371/journal.pone.0005655.

A. Lucchi, K. Smith, R. Achanta, V. Lepetit, and P. Fua. A fully automated approach to segmentation of irregularly shaped cellular structures in EM images. In **International Conference on Medical Image Computing and Computer Assisted Intervention**, 2010.

J. H. Macke, N. Maack, R. Gupta, W. Denk, B. Schölkopf, and A. Borst. Contour-propagation algorithms for semi-automated reconstruction of neural processes. **Journal of Neuroscience Methods**, 167(2):349–357, 2008. DOI http://dx.doi.org/10.1016/j.jneumeth.2007.07.021.

D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In **Proceedings of the 8th International Conference on Computer Vision (ICCV)**, volume 2, pages 416–423, 2001.

D. R. Martin, C. C. Fowlkes, and J. Malik. Learning to detect natural image boundaries using local brightness, color, and texture cues. **Transactions on Pattern Analysis and Machine Intelligence**, 26(5):530–549, 2004. DOI http://dx.doi.org/10.1109/TPAMI.2004.1273918.

R. McEliece, D. MacKay, and J.-F. Cheng. Turbo decoding as an instance of Pearl's belief propagation algorithm. **IEEE Journal on Selected Areas in Communications**, 16(2):140–152, 1998. DOI http://dx.doi.org/10.1109/49.661103.

H. Meine and U. Köthe. The GeoMap: A unified representation for topology and geometry. In L. Brun and M. Vento, editors, **Graph-Based Representations in Pattern Recognition**, LNCS 3434, pages 132–141. Springer, 2005. DOI http://dx.doi.org/10.1007/b107037.

H. Meine, U. Köthe, and H. Stiehl. Fast and accurate interactive image segmentation in the geomap framework. In T. Tolxdorff, J. Braun, H. Handels, A. Horsch, and H. P. Meinzer, editors, **Proc. Bildverarbeitung für die Medizin**, pages 60–64. Springer, 2004. DOI http://dx.doi.org/10.1007/b107037.

F. Meyer. Un algorithme optimal de ligne de partage des eaux. In **Actes du 8ème Congrès AFCET**, pages 847–859, Lyon-Villeurbanne, France, 1991.

S. Meyers. **Effective C++ – 55 Specific Ways to Improve Your Designs**. Addison Wesley, 3rd edition, 2005.

T. P. Minka. Expectation propagation for approximate Bayesian inference. In **Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence**, pages 362–369, 2001.

G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In **Proceedings of the 32nd International Conference on Very Large Data Bases**, 2006.

J. R. Munkres. **Elements of Algebraic Topology**. Perseus Books, 1995.

K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In **Proceedings of Uncertainty in AI**, pages 467–475, 1999.

H. T. Nguyen, M. Worring, and R. van den Boomgaard. Watersnakes: Energy-driven watershed segmentation. **Transactions on Pattern Analysis and Machine Intelligence**, 25(3):330–342, 2003. DOI http://dx.doi.org/10.1109/TPAMI.2003.1182096.

T. Pavlidis. **Structural Pattern Recognition**, volume 1 of **Electrophysics**. Springer, 1977.

J. Pearl. **Probabilistic reasoning in intelligent systems: networks of plausible inference**. Morgan Kaufmann, San Francisco, CA, USA, 1988.

G. D. Reis and B. Stroustrup. Templates aliases (revision 3). Technical Report N2258=07-0118, ANSI/ISO C++ Standard Committee, 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2258.pdf.

X. Ren and J. Malik. Learning a classification model for segmentation. In **Proceedings of the 9th International Conference on Computer Vision (ICCV)**, volume 1, pages 10–16, 2003.

J. B. T. M. Roerdink and A. Meijster. The watershed transform: definitions, algorithms and parallelization strategies. **Fundamenta Informaticae**, 41(1-2):187–228, 2000.

D. Schlesinger. Exact solution of permuted submodular MinSum problems. In **Proceedings of the 6th Conference on Energy Minimization Methods in Computer Vision and Pattern Recognition**, 2007.

