DISSERTATION

SUBMITTED

TO THE

COMBINED FACULTY 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

Diplom-Physiker Christoph-Nikolas Straehle

Born in Heidelberg

Oral examination: ___________________

Interactive Segmentation, Uncertainty and Learning

Advisor: Prof. Dr. Fred A. Hamprecht

Abstract

Interactive segmentation is an important paradigm in image processing. To minimize the number of user interactions ("seeds") required until the result is correct, the computer should actively query the human for input at the most critical locations, in analogy to active learning. These locations are found by means of suitable uncertainty measures. I propose various such measures for the watershed cut algorithm along with a theoretical analysis of some of their properties in Chapter 2.

Furthermore, real-world images often admit many different segmentations that have nearly the same quality according to the underlying energy function. The diversity of these solutions may be a powerful uncertainty indicator. In Chapter 3 the crucial prerequisite in the context of seeded segmentation with minimum spanning trees (i.e. edge-weighted watersheds) is provided. Specifically, it is shown how to efficiently enumerate the k smallest spanning trees that result in *different* segmentations.

Furthermore, I propose a scheme that allows to partition an image into a previously unknown number of segments, using only minimal supervision in terms of a few must-link and cannot-link annotations. The algorithm presented in Chapter 4 makes no use of regional data terms, learning instead what constitutes a likely boundary between segments. Since boundaries are only implicitly specified through cannot-link constraints, this is a hard and nonconvex latent variable problem. This problem is adressed in a greedy fashion using a randomized decision tree on features associated with interpixel edges. I propose to use a *structured* purity criterion during tree construction and also show how a backtracking strategy can be used to prevent the greedy search from ending up in poor local optima.

The problem of learning a boundary classifier from sparse user annotations is also considered in Chapter 5. Here the problem is mapped to a multiple instance learning task where positive bags consist of paths on a graph that cross a segmentation boundary and negative bags consist of paths inside a user scribble.

Multiple instance learning is also the topic of Chapter 6. Here I propose a multiple instance learning algorithm based on randomized decision trees. Experiments on the typical benchmark data sets show that this model's prediction performance is clearly better than earlier tree based methods, and is only slightly below that of more expensive methods.

Finally, a flow graph based computation library is discussed in Chapter 7. The presented library is used as the backend in a interactive learning and segmentation toolkit and supports a rich set of notification mechanisms for the interaction with a graphical user interface.

Zusammenfassung

Interaktive Segmentierung ist ein wichtiger Bereich in der Bildverarbeitung. Hierbei ist es wünschenswert die Anzahl der Benutzerinteraktionen zu minimieren, die gebraucht werden um ein gewünschtes Ergebnis zu erzielen. Um das zu erreichen, sollte der Computer den Benutzer an den problematischsten Stellen, in Analogie zum aktiven Lernen, nach zusätzlichen Eingaben fragen. Solch problematische Stellen in der Segmentierung können durch geeignete Unsicherheitsmaße lokalisiert werden. In Kapitel 2 schlage ich verschiedene Unsicherheitsmaße für den watershed cut Algorithmus vor.

Oftmals haben viele verschiedene Segmentierungen nahezu die gleiche Energie bezüglich des Optimierungsproblems. Die Unterschiede dieser energetisch nahezu gleichwertigen Segmentierungen können ein aussagekräftiges Unsicherheitsmaß sein. In Kapitel 3 präsentiere ich einen Algorithmus, welcher die k günstigsten watershed cut Segmentierungen enumerieren kann. Damit wird die Vorraussetzung zur Analyse dieser Segmentierungen geschaffen.

Des Weiteren schlage ich einen Algorithmus vor, welcher es erlaubt ein Bild in eine vorher unbekannte Anzahl von Segmenten zu partitionieren. Dabei benutzt der Algorithmus nur sehr schwache Annotationen in Form von einigen must-link und cannot-link Angaben. Der in Kapitel 4 vorgestellte Algorithmus nutzt keine lokalen Datenterme sondern lernt direkt was eine wahrscheinliche Grenzfläche zwischen Objekten ist. Da die Grenzflächen nur implizit durch die cannot-link Angaben spezifiziert werden, ist dies ein schwieriges und nicht konvexes Problem. Ich schlage hierzu einen randomisierten Entscheidungsbaum vor, welcher ein strukturiertes Lernkriterium beim Aufbau nutzt und dabei auf Zwischenpixel-Kanten arbeitet. Eine Backtrackingstrategie hilft zu verhindern, dass ein schlechtes lokales Optimum gefunden wird.

Das gleiche Problem – einen Grenzflächenklassifikator ausgehend von schwachen Benutzerannotationen zu lernen – wird auch in Kapitel 5 betrachtet. Das Problem wird auf ein Multiple Instance Lernproblem zurückgeführt. Hier bestehen positive Taschen aus Pfaden im Graph, die eine Grenzfläche schneiden und negative Taschen aus Pfaden innerhalb einer Benutzerannotation.

Das Multiple Instance Lernverfahren ist auch Thema in Kapitel 6. Hier schlage ich einen Multiple Instance Lernalgorithmus basierend auf randomisierten Entscheidungsbäumen vor. Experimente auf den typischen Benchmarkdatensätzen zeigen, dass der Algorithmus etwas schlechter als teurere Methoden abschneidet aber bessere Ergebnisse liefert als alle existierende Ansätze, die auch auf Entscheidungsbäumen aufbauen.

Schlussendlich wird in Kapitel 7 die datenflussgraphbasierte Softwarebibliothek lazyflow vorgestellt. Die Softwarebibliothek wird als Basis für ilastik, das interaktive lern und segmentier toolkit benutzt, und unterstützt einen großen Satz an Benachrichtigungsmechanismen zur Interaktion mit einer graphischen Benutzeroberfläche.

Contents

Acknowledgments

I want to thank Prof. Dr. Fred A. Hamprecht for supervising me during the time that has led to this thesis. Already during my master thesis Prof. Hamprecht introduced me to the interesting field of pattern recognition. His research group – Multidimensional Image Processing – has always been a great group of friendly, helpful, knowledgeable and fun people. Secondly, i want to express my gratidude to Prof. Dr. Andrzejak who took the burden of being my second advisor and the time to delve into this project. I also want to thank Dr. Ullrich Köthe for all his expertise and help throughout the years. With my colleagues I have spent countless hours on programming, bug fixing and discussions while working on the ilastik project. I will always remember the countless tablesoccer matches i have lost to Christoph Decker. Bernhard Kausler has pushed me to write cleaner code. Thorben Krögers amazing C++ skills always impressed me. Together with Christoph Sommer, Anna Kreshuk, Luca Fiaschi, Martin Schiegg and Stuart Berg we have worked towards a ilastik release. I also wish to thank all my other colleagues, in particular Thorsten Beier, Xinghua Lou, Ferran Diego, Melih Kandemir, Burcin Erocal, Kemal Eren, Philipp Hanslovsky, Robert Walecki, Buote Xu, Ben Heuer, Chong Zhang, Niko Krasowski as well as Carsten Haubold. Furthermore i wish to thank Barbara Werner, Evelyn Wilhem, and Ole Hansen for their support in all administrative affairs. Im also grateful for Oliver Petra and Kai Karius support in writing some lazyflow operators. Finally, i am deeply indebted to my Parents Luise and Wolfgang Straehle and my girlfriend Jil Molitor for supporting me in any possible way throughout these years.

Chapter 1

Introduction

1.1 Interactive segmentation

Interactive segmentation is an important paradigm in image processing. It allows to partition an image into components under the supervision of a human labeler. Such functionality is tremendously useful for example in life sciences where biologists need to extract and measure objects of interest from microscopic images. The most basic interactive segmentation method consists of a drawing application which the user can use to segment an image by labeling the individual image parts. This manual segmentation process is slow and tedious, especially so for 3D datasets. Over the years many algorithms have been developed that help the user in segmenting image data. These algorithms speed up the segmentation process since they rely only on sparse annotations instead of the dense image labeling. The sparse annotations are used as a starting point for some kind of region growing procedure which stops either at a discernible image boundary or when two different regions start to overlap.

One such segmentation method are the *active contour* based methods such as [26, 27, 137]. In these methods, the evolving estimate of the structure of interest is represented by one or more contours. An evolving contour is a closed surface that evolves over time according to a partial differential equation (PDE). The evolution of the PDE is steered by internal and external forces which act on the normal vector of the closed contour. Internal forces can for example enforce a low curvature of the contour, while external forces usually encode image information and depend on the image location and content at the contour at a time.

Another important line of work are *variational methods* for image segmentation such as [120]. These methods minimize an energy functional

$$E(u) = \int_\Omega g(x)|\nabla u| d\Omega + \int_\Omega \lambda(x)|u - f| d\Omega, u \in [0, 1]$$

where the output $u = 1$ corresponds to a foreground image part and $u = 0$ to a background image part. The data fidelity term $\lambda(x)|u - f|$ enforces (depending on $\lambda(x)$) that the classifier

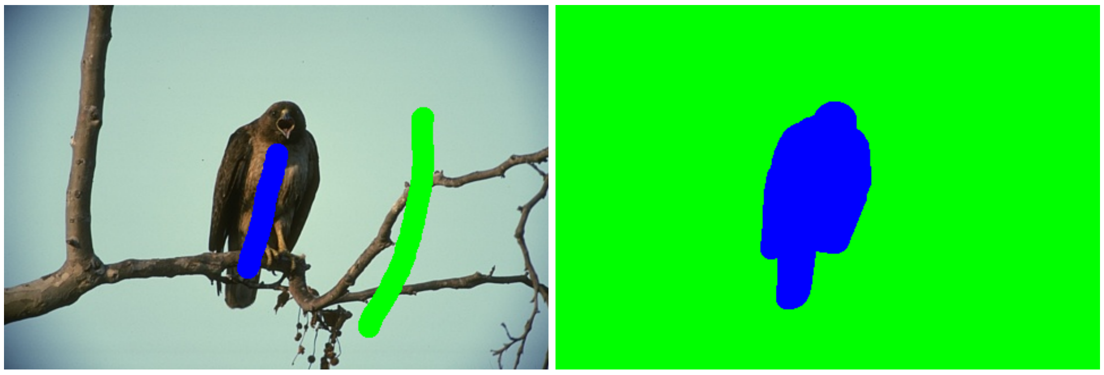


Figure 1.1: Interactive segmentation illustration. The left picture shows a image from the BSD300 [85] segmentation database and a set of foreground and background seeds a user might give. The right picture shows a potential segmentation result. The task of interactive segmentation algorithms is to produce a result close to the right picture with minimal supervision.

prediction f is obeyed. $g(x)|\nabla u|$ is the infamous total variation (TV) norm and enforces a smooth surface and small surface area of the foreground-background transition of u.

Variational methods treat the image space Ω as a continous domain. In contrast to this *graph based interactive segmentation methods* discretize the domain into a grid graph. Still, the underlying principle for interactive segmentation algorithms is the same: an energy function is minimized that consists of a data fidelity term, also called unary potential and a boundary length cost term that is similar to the total variation term. Some of the most important seeded segmentation methods on graphs can be unified in terms of the *power-watershed* framework [33]. It defines a segmentation as a labeling of a graph $G(V, E)$, where the optimal labeling $\mathbf{x}$ minimizes an energy function

$$\begin{aligned} E(\mathbf{x}, \mathbf{w}) = \sum_{v_i \in V} & w^p_{0,i} \|x_i\|^q + w^p_{1,i} \|1 - x_i\|^q \\ & + \sum_{e_{ij} \in E} w^p_{ij} \|x_i - x_j\|^q \end{aligned} \tag{1.1}$$

$x_i \in L$ is the label associated with node $v_i \in V$, $\mathbf{x}$ is the vector of all label assignments, and w_i, w_{ij} are node and edge weights, respectively. The first sum thus measures compatibility of the labeling with a given region model, whereas the second sum enforces smoothness of the solution. For different exponents, the power watershed specializes to the watershed cut algorithm ($p \to \infty$, q finite, [35]), the random walker (p finite, $q = 2$, [50]) and an Ising-type Markov random field amenable to graph-cut segmentation (p finite, $q = 1$, [18]). A number of coarse-grained [29], pre-computed [51] or warm-started [62] strategies have been suggested to speed up the response to user input for some of the methods.

One of the fastest algorithms for interactive segmentation however is the watershed cut [35] which can be computed in linear time in the number of pixels. Even when not using the specialized algorithm presented in [35] the computation is still extremely fast as it only requires the computation of a minimum spanning forest, or, in an augmented graph a minimum spanning tree computation. Despite of its age, the watershed algorithm is topic of current research and is a cornerstone of image processing, especially in 3D and 4D image processing [57, 36, 83, 13, 75, 60, 34]. In this thesis we use the watershed cut algorithm as a basis due to its favorable runtime requirements. The low computational complexity of a minimum spanning tree computation makes this algorithm applicable to large 3D datasets [114, 34, 83, 36]. In addition it was shown that the watershed cut is a very suitable algorithms for some the datasets considered in this thesis [114].

1.2 Watershed

The watershed transform was first introduced for image processing by Digabel and Lantuéjoul [40] in 1978. Since then many different motivations and algorithms have been proposed which are excellently summarized in [104]. The underlying principle is always similar and centered around the behavior of a drop of water on a topological surface. The image is treated as a height map, with bright pixels corresponding to high elevations and dark pixels corresponding to low elevations. Then, starting from the local minima the topological surface is flooded with water and so called dams or watershed lines are built where water from two different minima meets [124]. Another approach [14] is to look at a drop of water that falls onto the surface and flows along the path of steepest descent to a local minimum. From this viewpoint stems the notion of *catchment basins* which are separated by watershed lines, i.e. the lines from which the drop of water can flow to at least two different minima. The above algorithms treat the problem on a pixel grid graph where the input is a height-map on the *nodes* of the pixel grid graph, resulting in a so called *node-weighted watershed*. Another approach is to look at a height-map associated with the *edges* of the pixel-grid graph which results in a so called *edge-weighted watershed*. This approach has been pursued in [35]. The authors show that this edge-weighted watershed is equivalent to a minimum spanning forest computation where the individual trees represent the different local minima or catchment basins. Flooding an image height map or an edge weighted graph from local minima is an unsupervised segmentation paradigm. The watershed however can also be used in a supervised fashion suitable for interactive segmentations. In this setting the algorithm behaves in the same way but starts from a different set of seeds. Instead of associating each local minimum with a seed, only the user given labels are used as a starting point for the flooding process. In this way the watershed algorithm can be applied in an interactive fashion to solve various interactive segmentation tasks.

1.3 Graphs

So far we already mentioned graphs and edges a few times. More formally, a graph $G = (V, E)$ consists of a set of *vertices* or *nodes* V and a set $E \subseteq V \times V$, i.e. pairs of vertices. An unordered pair (u, v) is called an *edge*, and we call node u and node v neighbours. A path $P(p, q)$ in a graph $G = (V, E)$ from vertex p to vertex q is a sequence of vertices $(p_0, p_1, ..., p_l)$ such that $p_0 = p$, $p_l = q$ and $(p_i, p_{i+1}) \in E \forall i \in [0, l)$. If there exists a path $P(p, q)$ from vertex p to vertex q, we say q is *reachable* from p. A graph is *connected* if each vertex is reachable from every other vertex. In a graph a path $(p_0, p_1, ..., p_l)$ forms a *cycle* if $p_0 = p_l$ and $p_1, ..., p_l$ are distinct. A graph with no cycles is *acyclic*. A *forest* is an acyclic graph, a *tree* is a connected acyclic graph. A graph $G' = (V', E')$ is called a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$ and the elements of E' are incident with vertices from V' only. An *edge-weighted graph* $G = (V, E, w)$ is a graph with a weight function $w : E \to \mathbb{R}$. Instead of $w(e)$ we often write w_e or w_{uv} when we talk of an *edge weight* for edge $e = (u, v)$. A *spanning tree* $T = (V', E')$ of graph $G = (V, E)$ is a connected acyclic subgraph of G with $V' = V$.

1.4 Minimum spanning trees

As we will later see, a minimum spanning tree can be used to construct the watershed segmentation of an image. A minimum spanning tree $T = (V, E', w)$ is a connected acyclic subgraph of graph $G = (V, E, w)$ such that the total weight

$$W_{\text{total}} = \sum_{e \in E'} w_e$$

is minimal. This problem has a long history and several efficient algorithms for its solution have been found. On of the best known algorithms is the algorithm of Kruskal [73]. The algorithm sorts the edges $e \in E$ according to their weight w_e in increasing order. Then all edges e are inserted into the tree T according to their sort order if the edge e does not create a cycle in T. The result is a minimum spanning tree T of G.

Another famous algorithm is the algorithm of Prim [100]. It starts at an arbitrary vertex v of graph G and adds it to the initially empty tree T. Then, as long as T does not contain all vertices of G, pick the smallest weight edge e that connects the current tree T to a vertex $v \notin T$. Add e and v to T.

Both algorithms construct a minimum spanning tree which is characterized by the following Lemma:

Definition 1.4.1 (T-exchange [47]) *Let T be a spanning tree of graph G. A T-exchange is a pair of edges e,f where $e \in T$, $f \notin T$, and $T - e \cup f$ is a spanning tree.*

Lemma 1.4.2 ([47]) *A spanning tree T has minimum weight if and only if no T-exchange has negative weight.*

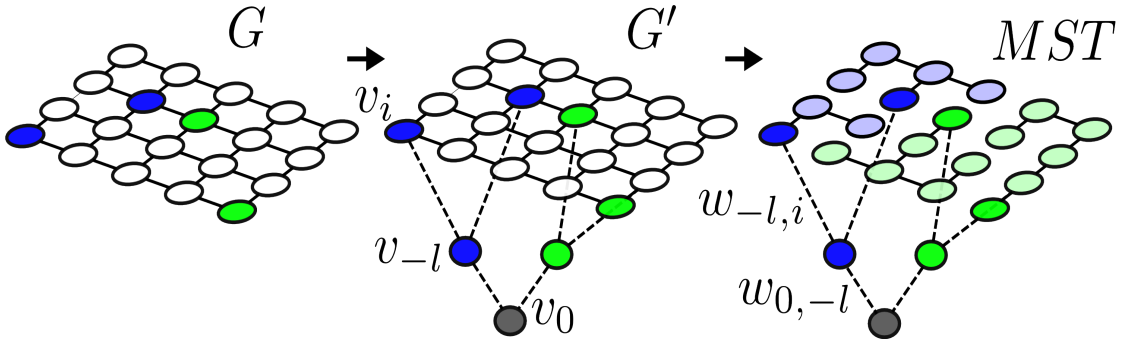


Figure 1.2: Seeded image segmentation using minimum spanning trees. The image nodes which the user has seeded (blue, green) are connected to virtual seed nodes v_{-l} which in turn are connected to a virtual root node v_0. The associated edge weights are set to 0 which ensures that the edges belong to the minimum spanning tree of the augmented graph G'. The final label of an image node depends on the subtree to which the node is assigned in the resulting minimum spanning tree.

1.5 Watershed cuts and minimum spanning trees

It is well known [35] that the watershed cut is equivalent to a minimum spanning forest calculation or a minimum spanning tree computation on an augmented graph. Such a suitably augmented graph $G' = (V', E')$ can be constructed by adding a supernode v_0 connected to newly added seed nodes $v_{-l}, l \in L$ for each class type that are connected to v_0 with zero weight edges. All labeled nodes (i.e. all supervoxels holding a user seed) are also connected to these seed nodes with zero-weight edges. These zero weight edges guarantee that the edges are part of a MST. Once the MST with root node v_0 has been constructed, subtrees originating from seed nodes v_{-l} form segments of the final segmentation. This graph construction is shown in Figure 1.2. The subtrees and the segmentation of the seeded watershed cut are defined as follows:

Definition 1.5.1 (Subtree T^i) *Let T be a spanning tree of G with root node v_0. The subtree $T^i = (V^i, E^i)$ is defined as the set of nodes V^i and edges E^i which in T can only be reached from the root node by a path containing v_i.*

Definition 1.5.2 (Segmentation $\mathbf{x}$) *Let T be a spanning tree of G with root node v_0. The label assignment of all nodes is called the segmentation $\mathbf{x}$ with $x_i = l$ if node v_i is element of the subtree T^{-l} of seed node v_{-l}. Thus, all nodes i with label $x_i = l$ and the edges connecting them form the subtree T^{-l} of T with root node v_{-l}.*

Why is a segmentation based on a minimum spanning tree computation called a watershed cut? Assume we are given an image where the pixel values $h(i)$ of pixel i correspond to the height of a topographic surface. Applying the classical watershed algorithm would start by flooding this surface starting from the user given seeds until the water from different basins meets at so called watershed lines. For the MST based algorithm, we construct a pixel grid graph and associated with each edge (i, j) a weight $w_ij = h(i) + h(j)$ which encodes the average height between the two pixels. When computing the MST with Prim's algorithm starting from the root node v_0 we always follow the smallest edge weight first and attach the node which is incident to this smallest edge weight to the current tree. The subtrees T^{-l} grow in this fashion until they meet and all nodes have been assigned to one of the subtrees. Since this procedure is very similar to the flooding based classical node weighted watershed the name watershed cut coined in [35] is suitable.

1.6 Watershed edge weights

We have presented the underlying principle of the watershed algorithm: flooding a topographic surface from local minima or user given seeds. However, given an image and a segmentation task it is unclear what this topographic surface should be, or, in the context of the watershed cut: what are the best edge weights for the segmentation task. It is clear that the edge weights which indicate boundaries should be high where a desired segmentation boundary is located and should be low where the object or segment of interest is homogeneous. Thus, depending on the segmentation task, edge weights can be constructed for example from the gradient of the image which is suitable to detect step like boundary, i.e. transitions from one homogeneous color to another homogeneous color. Another useful boundary indicator is the largest eigenvalue of the Hessian matrix which is suitable for image boundaries that are represented by a thin dark or bright line.

1.7 Decision trees

Despite the hand-crafted edge weights which were presented in the previous Section 1.6, edge weights can also be learned as a combination of different boundary indicators from user annotations. We will consider this boundary learning problem in Chapter 4 and Chapter 5. The boundary or edge-weight learning can be posed as a supervised learning problem where a classifier is trained from positive (boundary) and negative (no boundary) examples. Each example i consists of a D dimensional feature vector $\mathbf{x}_i \in \mathbb{R}^D$ and an associated class label $y_i \in \{0, 1\}$, where $y_i = 1$ would correspond to a positive example and $y_i = 0$ would correspond to a negative example. In a supervised machine learning setting many such examples are used to train a decision function that maps unseen query vectors $\mathbf{x} \in \mathbb{R}^D$ to either the positive or negative class. The basis for many supervised machine learning algorithms are decision trees. One of the best known works on decision trees is the book Classification and regression trees by Breiman et al. [20]. The authors describe the basics of decision trees and their application to classification and regression tasks. A decision tree is a collection of nodes an edges organized in a hierarchical fashion as a binary tree with a dedicated root node. Nodes are divided into inner nodes and leaf nodes. The leaf nodes contain a decision making predictor such as a class label or a regression value. The inner nodes i contain binary test functions $h(\mathbf{x}, \theta_i)$ whose output can be either true

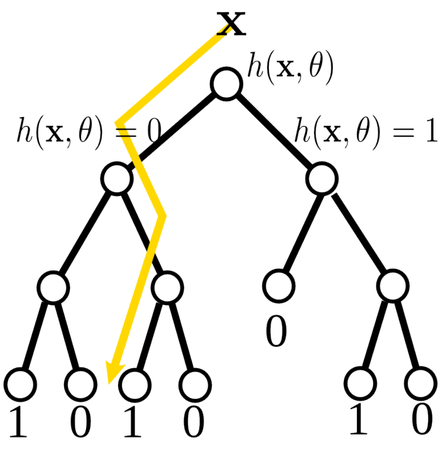


Figure 1.3: Decision tree illustration. Starting at the root node a test function $h(\mathbf{x},\theta)$ is applied to a data sample $\mathbf{x}$. Depending on wether the test function evaluates true (1) or false (0) the data sample is passed on to the right or left child of the current node. This process stops once the data sample reaches a leaf node and the corresponding class label stored at that node is returned.

or false. The output of the test function depends on the parameters θ_i associated with the inner node i and the feature values of the tested data sample $\mathbf{x} \in \mathbb{R}^d$. Often the binary test function $h(\mathbf{x},\theta_i)$ is a simple threshold test for a specific feature, i.e. it tests whether the feature value x_f of a data sample $\mathbf{x}$ for feature f is above or below a certain threshold. At prediction time, a data sample $\mathbf{x}$ is passed down the tree until it reaches a leaf node. The path to the leaf node is given by the results of the binary tests which are applied at the inner nodes. Starting at the root node a binary test function is applied to the data sample. If the outcome is true the sample is passed to the left child node, if the outcome is false the sample is passed to the right child node. This process is repeated recursively until the data sample reaches a leaf node and the class label or regression value stored at that leaf node is returned.

During the training procedure of a decision tree the structure of the tree is determined as well as the split node parameters θ_i. We denote the set of training samples that belongs to a split node i as S_i. An inner node or split node partitions this set of training samples into a left set S_i^L and a right set S_i^R. These two sets are passed on to the left and right child node of the node under consideration. Thus, $S_i^L = S_{2i+1}$ and $S_i^R = S_{2i+2}$. The contents of the set S_i^L and S_i^R depend on the split parameters θ_i. During tree construction these split parameters θ_i of a node are optimized, such that an impurity criterion for the partitioned set is optimized with respect to their ground truth labels:

$$\theta_i^* = \text{argmin}_{\theta_i} I(S_i, S_i^L, S_i^R, \theta_i)$$

where I is a suitable purity criterion, like Gini impurity [19] or Entropy [37]. This partitioning of the data starts with a root node 0 and the set of all training samples S_0 and is continued recursively until the set of samples assigned to a node is pure.

1.8 Thesis overview

The watershed edge weights discussed in Section 1.6 are calculated from simple image filters. The image, and the filter response are subject to noise which may result in ambiguous or noisy edge weights. These erroneous edge weights in turn can lead to a wrong segmentation result. In Chapter 2 and Chapter 3 we study how uncertainty estimates for the segmentation can be constructed, that allow the user to find such potentially erroneous parts of the segmentation.

Furthermore, in many segmentation tasks the best boundary indicator is not just a simple image filter but a combination of many such filters on different scales. In Chapter 4 and Chapter 5 we investigate how such combined edge weights can be learned from sparse user annotations that do not need to be placed on the boundary itself. Instead we learn the edge weights from must-link and cannot link constraints between user given brush strokes, which is beneficial since these strokes are much easier to give.

One of those the boundary weight learning algorithms I propose is based on Multiple Instance Learning, a weakly supervised learning technique. This lead to research on Multiple Instance Learning algorithms itself. Thus, in Chapter 6 I propose a Multiple Instance Learning algorithm based on an optimized linear combination of decision trees.

Finally, in Chapter 7 a flow graph based computation library is presented which forms the backend of ilastik [110], the interactive learning and segmentation toolkit.

Chapter 2

Watershed uncertainty

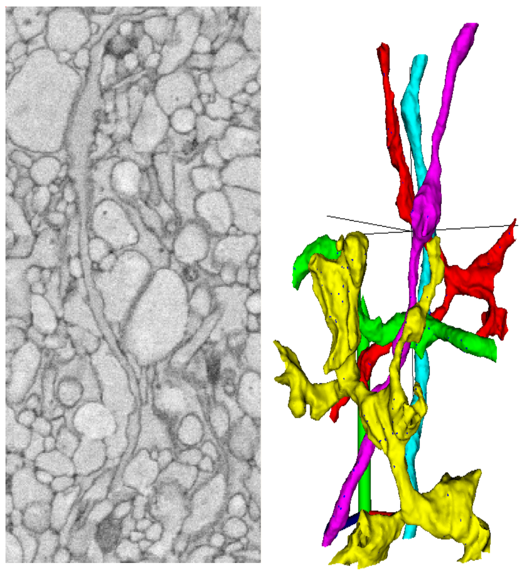
2.1 Introduction

Interactive segmentation is a popular paradigm in image analysis because it combines the number-crunching capabilities of a computer with the high-level understanding of a human. When the segmentation result is immediately updated after each interaction, the user can readily spot errors and correct these by a (hopefully small) number of additional inputs. Unfortunately, this elegant scheme breaks down in 3D because errors no longer “pop out” to the user’s attention as they do in 2D – it is not possible to visualize complicated 3D segmentations in a way that makes user inspection and intervention as easy as in two dimensions. Most commonly, volume data is presented on a 2D screen by means of three orthogonal views, and the user has to scroll through several, and possibly many, layers in order to find or rule out segmentation errors.

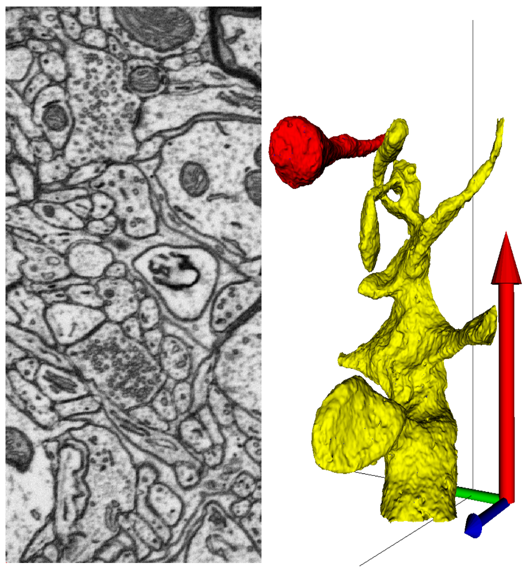
We propose to solve this problem by guided interactive segmentation [43], akin to active learning. Active learning (AL) [107] schemes aim at the steepest possible learning curve by querying for user input on locations which are regarded as most informative by a suitable selection criterion, so that user effort is focused on decisions with high impact. Accordingly, our algorithm not only proposes a segmentation based on the user’s inputs, but also estimates a *confidence* in the segmentation result which will guide the user to locations where the uncertainty is highest. Good uncertainty criteria are especially challenging in our context because segmentation quality is a non-local property: A very small error (e.g. a single wrongly deleted boundary) can have catastrophic global consequences (such as an erroneous merge of two very large regions). Purely local error estimates as used in most existing AL work on interactive segmentation [9, 43, 117] are not sufficiently sensitive to these non-local effects.

Our interactive segmentation framework is based on the watershed algorithm because it has a small computational footprint and is attractive for our target application: Microscopic images of neural tissue (see Fig. 2.1) are composed of very thin and elongated structures, which may pose a problem for the graph cut and random walk algorithms with their well-known shrinking bias.

For the neurobiological application example we compute watershed cuts on supervoxel graphs



(a)



(b)

Figure 2.1: Raw data and ground truth. (a) Serial blockface electron microscopy (SBEM) image slice and 3D rendering of some neural processes from the image stack. (b) Focused ion beam electron microscopy (FIBSEM) image slice and two neural processes.

[80, 114] (see section 2.4.2). The goal of interactive segmentation is therefore to merge all supervoxels belonging to the same object. User labels are interpreted as seeds for either a foreground object or the background, and regions are defined according to a watershed cut initiated by these seeds [35].

Specifically we make the following contributions:

- We define and characterize a number of uncertainty criteria that can be used in the context of interactive segmentation with the watershed cut algorithm.
- We conduct extensive comparisons of the practical performance of these criteria in 3D neuro-imaging application examples.
- We demonstrate empirically that correct segmentations are achieved much faster when user attention is guided by our best active learning criteria.

This chapter is based on the publication [113].

2.2 Background and related work

Interactive segmentation algorithms must be able to take user input ("seeds") into account and update results incrementally when new input arrives, and it must be sufficiently fast to ensure interactive response times. As explained in Chapter 1, some of the most important seeded segmentation methods can be unified in terms of the *power-watershed* framework [32]. It defines a segmentation as a labeling of a graph $G(V, E)$, where the optimal labeling $\mathbf{x}$ minimizes an energy function

$$\begin{aligned} E(\mathbf{x}, \mathbf{w}) = \sum_{v_i \in V} & w_{0,i}^p \|x_i\|^q + w_{1,i}^p \|1 - x_i\|^q \\ & + \sum_{e_{ij} \in E} w_{ij}^p \|x_i - x_j\|^q \end{aligned} \tag{2.1}$$

where $x_i \in L$ is the label associated with node $v_i \in V$, $\mathbf{x}$ is the vector of all label assignments, and w_i, w_{ij} are node and edge weights, respectively. The first sum thus measures compatibility of the labeling with a given region model, whereas the second sum enforces smoothness of the solution. For different exponents, the power watershed specializes to the watershed cut algorithm ($p \to \infty$, q finite, [79], [35]), the random walker (p finite, $q = 2$, [50]) and an Ising-type Markov random field amenable to graph-cut segmentation (p finite, $q = 1$, [18]). A number of coarse-grained [29], pre-computed [51] or warm-started [62] strategies have been suggested to speed up the response to user input.

A significant simplification of the solution space is achieved by moving from a grid-graph defined on the original voxels to a coarser graph of supervoxels. The weighted graphs that reflect the supervoxel adjacency typically have a large total number of nodes, a small number of seeded nodes (the user scribbles), and are sparse (i.e. the number of edges is of the same order as the number of nodes). This is a favorable situation for power-watershed methods.

A number of user guidance schemes have already been proposed in the context of interactive segmentation for the random walker [43, 117] or graph cuts [9, 98]. These works use uncertainty cues based on the margin in the case of the random walk or min-marginal energies in the graph cut case [68], both of which capture mostly local information. In addition a perturbation based local uncertainty estimator for the graph-cut has recently been proposed in [98].

We propose several non-local uncertainty estimators for the watershed cut, and show that they perform better than a local alternative. Closest in spirit to our work are the stochastic topological watershed variants that have been proposed for *un*seeded segmentation [3, 92]. These authors consider a topological watershed from randomized seeds, while we randomize the edge weights instead.

2.3 Uncertainty measures for watershed cuts

We reviewed the notion of a watershed cut and its relation to a minimum spanning tree (MST) in Section 1.5 Now we will present two different types of uncertainty estimators. The first type presented in Section 2.3.2 is based on a minimal perturbation property and expresses how the obtained segmentation boundary depends on single edges in a graph. The second type of uncertainty estimators in Section 2.3.4 takes into account the uncertainty of the edge weights themselves. It measures how much the overall segmentation changes when sampling noisy edge weights.

2.3.1 Watershed cuts and minimum spanning trees

The interactive segmentation algorithm in this chapter is based on watershed cuts [79, 35]. It starts from a supervoxel graph which is computed by a standard flooding-type watershed algorithm [124] on a suitable boundary indicator, in our case the largest eigenvalue of the Hessian matrix which measures "ridgeness" and thus indicates cell membranes. The region adjacency graph $G(V, E)$ of the supervoxels is equipped with edge weights that encode surface strength (in particular, the minimum value of the boundary indicator on the corresponding surface patch). User seeds provide hard assignments of some supervoxels to the background or one of the foreground regions.

As we explained in Section 1.5 the watershed cut is equivalent to a minimum spanning tree (MST) computation on a suitably augmented graph $G'(V', E')$ that contains a supernode v_0 connected to seed nodes $v_{-l}, l \in L$ for each class type that are connected to v_0 with zero weight edges. All labeled nodes (i.e. all supervoxels holding a user seed) are also connected to these seed nodes with zero-weight edges, which are guaranteed to remain in the MST. Once the MST with root node v_0 has been constructed, subtrees originating from seed nodes v_{-l} form segments of the final segmentation. This graph construction is shown in Figure 1.2.

Since we are concerned with non-local uncertainty estimates that measure influences on the segmentation, we frequently rely on the definition of the edges connecting different segments:

Definition 2.3.1 (Cut set $C(T)$) *Let T be a spanning tree of G. An edge $e = (i, j)$ is element of the cut set $C(T)$ if the vertices v_i and v_j belong to different subtrees T^{-l} (Definition 1.5.1) so that the segmentation label (Definition 1.5.2) $x_i \neq x_j$.*

In the following, we will introduce estimators arising from two different general principles. The first one analyzes the effect of single edge weight perturbations on the MST and the resulting segmentation, in a manner similar to [30]. The second estimator takes into account that the edge weights of the supervoxel graph are themselves subject to uncertainty, and measures how much the segmentation would change under perturbations of all weights.

As a baseline we compare our non-local estimators to a *local instability* defined by the margin of the seeeded watershed cut: the margin is the difference between the maximum weight encountered on the lowest possible path from the winning seed type to the node under consideration minus the maximum weight on the lowest possible path from the second best seed type to the same node.

2.3.2 Link instability via minimum perturbations

The first uncertainty measure we propose estimates the influence of *individual* edges on the final segmentation. In Lemma 2.3.4 we show that only the inclusion of edges $f \in C(T)$ from the current cut set $C(T)$ (i.e. $f \notin T$) into the minimum spanning tree changes the resulting segmentation. We propose to measure the instability of all edges $e \in T$ currently part of the MST T by calculating how often an edge e would be removed from the MST when considering perturbations that would enforce the inclusion of an edge $f \in C(T), \notin T$ into the MST. i.e. weight perturbations that change the segmentation.

Algorithm 2.3.2.1 counts how often an edge which is part of the MST would be removed from the spanning tree considering all edge weight perturbations that change the segmentation (Definition 1.5.2) induced by the MST. The correctness of Algorithm 2.3.2.1 is shown in the

Algorithm 2.3.2.1 Link Instability

```
1. Determine the cut set $C(MST)$.
2. Do a breadth first search starting from the root node of the MST and store at each node $n$
   the edge with maximum weight encountered on the path from the root node to node $n$.
3. For each edge in the cut set: The exchange partner is the edge with maximum weight
   stored in either end node of the cut edge. Increment the counter of that edge.
```

next section.

We note that the runtime of Algorithm 2.3.2.1 is linear in the number of edges of the graph and thus preserves the low computational overhead of the watershed cut. This follows from the fact that the computational complexity of the determination of $C(T)$, the breadth first search and the counter incrementation are all linear in the number of edges.

2.3.3 Correctness of algorithm 2.3.2.1

Our uncertainty measures estimate by how much the segmentation would change if the edges in the spanning tree were replaced, and how likely these replacements are. MST edge exchange has been investigated in [47], and we repeat a useful lemma and definition from there:

Definition 2.3.2 (e, f exchange) *[47] Let T be a spanning tree of graph G. A e, f-exchange is a pair of edges, e, f where $e \in T, f \notin T$, and $T \setminus e \cup f$ is a spanning tree. The weight of the exchange e, f is $w(f) - w(e)$. The weight of tree $T \setminus e \cup f$ is the weight of tree T plus the weight of the exchange e, f.*

Lemma 2.3.3 *[47] A spanning tree T has minimum weight if and only if no e, f-exchange has negative weight.*

Since we are only interested in changes of the MST that cause changes in the induced segmentation, we analyze the sufficient and neccessary conditions that yield a different segmentation:

Lemma 2.3.4 *An e, f-exchange resulting in a spanning tree $T' = T \setminus e \cup f$ induces a watershed segmentation different from T if and only if $f \in C(T)$.*

Proof *$\rightarrow$: Let e, f be an exchange with edge $e = (i, j) \in T$ and edge $f = (k, l) \in C$. Then, either node k or node l change their segmentation: before the exchange we obtain from the definition of the cut set C and $f \in C$: $x_k \neq x_l$, while after the exchange $f \in T$ and thus $x_k = x_l$.*

$\leftarrow$: Let e, f be an exchange with edge $e = (i, j) \in T$ and edge $f = (k, l)$. Assume $f \notin C$. First consider f to connect two nodes in the same subtree T^{-l} of e, i.e. $x_i = x_j = x_k = x_l = l$. Thus an e, f exchange will not change the segmentation of any node. Now consider f to connect two nodes in a different subtree $T^{-l'} \neq T^{-l}$ then e, i.e. $x_i = x_j = l$ and $x_k = x_l = l'$, then the e, f exchange will not produce a valid spanning tree: subtree $T^{-l'}$ contains a cycle, and subtree T^l is partitioned into two components.

Relying on the notion of an e, f-exchange (Definition 2.3.2) we now proof the correctness of the edge *link instability* Algorithm:

Lemma 2.3.5 *Algorithm 2.3.2.1 counts how often an edge in the minimum spanning tree is exchange partner in negative e, f exchanges resulting from all minimal single edge weight perturbations that induce a different seeded watershed cut segmentation (Lemma 2.3.4).*

Proof *Item 1: When considering all segmentation changing perturbations involving a single edge, it suffices to consider the e, f-exchanges where $e \in T$ and $f \in C(T)$. This follows from Lemma 2.3.4.*

Item 2: When considering the minimal perturbations that move edge $f \in C(T)$ into the minimum spanning tree via an e, f exchange, it suffices to consider the edges on the path from node i or node j to the root node v_0, with edge $f = (i, j)$: if the edge e were not on a path from node i or node j to the root node v_0, with edge $f = (i, j)$, exchanging e with f would lead to a cycle. The fact that e has to be the edge of maximum weight on either path follows from the fact that we look for the minimum perturbation of edge f that would result in a negative e, f-exchange.

From Item 1 and Item 2 follows that the algorithm is correct.

2.3.4 Uncertainty from stochastic graphs

Edge weights in the supervoxel graph are computed from local features. Since the raw data are noisy, the edge weights are necessarily noisy as well. We accomodate this uncertainty by moving from deterministic edge weights to stochastic ones, which are distributed according to a probability distribution reflecting the noise. In contrast to [3, 92] who obtain a stochastic watershed by random perturbations of the seed positions, we keep the seeds fixed and instead randomize the edge weights. In particular, we define the stochastic watershed cut by replacing the original edge weights w_{ij} with $w'_{ij} \sim PD_{ij}$, where PD_{ij} is the weight distribution of edge (i, j) that can be modeled by multiplicative noise, for example.

Consequently, the hard label assignment x_i of node v_i in the fixed-weight watershed cut will be replaced by the probability of a label assignment $p_i(l), l \in L$ which depends on the edge weight distributions PD_{ij} of *all* edges (i, j).

Ideally, we would like to compute these probabilities exactly, but the analysis in the following section shows that no efficient algorithm for this problem exists. This is why we study an approximation.

2.3.5 #P-hardness of stochastic watershed cut problem

In this section we will reduce the so called $l - m$ network reliability problem [122, 17] which is #P-hard to a stochastic watershed cut problem. This shows that the stochastic watershed cut problem, i.e. to calculate the probability of the label assignment x_i for all nodes is also #P-hard. #P-hard is a complexity class introduced in [122]. While an NP-hard decision problem asks wether there is a solution to a given problem, the #P-hard problem is to count the number of solutions to a problem. This complexity class has been extended in [17] to allow the computation of a real number instead of the number of solutions to a problem. First we give a definition of the $l - m$ network reliability problem [17].

Definition 2.3.6 (l-m Network reliability problem) *The two terminal network reliability problem is defined on an undirected graph $G(E, V)$ with edge weights $w_{ij} \sim Bernoulli(p_{ij})$ where p_{ij} is the probability of the connection between i and j being active (whereas an inactive edge is equivalent to a non-existing edge). The two terminal network reliability problem is then to calculate the probability of the existence of a path between two nodes l and m.*

Intuition We reduce the $l - m$ network reliability problem to a stochastic minimum spanning tree calculation by constructing a new graph G' based on G. We add a new root vertex 0 to G' and introduce two label vertices -1 and -2. Graph G' is constructed in such a way, that when a connection between l and m exists in G node m is a child of vertex -1 in a MST of G' and a child of vertex -2 in a MST of G' if no connection exists in the original graph G.

The new graph $G'(V', E')$ contains all edges and vertices of the original Graph G. In addition a root vertex 0 is introduced which is connected with zero edge weight $w_{-1\,0} = 0, w_{-2\,0} = 0$ to the new label vertices -1 and -2. Node -1 is connected to node l' with zero edge weight $w_{-1\,l'} = 0$. Thess zero edge weights ensure that the corresponding edges are included in any MST of G'.

In addition the newly introduced vertex -2 is connected with weight $w'_{-2\,i'} = \alpha,\ \alpha > 1$ to all nodes $i' \in V'$ that are also present in G.

The edge weight distributions of the original edges $w_{i\,j}$ of G are modeled as

$$w'_{i'\,j'} \sim Bernoulli(1 - p_{ij}) * \beta, \beta > \alpha$$

Thus a random trial in G that removes an edge (i, j) corresponds to an edge with weight $w'_{i'j'} = \beta$ in G'. A random trial which leaves edge (i, j) intact in G induces an edge with weight

$w'_{i'j'} = 0$ in G'.

Lemma 2.3.7 *The probability of the node m' being a child of node -1 in the $MST(G')$ is exactly the probability of a connection between m and l in the original graph G. Thus the two terminal network reliability problem can be reduced to a stochastic watershed cut.*

Proof *Consider a random draw of the edge weights. First we consider the case that the realized graph induced by the trial leaves l and m* connected *in G. It is easy to see that in this case m' must be a child of node -1 in the MST of the corresponding realization of G', since any spanning tree in which m' is a child of vertex -2 must include an edge $w'_{-2\ i'} = \alpha > 1$, while the* connectedness *of l and m in the original graph G implies by construction that a path $P(l', m')$ in G' exists with edge weights $w'_{i'\ j'} = 0, \forall(i', j') \in P(l', m')$. Thus any MST in G' with root node 0 will have node m' and l' in a subtree of node -1 (which is by construction connected with zero edge weight to l').*

Secondly we consider the case that the realized graph induced by the trial leaves l and m disconnected. *It is also easy to see that in this case m' must be a child of node -2 in any MST of the corresponding realization of G', since any spanning tree connecting m' to node -1 must include an edge $w'_{i'j'} = \beta > \alpha$ since the* disconnectedness *in G implies that any $P(l', m')$ in G' includes at least on edge of such weight (by construction of the edge weights in G' which assigns weight $w'_{i'j'} = \beta$ when the bernoulli trial in the original graph G removes edge (i', j')). Thus any minimum spanning tree connects node m' to node -2 since this incurs the cheaper maximum cost of $w'_{-2\ m'} = \alpha < \beta$.*

We showed that any random trial that leaves l and m connected in G implies that m' is a child of node -1 in the $MST(G')$, while any random trial that disconnects l and m in G implies that m' is a child of -2 in the $MST(G')$.

Since the final probability is defined by the outcome of all possible trials and the outcomes are linked in the described way it has been shown that the $l - m$ terminal network reliability can be answered by calculating the probability for m' being a child of a newly introduced node -1 in a MST of a newly constructed Graph G'. This is a stochastic watershed cut problem.

Sampling scheme

Since computing the exact label distribution is infeasible, we propose to sample $t_{\max}$ complete graphs $G^t, t \in \{1, .., t_{\max}\}$ from the space of feasible graphs by sampling their edge weights w^t_{ij} from the independent probability distributions PD_{ij}.

For each randomly drawn graph G^t, the seeded watershed cut is computed by calculating the minimum spanning tree and assigning the node labels $x^t_i = l$ according to Definition 1.5.2. After all repetitions, the probability of node v_i carrying label l can be estimated as $p_i(l) = \frac{1}{t_{\max}} \sum_{t'=1}^{t_{\max}} \delta(x^{t'}_i, l)$. The final segmentation after $t_{\max}$ repetitions is defined as $x_i = \underset{l}{\operatorname{argmax}}\, p_i(l)$, i.e. x_i is assigned in a winner-take-all fashion to the label l to which node i was most often assigned during the trials.

The computational complexity of this sampling scheme depends linearly on the number of sampled graphs and the individual minimum spanning tree computations can be executed in parallel.

Stochastic uncertainty estimators

Uncertainty estimators based on the stochastic watershed cut can be defined in various ways, a natural one being the probability margin between the winning label $x_i = l$ and the one with the next highest class count, i.e. $m_i = p_i(l) - p_i(z')$ where $z' = \underset{\mathbf{l'} \neq \mathbf{l}}{\operatorname{argmax}}\, c_i^{l'}$.

However, this is only a local estimator of uncertainty, whereas critical edges should be characterized by their *non-local* effects. By combining the link instability according to Algorithm 2.3.2.1 with stochastic watershed cuts by accumulating the link instability of the edges over all t_{max} trials, a measure incorporating the influence of a single edge on the *global* segmentation can be obtained. This estimator is called *stochastic link instability*.

Algorithm 2.3.5.1 Stochastic segmentation instability

```
• Do $t_{\text{max}}$ times
    1. Construct graph $G^t$ by sampling $w_{ij}^t$ from $PD_{ij}$.
    2. Construct $MST(G^t)$ with root node $0$, and store the segmentation $\mathbf{x}^t$.
    3. Calculate and store the size of the subtree $h_i^t = |T^i|$ of each node $i$.
• Calculate $p_i(l) = \frac{1}{t_{\text{max}}} \sum_{t'=1}^{t_{\text{max}}} \delta(x_i^{t'}, l)$
• Calculate the final segmentation from the winning label for each node: $x_i = \underset{l}{\operatorname{argmax}}\, p_i(l)$.
• Calculate the cut set $C$ induced by $\mathbf{x}$.
• Aggregate for each node $i$ with edge $(i, j) \in C$, i.e. for all nodes $i$ touching the segmentation border, the size of the subtrees $h_i^t$ over all trials where the label $x_i^t$ differed from the winning label $x_i$: $H_i = \sum_{t':x_i^t \neq x_i} h_i^{t'}$
```

Finally we propose another non-local alternative. Algorithm 2.3.5.1 takes advantage of an important property of the randomization of edge weights: the changing segmentation boundary (cut set $C(T^t)$) that results from each trial t of the stochastic watershed cut. This effect can be incorporated into an uncertainty estimator which attributes the magnitude of the aggregated segmentation boundary movement throughout the trials to individual edges.

The intuition behind the *stochastic segmentation instability* measure H_i is that very unstable segmentation boundaries indicate ambiguity in the data and need user verification. The definition of H_i (Algorithm 2.3.5.1) ensures that nodes receive high uncertainty when their label differs

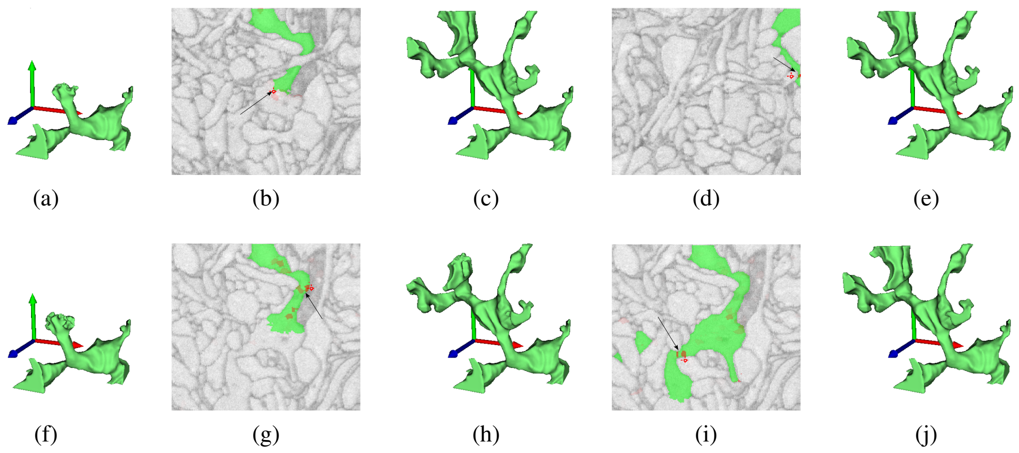


Figure 2.2: User guidance example for two of the proposed estimators. The top row displays the stochastic watershed using the stochastic segmentation instabiltiy estimator (Algorithm 2.3.5.1) , the bottom row the stochastic link instability estimator (Section 2.3.5). Displayed from left to right are the initial segmentation and two refinements based on seeding at the position of highest uncertainty (uncertainty is indicated by red color, the position of highest uncertainty by an arrow).

frequently from the winning label, or the affected subtrees are large. Nodes of highest criticality exhibit both problems.