A. Schrijver. **Theory of linear and integer programming**. John Wiley & Sons, Inc., New York, NY, USA, 1986.

A. Schrijver. **Combinatorial Optimization: Polyhedra and Efficiency**. Springer, 2003.

D. Sieling. Minimization of decision trees is hard to approximate. **Journal of Computer and System Sciences**, 74(3):394–403, 2008. DOI http://dx.doi.org/10.1016/j.jcss.2007.06.014.

O. Sporns, G. Tononi, and R. Kötter. The human connectome: A structural description of the human brain. **PLoS Computational Biology**, 1(4): e42, 2005. DOI http://dx.doi.org/10.1371/journal.pcbi.0010042.

B. Stroustrup. **The C++ Programming Language**. Addison-Wesley, 3rd edition, 2000.

B. Stroustrup. A brief look at C++0x. **The C++ Source**, 2006.

B. Stroustrup and G. D. Reis. Initializer lists (revision 3). Technical Report N2215=07-0075, ANSI/ISO C++ Standard Committee, 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2215.pdf.

H. Sutter and A. Alexandrescu. **C++ Coding Standards: 101 Rules, Guidelines, and Best Practices**. Addison Wesley, 2004.

R. H. Swendsen and J.-S. Wang. Nonuniversal critical dynamics in Monte Carlo simulations. **Physical Review Letters**, 58(2):86–88, 1987. DOI http://dx.doi.org/10.1103/PhysRevLett.58.86.

S. C. Turaga, K. L. Briggman, M. Helmstaedter, W. Denk, and H. S. Seung. Maximin affinity learning of image segmentation. In **Advances in Neural Information Processing Systems**, 2009.

S. C. Turaga, J. F. Murray, V. Jain, F. Roth, M. Helmstaedter, K. Briggman, W. Denk, and H. S. Seung. Convolutional networks can learn to generate affinity graphs for image segmentation. **Neural Computation**, 22(2):511–538, 2010. DOI http://dx.doi.org/10.1162/neco.2009.10-08-881.

I. Vanhamel, I. Pratikakis, and H. Sahli. Multiscale gradient watersheds of color images. **Transactions on Image Processing**, 12(6):617–626, 2003. DOI http://dx.doi.org/10.1109/TIP.2003.811490.

T. Veldhuizen. Arrays in Blitz++. In **Proceedings of the 2nd International Conference on Scientific Computing in Object-Oriented Parallel Environments**, number 1505 in LNCS, pages 223–220. Springer, 1998.

M. J. Wainwright and M. I. Jordan. **Graphical Models, Exponential Families, and Variational Inference**. Now Publishers Inc., Hanover, MA, USA, 2008. URL http://www.eecs.berkeley.edu/~wainwrig/Papers/WaiJor08_FTML.pdf.

M. J. Wainwright, T. Jaakkola, and A. S. Willsky. MAP estimation via agreement on trees: message-passing and linear programming. **Transactions on Information Theory**, 51(11):3697–3717, 2005. DOI http://dx.doi.org/10.1109/TIT.2005.856938.

Y. Weiss and W. Freeman. On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs. **Transactions on Information Theory**, 47(2):736 –744, 2001.

T. Werner. A linear programming approach to max-sum problem: A review. **Transactions on Pattern Analysis and Machine Intelligence**, 29 (7):1165–1179, 2007. DOI http://dx.doi.org/10.1109/TPAMI.2007.1036.

J. G. White, E. Southgate, J. N. Thomson, and S. Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. **Philosophical Transactions of the Royal Society of London. B, Biological Sciences**, 314(1165):1–340, 1986. DOI http://dx.doi.org/10.1098/rstb.1986.0056.

U. Wolff. Collective monte carlo updating for spin systems. **Physical Review Letters**, 62(4):361–364, 1989. DOI http://dx.doi.org/10.1103/PhysRevLett.62.361.

J. S. Yedidia, W. T. Freeman, and Y. Weiss. Bethe free energy, Kikuchi approximations and belief propagation algorithms. Technical report, Mitsubishi Electric Research Laboratories, 2000.

L. Zhang and Q. Ji. Image segmentation with a unified graphical model. **Transactions on Pattern Analysis and Machine Intelligence**, 32: 1406–1425, 2010. DOI http://dx.doi.org/10.1109/TPAMI.2009.145.