2.4 Evaluation

2.4.1 Robot user

To evaluate the proposed uncertainty estimators objectively, we have designed an interactive segmentation robot [94]. The automaton tries to segment all neural processes in the data using two different seeding strategies. In the **ground truth strategy** the robot places two initial seeds, one inside the object of interest, one outside and then loops until convergence:

1. Calculate the set differences between ground truth and current segmentation.
2. Place a correcting single voxel seed in the center (maximum of the Euclidean distance transform) of the largest false positive or false negative region.
3. Re-run the segmentation algorithm with the new set of seeds.

Note that this segmentation robot requires knowledge of the complete three-dimensional ground truth for each iteration. This is clearly unrealistic, because if this knowledge were so readily available, then interactive segmentation would not be required in the first place.

The **uncertainty query strategy**, on the other hand, does not require full knowledge of the entire ground truth at each step. The robot begins by placing two initial seeds, one inside the object of interest, and one outside. It then loops until convergence:

1. Query the segmentation algorithm for the most uncertain region, using one of the confidence measures defined in Section 2.3.

2. Query the ground truth for the true label at the corresponding position.

3. Place a suitable seed at that position and re-run the segmentation algorithm.

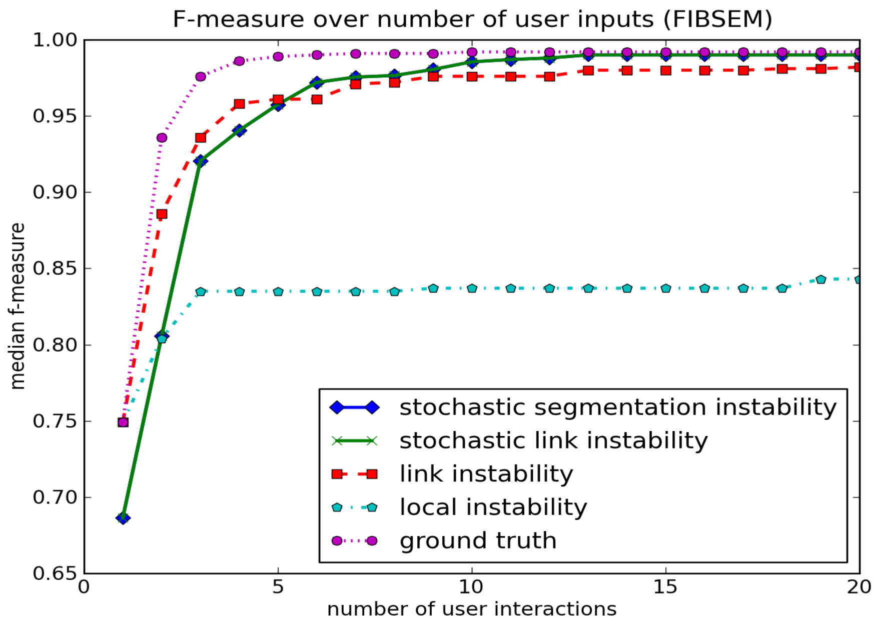
2.4.2 Experiments

To evaluate the proposed uncertainty estimators for the seeded watershed cut we compare the estimators on a 3D segmentation problem from the neurosciences in a user guided segmentation setting. The nearly isotropic and densely annotated ground truth data is a subset of $400 \times 200 \times 200$ voxels from a 2000^3 volume of neural tissue acquired with a serial blockface electron microscopy (SBEM [38], Figure 2.1) and a $900 \times 450 \times 450$ densely annotated subset from a 2000^3 volume acquired with focused ion beam electron microscopy (FIBSEM [67], Figure 2.1). The reconstruction of the neural processes in this tissue is a segmentation problem that exhibits many properties that make it suitable for the seeded watershed cut [114].

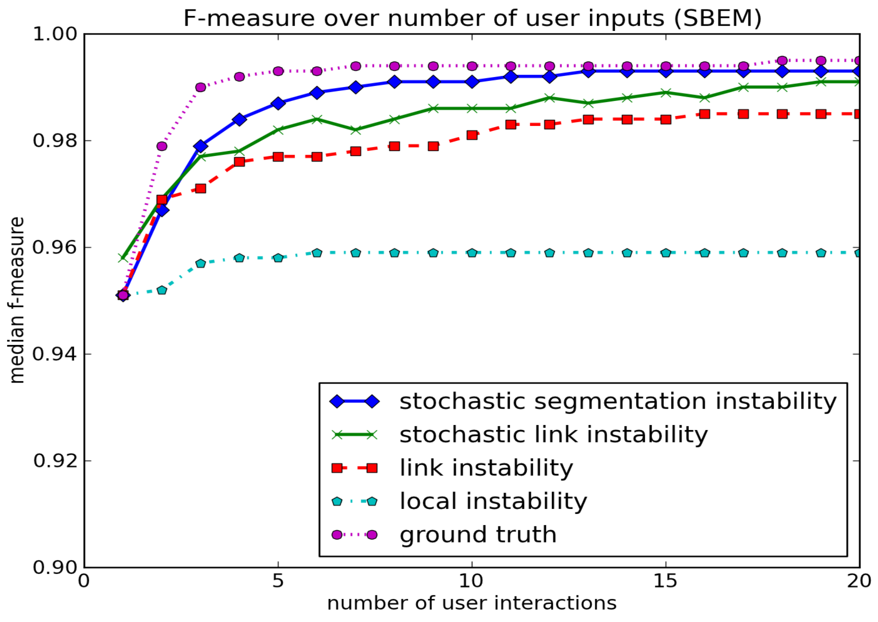
We have tested all proposed uncertainty estimators with the *uncertainty query strategy* of the robot user against the *ground truth strategy*, which can be seen as an upper bound labeling strategy. During the segmentation process we recorded the resulting segmentation f-measure after each additional seed to compare the convergence rate of the robot for the different uncertainty estimators. Figure 2.3 shows the median across all neural processes in the respective ground truth. The parameters, namely the *bias* of the background seed (a background seed preference, [114]) and the amount of *perturbation* β in the case of the estimators based on the *stochastic watershed cut* were determined by a grid search over a training set consisting of 10% of the neural processes. For simplicity, the edge weights for the trials t were sampled as $w_{ij}^t \sim \text{Unif}(w_{ij}, (1.0+\beta) * w_{ij})$. These edge weight distributions are simplistic, and even better results may be obtained when using more appropriate distributions. Figure 4.5 b displays the averaged standard deviation for $p_i(l)$ over 100 runs of the stochastic watershed cut with different trial counts t_{max}. While the average standard deviation does not converge for the trial counts considered here, Figure 4.5 a indicates, that the number of randomly sampled graphs has virtually no effect on the predictive quality of the two proposed stochastic uncertainty estimators, already a trial size of $t_{\text{max}} = 5$ can be used for successful user guidance. Thus our estimators incur only a small additional computational overhead compared to the standard watershed cut, which can be calculated in quasi linear time [28] (inverse ackerman complexity).

2.5 Conclusion

We have presented and evaluated several novel uncertainty estimators for the seeded watershed cut. The proposed estimators are based on a perturbation principle and stochastic edge weights, respectively. The proposed estimators were evaluated on a 3D biological neuroimaging application example that can profit from good uncertainty estimates and which exhibits many properties that make the seeded watershed cut a suitable algorithm. We showed that the proposed non-local uncertainty estimators yield a tremendous improvement in the number of user interactions compared to a simple local margin based approach which fails to query for more informative labels after the first few iterations. The proposed non-local estimators yield segmentation improvements that come close to an error correction strategy that relies on complete knowledge of the ground truth while incurring only an insignificant overhead compared to a standard watershed cut.

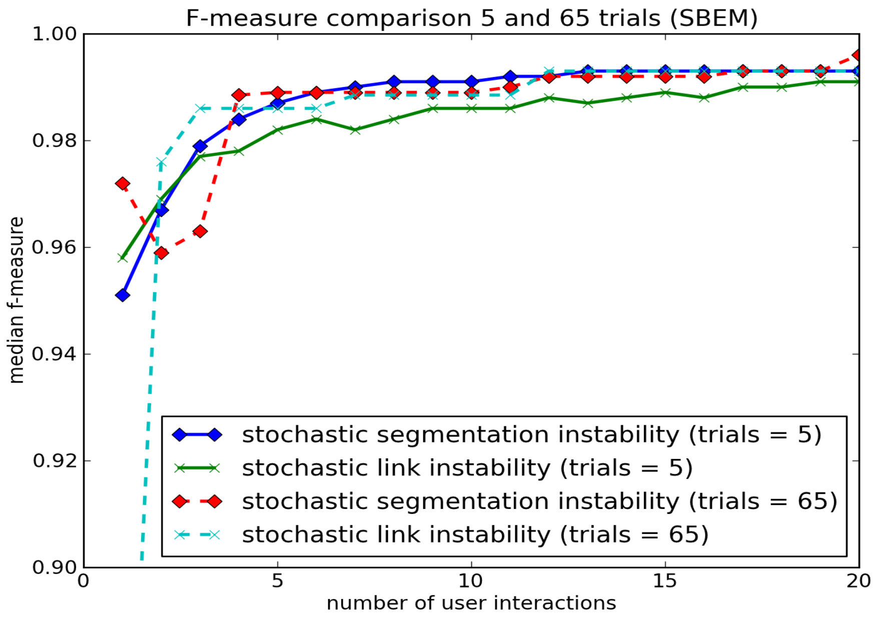


(a)

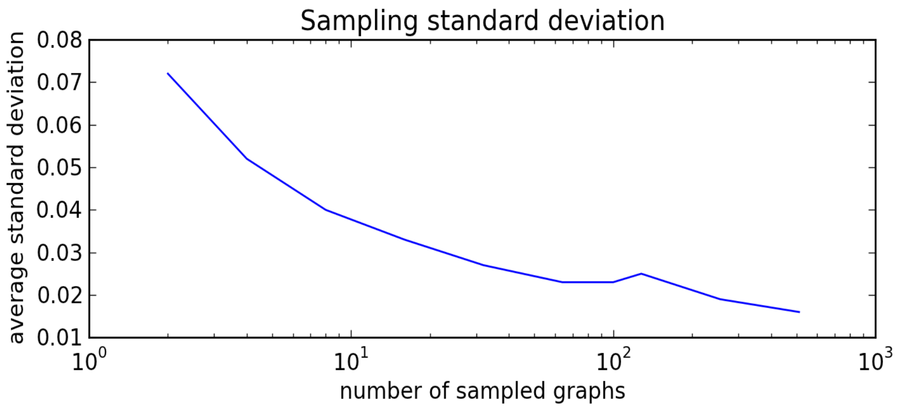


(b)

Figure 2.3: Median F-measure over number of user interactions using the different uncertainty estimators as the query strategy for the segmentation robot for the two different datasets. (a) FIBSEM. (b) SBEM.



(a)



(b)

Figure 2.4: (a) Median F-measure over number of user interactions using 5 and 65 randomly sampled graphs. (b) Averaged standard deviation for $p_i(l)$ using 100 runs of the stochastic watershed over the number of randomly sampled Graphs along the x-axis.

Chapter 3

Enumerating the k best segmentation changing spanning trees

3.1 Introduction

The most popular algorithms for interactive segmentation are Ising type Markov random fields (MRFs) [18], the random walker [50] and the seeded watershed [91]. Their smoothness terms penalize label changes with the L_1,L_2 and L_∞ norms respectively [32]. However, in the process of finding the single lowest energy solution to the graph partitioning problem a lot of information is lost and other modes of the solution space which may convey important aspects of the problem are ignored. An enumeration of more than one low energy solution allows to obtain a more robust segmentation, and can help defining an uncertainty of the resulting segmentation which may be used in downstream processing. Thus, recent work on these algorithms has focused on finding the M lowest energy solutions [45, 95, 130], ideally subject to a diversity constraint [10]. These references solve the problem for Ising-type MRFs. However, systematic empirical studies [114] show that the seeded watershed outperforms MRFs in certain datasets and offers computational advantages. The near linear runtime of a seeded watershed stems from its connection to the minimum spanning tree (MST) of the image graph [90, 35]. The present work presents the first viable algorithm that provides analogous M-best results for the seeded watershed cut. We build on the seminal work of Gabow [47] to enumerate only those spanning trees (ST) in an edge-weighted graph that lead to a change in the resulting segmentation. Furthermore we give a modification of Gabow's algorithm that allows to enumerate the M-best diverse solutions similar to [10], by enforcing a user specified distance between some of the generated segmentations. Such a diverse set of solutions can in turn be combined into a final segmentation [21].

This chapter is based on the publication [115].

3.2 Related work

The M-best solutions problem has been studied in the context of discrete graphical models [15] where it is known as the M-best MAP problem. Several types of algorithms have been proposed for the M-Best MAP problem: junction tree based exact algorithms [95, 106], dynamic programming based algorithms [105] and max marginal based algorithms [130]. An interesting extension of the M-best MAP problem was proposed and studied in [10]. Here, the authors give an algorithm to enumerate a *diverse* set of solutions. This is an attractive approach since the M-best solutions tend to be very similar to the MAP solution for a low M and thus important aspects of the solution space cannot be found. Generating such a set of diverse solutions was shown in [10] to remedy the problem of M-best solutions: when the initial M-best solutions are too close, important aspects of the solution space may only be found by enforcing a certain amount of diversity. The authors show that this diverse set of solutions which differ in a user specified amount from the MAP solution do not exhibit this problem.

The seeded watershed algorithm [91, 124] which we adapt enjoys great popularity in applications where large amounts of data have to be processed, including medical and biological 3D image analysis [114], and where the shrinking bias typical of MRFs is detrimental.

We rely on the equivalence of the edge weighted seeded watershed and the minimum spanning tree (MST) algorithm [90, 35, 42] and draw on the seminal work of Gabow [47] who solved the k-smallest minimum spanning trees problem.

3.3 Image segmentation with minimum spanning trees

As explained in Chapter 1 we formulate the interactive image segmentation problem as a graph partitioning problem on the pixel neighborhood graph $G(E, V)$. All neighboring pixels $v \in V$ are connected with edges $(i, j) \in E$. All edges have an associated edge weight $w_{ij} \in \mathbb{R}$ which expresses the dissimilarity between the neighboring pixels. The edge weights w_{ij} can be computed for example from the color gradient or another suitable boundary indicator. It is well known [90, 35] that the seeded watershed cut on a graph G is equivalent to a minimum spanning tree computation on a suitably augmented graph $G'(V', E')$ that contains a supernode v_0 connected to seed nodes v_{-l} for each label class $l \in L$. These seed nodes are connected to the root node v_0 with zero weight edges. All labeled nodes (i.e. all supervoxels holding a user seed) are also connected to these seed nodes with zero-weight edges $w_{i,-l} = 0$, which are guaranteed to remain in the MST. Once the MST with root node v_0 has been constructed, subtrees originating from seed nodes v_{-l} form segments of the final segmentation. This graph construction is illustrated in Figure 1.2.

3.4 Gabow's algorithm for the k smallest spanning trees

The MST segmentation algorithm outlined in the previous section finds a single smallest spanning tree of the augmented graph. In the next section, we will propose to generalize the algorithm by Gabow [47] to enumerate spanning trees that result in *different* segmentations. To lay the foundation for our extension, we start with a description of Gabow's original algorithm.

Gabow's algorithm starts with a minimum spanning tree for the graph generated by e.g. Kruskal's algorithm. This MST constitutes the first solution. The algorithm then enumerates different spanning trees in the order of increasing weight by swapping out an edge e belonging to the current spanning tree, and replacing it by another edge f which is currently not in the tree.

To obtain the smallest spanning tree under such a so-called e, f-exchange, it finds the pair e, f that gives the smallest weight increase $w(f) - w(e)$.

The main idea of the algorithm is to maintain a set of branchings and two lists associated with each branch, called IN and OUT, which prevent the algorithm from enumerating a spanning tree twice. All edges contained in the IN list have to stay in the spanning tree and all edges contained in the OUT list cannot enter the spanning tree.

To enumerate all spanning trees in order, the algorithm finds the smallest weight e, f-exchange which is feasible according to the IN and OUT lists of the current state and branches on this exchange. Branching is done by considering two different cases: one branch is constructed by adding the f edge to the OUT list, the other branch is established by adding f to the IN list. Any further branching which is executed in the two cases inherits the respective IN and OUT lists from its parent state. Thus, any spanning tree constructed in the first branch excludes edge f and any spanning tree constructed in the second branch includes edge f. By visiting all branchings strictly in the order of increasing tree weight, the first k minimum spanning trees are constructed in the correct order. This process is illustrated in Figure 3.1.

3.5 Enumerating changing segmentations

While Gabow's algorithm finds the k smallest spanning trees of a given graph, it cannot be used to find the different modes of a segmentation: a grid graph has exponentially many spanning trees (10^{40} for a 4-connected grid graph of size 10×10, [119]). In typical images, exceedingly many spanning trees lead to the same segmentation. This effect is visible in the illustration of Gabow's algorithm in Figure 3.1: while the algorithm always produces a new spanning tree, the associated segmentation does not necessarily change. This is especially true when there are large basins, or areas with low edge weights, as are typical for graphs constructed from natural images.

Thus, when generating the k smallest spanning trees, many if not all (when k is small) of the trees correspond to the same segmentation result. Luckily we can identify the sufficient and necessary condition that leads to a different segmentation result (compared to the previous state) in an e, f-exchange in Gabow's algorithm.

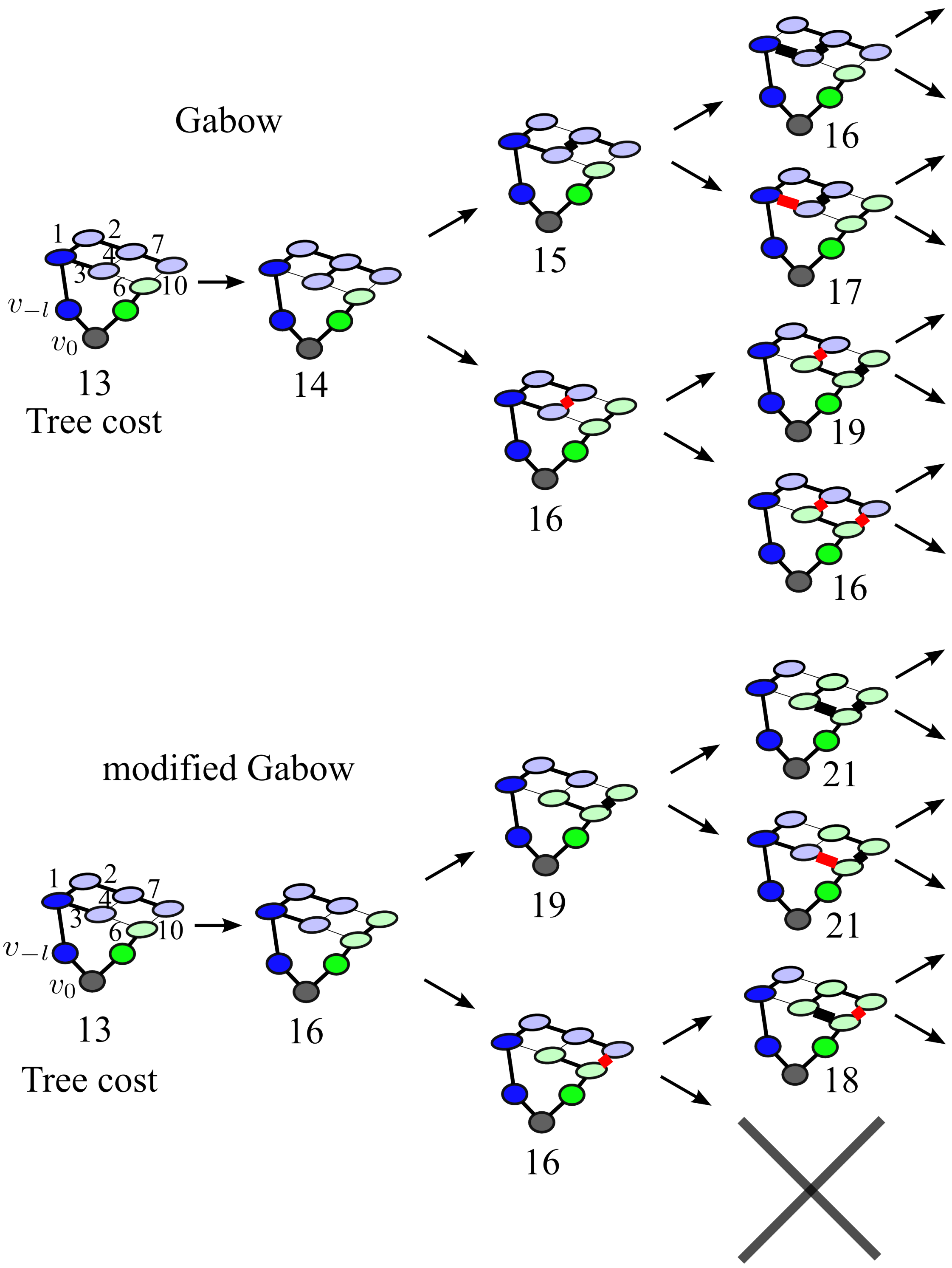


Figure 3.1: Illustration of Gabow's algorithm and our modification (best viewed in color). The algorithm of Gabow branches after each e, f-exchange into an upper case where the edge f (indicated by thick black stroke) must stay in the tree and a lower case where the edge f (thick red stroke) must stay out of the tree. Our modified algorithm works in the same way, but only considers edges $f \in C(ST)$ which are part of the cut set for the current spanning tree. By definition, this induces a changed segmentation in each step. In contrast, in the original Gabow algorithm the segmentation often stays unchanged. Some of the k smallest spanning trees have a cut set fully contained in the OUT list and hence do not allow further branching: the set of viable edges for an exchange has been depleted (crossed out state).

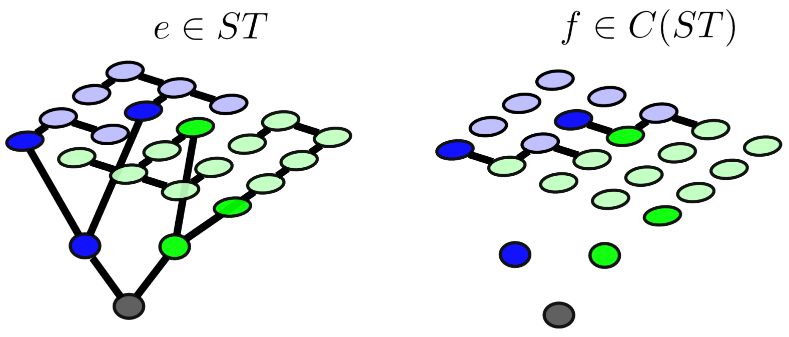


Figure 3.2: Illustration of cut edges. On the left, all edges of a spanning tree (ST) are shown in bold. In the middle all edges not belonging to the spanning tree a shown. The subset of the middle edges which belong to the seeded watershed cut – i.e. the edges which connect the different segments – are shown on the right. We modify Gabow's algorithm by only considering these cut edges in any e, f-exchange. This enforces a changing segmentation between any two STs in the hierarchy of Figure 3.1, but does not guarantee that all resulting segmentations are unique.

In the case of a spanning tree segmentation, the assigned label (color) of a node depends on the subtree to which the node is connected in the spanning tree: the node is assigned the label of the virtual seed node v_{-l} of which it is a child in the spanning tree (see Figure 1.2). All edges in the spanning tree connect nodes of the *same* color. An edge connecting two nodes of *different* color cannot be part of the spanning tree segmentation.

Now, if an e, f-exchange removes an edge e from the tree and replaces it with an edge f that connects two nodes of the same color it is clear that the segmentation cannot change: the resulting spanning tree is merely a different way to express the same segmentation.

We now come to the core idea: to enforce a different segmentation, the edge f that is swapped in has to connect two nodes of *different* colors. After swapping in the edge f, both nodes belong to the same subtree and thus one of the two nodes changes its color. The resulting segmentation is different.

We call these edges f that connect nodes of different color in a ST segmentation "cut edges" $f \in C(ST)$. See Figure 3.2 for an illustration.

At this point, we are able to modify Gabow's algorithm to not only enumerate different spanning trees in order of increasing weight, but to enumerate *changing* segmentations in the order of increasing weight.

A small change is sufficient to reach the desired behavior: by adding all edges f which are not part of the cut set to a list OUTC $= \{f : f \notin C(ST)\}$, Gabow's algorithm can only consider edges $f \in C(ST)$ for any e, f-exchange since all edges in the OUT and OUTC list are not eligible. Thus the segmentation of any spanning tree that is generated differs from the previous segmentation. The OUTC list is always updated once a new spanning tree has been generated and ensures the desired behavior.

Our modification of Gabow's algorithm does not change its computational complexity, the scanning of the edges and the calculation of the OUTC list take time proportional to the number of edges in the graph which is of the same order as the normal Gabow algorithm takes in each iteration.

3.5.1 Algorithm correctness

Our algorithm may enumerate a segmentation (though not a spanning tree) more than one time and in that sense is an approximation. However, our modification of Gabow's algorithm generates all possible segmentations in their order of increasing weight and finds the lowest cost spanning tree that represents each segmentation. We first show that any spanning tree of minimum weight that represents a segmenation is found by our algorithm.

Definition 3.5.1 *a minimum spanning tree S of a given segmentation can be found by computing the cut edges $C(S)$ of the desired segmentation: it contains all edges $c = (i, j), x_i \neq x_j$ that connect nodes of different color $x_i \neq x_j$. The minimum spanning tree S of this segmentation can then be found by removing these edges from the graph (or adding them to the OUT list) and computing the minimum spanning tree of the modified graph.*

A set of constraints IN_S, OUT_S that induce the minimum spanning tree S for a given segmentation can be found easily: the largest such set is $IN_S = \{e : e \in S\}, OUT_S = \{e : e \notin S\}$. Computing a spanning tree obeying these constraints will produce S. This shows the existence of at least one such set of constraints.

Theorem 3.5.2 *Let T be a minimum spanning tree of graph G and let S be the minimum weight spanning tree that induces a given segmentation. Let IN_S, OUT_S be a set of constraints that induce spanning tree S. Then there exists a series of branchings that leads to IN_S, OUT_S.*

Proof Let $C(T)$ be the cut edges of spanning tree T. Then any edge $f \in C(T)$ which is eligible for an e, f-exchange in our algorithm is either (a) $\notin S$ or (b) $\in S$. The algorithm now picks the $f \in C(T)$ that has the smallest exchange weight $w(f) - w(e)$ using their best respective exchange partner e. In case (a) if $f \notin S$ we follow the branch where f is added to the OUT list. This leads to a new state that has the same induced segmentation (and cut set) but a modified OUT list. In the new state the next best exchange pair e', f' with $f' \in C(T)$ will be considered and the same case consideration applies. In case (b) when $f \in S$ we follow the other branch by adding f to the IN list. This leads to a new state with a new segmentation (and new cut set), but starting from this new state the same consideration applies. Both cases (a) and (b) lead to a state which is closer to a set of constraints IN_S, OUT_S. After a finite number of branchings we end up with a set of constraints IN_S, OUT_S that define the minimum cost spanning tree that induces a given segmentation.

The proof shows that all segmentations that are representable by a spanning tree will be found by our algorithm, forbidden branchings as in Figure 3.1 do not pose a problem: they occur when the OUT list contains all edges in the current cut set. By construction of the proof, for all edges

$f \in OUT$ we know $f \notin S$. Since the current cut set is fully contained in the OUT list no edge in the current cut set can be part of the segmentation inducing spanning tree S. But if no edge of the current cut set is $\in S$ then the current state already defines the correct segmentation.

Definition 3.5.3 *[47]: T-exchange. Let T be a spanning tree of graph G. A T-exchange is a pair of edges e,f where $e \in T$, $f \notin T$, and $T - e \cup f$ is a spanning tree.*

Lemma 3.5.4 *[47]: A spanning tree T has minimum weight if and only if no T-exchange has negative weight.*

Theorem 3.5.5 *Let T be a minimum spanning tree of graph G and let f be an edge not in T. Let e, f be a T-exchange having the smallest weight of all exchanges e', f. Then $T - e \cup f$ is a minimum weight spanning tree of graph G under the constraint that f is in this tree.*

Proof The proof is similar to the proof of Theorem 2 in [47]. Let $S = T - e \cup f$. Suppose S does not have minimum weight. By Lemma 3.5.4, there is a S-exchange g, h having negative weight. We derive a contradiction below. Let $T - e$ consist of the two trees U, V. Edge e joins U and V. Edge f must also join U and V since e, f is a T-exchange which produces a valid spanning tree. Edge h also joins U and V. For if not, assume without loss of generality that h joins two vertices in U. Since g, h is an S-exchange g is also in subtree U and thus g, h is also a T-exchange. Thus, by Lemma 3.5.4 it has positive weight which violates our assumption, thus h must join U and V. Edge $g \neq f$ is in $U \cup V$ since exchange g, h cannot remove edge f from S due to the constraint. Assume without loss of generality that $g \in U$. Now let $U - g$ consist of the two trees W, X. Edge e is incident to one of those trees, say W. Since $T - e - g \cup f \cup h$ is a spanning tree, either f or h is incident to X (since edge h must join U and V). If h is incident to X then g, h is also a negative a T-exchange which would violate Lemma 3.5.4 and leads to a contradiction. If f is incident to X then g, f is also a T-exchange, but since $w(g) \leq w(e)$ (since by construction we picked the smallest possible e, f exchange, i.e. the largest eligible weight edge e) this would imply the T-exchange e, h being negative since $w(h) \leq w(g) \leq w(e)$: also a contradiction.

Induction and Theorem 3.5.5 gives us the property that our algorithm always produces spanning trees of increasing weight. No permitted e, f-exchange during the execution of the algorithm is of negative weight.

From Theorem 3.5.2 we have that the algorithm produces any segmentation at some point. In combination with Theorem 3.5.5 we have that our algorithm produces all possible segmentations in the order of increasing weight.

3.6 Enumerating diverse segmentations

When enumerating the M-best solutions and choosing a sensible M, say 50, there is a certain danger that the returned solution set is very similar and only differs marginally from the lowest energy solution. To remedy this problem in the M-best MAP setting the authors in [10] propose to enumerate diverse solutions S that obey a given minimum distance $\Delta(MAP, S)$ to the MAP

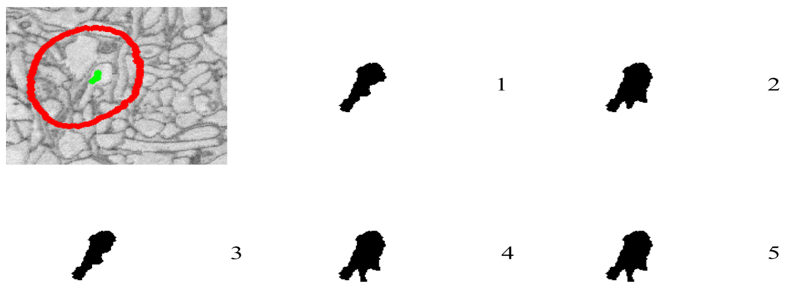


Figure 3.3: Modified Gabow example. The top-left image shows an electron micropscopy image of cells in neural tissue [22] and user given seeds (red,green). The $k = 1, 2, 3, 4, 5$ first segmentations generated by our modified Gabow algorithm are shown in black and white. The algorithm successfully finds different modes of the segmentation.

solution. We now show that a similar constraint can be incorporated into our algorithm without increasing computational complexity. We will modify our algorithm that enumerates changing segmentations in such a way that it returns a changing segmentation which differs in at least Δ nodes from the previous segmentation.

The core part of our modified Gabow algorithm from the previous section is the e, f-exchange with a restriction that only allows edges f from the current cut set to be moved into the tree. This restriction of f enforces that at least one node changes its color because it is attached (via edge f) to a differently colored subtree. The exact number of nodes that change their color depends on the edge e which is removed from the tree: all nodes and edges below edge e are reconnected by edge f to another subtree with different color. Thus, by also restricting the edges e for an e, f-exchange we can control how many nodes will change their color in that exchange.

The exact implementation is straightforward: in each step of our modified algorithm, we compute for each edge e the number of nodes $\#(e)$ below this edge. This can be done in time linear in the number of edges. Then, we add all edges e which do not fullfill the user specified diversity requirement Δ to an IND list: IND $= \{e : \#(e) < \Delta\}$. The algorithm is adapted to use both lists, IN and IND. Thus, the IND edges are not considered for an e, f-exchange in the current iteration since they must stay in the tree and any eligible e, f-exchange must change the color of at least Δ nodes. The IND list is updated in each iteration of the algorithm.

3.7 Experiments

The proposed algorithm is a heuristic because it may produce multiple spanning trees that induce the same segmentation, see section 3.5.1. To study how many unique segmentations are generated, we choose a segmentation task that is suitable for a minimum spanning tree segmentation [114]: the segmentation of single cells in neural tissue. For each out of 10 neural processes we

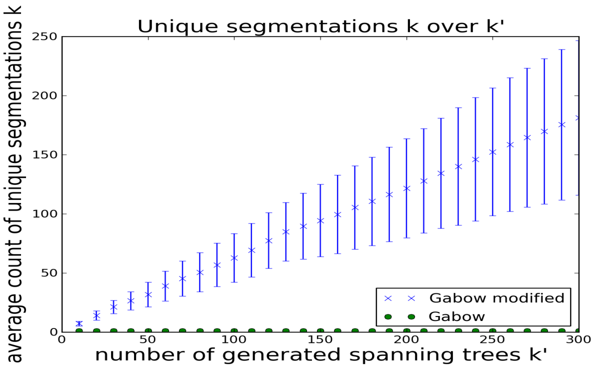


Figure 3.4: This plot shows how many unique segmentations are generated when varying the parameter k' that determines how many spanning trees are generated. Shown are the results for Gabow's original algorithm and for our modified version. Gabow's original algorithm fails to generate different segmentations: the spanning trees differ, but the induced segmentation is always the same. In contrast, in the proposed algorithm, more than half of the generated spanning trees induce a unique segmentation.

ran Gabow's original algorithm and our modified version and compared the induced segmentation of each generated spanning tree to all previous generated segmentations to see how many unique segmentations each of the algorithm generates. As can be seen in Figure 3.4, Gabow's original algorithm fails to generate even two unique segmentations in $k' = 300$ iterations. Each generated spanning tree differs, but all spanning trees induce an equivalent segmentation. The same effect can be observed in the toy example in Figure 3.1. Our modification ensures that each generated spanning tree induces a different segmentation compared to the previous state. It works well in practice: more than half of the generated MST's induce a unique segmentation.

3.8 Conclusion

Recently a way to enumerate the diverse M-best solutions for Markov random fields has been proposed in [10]. We present an algorithm that allows to enumerate locally changing segmentations in order of increasing spanning tree weight. We prove that the algorithm finds the smallest weight spanning tree that represents a given segmentation. We experimentally validate the algorithm and show that it can be used to effectively enumerate different smallest spanning tree segmentations. Furthermore we show how a diversity constraint can be incorporated into the algorithm that allows to enumerate segmentations which differ in a user specified number of nodes. We expect the proposed algorithm to be of value in the pursuit of meaningful uncertainty measures as well as user guidance in a 3D segmentation setting. We also trust that more robust segmentations can be obtained by taking into account the information contained in a diverse set of solutions.

Chapter 4

Image partitioning using structured decision trees

4.1 Introduction

This chapter describes a new method to learn *edge* models from sparse user scribbles marking *regions* and is based on the publication [112]. Consider Figure 4.1 and assume that we want to segment each kiwi. Normally, scribbles as shown on the left would be used to learn region appearance models that can then serve as potential functions in energy-minimizing segmentation methods such as graph cuts. However, this does not work here because the individual objects are indistinguishable by region appearance. Another popular approach would use the scribbles as seeds for a suitable region growing algorithm such as a seeded watershed or random walker. However, such an approach would need a seed or scribble for each and every object.

When region appearance alone is not informative, segmentation must be based on an edge model. Our ambition is to train such a model with minimal labeling effort on the user's part. Traditional learning methods require the user to place edge scribbles exactly on the desired edges. The required localization accuracy makes this a time consuming task. Section 4.1.1 describes recent proposals for a simplified edge labeling. In contrast, we strive to use cheap region scribbles like in Figure 4.1 to train edge models instead of the usual region models.

Clearly, region scribbles cannot be used for edge learning directly because they are typically located far away from edges. However, they provide a large number of constraints that can control edge learning indirectly:

- Each pair of pixels from the same scribble defines a *must-link* constraint, i.e. there must be at least one connecting path that does not cross any edge.

- Pixels from different scribbles define a *cannot-link* constraint, i.e. any connecting path must cross at least one edge.

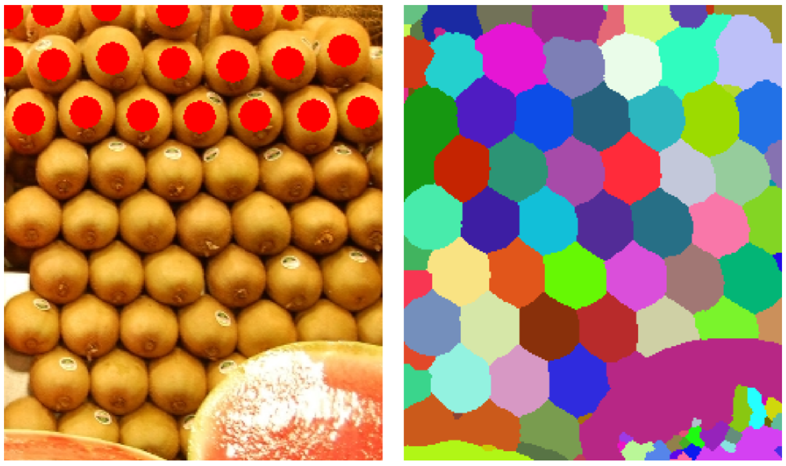


Figure 4.1: Segmentation example. Each connected component of the user scribbles is treated as an individual label and the decision tree is trained using must-link constraints inside each component and cannot-link constraints between label components. The resulting tree learns an edge model consistent with the user-provided constraints and successfully generalizes to the unlabeled part of the image, where it segments many objects successfully.

We call this a “link-or-cut edge learning” problem. It turns out that these constraints contain sufficient information for successful training of an edge model, and we propose a structured decision tree-type algorithm to solve this problem.

Since the training data does not contain direct edge annotations, the training error cannot be defined as the fraction of mis-classified pixels or edges. However, such a simplistic criterion is unsuitable for segmentation quality assessment anyway [121, 56]: It cannot penalize the global consequences (big changes in the resulting connected components) that may be caused by local errors such as a fine gap in an object contour. Instead, we define the training error in terms of a clustering quality score similar to the Rand Index (eq. 4.1), which can be directly derived from the pairwise constraints. This has profound consequences for the learning algorithm: Local decisions should be conditioned on the state of the *entire* segmentation, and our new learning algorithm reflects this requirement.

The proposed algorithm recursively builds a decision tree that predicts the state of each edge of the image graph. During tree construction we use a non-local split criterion which takes into account the global connectivity consequences of local edge predictions.

To summarize, the proposed algorithm relies on cheap must-link and cannot-link annotations and has the following virtues:

- it requires no region appearance terms,
- the number of segments need not be specified in advance,

- weak supervision in terms of sparse annotations is sufficient and no explicit edge labels are needed,
- the training optimizes a global clustering score in a decision tree.

To the best of our knowledge, this is the first time a decision tree is trained using a non-local structured split criterion.

4.1.1 Related work

Previous work on edge learning includes many approaches which are based on training data with exact boundary localization, e.g. [70, 135, 69, 86, 41]. The method presented in [99] learns an optimal edge labeling policy based on context and gestalt features, but also requires dense ground truth. An interesting approach using weaker supervision is the livewire method in [8] which snaps a path to the most probable boundary predicted by a classifier which is trained online. Another approach using weaker supervision is presented in [6]. The author learns an edge model from inaccurate boundary annotations.

Small errors in the boundary predictions have large global consequences when calculating the connected components. To avoid this some authors introduce higher order constraints (e.g. boundary closedness) to obtain a consistent segmentation [65, 1, 64]. A novel take at edge learning based on a non local clustering quality measure is used in [118] to learn a neural network. A non-local warping error that takes topological constraints into account is also proposed in [56] but the authors also use a dense labeling during neural network training.

The decision tree based edge learning algorithm that we propose is related to [96]. The authors learn edge on/off probabilities using a decision tree, but the method requires a dense labeling as input and does not take the effect on the connected components of the graph into account – it acts locally. Our special split criterion is inspired for example by [59] where a special loss function in the split nodes of a decision tree is optimized. But in contrast to our approach their objective is local, as is the case in [58]. The authors of [71] have introduced a decision tree algorithm that works on locally structured labels. We build on this idea and extend it to a structured loss function on a complete image graph. In [72] a decision tree is proposed whose intermediate learning state is used as a feature for further tree growing. This is the basis for our proposed algorithm which evaluates a structured split criterion with regard to the intermediate tree state.

The must-link and cannot-link constraints that we use to train our learning algorithm have also been used by [53, 132]. These algorithms partition an image graph into connected components using said constraints. Both partitioning algorithm use a single scalar value associated with each edge whereas our method can take a multitude of edge features into account and learns an edge model from the given constraints. The same type of partitioning constraints has been considered in [54] where the authors introduce must-link constraints in the context of the normalized cut algorithm. In the context of the image foresting transform cannot-link constraints have been investigated in [81]. To summarize, must-link and cannot-link constraints haven been investigated in the literature, but our approach is novel since we use these constraits for weakly supervised boundary learning.

4.2 Problem definition and objective function

We consider a segmentation problem defined on a graph $G(\mathcal{E}, \mathcal{V})$ in which the nodes $n_i \in \mathcal{V}$ correspond to the pixels of an image, and the edges $(i, j) \in \mathcal{E}$ correspond to the pixel neighborhood of the image. We assume a suitable set of edge features w_{ijf} (such as color gradients or structure tensor eigenvalues on different scales) is available and can be attributed to each edge (i, j). In addition we are given a sparse constraint matrix $C \in \{-1, 0, 1\}^{N \times N}$ that defines whether a pair of pixels (i, j) must be in the same component ($C_{ij} = 1$), or in two different components ($C_{ij} = -1$). The decision variables $x_{ij} \in \{0, 1\}$ determine whether an edge (i, j) is removed from ($x_{ij} = 0$), or remains ($x_{ij} = 1$) in the graph G. The objective function $F(\mathbf{c}, \mathbf{x})$ which we seek to maximize depends on the set of constraints $\mathbf{c}$ and the connected components or partitions $\pi(\mathbf{x})$ implied by the binary edge indicator variables x_{ij}:

$$F(\mathbf{c}, \pi(\mathbf{x})) = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TP + FP)(TN + FN)}} \tag{4.1}$$

where $TP(\mathbf{c}, \pi(\mathbf{x}))$ is the number of pairs of pixels which are correctly (according to the constraints $\mathbf{c}$) assigned to the same component and $FP(\mathbf{c}, \pi(\mathbf{x}))$ is the number of pairs of pixels which are incorrectly assigned to the same component. $TN(\mathbf{c}, \pi(\mathbf{x}))$ and $FN(\mathbf{c}, \pi(\mathbf{x}))$ are the number of true negative and false negative pairs respectively. Equation 4.1 is known as Matthews correlation coefficient [87].

Thus, in the optimum $\hat{\mathbf{x}} = \operatorname{argmax}_{\mathbf{x}} F(\mathbf{c}, \pi(\mathbf{x}))$, the binary indicator vector $\hat{\mathbf{x}}$ corresponds to a partitioning of the graph into a set of connected components $\pi(\mathbf{x})$ that satisfy the constraint matrix. Note that the number of connected components defined by $\pi(\mathbf{x})$ can be larger than the number of components defined by the constraints $\mathbf{c}$: the unconstrained pixels of the image can be partitioned in many different ways.

4.2.1 Overview: global optimization using decision trees

The objective function $F(\mathbf{c}, \pi(\mathbf{x}))$ defines a *global* objective which depends on the structure of the graph G. Maximizing F may be a simple task when the constraint set is very small: many different possible edge assignments $\mathbf{x}$ may partition the graph correctly. But depending on the size of the constraint set and the structure of the graph the problem can become computationally infeasible. The objective function is highly non-smooth and non-convex: switching a single binary indicator variable x_{ij} – for example on the boundary between two objects – may have a huge influence on the value of F.

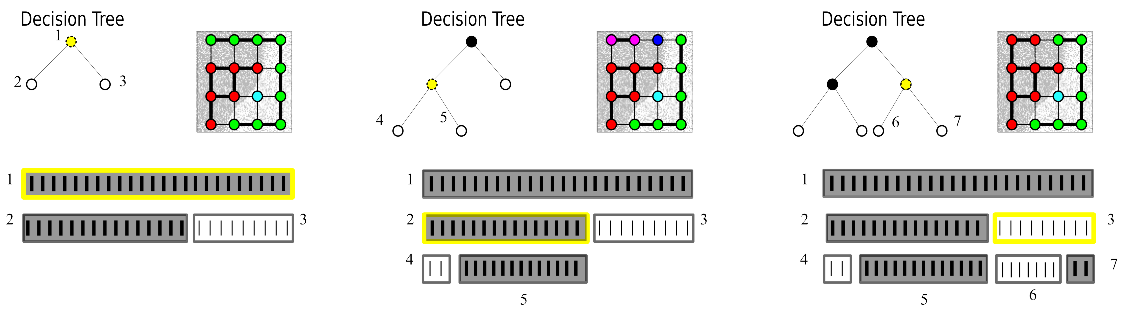


Figure 4.2: Illustration of our decision tree building process. Starting at the root node (1) the edges are partitioned into two sets, one of which is assigned $x_{ij} = 1$ (thick, edges stays in the graph) while the edges of the other partition are assigned $x_{ij} = 0$ (thin, edges are removed from the graph). In the next steps this initial decision on the edge states is revised by partitioning the two edge sets recursively further into an on and off set such that the objective function $F(\mathbf{x}, \mathbf{c})$ is maximized. The value of the objective function depends on the connected components of the graph G that are induced by the edge state defined in the leaf nodes. The connected components are distinguished by the node colors.

In contrast to existing approaches which use classifiers to learn local pixel class probabilities or which learn pairwise boundary probabilities [96] from strong edge/no-edge examples, our aim is to *learn from weak labels*: the learner will be trained only from the *constraint set* $\mathbf{c}$ derived from sparse user scribbles and the *structure of the problem*, the pixel neighborhood graph G.

We have chosen to optimize Equation 4.1 using a greedy method inspired by traditional decision trees. As explained in Section 6.3 decision trees are constructed in the following fashion: Given a set of examples S_i associated with a decision tree node i, a tree is built starting at the root node 0 by partitioning the set of examples into two subsets S_L and S_R. The decision how the set S_i is partitioned depends on the parameters θ_i for node i and the features of the samples. These parameters are obtained by optimizing a split criterion function $\hat{\theta}_i = \text{argmax}_{\theta_i} CF_i(S_L, S_R, \theta_i; S_i)$. Examples of such split criteria CF include the *Gini-Impurity* or the *Information Gain*. Important is the *purely local dependency* of the split functions that are usually used – the function which is optimized only depends on the split parameters θ_i of the node i that is optimized and the set of training examples S_i over which this node optimizes and their associated features.

We however propose to optimize a *global* function at each split node. In other words, we condition the split criterion function CF_i on the parameters $\theta_0, ..., \theta_{i-1}$ of all other split nodes of the tree. Intuitively speaking, in each node we optimize the split parameters θ_i of that node, given all split decisions of all other nodes at the current state of the tree. Formally we find the parameters of node i:

$$\hat{\theta}_i = \text{argmax}_{\theta_i} CF_i(S_L, S_R, \theta_i; S_i, \theta_0, ..., \theta_{i-1})$$

In our case we seek to optimize the objective function F over the connected components of a graph G obeying constraints on pairs of nodes. The samples and features of the decision tree consist of edges and their associated features on this graph.

4.2.2 Decision tree building algorithm

Our algorithm seeks to discriminate object boundaries by their features such that the boundaries satisfy a set of must-link and cannot-link constraints. Tree construction starts by trying to satisfy as many of the given constraints as possible by thresholding a single feature and thereby splitting the edges of the graph into one set that is removed from and another one that remains in the graph. The decision on the split parameters θ_i of node i is optimized by sorting the edges S_i associated with decision tree node i on m_{try} different feature values. For each of the m_{try} features all possible split points are evaluated by first removing all edges S_i associated with the decision tree node from the graph G. It is important to realize that only the edges of the currently considered split node are removed, the presence and absence of all other edges, as defined by the current state of the decision tree, remains unchanged. In a second step, the edges associated with split node i are re-inserted into the graph one after the other in the order of increasing feature weight. After each edge insertion, its effect on the connected components of the graph is efficiently evaluated using a union find data structure. In addition, we count how many true positive and false positive pairs are generated with respect to the global constraint set $\mathbf{c}$ which specifies which nodes should be in the same component and which nodes should be in different components. Using the TP, FP, TN and FN counts we compute the value of the objective function F and remember the best split position that we encountered while inserting the edges into the graph. The same procedure is executed in descending sort order. After determining the feature and split position that yield the highest objective function value, two child nodes are added to the currently considered node i and the split parameters θ_i of the node are set accordingly. These two child nodes determine the new state of their associated edges until they are further refined in a recursive fashion. The recursive partioning continues until no further improvement in the objective function can be made.

Splitting a node and the associated edge set further thus re-optimizes the state of the edges associated with that node: the final leaf node with which an edge is associated defines the edge state within the tree. This process is illustrated in Figure 4.2.

It is important to see that during this recursive partitioning the optimization in each leaf node involves only the edges associated with that particular node. The state of the other edges is determined by the already existing leaf nodes and is assumed fixed. Thus each leaf node is optimized conditioned on the graph state given by the current complete decision tree.

While the insertion of edges and its effect on the connected components of the graph can be computed very efficiently, handling edge removal is more difficult. Handling edge removal requires either extremely intricate algorithms [116] or a linear scan over possibly *all* edges in G even though the removed edge set is very small. For this reason we compute the connected components of the graph a single time once all edges associated with a decision tree node are removed. We then trace all changes to the union find data structure and to the TP, FP, TN and FN pair counts caused by inserting an edge into the graph when we test for a split position. This allows us to unwind all changes once the objective function has been evaluated for all split points along one feature and to efficiently begin testing for better splits using the next feature without recomputing the connected component state from scratch.

4.2.3 Backtracking for greedy global decision trees

It is intuitively clear that the greedy tree building procedure that was outlined in the previous section may get stuck in local minima: when partitioning the edge set recursively, the sets assigned to the leaf nodes quickly get smaller and the edges associated to one decision tree leaf are not necessarily close to each other in the underlying graph. It becomes very likely that the edges in any single leaf are unable to form a linking path between two isolated connected components regardless of their labeling – this implies that it would be impossible to satisfy any constraint that would require linking those components in the current state of the decision tree. Similarily it becomes more unlikely that the edges in small leaves are sufficient to build a cut across a connected component – thus it can become impossible to satisfy any constraint that would require splitting some component into two isolated parts because the edges that could form such a cut are distributed across different leaf nodes of the decision tree. For these reasons, we propose a

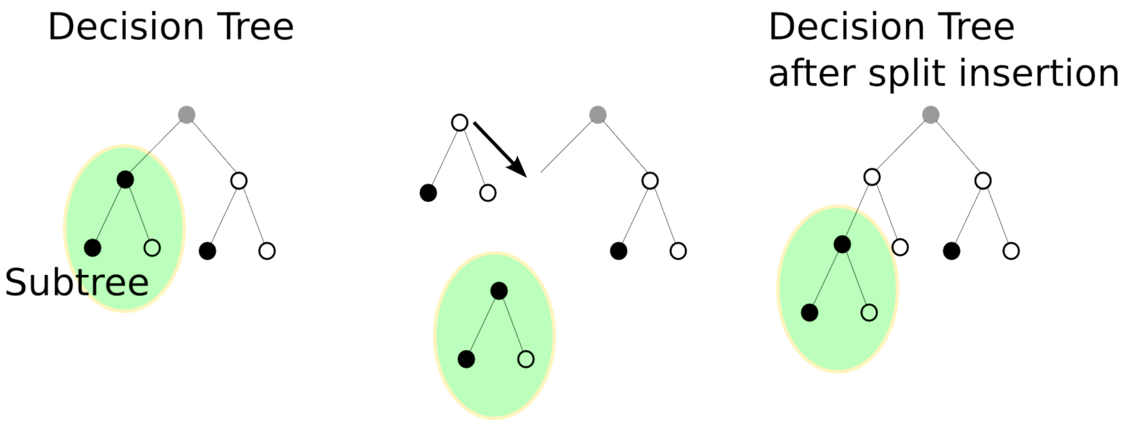


Figure 4.3: Illustration of split node insertion for backtracking. First a random subtree of the decision tree is selected (green overlay). Then a split node is inserted above this subtree whose split function is optimized under the assumption that the left partition below the inserted node is passed onto the existing subtree, while the right partition is passed to a new leaf node and is assigned either to 0 or 1. The insertion of a split at a higher tree level effectively optimizes over a larger set of samples compared to adding a split ad a leaf node. The insertion split takes away some samples from an existing part of a decision tree and overrides the existing subtree partially.

novel *backtracking scheme* during decision tree building: we allow for split nodes to be *inserted* *at arbitrary positions* in the tree, not only at the leaves of the tree. Since we allow these nodes to be inserted at any tree level, these split nodes can optimize over a larger set of edges compared to a node at a decision tree leaf. In the extreme case of inserting such a node above the current root of the tree, the new node can reoptimize over all edges of the graph. This novel *insertion split* partitions the edge set arriving at an inserted node into two parts, such that the left partition is passed to the already existing subtree below the inserted split node as before while the right partition is either assigned to $x_{ij} = 0$ or $x_{ij} = 1$. Thus an existing learned combination of rules, defined by the subtree below the inserted node, is partially reused and the decision for a subset of the edges below/above a feature threshold is reconsidered.

The optimization of an inner node of the decision tree is executed in the same manner as already described for a leaf node. The only difference is that when scanning over the edges in the partition in increasing/decreasing feature weight order, not all edges are re-inserted into the graph. Instead, the current state of the edge x_{ij} which is determined by the subtree of the node currently under consideration is used as a mask. In a first trial only edges with current state $x_{ij} = 1$ are re-inserted. This corresponds to overriding the subtree for the edges right (increasing sort order) / left (decreasing sort order) of the split position with a $x_{ij} = 0$ assignment. In a second trial, only the edges with $x_{ij} = 0$ are removed from the graph before inserting all edges in increasing/decreasing order. This corresponds to overriding the subtree for some edges with a $x_{ij} = 1$ assignment left or right of the split position.

4.2.4 Decision tree prediction algorithm

Once a decision tree has been built in the described manner, it can be used to determine a segmentation of the graph by predicting the binary indicator x_{ij} for all edges. Prediction proceeds as in any normal decision tree: samples (in our case edges (i, j)) are passed down the tree, starting from the root node 0 by comparing the value w_{ijf} of edge feature f with the split value that is stored in a tree node. Edges with a smaller (larger) feature value are passed to the left (right) child of the current node. Once a leaf node is reached, the x label of this leaf node is assigned to the edge. Now all edges with $x_{ij} = 1$ are switched on in the graph and its connected components are determined.

In an **unsupervised segmentation** setting, the resulting connected components of the graph are the final output.

In a **foreground/background segmentation** setting as shown in Figure 4.4, the number and type of user labels that are located inside each component are determined and the component is labeled with the winning label. Since not necessarily all induced components contain user given labels, the unlabeled components are assigned a label by determining the closest (node distance) labeled component in the adjacency graph.

4.3 Experiments

Unfortunately, at present there is no suitable benchmark for the sparse must-link/cannot-link edge learning problem that we propose. To indicate the usefulness of the proposed method we apply our method to a related benchmark dataset [131] and show that our method is applicable to a range of typical unsupervised segmentation problems. Examples of such problems are given in Figure 4.1 and Figure 4.6.

Edge features: a range of simple local filters (Gaussian smoothing, Hessian eigenvalues and Gradient magnitude) computed over several scales ($\sigma = 1.0, 1.3, 1.6, 2.5$) have been used as interpixel edge features. These filter responses haven been calculated in RGB and the LUV colorspace respectively. To obtain features that can be associated with an interpixel edge, these pixel features have been linearly interpolated from two neighboring pixels.

Postprocessing: When our algorithm satisfies a cannot-link constraint between nodes, it does so by introducing a closed boundary between these nodes. This boundary often consist of many isolated 1-pixel components, as can be observed in Figure 4.6. This thick boundary is a result of the ambiguity in the data. A simple way to obtain a visually more pleasing segmentation as in Figure 4.1 is to perform a seeded region growing from all large regions, and to reassign the 1-pixel components to the nearest larger component.

Analysis of training and test error is given in Figure 4.5. The scores were obtained by sparsely labeling all objects in the images of Figure 4.1 and Figure 4.6 and splitting the individual images into two parts. Training was done on the left half and testing on the right half and vice versa. The two examples with inhomogenous boundary appearance (apples, kiwis) profit from deeper trees, as can be seen from the test score which increases until a depth of 4. The examples with homogenous boundary appearance (cells, neural tissue) do not profit from more decision tree levels - the tree starts to overfit after the first level. This ovefitting behaviour of single decision trees is well known. In the future we intend to remedy this problem by training an ensemble of randomized trees which are trained on different subsets of the training data.

Quantitative evaluation: In addition to the qualitative examples, we test our algorithm on the LHI [131] interactive segmentation benchmark. The Benchmark consists of several natural images and provides ground truth and three different types of foreground/background user scribbles with varying difficulty. We adapt the problem to our algorithm by introducing must-link constraints for all foreground-foreground label pairs and all background-background label pairs. In addition we introduce cannot-link constraints for all mixed foreground-background label pairs. We ran the benchmark on all three kinds of user scribbles and calculated the average foreground object precision ($S^+_{gt} \bigcap S^+_{res} / S^+_{gt} \bigcup S^+_{res}$) over all images.

The results show that our purely edge based decision tree achieves a segmentation quality that surpasses many established methods, without learning local class probabilities (unary potentials describing region appearance) from the user labels. These local class probabilities usually work very well on the benchmark images. In addition the other methods rely on a hand-crafted edge probability. We show experimentally that it is possible to achieve the same segmentation quality without relying on local class probabilities and without hand-crafting binary potentials for edges.

Bai et al. [7]	0.50
Gradi [50]	0.56
Couprie et al.[32]	0.58
our algorithm w/o. insertion splits	0.68
Boykov et al. [18]	0.69
our algorithm	0.71
Unger et al. [120]	0.73
Zhao et al.[134]	0.79

Table 4.1: Quantitative Evaluation on the LHI interactive segmentation dataset [131]. The dataset consist of several foreground-background segmentation tasks with varying seed quality (see Figure 4.4 for examples). Results for other algorithms were taken from [134].

Our method exclusively relies on the edge probabilities learned from sparse user scribbles.

4.4 Conclusion

We propose "link-or-cut edge learning", i.e. to learn an edge model from sparse region scribbles interpreted as must-link and cannot-link constraints. To solve this problem, a novel global structured learning scheme based on decision trees is introduced. We explain how decision trees can be trained using a global structured loss criterion and show how they can be used to learn an edge model on an image graph. In addition, we show how local minima during tree construction can be overcome by a split node insertion that reuses the already learned decision structure. When applied to interactive foreground/background segmentation problems on natural images, the proposed algorithm produces results comparable to other methods which do rely on local appearance models. The real strength of the proposed method, however, does not lie in the foreground/background segmentation, but in the discrimination of multiple, and possibly similarly-looking foreground objects. Unfortunately the presented approach does have some limitations which we want to discuss. A current limitation is the reliance on axis-orthogonal splits. If there is no single feature that can discriminate the edges around an object relatively well, the algorithm cannot construct a closed cut around this object and cannot escape from this situation – the objective function only improves, if an object is completely separated from its cannot-link partners. The same problem holds for the must-link constraint: if there is no single feature that can be used to build a linking path between two must-link partners, the objective function cannot be improved.

In future work we hope to remedy some of these problems, either by using a relaxed version of the objective function that allows to increase the objective value also by partially separating a node. In addition one could introduce oblique splits that may prevent some of the problems since the algorithm could construct a suitable linear combination of existing features. Another promising avenue is to extend the supervised segmentation learning algorithm using region homogeneity priors.

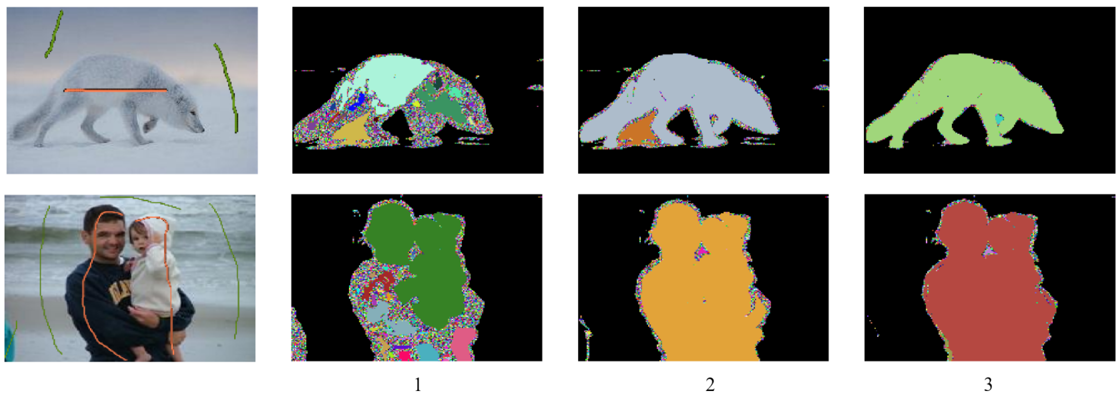


Figure 4.4: Supervised segmentation example. The images shown are two examples from the LHI interactive segmentation benchmark, also displayed are the benchmark provided scribbles. Our decision tree iteratively partitions the image graph into connected components (indicated by same color). In each tree level (1,2,3) the decisions are refined such that a global objective function over the image graph is optimized that enforces the pairwise connectivity and exclusion constraints that are implicitly defined by the labels.

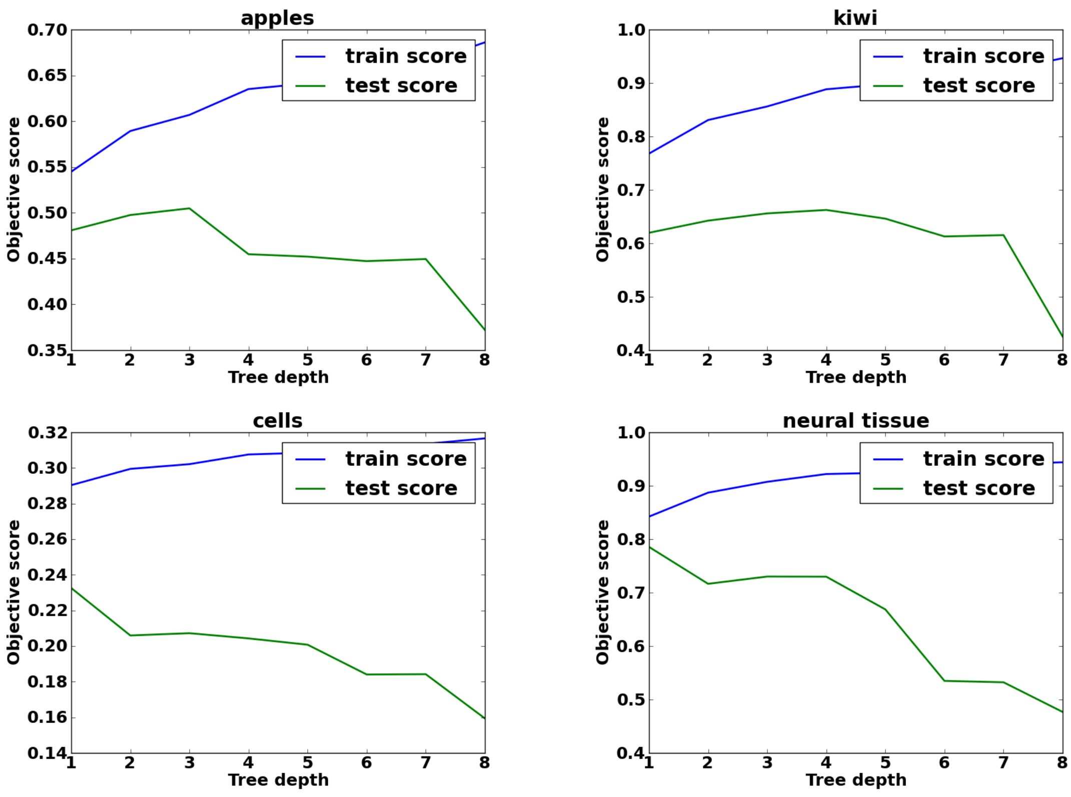


Figure 4.5: The plots show the training and testing score over the decision tree depth. The two examples with inhomogenous boundary appearance (apples, kiwis) profit from deeper trees, as can be seen from the test score which increases until a depth of 4. The examples with homogenous boundary appearance (cells, neural tissue) do not profit from more decision tree levels - the tree starts to overfit after the first level.

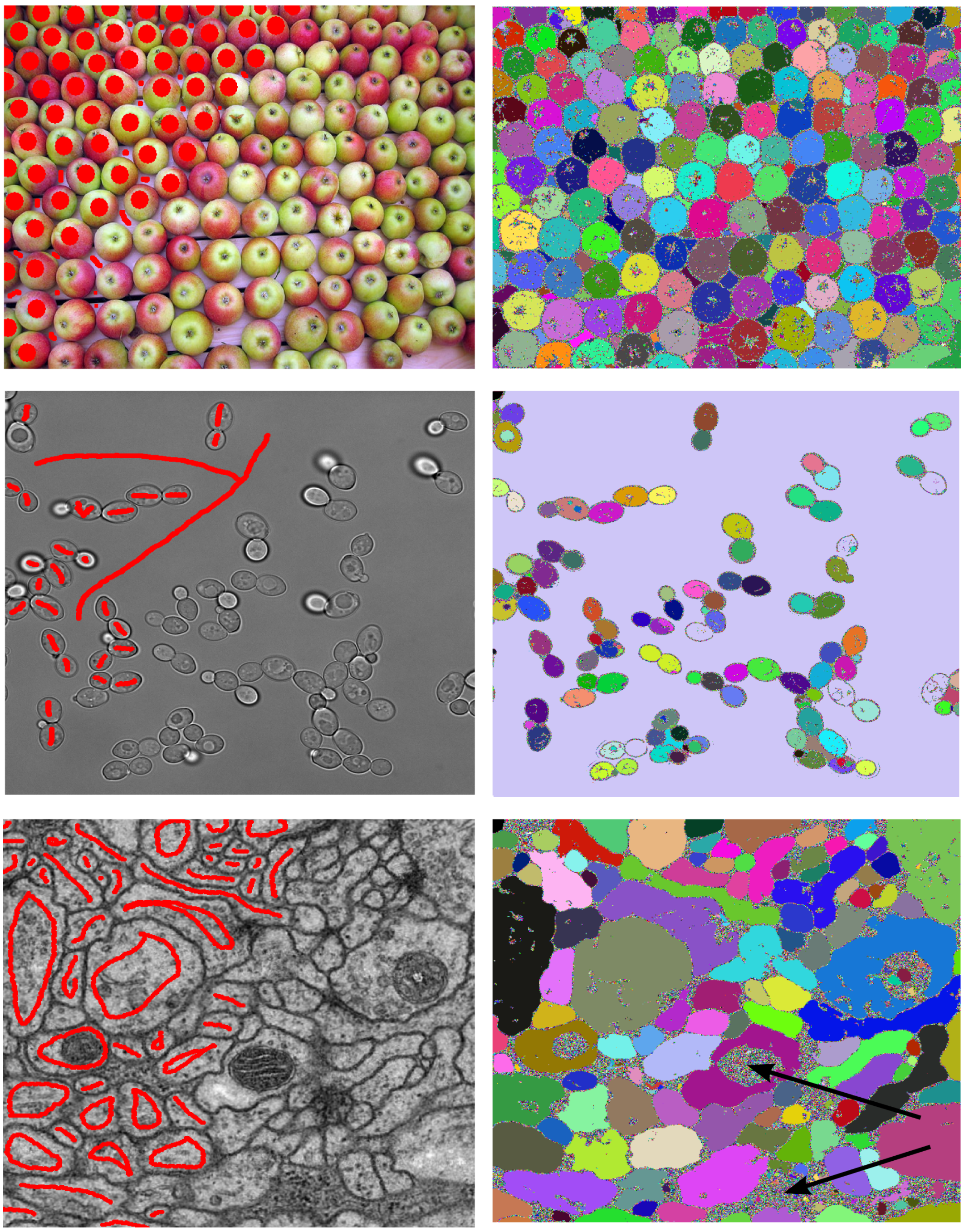


Figure 4.6: Segmentation examples of different characteristics, all segmented with the same parameter settings. The last two rows show yeast cells in light microscopy, and dendrites in electron microscopy [24, 25], respectively. Individual objects do not differ in appearance and can be discriminated via their boundaries only. These are learned in a weakly supervised fashion from the connectivity constraints that are implicit in the seeds. In contrast to seeded segmentation only a subset of objects need to be marked and the learned classifier can be applied to similar images. Region types that are not represented in the training set (no labels) receive an incoherent prediction (bottom example, arrows).

Chapter 5

Multiple instance based edge learning

5.1 Introduction

This chapter describes a new method to learn *edge* models from sparse user scribbles marking *regions*. Consider Figure 5.1 and assume that we want to segment each kiwi. Normally, scribbles as shown on the left would be used to learn region appearance models that can then serve as unary potential functions in labeling methods such as graph cuts. However, this does not work here because the individual objects are indistinguishable by region appearance. When region appearance alone is not informative, segmentation must be based on an edge model. Based on the edge model one would use the scribbles as seeds for a suitable region growing algorithm such as a seeded watershed or random walker. When the edge model is of high accuracy even a simple connected component computation can produce the desired segmentation.

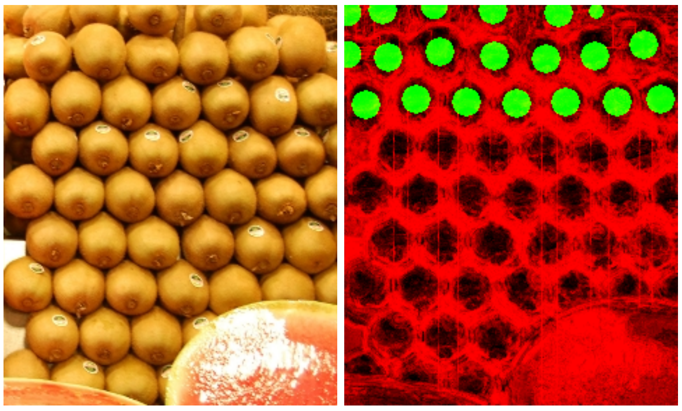


Figure 5.1: Qualitative boundary learning result. The boundary prediction (red, right image) was learned from the user given region scribbles (green, right image).

Our ambition is to train such a model with minimal labeling effort on the user's part. Traditional learning methods require the user to place edge scribbles exactly on, or close to [6], the desired edges. The required localization accuracy makes this a time consuming task. In contrast, we strive to use cheap *region* scribbles like in Figure 5.1 to train edge models instead of the usual region models.

Clearly, region scribbles cannot be used for edge learning directly because they are typically located far away from edges. However, they provide a large number of constraints that can control edge learning indirectly:

- Each pair of pixels from the same scribble defines a *must-link* constraint, i.e. there must be at least one connecting path that does not cross any edge.
- Pixels from different scribbles define a *cannot-link* constraint, i.e. any connecting path must cross at least one edge.

Our main contribution is to point our that this can be formulated in a multiple instance learning setting. *Multiple instance learning (MIL)* [39] is a learning technique that relies on very weak supervision. As opposed to standard supervised learning settings where a ground-truth label is available for each instance, in the MIL setting the labels of individual instances remain unobserved (latent variables). Instead, a single label is assigned to each set, called *bag*, of instances. A positive bag label indicates that there exists at least one instance in that bag with positive label. In a bag with a negative label, all instances are known to have a negative label.

More formally, the MIL problem considers several bags $B^b, b \in \{1, ..., N\}$. Each bag B^b consists of a number of instances $\mathbf{x}_i^b \in B^b, i \in \{1, ..., N^b\}$. A bag B^b has a positive label $Y^b = 1$ if there is a $\mathbf{x}_i^b \in B^b$ with $y_i^b = 1$. This is called the *positive identifiability* constraint [77]. A bag B^b has a negative label $Y^b = 0$ if $y_i^b = 0$ for all $\mathbf{x}_i^b \in B^b$. This is called the *negative exclusion* constraint [77]. An instance of a positive bag is called a *witness* if it has a positive label, and *non-witness* if it has a negative label. Given a previously unobserved bag of instances, an MIL model can either predict a label at the bag level, as in [136], or at the level of individual instances, as in [2]. The latter is a more difficult problem and requires a model with greater descriptive power. For a more detailed introduction to the MIL problem and related algorithms we refer to Chapter 6.

5.2 Edge learning as a MIL problem

We consider an image partitioning problem defined on a graph $G(E, V)$ in which the nodes $n_i \in V$ correspond to the pixels of an image, and the edges $(i, j) \in E$ correspond to the pixel neighborhood of the image. We assume a suitable vector of edge features $\mathbf{w}_{ij}$ (such as color gradients and structure tensor eigenvalues on different scales) is available and can be attributed to each edge (i, j). In addition we are given a sparse constraint matrix $c_{ij} \in \{-1, 0, 1\}$ that defines whether a pair of pixels (i, j) must be in the same component ($c_{ij} = 1$), or in two different components ($c_{ij} = -1$). Associated with each edge are decision variables $x_{ij} \in \{0, 1\}$ that determine whether an edge (i, j) is removed from ($x_{ij} = 1$), or remains ($x_{ij} = 0$) in the graph G. These decision variables define the solution to the graph partitioning problem via the connected components of a modified graph G' consisting only of the non-boundary edges (i, j) with $x_{ij} = 0$. Our aim is to learn a classifier that predicts all edge decision variables x_{ij} such that the connected components of the graph G' satisfy all user given grouping constraints C.

We now show that it is possible to formulate the edge learning problem from sparse user scribbles as a multiple instance learning problem (for a introduction to multiple instance learning see Chapter 6) by expressing the must-link and cannot-link constraints as positive and negative bags: any path $P \subseteq E$ on G between two cannot-link nodes a, b must, by definition, contain at least one edge $e \in P$ which is a boundary edge. This is, because node a and node b are located in different partitions, which in turn must be separated by at least one boundary. This boundary crossing constraint on the edges of path P corresponds exactly to the concept of a positive bag in the MI setting: the positive bag must contain at least one positive instance. The negative bag concept on the other hand is directly related to must-link constraints within user scribbles: we treat each edge between two neighboring nodes which received a must-link constraint (i.e. two neighboring pixels inside a user scribble) as a non-boundary edge: the two nodes must be connected and there is no boundary between them. Thus, we can add all such edges between must-link nodes to a single negative bag which in the MI setting must not contain any positive instance, i.e. no boundary. Thus, we relate boundary edges ($x_{ij} = 1$) with positive instances and non-boundary edges ($x_{ij} = 0$) with negative instances.

In principle a MI classifier can be trained by constructing the positive bags from all possible paths between all cannot-link nodes. But unfortunately it is infeasible to sample all paths P_i between even a single cannot-link node pair since there are exponentially many such paths on a graph.

One solution to this problem is a cutting planes like approach: we sample k paths initially and iteratively add violated paths to the learning problem. The algorithm works as follows: from the initially sampled paths positive bags are constructed by adding all edges $e \in P_i$ on path i to a positive MI bag B^i_+. In addition a negative bag B_- is constructed from all must-link edges which is also added to the MI problem. Next, a MI learning algorithm is trained on the constructed problem. The trained MI classifier is then used to predict an instance label x_{ij} for all edges (i, j) of the image graph. This predicted instance label is the boundary indicator for each edge: $x_{ij} = 1$ is a positive instance which corresponds to a boundary, $x_{ij} = 0$ is a negative instance which corresponds to a non-boundary.

Now the graph G' is constructed from all non-boundary ($x_{ij} = 0$) edges. The connected components of G' define the solution to the graph partitioning problem in the current iteration. On this graph G' k still violated paths, i.e. paths which connect two cannot-link nodes, are sampled. These newly sampled paths are also added as positive bags to the MI problem. This whole process is repeated until no more paths between cannot-link nodes can be found in G' - the segmentation problem is then solved and the MI classifier has learned a positive instance boundary concept that separates all cannot-link nodes from each other by predicting a closed boundary between them.

5.3 Experimental evaluation

We evaluated the proposed boundary learning method on several images which are suitable for such an algorithm: these images contain many similar objects touching each other and thus cannot be discriminated by training a normal foreground/background classifier since all objects look alike featurewise.

The iterative learning/prediction scheme from Section 5.2 requires a MI learning algorithm with low computational requirements since the problem size increases in each iteration and can consist of hundreds of thousands of instances. For this reason the MIForest [76] is used

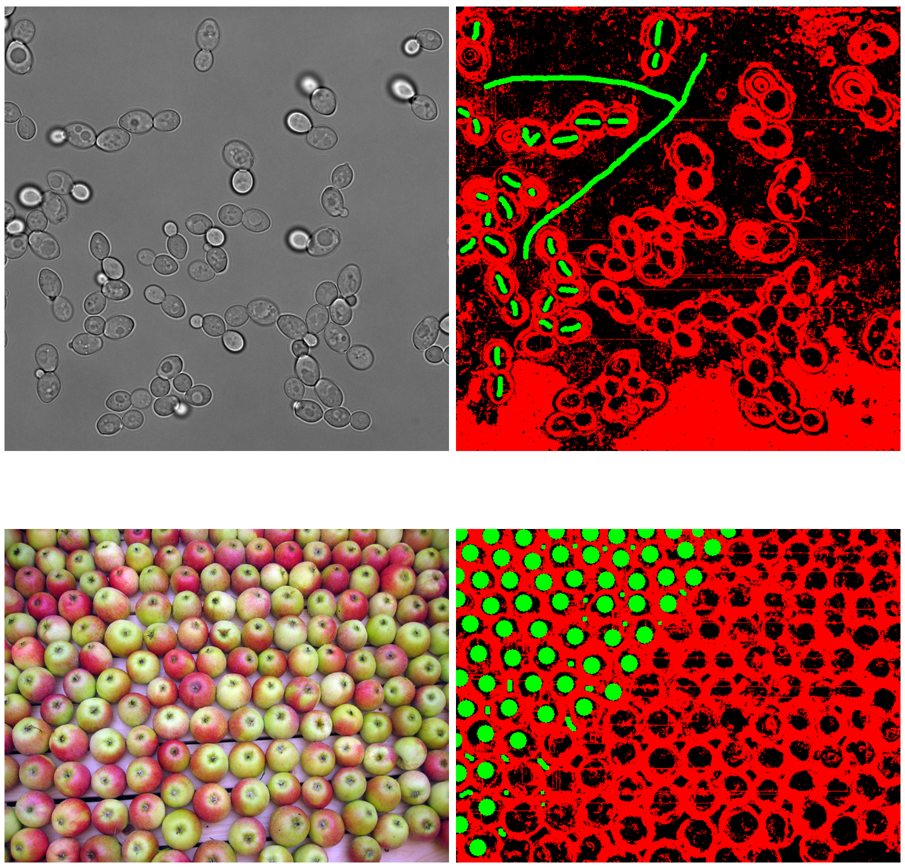


Figure 5.2: Qualitative boundary learning result. The boundary prediction (red, right image) was learned from the user given region scribbles (green, right image).

throughout the experiments due to its low runtime complexity.

The example images and experiments show that the MIForest fails to learn a reasonable boundary concept: nearly all unlabeled parts of the image are predicted as boundary at test time.

We identified two interconnected reasons for this undesirable behavior. The first reason are the algorithm internals of MIForest (and mi-SVM, MI-SVM and others): the algorithm infers the latent instance labels in an iterative fashion by training a normal supervised random forest. The MIForest starts from an instance label initialization of $y_i = 1$ if instance i is part of of positive bag and $y_i = 0$ if instance i is part of a negative bag. These labels are used for training a normal random forest and the prediction of the trained forest is used as the instance labels in the next iteration. This process is repeated until the instance labels do not change from one iteration to the next.

The second reason is the class imbalance between positive bags and negative bags: since the iterative learning/prediction scheme from Section 5.2 adds several positive bags per iteration until all constraints are satisfied, the number of positive bags and instances outweighs the negatives by far.

The simplistic supervised classifier that is internally used by MIForest and the huge class imbalance together lead to a multiple instance classifier that prefers assigning most edges to the boundary class.

5.4 Conclusion

We propose a method that can make use of sparse partial region annotations to learn a boundary concept on an image, even when the regions look alike and cannot be discriminated from the background. The proposed method maps the boundary learning task to a multiple instance learning problem. Positive bags correspond to paths between cannot-link labels where a positve instance in a bag maps to a boundary edge on the path. Negative bags in the multiple instance learning problem correspond to paths within a user given brush stroke: all edges on the path belong to the non-boundary class. The method is evaluated on several natural images but fails to produce visually pleasing results due to the many boundary predictions that are produced. Much of the image is predicted as being a positive instance. Further research is neccessary to remedy this problem.

Chapter 6

Multiple instance optimized random forest

6.1 Introduction

Dramatic improvements in instrumentation, and online data collection on an unprecedented scale, result in a tremendous increase in the number of observations. In contrast, the rate with which a human expert can annotate such observations for the purpose of supervised machine learning is a (biologically limited) constant. It is hence attractive to trade human effort for computational expense, by learning from *weak supervision*. In brief, learning highly descriptive models from weak supervision appears as a key challenge in next generation data analysis.

Multiple instance learning (MIL) [39], which was used in Chapter 5 to learn image boundaries, is a learning technique that relies on very weak supervision. As opposed to standard supervised learning settings where a ground-truth label is available for each instance, in the MIL setting the labels of individual instances remain unobserved (latent variables). Instead, a single label is assigned to each set, called *bag*, of instances. A positive bag label indicates that there exists at least one instance in that bag with positive label. In a bag with a negative label, all instances are known to have a negative label.

More formally, the MIL problem consists of several bags $B^b, b \in \{1, ..., N\}$. Each bag B^b consists of a number of instances $\mathbf{x}_i^b \in B^b, i \in \{1, ..., N^b\}$. A bag B^b has a positive label $Y^b = 1$ if $\exists \mathbf{x}_i^b \in B^b$ with $y_i^b = 1$. This is called the *positive identifiability* constraint [77]. A bag B^b has a negative label $Y^b = 0$ if $y_i^b = 0 \; \forall \mathbf{x}_i^b \in B^b$. This is called the *negative exclusion* constraint [77]. An instance of a positive bag is called a *witness* if it has a positive label, and *non-witness* if it has a negative label. Given previously unobserved bag of instances, an MIL model can either predict a label at the bag level, as in [136], or at the level of individual instances, as in [2]. The latter is a more difficult problem and requires a model with greater descriptive power.

In this chapter, which is based on publication [111], we introduce a decision tree based solution to MIL. Our model inherits the desirable properties of decision trees such as small computational footprint, ease of implementation, and high interpretability. In particular, we extend the work of Blockeel et al. on multi-instance decision trees [16] in several aspects. First, we learn multiple randomized decision trees [19]. Secondly, we employ a non-linear split criterion on multiple features at a time, in place of the commonplace axis-orthogonal approach. Thirdly, we increase the robustness of our model against noise by regularizing instance weights. Lastly and importantly, we introduce a means to learn the optimal combination of the decision outputs of all trees in the forest. In a lesion study, we analyze the contribution of each of these extensions to overall performance. The combined model clearly outperforms existing decision tree based methods [76, 16].

6.2 Related Work

The first algorithm to solve the multiple instance learning problem was proposed by [39]. Here, the authors assume that the positive instances reside in a single axis parallel rectangle (APR). The diverse density framework in [84, 133] assumes that positive instances form a Gaussian-like pattern around some concept points. Wang et al. [127] propose a nearest neighbor and nearest neighbor of nearest neighbor based approach. Another SVM-based approach is presented in [48, 74]. Here, the authors design kernel functions on bags, instead of on instances. Mangasarian et al. [82] use a succession of linear programs to solve the MIL problem. Boosting based approaches include, for example, Viola et al. [125]. In addition, some deterministic annealing type approaches have been proposed [49, 76] that try to uncover the instance labels in several training iterations. Joulin and Bach [61] propose a convex formulation for estimating the latent instance labels. An interesting extension to the finite-sized MIL algorithms has been proposed in [4] that learns from infinite size manifold bags. Also noteworthy is the method proposed in [12] which tackles the runtime complexity of multiple instance learning algorithms using a fast bundle method. Settles et al. [108] apply the idea of active learning to MIL by querying the labels of instances in positive bags . Li et al. [78] reports increased performance in object tracking if learning is performed from batches of bags.

A large group of existing MIL methods commonly follow the large margin principle [2, 49, 31, 52]. These models are based on extensions of the support vector machine (SVM) optimization problem with the positive identifiability and negative exclusion constraints. MI-SVM [2] adds one constraint per one instance in a positive bag that has the largest discriminant value. mi-SVM [2] treats the instance labels in positive bags as latent variables to be learned from data. Although mi-SVM is a more flexible model, MI-SVM exhibits better prediction performance, since the prediction of the latent variables in mi-SVM strongly biases false positives to maximize the margin. This inherent difficulty in regularizing the intra-bag distribution of instance labels has been pointed out by [52], and solved alternatively by enforcing the instances in positive bags to lie on a manifold perpendicular to decision boundary.

An alternative approach to MIL is via probabilistic modeling. A seminal work in this line is by Kim and de la Torre [63], who introduce an extension of Gaussian process classification to the MIL setting. The model relies on a very similar principle to MI-SVM, namely describing positive bags by a single instance having the largest separation from the border. Apart from the fact that it enjoys a principled parameter tuning mechanism, this model shares all the weaknesses of MI-SVM, such as not being robust to false positives in positive bags, due to the fact that it ignores the information stored in all the instances in a positive bag but one. Raykar proposes a Bayesian formulation [103] to MIL that can effectively perform feature selection.

A third alternative approach is decision tree based MIL. Blockeel et al. [16] adapted decision trees to the MIL problem by introducing a priority queue into the tree construction process. Leistner et al. [76] propose a deterministic annealing procedure for uncovering the hidden instance labels in several forest training iterations. In this chapter, we introduce another MIL algorithm on this track, which is an extended version of of the work of Blockeel et al. [16].

Recent examplary applications of MIL include diabetic retinopathy screening [101], cancer detection from tissue images [129], visual saliency estimation [128], and content-based object detection and tracking [109, 5]. MIL is also useful in drug activity prediction where each molecule constitutes a bag, each configuration of a molecule an instance, and binding of any of these configurations to the desired target as a positive label, as first introduced by Dietterich et al. [39]. More recent applications of MIL to this problem include finding the interaction of proteins with Calmodulin molecules [11], and finding bioactive conformers [46].

6.3 Decision trees

We quickly review the decision tree algorithm already covered in Section 1.7.
Let $\mathcal{D} = \{(\mathbf{x}_1, y_i), (\mathbf{x}_2, y_2) \cdots, (\mathbf{x}_N)\}$ be a data set of N instances where
$\mathbf{x}_i = [x_{i1}, x_{i2}, \cdots, x_{iD}]$ are D-dimensional vectors of observed instances and y_i are associated labels. Suppose that $M(\mathbf{y}_s)$ is a scalar measure based on the labels of an instance set s, denoted by $\mathbf{y}_s$. A decision tree is built by the following steps:

- For each observed value x_{ij} of each feature j, group the instances into two groups according to the split rule $f_j > x_{ij}$ where f_j is an arbitrary value for feature j. Calculate a goodness measure for the split $\theta_{ij} = M(\mathbf{y}_{f_j > x_{ij}}) + M(\mathbf{y}_{f_j \leq x_{ij}})$.
- Create a node for the feature $\hat{j}$ which gives the highest θ_{ij}, and two child nodes, and assign all instances with $f_j > x_{ij}$ to one node, and the rest to the other node.
- For each child node, repeat this process on their assigned set of instances recursively until all nodes have instances belonging to a single class.

Let f_c denote the ratio of instances in $\mathbf{y}_s$ that belong to class c. Two widely used examples of goodness measures are *Gini Impurity*:

$$I_G = 1 - \sum_{c=1}^{C} f_c^2,$$

and *Information Gain*:

$$I_E = -\sum_{c=1}^{C} f_c \log f_c.$$

Gini impurity is widely used by the CART (Classification And Regression Tree) algorithm [20], and the information gain is more often preferred with the C4.5 algorithm [102]. The outlined axis-orthogonal splitting procedure can be replaced by more complex splitting tests over multiple features at a time, see for example [88].

6.4 Random forests

Decision tree learning has several desirable properties such as being computationally very fast, and being very interpretable. However, the main drawback of this algorithm is the suboptimal prediction performance it provides. One reason for the low accuracy of decision trees is that features are handled one-by-one in the decision process, hence complex correlations between features are not captured sufficiently. Another reason is that the goodness measures are calculated over the entire set of available instances, which makes the algorithm prone to overfitting.

Breiman has showed that a simple extension to the decision tree algorithm, called the *random forest*, indeed brings a significant improvement in prediction performance, while keeping the other good properties of decision trees [19]. The random forest takes a decision tree as a weak classifier, and performs ensemble learning together with a bagging procedure. In particular, the algorithm learns multiple decision trees on randomly chosen features based on randomly chosen sets of training instances. The decision for a newly seen instance is made by combining the outcomes of the learned trees in the forest.

6.5 Multi-instance tree learning

As a basis for our method we rely on the *multi-instance tree learning* (MITI) algorithm presented in [16]. This greedy decision tree algorithm initially assumes that each instance in a positive bag has a positive label. Importantly, the algorithm from [16] does not follow the depth first recursive partitioning scheme outlined in Section 6.3. Instead, the authors propose a priority queue based node expansion. The algorithm maintains a priority queue of split nodes, and iteratively takes the node with the highest priority from the queue. The node that was taken from the queue is split according to the goodness measure, and the resulting two child nodes are inserted into the priority queue. The algorithm uses the number of positive instances in a node as the priority for the queue.

Furthermore, the authors assign to each instance $\mathbf{x}_i^b$ a weight w_i^b. At the beginning this weight is set to $\frac{1}{|B_b|}$, the inverse of the size of the bag b to which instance i belongs. This ensures that the weights of all the instances in a bag sum up to 1 - otherwise large bags would cause a bias. The instance weight is used throughout the entire tree construction process and is also considered during the evaluation of the goodness of a split. Thus, a weighted Gini-impurity measure for a set $\mathcal{S}$ of instances $\mathbf{x}_i^b$ is

$$G(\mathcal{S}) = \left(\sum_{\mathbf{x}_i^b \in \mathcal{S}} w_i^b\right)^2 - \left(\sum_{\mathbf{x}_i^b \in \mathcal{S}+} w_i^b\right)^2 - \left(\sum_{\mathbf{x}_i^b \in \mathcal{S}-} w_i^b\right)^2 ,$$

where $\mathcal{S}+$ denotes the subset of instances in $\mathcal{S}$ with positive labels, and $\mathcal{S}-$ denotes the ones with negative labels.

In addition the authors propose to change the assigned weight of the instances once a purely positive leaf has been discovered. In this case the other instances of the bags that are covered by the leaf will be discounted by setting their weight to 0. More formally, once a positive leaf Π_k has been found during tree construction we set $w_i^b = 0, \mathbf{x}_i^b \in B^b, i \neq j, \mathbf{x}_j^b \in \Pi_k$. The intuition here is that that once a positive bag is explained by one or more positive instances in a purely positive leaf node, the decision tree does not need to find another positive instance to explain the positive label of the associated bag, thus the weight w_i^b of the other instances of the bags that are covered by the leaf node can be set to 0. This essentially discards these instances from the further tree growing process, since these presumed negative instances have no influence on the weighted Gini impurity during split evaluation.

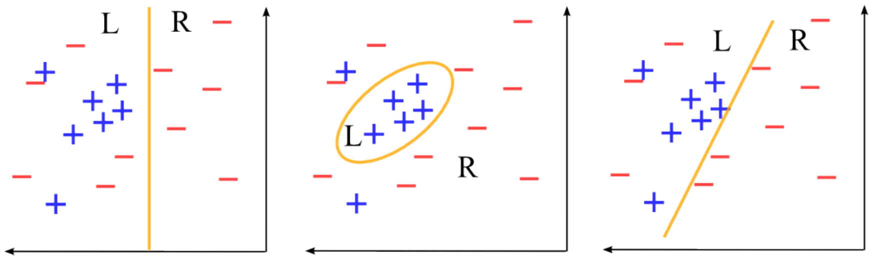


Figure 6.1: Overview of the different decision tree split types. The standard axis-orthogonal split type (left) is less discriminative then the oblique hyperplane split (right) and the ellipsoid split (middle). The ellipsoid split is defined by the contour line of a normal distribution fitted to the positive instances.

The overall process amounts to building a decision tree in a special order, where the largest positive subset in the decision tree is expanded first. This prioritization induces the tree learner to focus on finding pure positive subsets. In other words, it enforces the rule that it is legal for non-witness positive instances to end up in negative nodes, but it is not desirable for negative instances to end up in positive nodes. Thus trying to first find pure positive subsets best expresses the multiple instance constraint. In [16] the authors show that the resulting tree growing procedure outperforms other decision tree based algorithms in MIL problems.

In the following sections we extend the MITI algorithm in several ways and show that this approach yields an algorithm that outperforms the existing decision tree based MIL algorithms, and gives results similar to other existing algorithms. Our extensions to MITI are:

- learning a random forest instead of a single deterministic decision tree,
- using a regularization and a weight redistribution to increase robustness to noise (as detailed in 6.5.1),
- using a non-linear splitting method on pairs of features, instead of line search in individual features (as detailed in 6.5.2),
- learning how to combine the decision outputs of the trees in the forest from data, subject to a multiple instance constraint (as detailed in 6.5.4).

6.5.1 Tree regularization and weight redistribution

As opposed to [16] who grow a decision tree until impurity, we use a minimum leaf node size as the stopping criterion for the decision tree growing process. This early stopping acts as a regularizer, since it allows leaf nodes to contain positive and negative instances and the positive instances in such impure leaves are assumed to be false positives. In this case we redistribute the weights of the positive instances in an impure leaf node to the other instances of the corresponding bag which are not yet assigned to a leaf node. More formally, for an impure leaf node Π_k that is below a size threshold we set $w_i^b = w_i^b + \frac{1}{|B^b|}, \mathbf{x}_i^b \in B^b, Y^b = 1, i \neq j, \mathbf{x}_j^b \in \Pi_k$ where $|B^b|$ is the size of bag B^b. Thus, these remaining positive instances of a bag B^b contribute more strongly to the further decision tree building process and the reweighted positive instances are more likely to end up in a positive leaf node.

6.5.2 Inside/outside split concepts

In typical multiple instance learning tasks, positive instances are often distributed around few cluster centers while the negative class is distributed more evenly. This fact is exploited in many multiple instance learning algorithms, such as [84, 133, 31]. The orthogonal axis splits used in traditional decision trees, such as in [16], appear as a bottleneck in model performance since they assume an axis-orthogonal boundary between classes.

As a solution to this problem, we strenghten the modeling power of our decision trees with a more complex split scheme. Instead of the line-search based splits on single features used in [16], we employ two types of split criteria on multiple features at a time (see Figure 6.1):

- Ellipsoid splits
- Hyperplane splits

These complex split criteria are inspired from the oblique random forest idea studied in [89], and the Gaussian density estimation forests in [37]. In the case of *ellipsoid splits*, we pick a random subset of the feature dimensions and calculate the weighted mean and weighted sample covariance matrix of the positive samples that are assigned to that split node N:

$$\mu = \sum_{\mathbf{x}_i^b \in \mathbf{X}_+} \frac{w_i^b \mathbf{x}_i^b}{W},$$

where $W = \sum_{\mathbf{x}_i^b \in \mathbf{X}_+} w_i^b$, and

$$\Sigma_{kl} = \left[\frac{\sum_{\mathbf{x}_i^b \in \mathbf{X}_+} w_i^b}{\left(\sum_{\mathbf{x}_i^b \in \mathbf{X}_+} w_i^b\right)^2 - \sum_{\mathbf{x}_i^b \in \mathbf{X}_+} (w_i^b)^2} \right] \times \sum_{\mathbf{x}_i^b \in \mathbf{X}_+} w_{ij} (x_{ik}^b - \mu_k)(x_{il}^b - \mu_l)$$

where $\mathbf{X}_+$ is the set of positive instances, $\mathbf{x}_i^b$ is the instance i of bag b, and x_{ik}^b is the kth feature of the same instance. We then sort the instances with respect to their Mahalanobis distance

$$\lambda = \sqrt{(\mathbf{x}_i^b - \mu)^T \mathbf{\Sigma}^{-1} (\mathbf{x}_i^b - \mu)}$$

and search for the optimal split threshold $\lambda > \tau$ with respect to the Gini impurity of the induced partitioning into a left and right subset.

In the case of *hyperplane splits*, we choose a random subset of the feature dimensions and assign a random weight to each dimension, afterwards we search for the optimal linear intercept with respect to the Gini impurity. This split type allows to discriminate better on correlated features than the axis orthogonal split type used in [16].

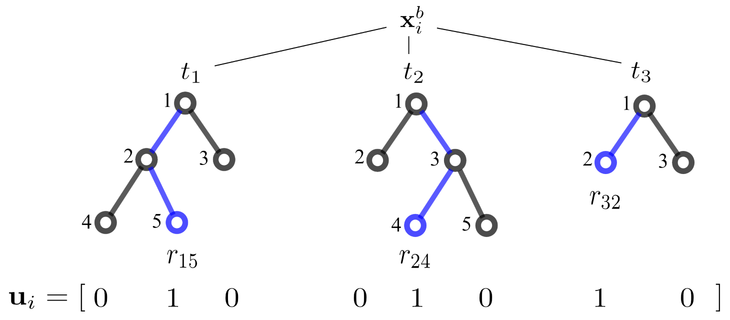

$\mathbf{x}_i^b$
t_1
t_2
t_3
1
2
3
4
5
r_{15}
r_{24}
r_{32}
$\mathbf{u}_i = [\,0 \quad 1 \quad 0 \qquad 0 \quad 1 \quad 0 \qquad 1 \quad 0\,]$

Figure 6.2: The classifier response $f(\mathbf{x_i^b}) = \mathbf{u}_i^T \mathbf{r}$ for the instances is calculated as the scalar product between leaf weights $\mathbf{r}$ and the indicator vector $\mathbf{u}_i$ for instance $\mathbf{x}_i^b$. The indicator vector has 1 entries for the leaf nodes in which that instance $\mathbf{x}_i^b$ ends up when predicting the instance with the individual trees t_k. The leaf node weights $\mathbf{r}$ are then optimized in a post processing step subject to positive identifiability and negative exclusion constraints.

6.5.3 Forest vote bias correction

Our model learns multiple decision trees, each on a random subset of training instances and features. Each of these trees have their own decision output for a newly seen instance. Hence, a mechanism is required to combine these decisions to obtain the final outcome for that instance.

In the traditional random forest formulation, this issue is solved simply by majority voting. In the case of multiple instance decision trees this can have detrimental effects: since each tree deactivates all positive instances of a positive bag when it discovered a positive witness for that bag the decision boundary is biased in favor of the negative instances since instances of the negative bags are never deactivated.

Another undesirable situation occurs when the true positive instances are arranged in several distinct clusters. Due to the deactivation mechanism, each tree can only identify a single positive cluster per positive bag. The other clusters have a certain probability of being misclassified by a single tree. Once again, true positive samples may not receive sufficiently many positive votes to reach the majority vote threshold. Consequently, a simple majority vote combination of the individual trees will lead to a strong negative bias of the forest.

We propose to counter the negative class bias of the forest in a simple way by optimizing a threshold with regard to the MI-constraint. We count the number of positive tree votes for each instance, and the instance that receives the most positive votes from each bag is recorded. For these most positive instances the vote count threshold that best separates the positive from the negative bags is determined by a line search with respect to the induced bag accuracy. We experimentally show that countering the negative bias of the forest is crucial for good performance (see section 6.6).

6.5.4 Forest response optimization

Threshold optimization counters the class imbalance bias (which arises from deactivation of positive training samples) by lowering the *number* of positive votes an instance must receive in order to be assigned to the positive class. This strategy does not alter the votes themselves, but only their interpretation. Alternatively (or in addition), we can improve voting performance by optimizing the *response* of all leaf nodes. This optimization is only beneficial if it is conducted *jointly* over all trees in the ensemble, because the standard leaf response (the majority label of the instances assigned to each leaf) is clearly optimal as long as each tree is considered in isolation. Since we want to retain the property that the ensemble response is computed as a linear combination of tree responses, this leads us to a constrained linear optimization problem similar to the ones discussed in only [44, 88]. These works introduce a sparsity constraint in order to prune as many nodes as possible without degrading performance. In contrast, we attempt to optimize leaf responses under the multiple instance constraints.

To define the linear system, we introduce the *indicator vector* $\mathbf{u}_i$ of an instance i. It is a binary vector whose length equals the total number of leaf nodes in the forest, and whose elements represent the membership of i to the different leaves. That is, an element of $\mathbf{u}_i$ takes a value of

1 precisely when the instance i is assigned to the corresponding leaf. For example, the indicator vector for the ensemble in Figure 6.2 has length 8. When the features vector $\mathbf{x}_i^b$ is propagated down each tree, the leaves (1,5), (2,4), and (3,2) marked in blue are reached, and the three corresponding entries in the indicator vector are set to 1. In addition, we define the leaf response vector by $\mathbf{r} = [r_{11}, r_{12}, \cdots, r_{TL}]$ (where r_{tl} is the response of leaf tl). Then, the total response of the ensemble can be written as the scalar product

$$f(\mathbf{x}_\mathbf{i}^\mathbf{b}) = \mathbf{u}_i^T \mathbf{r}$$

To optimize the weights $\mathbf{r}$, we take the indicator vectors of all training examples and stack them next to each other into the indicator matrix $\mathbf{U}$. The training response is thus $\mathbf{f} = \mathbf{U}^T\mathbf{r}$. Note that all instances are assigned to all trees during global optimization, irrespective of whether they where out-of-bag or de-activated in the tree construction phase. We now seek the vector $\mathbf{r}^*$ that minimizes the training loss under the multiple instance constraints. That is, f_i should take the value 0 for all instances from negative bags, and 1 for the witness of each positive bag, whereas the response for non-witness members of positive bags is ignored. Under our linear model, the witness is defined as the positive bag member that receives the maximum response. This leads to the following formal definition of the loss

$$L_\mathbf{r}^b(\mathbf{X}^b, Y^b) = \begin{cases} (\max_{\mathbf{x}_i^b \in \mathbf{X}^b}[f(\mathbf{x}_i^b)] - 1)^2 & , Y^b = 1, \\ \sum_{\mathbf{x}_i^b \in \mathbf{X}^b} (f(\mathbf{x}_i^b))^2 & , Y^b = 0. \end{cases}$$

This loss is combined with a quadratic regularizer to obtain the global optimization problem

$$\mathbf{r}^* = \operatorname*{argmin}_\mathbf{r} \mathbf{r}^T\mathbf{r} + \sum_{b \in \mathcal{B}} L_\mathbf{r}^b(\mathbf{X}^b, Y^b).$$

While this is a non-convex optimization problem (the arguments of the square functions in the loss can have arbitrary sign), we found in practice that good local optima can be found using gradient descent by initializing the leaf weights $\mathbf{r}$ with the fraction of positive instances assigned to them:

$$r_l = \frac{\sum_{\mathbf{x}_i^b \in \Pi_l, y_i^b = 1} 1}{\sum_{\mathbf{x}_i^b \in \Pi_l} 1}$$

The resulting optimization algorithm is efficient and takes only a small fraction of the total training time.

6.6 Experiments

We evaluated the generalization performance of our algorithm on five publicly available benchmark data sets for MIL. The randomized decision tree parameter *mtry* which determines how many different splits are tested in each split node during tree construction was set to the standard square root of number of feature dimensions proposed by Breiman [19]. The number of dimensions for the hyperplane splits was set to 5 and the number of dimensions for the Gaussian ellipsoid split were set to 2. These choices ensure that the both split types have the same number of free parameters (a 2D Gaussian has 3 free covariance matrix entries and 2 free mean vector entries). Additionally, we set the minimum leaf node size regularization parameter to the number of free parameters in the split nodes, i.e. 5. All performance scores reported below are obtained by averaging 5 runs of 10-fold cross-validation.

6.6.1 Drug activity prediction

Drug activity prediction has been the first problem on which the concept of multiple instance learning was introduced [39]. In this setup, each molecule is formulated as a bag, and each configuration of that molecule as an instance in that bag. An instance is positively labeled if the corresponding configuration of the molecule binds with another predetermined target molecule. For a bio-chemist, what matters is whether a molecule binds with its target, but which particular configuration binds is less important. Hence, this setup is exactly suitable to be defined as an MIL problem. We evaluate our model on the two standard benchmark data sets for the drug activity prediction: MUSK1 and MUSK2. The MUSK1 data set consists of 47 positive bags, 45 negative bags and 476 instances overall. Each instance is represented by a 166 dimensional feature vector. MUSK2 consists of 39 positive bags, 63 negative bags and 6598 instances of 166 dimensions.

6.6.2 Image classification

Content based image classification is another application domain for multiple instance learning. Each image consists of several regions, only one of which contains an object of interest. When only image-level labels are available, such as the image contains a dog or the image does not contain a dog the task to identify all images which contain a dog can be cast as a multiple instance learning problem. In the benchmark data set the images are segmented into different regions, each segmented region is described by a 230 dimensional feature vector. Each bag corresponds to a single image and each region of an image constitutes an instance. All data sets are grouped into 100 positive and 100 negative bags. The Elephant dataset consists of 1391, the Tiger data set of 1220 and the Fox data set of 1320 instances.

6.6.3 Influence of proposed extensions

As seen in Table 6.1, all extensions except constraining the minimum leaf node size contribute to prediction accuracy, and the highest performance is achieved when the extensions are employed altogether. Most pronounced is the influence of combining multiple decision trees into a forest, which is indicated by the low accuracy of the single tree model. The difference of our single tree accuracy in comparison to the MITI method can be explained by the randomized training procedure that we employ to ensure uncorrelated trees. To investigate the influence of the forest reponse optimization, we replaced this part of our model with the simpler forest vote bias correction step on the instance predictions of the forest. We found that the response optimization is an essential part of the proposed model, as can be seen in Table 6.1. Deactivating also the forest vote bias correction from Section 6.5.3 leads to predictions at the chance level. Less pronounced but still positive is the contribution of the Ellipsoid split that we propose. The leaf node regularization via a minimum leaf node size and a redistribution of the positive weights does not have a visible positive effect. The Musk2 data set benefits from this extension while the effect on the Musk1 data set is detrimental.

6.6.4 Discussion

Table 6.2 shows the accuracy of the models in comparison. The top-most group contains decision tree based algorithms including our proposed model (MIOForest). Our model gives the highest accuracy in this group of models in all five data set, while in Musk2 data set it is tied with [16].

The group below it contains kernel-based based methods, which are either built on SVMs or Gaussian processes. In terms of average performance over the five data sets, our model ranks as fourth slightly after PC-SVM [52] and [23], and GP-MIL [63]. Among these models, PC-SVM depends on a Quadratically Constrained Quadratic Programming (QCQP), and GP-MIL involves inversion of a data-sized matrix, which make these models asymptotically more complex than the method we propose. The average prediction performance of our model is 2% less than these expensive models. We think the proposed algorithm is a worthwhile new approach to the multiple instance learning problem that inherits the nice computational properties of the random forest: $\mathcal{O}(n_{\text{features}} \times N \log N)$ per tree and its inherent parallelism during training of the individual trees.

The third group from the top consists of two variants of an algorithm that is based on a plain SVM with a special type of kernel function that calculates the dissimilarity between two bags. Due to the fact that each bag is represented as a single instance during classification, these algorithms are very fast. In addition their accuracy is very competitive. However, the problem of these algorithms is that they are able to make predictions only at the bag level, as opposed to all the other algorithms in comparison, including the method we propose.

The methods in the last group includes methods depending on various other approaches. MILES [31] and SIL-SVM [23] basically solve the MIL problem using single-instance learning, and EM-DD [133] uses a diverse density based approach combined with expectation maximization.

6.7 Conclusion

We have extended the multi instance tree learning algorithm proposed in [16] with a regularization and weight redistribution scheme, inside/outside split concepts, randomization and a leaf node response optimization. We have shown that our method exhibits superior performance to other decision tree based algorithms on a number of benchmark data sets, and that its accuracy is only slightly worse than rather expensive MIL algorithms that yield instance-level predictions.

There exist several interesting future extensions of the model we propose. For instance, exploring additional ways to control the witness/non-witness co-occurrences in positive bags is a promising strategy for future work. Employing projection constraints towards the most discriminative direction in a manner introduced in [52] easily be added to our method at the response optimization stage of the algorithm. In addition, the idea from [136] to treat the instance from the bags as non i.i.d. samples is another attractive direction of research that might be incorporated in the response optimization stage. Furthermore the instance-to-bag principle presented in [126] could be beneficial also for the decision tree based approach used here.

Table 6.1: Analysis on the influence of the proposed extensions. We analyze the impact of the extensions by comparing the accuracies between including and excluding each extension. The results illustrate that all extensions except constraining the minimum leaf node size have a positive influence on model accuracy, and the highest influence comes from response optimization. The best performer version of the algorithm on each data set is shown in bold.

	Drug Activity		**Image Classification**			
	Musk1	**Musk2**	**Elephant**	**Fox**	**Tiger**	**Average**
MIOForest	**89**	87	**86**	**68**	**83**	**82**
MIOForest w.o. forest (single randomized tree)	79	75	79	62	74	74
MIOForest w.o. Ellipsoid Split	87	86	86	67	82	82
MIOForest w.o. response optimization	82	79	81	65	81	78
MIForest w.o response optimization and bias correction	51	50	50	50	38	48
MIOForest with min. leaf size	88	**88**	85	**68**	**83**	**82**

Table 6.2: Five times 10-fold Cross-validated prediction accuracies of our model (MIOForest) and the existing methods on MUSK1 and MUSK2 drug activity prediction data sets and classification of elephant, fox, and tiger images. The best performer among decision tree based models is shown in bold, and the best performer among all models is shown in bold and italics. Our model gives the highest accuracies in four of the five data sets among decision tree based models, and its performance is slightly worse than more expensive methods.

		Drug Activity		**Image Classification**			
Category	**Method**	**Musk1**	**Musk2**	**Elephant**	**Fox**	**Tiger**	**Average**
Decision tree based methods	our method	**89**	87	**86**	**68**	**83**	**82**
	MIForest [76]	85	82	84	64	82	79
	MITI [16]	84	**88**	N/A	N/A	N/A	N/A
	RandomForest [19]	85	78	74	60	77	75
Kernel based methods	MI-Kernel [2]	88	89	84	60	84	81
	MI-SVM [2]	78	84	81	59	84	77
	mi-SVM [2]	87	84	82	58	79	78
	AW-SVM [49]	86	84	82	64	83	80
	AL-SVM [49]	86	83	79	63	78	78
	MILBoost-Nor [125]	71	61	73	58	56	63
	sbMIL [23]	***92***	88	89	***70***	83	**84**
	PC-SVM [52]	91	***91***	***89***	66	84	**84**
	GPMIL [63]	89	87	84	66	***88***	83
Methods predicting only bag labels	mi-Graph [136]	90	90	87	62	86	83
	MI-Graph [136]	89	90	85	61	82	81
Other methods	MILES [31]	88	83	81	62	80	79
	EM-DD [133]	85	85	78	56	72	75
	SIL-SVM [23]	88	87	85	53	77	78

Chapter 7

Lazyflow: flow graph based computation framework

Lazyflow is a flow graph based computation framework written in the Python programming language. It is the cornerstone of all computations in ilastik [110], the interactive learning and segmentation toolkit.

Expressing a computation as a data flow graph is often beneficial. It allows to analyze dependencies easily and to reuse subblocks of the computation. A lazyflow graph is a directed acyclic graph that expresses a computation. A data flow graph in lazyflow consists of so-called operators and the edges connecting different operators. An operator is a small unit of computation that has a set of input slots and output slots. These slots are used to define the inputs to the computation by connecting them either to the outputs of a predecessor operator or by connecting them to data directly.

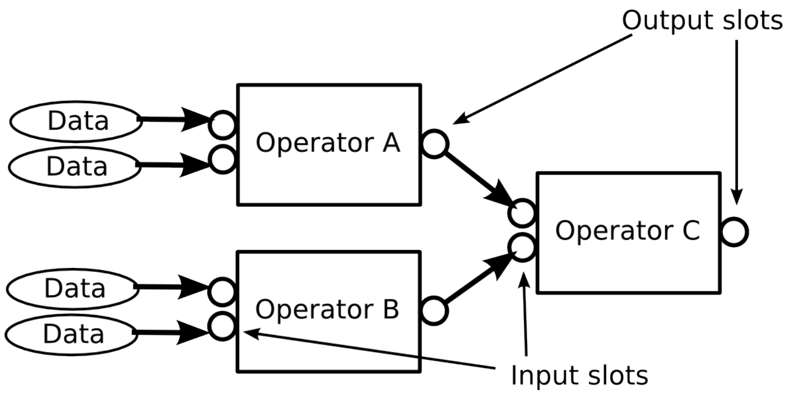


Figure 7.1: Illustrations of a data flow graph in lazyflow. Units of computations are expressed as operators whose inputs are either connected to a data source or to the output of another operator.

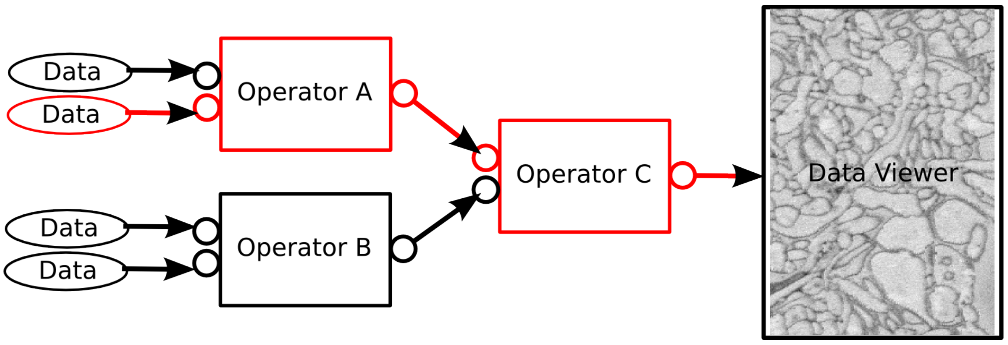


Figure 7.2: Illustrations of dirty propagation in lazyflow. Once some input to a calculation is changed (red Data on the left) this change is propageted through the graph (red) by notifying the dependent successors. This mechanism is highly useful in conjunction with a graphical user interface.

There are two different ways to compute the calculation defined by such a data flow graph. The first way is a push based computation in which the graph is traversed left to right by calculating the output of each operator and feeding that output to the input of the next operator. This approach which is used by many flow graph based computation libraries has a big drawback however: when computing the result of an operator the required input to the next operator has to be known exactly. In Figure 7.1 the exact required inputs for Operator C have to be known, to be able to first compute the outputs of Operator A and B. If we want to be able to parametrize the computation of Operator A, for example by requesting only a subset of the full output of Operator A, the parametrization has to be known beforehand.

The second approach to execute the computation of a data flow graph is a pull based mechanism. The pull mechanism traverses the graph right to left by determining the inputs required for the calculation of an output and then requesting these inputs from the predecessor operators. This pull based mechanism has the advantage that output parametrizations do not have to be known beforehand. When the output of Operator C is requested, Operator C determines what the required input for this computation is and then proceeds by requesting that input from its predecessor operators A and B.

Another advantage of using a data flow graph to express a computation is the ability to track dependencies of the data which are encoded in the graph. This allows to determine changed outputs easily once the input to a computations changes which is a highly useful mechanism when combining such a flow graph with a graphical user interface (GUI). In Figure 7.2 such a dirty propagation is illustrated. When the user changes the parameter of a computation, this change is propagated through the graph and all dependent outputs are flagged as dirty. This notification mechanism can be used by a GUI to request the changed data and to update the displayed results.

The development of lazyflow was driven by several key requirements which constitue the main features of lazyflow:

- Python programming language. Since the application whose requirements lead to the development of the lazyflow library is written in Python the choice of the programming language was clear. Python allows to rapidly prototype and develop software due to its dynamic nature.
- Flow graph based computations. Expressing computations as data flow graphs allows easy reusing of components and automatic parallelization of computations at branchings of the flow graph.
- Lazy computations. ilastik is an end user application for biologists that allows to process massive amounts of data interactively. On such large datasets processing all data is not an option; only the small currently visible parts of the data should be processed to provide interactive response times.
- Automatic data dependency tracking. Lazyflow should provide the neccessary communication channels that allow to track dependencies in the computation pipeline. Changing a parameter in the flow graph should message this change to all dependent calculations and outputs.
- Intuitive application programming interface. Since ilastik computations should be developable by non-experts, the API of lazyflow should be simple to use.
- Rich set of notification mechanism for tight integration with a graphical user interface.

In the next sections we will describe the lazyflow computation framework that was developed with these key requirements in mind.

7.1 Related work

The advantages of a flow graph based computation pipeline, such as dependency tracking or parallelization have lead to many implementations of this paradigm. The best known example of a flow graph based programming environment is propably LabVIEW [93]. It is a graphical environment that allows to perform scientific calculations or data manipulations on measurements by connecting input nodes and computation nodes. LabVIEW also supports to extend the functionality by creating custom plugins, however the environment is proprietary and not free. Another example of such a graphical environment that allows to create a data processing pipeline is KNIME [66]. It is a open source java program especially tailored towards data analytics. It supports a wide range of data processing nodes and can easily be extended by writing plugins in the Java programming langauge. However it does not support partial region of interest calculations and is not suited for large multidimensional array processing. The use of the java programming language is another obstacle since the software developed in our group is written in python. A flow graph based computation library written in C++ and suited for multidimensional array processing is itk [55]. It allows to develop plugins and includes several processing nodes suitable for image manipulation and registration. Lazyflow was influenced by its capabilities but adds several important aspects such as higher level slots (Section 7.5) the ability of operator wrapping (Section 7.6) or composite operators (Section 7.7). OpenAlea [97] is a Python based open source framwork primarily aimed at the plant research community. It supports a graphical user interface for creating flow graphs from predefined processing nodes and allows to extend the functionality by writing custom nodes in Python. It lacks however support for partial computations, the handling of large multidimensional arrays and other important features of lazyflow such as the ones already mentioned. Another python based flow graph library is joblib [123]. It is however push based which prevents its use in a pull based computation framework that uses partial region of interest calculations. It also lacks the rich set of notification mechanisms neccessary for a tight gui integration that lazyflow provides.

7.2 Lazyflow operators

Lazyflow operators encapsulate a small part of a computation. The requirements from a user perspective on an operator are.

- Define inputs and outputs of the computation
- Define how properties of the output slots depend on properties of the input slots
- Define the computation
- Define how dirty inputs propagate to the outputs of a computation

These requirements are directly reflected in the application programming interface (API) of lazyflow, as we will see in the following sections.

Inheriting from the operator class and defining inputs and outputs

To implement an operator, the user has to derive from the Operator base class and has to implement several methods that define the behaviour of the operator.

```
from lazyflow.graph import Operator, InputSlot, OutputSlot
from lazyflow.stype import ArrayLike

class SumOperator(Operator):
    inputA = InputSlot(stype=ArrayLike)
    inputB = InputSlot(stype=ArrayLike)

    output = OutputSlot(stype=ArrayLike)
    def setupOutputs(self):
        pass

    def execute(self, slot, subindex, roi, result):
        pass

    def propagateDirty(self, slot, subindex, roi):
        pass
```

The setupOutputs method

This method is called when all InputSlots of an operator are connected and ready. The connection partner can either be the OutputSlot of another operator or directly a data source. The method receives no arguments and is responsible for setting up the neccessary meta information for the output slots. Once an operator has set up the meta information of an OutputSlot, this OutputSlot will become ready. Meta information on the OutputSlots can contain such information as the shape of the OutputSlot or the data type of the OutputSlot. This information is not known beforehand and can only be determined once all InputSlots are connected. The most simple example is a pipe Operator which just copies its input to the output. The shape of the OutputSlot is only known once an InputSlot has been connected.

The execute method

The execute method receives several arguments: slot, subindex, roi and result. The execute method is called, when somebody requests data from an OutputSlot of an operator. The OutputSlot for which data was requested is given in the *slot* argument. In addition, when requesting data from an OutputSlot the user can specify a region of interest (roi). This region of interest that the user specified is given in the roi argument. The result argument holds a preallocated data area of correct size (compatible with the roi) into which the operator must write the result of its computation.

The propagateDirty method

The propagateDirty method receives as arguments a slot, subindex and a roi. This method is called by the operator base class whenever an InputSlot of the operator is set to dirty. This InputSlot is given in the slot argument. The region of interest of the slot which was set to dirty is given in the roi argument. The responsibility of the operator is to determine which OutputSlot becomes dirty in which region depending on the arguments that it was called with. In a simple case, such as an Operator which just pipes its InputSlot to the OutputSlot, the OutputSlot would be set to dirty in the same region that the InputSlot became dirty.

Simple SumOperator example

Below is a simple but fully functional example of an operator that has two input slots and one output slot. The computation which the operators performs is adding the two inputs.

```
from lazyflow.graph import Operator, InputSlot, OutputSlot
from lazyflow.stype import ArrayLike

class SumOperator(Operator):
    inputA = InputSlot(stype=ArrayLike)
    inputB = InputSlot(stype=ArrayLike)

    output = OutputSlot(stype=ArrayLike)

    def setupOutputs(self):
        # get the shape of the operator inputs
        shapeA = self.inputA.meta.shape
        shapeB = self.inputB.meta.shape

        # check that the inputs have the same shape
        assert shapeA == shapeB

        # setup the shape of the output slot
        self.output.meta.shape = shapeA

        # setup the dtype of the output slot
        self.output.meta.dtype = self.inputA.meta.dtype

    def execute(self, slot, subindex, roi, result):
        # the following two lines request the inputs of the
        # operator for the specifed region of interest

        a = self.inputA.get(roi).wait()
        b = self.inputB.get(roi).wait()

        # the result of the computation is written into the
        # pre-allocated result array

        result[:] = a+b

    def propagateDirty(self, slot, subindex, roi):
        # the method receives as argument the slot
        # which was changed, and the region of interest (roi)
        # that was changed in the slot

        # in this case the mapping of the dirty
        # region is simple, it corresponds exactly
        # to the region of interest that was changed in
        # one of the input slots
```

```
            self.output.setDirty(roi)
```

Using operators in lazyflow

To use an operator in lazyflow one first has to construct an instance of the operator:

```
# instantiate two SumOperators
op1 = SumOperator()
op2 = SumOperator()
```

Now these operators must receive some input, either by connecting them to another operator or by directly connecting them to a data source. An operator whose inputs are not connected is not functional because its setupOutputs method has not been called which implies that the OutputSlots of the operator are not yet ready. Below we give an example of how the inputs of the two operators can be connected:

```
# instantiate two SumOperators
op1 = SumOperator()
op2 = SumOperator()

# first we provide some data to op1 using the
# setValue method of its InputSlots
op1.inputA.setValue(numpy.zeros((10,20)))
op1.inputB.setValue(numpy.ones((10,20)))

# next we connect op2 to the output of op1
op2.inputA.connect(op1.output)
# and now connect the remaining InputSlot of op2 directly to some data
op2.inputB.setValue(numpy.ones((10,20)))
```

The above example is now a fully functional lazyflow graph which can be executed. To perform the calculation we request the data from the OutputSlot output:

```
result = op2.output[:].wait()
```

7.3 Lazyflow slots

Lazyflow slots serve as the input and output facilities for lazyflow operators. Input and output slots for an operator are defined as follows:

```
from lazyflow.graph import Operator, InputSlot, OutputSlot
from lazyflow.stype import ArrayLike

class SumOperator(Operator):
    inputA = InputSlot(stype=ArrayLike)
    inputB = InputSlot(stype=ArrayLike)

    output = OutputSlot(stype=ArrayLike)
```

In an operator instance the slots can be accessed by their name and provide methods to connect to other slots or to connect to data directly:

```
# instantiate a SumOperators
op = SumOperator()

# connecting to another slot
op.inputA.connect(other_operator.output)

# setting a data value directly
op.inputB.setValue( some_data )
```

In addition to these methods already discussed, slots provide many notification mechanisms which are very useful when using lazyflow in the context of a graphical user interface application.

notifyDirty

The notifyDirty method of a slot notifies an subscriber of a dirty event. Such dirty events are propagated through a lazyflow graph to all dependent operators and outputs. A graphical user interface that displays some computation results of an output slot can subscribe to this event and update the displayed data once it becomes dirty. Such a dirty notification can be triggered for example by changing a computation parameter.

```
some_operator.some_slot.notifyDirty(callback)
```

After registering a callback the corresponding callback function is called when the slot gets dirty. First argument of the callback is the slot, second argument the dirty region of interest (roi).

notifyMetaChanged

The notifyMetaChanged method of a slot can be used to register a callback for meta change events on this slot. Meta information of a slot may contain information such as the dimensionality or shape of a computation result. A meta information change event is triggered for example when connecting a data set of different shape or dimensionality to an input slot. In this case, all dependent output slots will also have to change the shape of the computation result and trigger such an event. A graphical user interface may register to this type of event by registering a callback, the callback can then update the displayed data or reconfigure the data view.

```
def callback(slot):
    # do some work
    pass

some_operator.some_slot.notifyMetaChanged(callback)
```

notifyReady

The notifyReady method of a slot can be used to register a callback like this:

```
def callback(slot):
    # do some work
    pass

some_operator.some_slot.notifyReady(callback)
```

The callback is called whenever a slot becomes ready. A ready slot in lazyflow is a slot from which computation results can be requested. Unready slots throw an error when a computation is requested. Such a notification is useful in the context of a graphical user interface, since the data view may only query for computation results on ready slots.

notifyUnready

The notifyUnready method is the counterpart of the notifyReady notification. The callback is called whenever a slot becomes unready. An unready slot in lazyflow cannot be used to request a calculation result, thus a graphical user interface, which displays some computation result of a slot, may want to register to this kind of event to prevent from trying to display unready data.

```
def callback(slot):
    # do some work
    pass

some_operator.some_slot.notifyUnready(callback)
```

notifyConnect

The connect event is triggered once a lazyflow slot is connected to a partner. This type of event is mostly used internally by lazyflow. An operator registers for this kind of event on all of its input slots and calls its setupOutputs method once all inputs are properly connected. The notifyConnect method of a slot can be used to register a callback like this:

```
def callback(slot):
    # do some work
    pass

some_operator.some_slot.notifyConnect(callback)
```

notifyDisconnect

The notifyDisconnect method of a slot is the counterpart of the notifyConnect method. It is also mainly used internally by lazyflow. An operator registers for this kind of event for all its input slots. Once an input slot becomes disconnected, all output slots of the operator are set to unready. In addition, when the slot becomes connected again, this recalls the setupOutputs method of the operator.

```
def callback(slot):
    # do some work
    pass

some_operator.some_slot.notifyDisconnect(callback)
```

7.4 Lazyflow requests

Lazyflow requests encapsulate a computation. They allow for easy cancellation or notification of computations. All queries for computation results to lazyflow operators return a lazyflow request object that can be used to handle the computation which is processed in the background by a threadpool.

A simple example of the creation of a request object in the context of a lazflow operator looks like this

```
# instantiate a SumOperators
op = SumOperator()

# querying the output slot returns a request object
request = op.output[:]
```

Creating custom requests

While lazyflow operator slots return request objects when queried for output, a request object can also be created outside lazyflow to encapsulate a parallel computation. An example is given below:

```
from some_img_lib import smooth
from functools import partial
from lazyflow.request import Request

def f(image, sigmaA, sigmaB):
    r2 = Request( partial(smooth, image, sigmaA) )
    r3 = Request( partial(smooth, image, sigmaB) )

    # Start executing r3
    r3.submit()

    # Wait until both requests are complete
    smoothedA = r2.wait() # (Auto-submits)
    smoothedB = r3.wait()

    result = smoothedA - smoothedB
    return result
```

```
r1 = Request( partial(f, my_image, 1.0, 3.0) )
diff_of_smoothed = r1.wait()
```

Request submission

The computation encapsulted by a lazyflow request is started via its submit method. The computation is executed by a threadpool in the background.

```
request.submit()
```

Synchronous waiting for a request

Instead of submitting a request for background processing, the user can also decide to synchronously wait for the computation result via the requests wait method. This is a blocking method call that returns the result of the computation.

```
result = request.wait()
```

Request cancellation

Once a request has been submitted for processing, the user can cancel the computation via the request's cancel method:

```
requestA.cancel()
```

Lazyflow internally tracks all requests that have been created by requestA and cancels these requests too.

Request computation finished notification

Once a request has been submitted for background processing, the user can specify a callback that is executed once the request has been processed and its computation is finished. If the computation was already finished before the user specified the callback, the callback is executed immediately.

```
request.notify_finished(callback)
```

It is important to know that this callback is executed in a thread of the threadpool.

Request computation cancelled notification

The user can also specify a callback that is executed after a request has been cancelled:

```
request.notify_cancelled(callback)
```

Request computation failed notification

Since a computation encapsulated by a request can fail with an exception, we provide a callback mechanism for a corresponding notification:

```
request.notify_failed(callback)
```

This callback is executed in the context of a thread from the threadpool.

7.5 Higher level slots

Consider a lazyflow operator that works on multiple images at once. An example would be a sum operator that adds more than two images. Implementing such an operator is simple if the number of images on which the operator works is fixed: just define as many input slots as needed. But if the operator should work with a variable number of inputs the situation is more complicated. To handle this use case lazyflow supports so-called higher level slots. These slots act as a list of inputs. A slot of level 1 consists of a list of input slots, a slot of level 2 consists of lists of lists of input slots. These higher level slots are defined like this:

```
from lazyflow.graph import Operator, InputSlot, OutputSlot
from lazyflow.stype import ArrayLike

class SumOperator(Operator):
    input = InputSlot(level = 1) # note the level keyword argument !
```

Accessing such higher level slots inside and outside an operator is simple since they behave like a Python list:

```
sumop = SumOperator()

sumop.input[0] # this accesses the first slot inside the sumop.input higher
     level slot
```

Since higher level slots behave like a Python list they have a certain length or size. To resize a higher level slot to a specified number of elements the resize function can be used:

```
sumop.input.resize(4) # resize the input level 1 slot to contain 4 input
    slots
```

Such resizing happens automatically when an input level 1 slot is connected to another output level 1 slot. In this case, the input slot is resized to match the number of elements in the output slot to which it is connected.

To allow the developer of an operator or graphical user interface to react to such resize events higher level slots support a wider range of notification events which we present now.

notifyResize

Calls the corresponding callback function before the slot is resized. Such a resize event can occur for example when connecting a slot to a slot of different size or when manually calling the resize method of a slot. The first argument of the function is the slot, the second argument is the old size and the third argument is the new size.

```
def callback(slot, old_size, new_size):
  # do something
  pass

operator.level1slot.notifyResize(callback)
```

notifyResized

Setting up this notification calls the corresponding callback function after the slot is resized. The first argument of the function is the slot, the second argument is the old size and the third argument is the new size.

```
def callback(slot, old_size, new_size):
  # do something
  pass

operator.level1slot.notifyResized(callback)
```

notifyRemove

Lazyflow will call the specified callback function before a slot is removed. Such a slot removal can happen during a resize event of a slot when the size of the slot is decreased. The first argument of the function is the slot, the second argument is the old size and the third argument is the new size.

```
operator.level1slot.notifyRemove(callback)
```

notifyRemoved

Calls the given callback function after a slot is removed. The first argument of the function is the slot, the second argument is the old size and the third argument is the new size.

```
operator.level1slot.notifyRemoved(callback)
```

notifyInserted

Allows to specify a callback function which is called after a slot has been added. Such an event can occur during a resize operation when the size of a slot is increased. The first argument of the function is the slot, the second argument is the old size and the third argument is the new size.

```
operator.level1slot.notifyResize(callback)
```

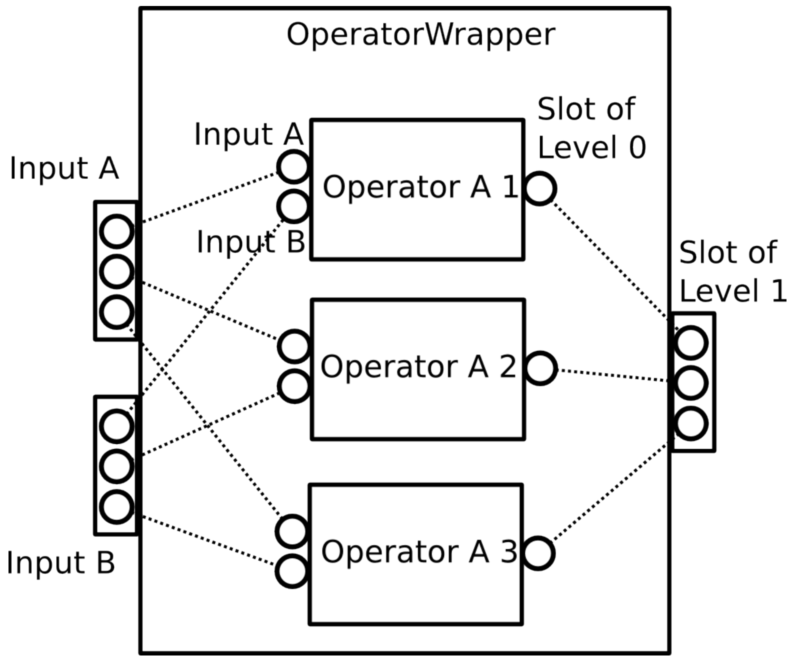


Figure 7.3: Illustrations of the OperatorWrapper class. It replicates the input and outputs slots of a given operator (OperatorA) with a level of +1, i.e. for each slot it creates a list of slots. The slots of this list slots are forwarded to the inner slots of instances of the wrapped operator.

notifyPreInsertion

Calls the given callback function immediately before a slot is going to be inserted into a multi-slot. Same signature as the notifyInserted signal.

```
operator.level1slot.notifyPreInsertion(callback)
```

7.6 The OperatorWrapper

Consider a lazyflow computation graph where all operators work on lists of images at once. I.e. all operators in the graph have output and input slots of level 1. Now assume we want to apply another operator, say a threshold operator, at the end of the computation pipeline. If we already have implemented such a threshold operator which works on a single image, i.e. a level 0 slot, it would be a burden to write another threshold operator which justs can work on multiple images. Exactly for this use case, there exists the OperatorWrapper class. It takes a given lazyflow operator as input, replicates all its inputs and outputs, but with a level of +1. It then reacts to all kinds of slot resize events and instantiates or deletes instances of the input operator and connects newly created instances of created operators to its input and output slots.

I.e. the wrapped operator behaves exactly like the given input operator, but works on many inputs at once. All of this comfort can be used with a single line of code:

```
opMultiThreshold = OperatorWrapper( OpThreshold ) # creates a higher level
    +1 operator from a given input operator
```

This opMultiThreshold can now be used in the assumed lazyflow computation graph without having to rewrite a threshold operator just for multiple images.

7.7 Composite operators

While developing an operator one often realizes that parts of the needed functionality already exist in form of other operators. Combining many operators into a single composite operator is a frequent use case and lazyflow supports this functionality. Existing operators can be instantiated and connected to input and output slots easily in the init method of the composite operator. An example is given below.

```
from lazyflow.graph import Operator, InputSlot, OutputSlot
from lazyflow.stype import ArrayLike
import numpy

class SumThresholdOperator(Operator):
    inputA = InputSlot(stype=ArrayLike)
    inputB = InputSlot(stype=ArrayLike)
    threshold = InputSlot()

    output = OutputSlot(stype=ArrayLike)

    def __init__(self):
        # instantia a SumOperator
        self.sumOp = SumOperator()

        # forward the input slots to the sum operator
        self.sumOp.inputA.connect(self.inputA)
        self.sumOp.inputB.connect(self.inputB)

    def execute(self, slot, subindex, roi, result):
        # request the result from the SumOperator
        sumResult = self.sumOp.output.get(roi).wait()

        # get the threshold value
        threshold = self.threshold.value

        # threshold and return the sumResult
        return numpy.where(sumResult > threshold, 1, 0)
```

In the above example the setupOutputs and propagateDirty methods we left out for brevity.

7.8 Summary

Lazyflow is a flow graph computation library. It was designed around several key requirements. The first requirement was the Python programming language since ilastik, the interactive learning and segmentation toolkit is written in this language and lazyflow should serve as its computation backend. The second requirement was the flow graph based computation principle: we want developers to be able to quickly prototype and develop a new pipeline by reusing existing code fragments. These code fragments are encapsulated in so-called lazyflow operators with defined input and output slots which can easily be chained together. The third requirement was the lazy computation principle: the flow graph computation is not processed in a push based manner, but in a pull based manner. Thus when requesting output from an operator, the user can specify a region of interest for the results and only this region of interest is actually computed. Each operator backprojects the region of interest to its own inputs and in turn only requests subsets of the results from its predecessor operators. In this way a graphical user interface can be developed which supports fast response times since only visible subsets of the data need to be processed. A fourth requirement was the need for automatic parallel processing of the computation requests. This lead to the development of the request system, which encapsulates a computation. Each query for a computation result returns a request object, which is processed in the background by a thread pool. When an operator triggers multiple requests to upstream operators, all these requests are processed in parallel. Another requirement was the need to work on multiple images at once, which is an interaction mode ilastik supports. This requirement lead to the development of the so-called higher level slots which act as lists of input slots. To support reusing single image operators for such multi image pipelines lazyflow provides the Operator-Wrapper class which conveniently takes care of instantiating as many single image operators as needed and takes care of connecting and configuring the input and output slots. Reusing existing operator inside newly created operators is supported with the composite operator principle. In lazyflow it is easy to instantiate already available operators inside a new operator and to reuse code. The requirements a graphical user interface demands lead to the various notification and callback mechanisms that all lazyflow components such as slots, requests and higher level slots support. These callbacks make the implementation of a graphical user interface easy and provide the necessary means for communicating with lazyflow.

Chapter 8

Conclusion

Interactive segmentation is an important paradigm in image processing. However, the initial segmentation result may not always be what the user expects. Spotting erroneous areas of the segmentation is easy in 2D: the user can examine the segmentation result with one glance. In 3D segmentation tasks finding errors is much more laborsome, as the user has to inspect many orthogonal slice views of the data to find mistakes of the algorithm. In Chapter 2 I investigated this problem in the context of the seeded watershed cut and proposed several uncertainty measures which guide the user effectively to potentially interesting regions of the segmentation which may need correction. The proposed uncertainty measures were evaluated and shown to be superior to a simple local margin based measure. In fact, the best of the proposed uncertainty measures results in performance close to a ground truth oracle.

Another important problem in the context of segmentation algorithms is the enumeration of the M-best solutions. These M-best solutions can be used to evaluate the segmentation results with regard to higher order priors which cannot be incorporated into the energy formulation directly due to their computational hardness. Another interesting application of the M-best solutions is to find different modes of the segmentation which differ from the lowest energy solution. These differing solutions may be close in terms of the energy but far in terms of the Hamming distance of the segmentation result. Fusing different modes of the segmentation into a consistent result or deriving a segmentation uncertainty from the different modes are natural applications of the M-best solutions. This problem has been studied in the literature recently in the context of Markov random fields. In Chapter 3 I study the M-best solutions problem in the context of watershed cuts and provide an algorithm that enumerates many unique segmentations in the order of increasing cost.

Another interesting question in the context of the seeded watershed cut segmentation algorithm is the choice of the edge weights for a segmentation problem. Usually these edge weights are hand crafted for each segmentation task and often simple image filters such as the image gradient or the largest eigenvalue of the Hessian matrix are used, depending on the type of the segmentation problem at hand. In Chapter 4 I propose a decision tree type algorithm that can partition an image into an unknown number of components by learning a combination of edge features that represents an image boundary. The algorithm is interesting since it uses a global structured split criterion defined on the connectivity structure of a graph and requires only very weak annotations, namely region scribbles, to learn a boundary model from the must-link and cannot-link constraints expressed by the user given annotations. Other approaches require precise boundary annotations and thus much greater labeling effort. The learned boundary model that is encoded in the decision tree can either be used directly to segment an unseen image without supervision, or it can be used to calculate a boundary probability for each edge using a randomized ensemble of such decision trees. This boundary probability for each edge can in turn be used as input for a seeded watershed cut segmentation algorithm.

In Chapter 5 I propose another way to solve the same problem. I express the must-link and cannot-link constraints inherent in the user given scribbles as a multiple instance learning problem. In this setting the must-link constraints correspond to negative bags and the paths connecting two cannot-link scribbles correspond to positive bags. The positive bag in a multiple instance learning problem must contain at least one positive sample which I relate to a boundary which must be crossed by a path connecting two cannot-link strokes. I study if this intuitive mapping of a weakly supervised boundary learning problem can be solved using of the shelf multiple instance learning algorithms. Unfortunately the resulting segmentations are of low quality, nearly all image pixels outside the user given scribbles are predicted as the boundary class. From the multiple instance learning algorithm point of view, this is a correct solution, but from a practical point of view this is not a desirable outcome.

The usage of multiple instance learning algorithms in the context of the boundary learning problem was inspiring and lead to the development of a new multiple instance learning algorithm in Chapter 6. The algorithm is based on decision trees that make use of the positive identifiability constraint during learning. The outputs of many such randomized decision trees are linearly combined in a fashion that obeys the multiple instance constraints. Experiments on the standard benchmark datasets show that the proposed algorithm outperforms previous decision tree type algorithms.

Finally, in Chapter 7 a flow graph based computation framework was presented. Its key features are the pull based computation principle which allows for lazy region of interest computations that are important in the context of a graphical user interface to allow partial computations of visible areas. Furthermore, the framework supports push based dirty notifications upon parameter changes, intuitive development of computation nodes, semiautomated parallelization at branchings of the data flow graph, a rich notification infrastructure which is necessary in the context of a graphical user interface and advanced concepts like higher level slots and composite operators.

List of Publications

Peer-reviewed Conference Proceedings

- C. Straehle, M. Kandemir, U. Köthe, F.A. Hamprecht: Multiple instance learning with response-optimized random forests. in: International Conference on Pattern Recognition (ICPR), 2014
- C. Straehle, U. Köthe, F.A. Hamprecht: Weakly supervised learning of image partitioning using decision trees with structured split criteria in: IEEE International Conference on Computer Vision (ICCV), 2013
- C. Straehle, S. Peter, U. Köthe, F.A. Hamprecht: K-smallest spanning tree segmentations in: German Conference on Pattern Recognition (DAGM/GCPR), 2013
- S. Wanner, C. Straehle, B. Goldlücke: Globally Consistent Multi-Label Assignment on the Ray Space of 4D Light Fields in: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2013
- C. Straehle, U. Köthe, K. Briggman, W. Denk, F.A. Hamprecht: Seeded watershed cut uncertainty estimators for guided interactive segmentation in: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2012
- C. N. Straehle, U. Köthe, G. Knott, F. A. Hamprecht: Carving: Scalable Interactive Segmentation of Neural Volume Electron Microscopy Images in: International Conference on Medical Image Computing and Computer Assisted Intervention (MICCAI), 2011

Bibliography

[1] Bjoern Andres, Jörg H Kappes, Thorsten Beier, Ullrich Köthe, and Fred A Hamprecht. Probabilistic image segmentation with closedness constraints. In *International Conference on Computer Vision*, pages 2611–2618. IEEE, 2011. 37

[2] Stuart Andrews, Ioannis Tsochantaridis, and Thomas Hofmann. Support vector machines for multiple-instance learning. *Advances in Neural Information Processing Systems*, 15:561–568, 2002. 50, 55, 56, 68

[3] Jesús Angulo and Dominique Jeulin. Stochastic watershed segmentation. In *International Symposium on Mathematical Morphology*, volume 8, pages 265–276, 2007. 13, 16

[4] Boris Babenko, Nakul Verma, Piotr Dollár, and Serge Belongie. Multiple instance learning with manifold bags. In *International Conference on Machine Learning*, pages 81–88, New York, NY, USA, 2011. ACM. 56

[5] Boris Babenko, Ming-Hsuan Yang, and Serge Belongie. Robust object tracking with online multiple instance learning. *Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1619–1632, 2011. 57

[6] Shai Bagon. Boundary driven interactive segmentation. In *Conference on Information Science and Applications*, pages 1–5. IEEE, 2012. 37, 50

[7] Xue Bai and Guillermo Sapiro. Geodesic matting: A framework for fast interactive image and video segmentation and matting. *International Journal of Computer Vision*, 82(2):113–132, 2009. 44

[8] William A Barrett and Eric N Mortensen. Interactive live-wire boundary extraction. *Medical Image Analysis*, 1(4):331–341, 1997. 37

[9] Dhruv Batra, Adarsh Kowdle, Devi Parikh, Jiebo Luo, and Tsuhan Chen. icoseg: Interactive co-segmentation with intelligent scribble guidance. In *Conference on Computer Vision and Pattern Recognition*, pages 3169–3176. IEEE, 2010. 11, 13

[10] Dhruv Batra, Payman Yadollahpour, Abner Guzman-Rivera, and Gregory Shakhnarovich. Diverse m-best solutions in markov random fields. In *European Conference on Computer Vision*, pages 1–16. Springer, 2012. 25, 26, 31, 33

[11] Asa Ben-Hur et al. Multiple instance learning of calmodulin binding sites. *Bioinformatics*, 28(18):i416–i422, 2012. 57

[12] Charles Bergeron, Gregory Moore, Jed Zaretzki, Curt M Breneman, and Kristin P Bennett. Fast bundle algorithm for multiple-instance learning. *Transactions on Pattern Analysis and Machine Intelligence*, 34(6):1068–1079, 2012. 56

[13] Elena Bernardis and Stella X Yu. Pop out many small structures from a very large microscopic image. *Medical Image Analysis*, 15(5):690–707, 2011. 5

[14] Andreas Bieniek, Alina Moga, et al. A connected component approach to the watershed segmentation. *Computational Imaging and Vision*, 12:215–222, 1998. 5

[15] Andrew Blake, Pushmeet Kohli, and Carsten Rother. *Markov random fields for vision and image processing*. MIT Press, 2011. 26

[16] Hendrik Blockeel, David Page, and Ashwin Srinivasan. Multi-instance tree learning. In *International Conference on Machine learning*, pages 57–64. ACM, 2005. 56, 57, 58, 59, 60, 61, 65, 66, 68

[17] Hans L Bodlaender and Thomas Wolle. A note on the complexity of network reliability problems. *Transactions on Information Theory*, 47:1971–1988, 2004. 17

[18] Yuri Y Boykov and M-P Jolly. Interactive graph cuts for optimal boundary & region segmentation of objects in nd images. In *International Conference on Computer Vision*, volume 1, pages 105–112. IEEE, 2001. 4, 13, 25, 44

[19] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. 9, 56, 58, 64, 68

[20] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A. Olshen. *Classification and regression trees*. CRC press, 1984. 8, 58

[21] William Brendel and Sinisa Todorovic. Segmentation as maximum-weight independent set. In *Advances in Neural Information Processing Systems*, pages 307–315. MIT Press, 2010. 25

[22] Kevin L Briggman and Winfried Denk. Towards neural circuit reconstruction with volume electron microscopy techniques. *Current Opinion in Neurobiology*, 16(5):562–570, 2006. 32

[23] Razvan C Bunescu and Raymond J Mooney. Multiple instance learning for sparse positive bags. In *International Conference on Machine learning*, pages 105–112. ACM, 2007. 65, 68

[24] Albert Cardona, Stephan Saalfeld, Stephan Preibisch, Benjamin Schmid, Anchi Cheng, Jim Pulokas, Pavel Tomancak, and Volker Hartenstein. An integrated micro-and macroarchitectural analysis of the drosophila brain by computer-assisted serial section electron microscopy. *PLoS biology*, 8(10):e1000502, 2010. 47

[25] Albert Cardona, Stephan Saalfeld, Johannes Schindelin, Ignacio Arganda-Carreras, Stephan Preibisch, Mark Longair, Pavel Tomancak, Volker Hartenstein, and Rodney J Douglas. Trakem2 software for neural circuit reconstruction. *PLoS One*, 7(6):e38011, 2012. 47

[26] Vicent Caselles, Francine Catté, Tomeu Coll, and Françoise Dibos. A geometric model for active contours in image processing. *Numerische mathematik*, 66(1):1–31, 1993. 3

[27] Vicent Caselles, Ron Kimmel, and Guillermo Sapiro. Geodesic active contours. *International Journal of Computer Vision*, 22(1):61–79, 1997. 3

[28] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000. 21

[29] Christophe Chefd'Hotel and Alexis Sebbane. Random walk and front propagation on watershed adjacency graphs for multilabel image segmentation. In *International Conference on Computer Vision*, pages 1–7. IEEE, 2007. 4, 13

[30] Chao Chen, Daniel Freedman, and Christoph H Lampert. Enforcing topological constraints in random field image segmentation. In *Conference on Computer Vision and Pattern Recognition*, pages 2089–2096. IEEE, 2011. 14

[31] Yixin Chen, Jinbo Bi, and James Ze Wang. Miles: Multiple-instance learning via embedded instance selection. *Transactions on Pattern Analysis and Machine Intelligence*, 28(12):1931–1947, 2006. 56, 60, 65, 68

[32] Camille Couprie, Leo Grady, Laurent Najman, and Hugues Talbot. Power watersheds: A new image segmentation framework extending graph cuts, random walker and optimal spanning forest. In *International Conference on Computer Vision*, pages 731–738. IEEE, 2009. 13, 25, 44

[33] Camille Couprie, Leo Grady, Laurent Najman, and Hugues Talbot. Power watershed: A unifying graph-based optimization framework. *Transactions on Pattern Analysis and Machine Intelligence*, 33(7):1384–1399, 2011. 4

[34] Camille Couprie, Laurent Najman, and Hugues Talbot. Seeded segmentation methods for medical image analysis. In *Medical Image Processing*, pages 27–57. Springer, 2011. 5

[35] Jean Cousty, Gilles Bertrand, Laurent Najman, and Michel Couprie. Watershed cuts: Minimum spanning forests and the drop of water principle. *Transactions on Pattern Analysis and Machine Intelligence*, 31(8):1362–1374, 2009. 4, 5, 7, 8, 12, 13, 14, 25, 26

[36] Jean Cousty, Laurent Najman, Michel Couprie, Stéphanie Clément-Guinaudeau, Thomas Goissen, and Jerôme Garot. Segmentation of 4d cardiac mri: Automated method based on spatio-temporal watershed cuts. *Image and Vision Computing*, 28(8):1229–1243, 2010. 5

[37] Antonio Criminisi, Jamie Shotton, and Ender Konukoglu. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Foundations and Trends in Computer Graphics and Vision*, 7(2–3):81–227, 2012. 9, 60

[38] Winfried Denk and Heinz Horstmann. Serial block-face scanning electron microscopy to reconstruct three-dimensional tissue nanostructure. *PLoS biology*, 2(11):e329, 2004. 21

[39] Thomas G Dietterich, Richard H Lathrop, and Tomás Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial intelligence*, 89(1):31–71, 1997. 50, 55, 56, 57, 64

[40] H Digabel and Christian Lantuéjoul. Iterative algorithms. In *European Symposium on Quantitative Analysis of Microstructures in Material Science, Biology and Medicine*, volume 19, page 8. Riederer Verlag, 1978. 5

[41] Piotr Dollar, Zhuowen Tu, and Serge Belongie. Supervised learning of edges and object boundaries. In *Conference on Computer Vision and Pattern Recognition*, volume 2, pages 1964–1971. IEEE, 2006. 37

[42] Alexandre X Falcão, Jorge Stolfi, and Roberto de Alencar Lotufo. The image foresting transform: Theory, algorithms, and applications. *Transactions on Pattern Analysis and Machine Intelligence*, 26(1):19–29, 2004. 26

[43] Alireza Fathi, Maria Florina Balcan, Xiaofeng Ren, and James M Rehg. Combining self training and active learning for video segmentation. In *British Machine Vision Conference*, volume 29, pages 78–1. BMVA Press, 2011. 11, 13

[44] Jerome H. Friedman and Bogdan E. Popescu. Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3):pp. 916–954, 2008. 62

[45] Menachem Fromer and Amir Globerson. An LP view of the M-best MAP problem. In *Advances in Neural Information Processing Systems*, volume 1, page 3. MIT Press, 2009. 25

[46] Gang Fu, Xiaofei Nan, Haining Liu, Ronak Patel, Pankaj Daga, Yixin Chen, Dawn Wilkins, and Robert Doerksen. Implementation of multiple-instance learning in drug activity prediction. *BMC Bioinformatics*, 13(Suppl 15):S3, 2012. 57

[47] Harold N Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM Journal on Computing*, 6(1):139–150, 1977. 6, 15, 25, 26, 27, 31

[48] Thomas Gärtner, Peter A Flach, Adam Kowalczyk, and Alex J Smola. Multi-instance kernels. In *International Conference on Machine Learning*, volume 2, pages 179–186. ACM, 2002. 56

[49] Peter V Gehler and Olivier Chapelle. Deterministic annealing for multiple-instance learning. In *International Conference on Artificial Intelligence and Statistics*, pages 123–130, 2007. 56, 68

[50] Leo Grady. Random walks for image segmentation. *Transactions on Pattern Analysis and Machine Intelligence*, 28(11):1768–1783, 2006. 4, 13, 25, 44

[51] Leo Grady and Ali Kemal Sinop. Fast approximate random walker segmentation using eigenvector precomputation. In *Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008. 4, 13

[52] Yanjun Han, Qing Tao, and Jue Wang. Avoiding false positive in multi-instance learning. In *Advances in Neural Information Processing Systems*, pages 811–819. MIT Press, 2010. 56, 65, 66, 68

[53] Ping He, Xiaohua Xu, and Ling Chen. Constrained clustering with local constraint propagation. In *European Conference on Computer Vision - Workshop*, pages 223–232. Springer, 2012. 37

[54] Matthias Heiler, Jens Keuchel, and Christoph Schnörr. Semidefinite clustering for image segmentation with a-priori knowledge. In *Pattern Recognition*, pages 309–317. Springer, 2005. 37

[55] ITK. Insight Segmentation and Registration Toolkit. `http://www.itk.org`. 72

[56] Viren Jain, Benjamin Bollmann, Mark Richardson, Daniel R Berger, Moritz N Helmstaedter, Kevin L Briggman, Winfried Denk, Jared B Bowden, John M Mendenhall, Wickliffe C Abraham, et al. Boundary learning by optimization with topological constraints. In *Conference on Computer Vision and Pattern Recognition*, pages 2488–2495. IEEE, 2010. 36, 37

[57] Viren Jain, H Sebastian Seung, and Srinivas C Turaga. Machines that learn to segment images: a crucial technology for connectomics. *Current Opinion in Neurobiology*, 20(5):653–666, 2010. 5

[58] Jeremy Jancsary, Sebastian Nowozin, and Carsten Rother. Loss-specific training of non-parametric image restoration models: A new state of the art. In *European Conference on Computer Vision*, pages 112–125. Springer, 2012. 37

[59] Jeremy Jancsary, Sebastian Nowozin, Toby Sharp, and Carsten Rother. Regression tree fields—an efficient, non-parametric approach to image labeling problems. In *Conference on Computer Vision and Pattern Recognition*, pages 2376–2383. IEEE, 2012. 37

[60] Johannes Jordan and Elli Angelopoulou. Supervised multispectral image segmentation with power watersheds. In *International Conference on Image Processing*, pages 1585–1588. IEEE, 2012. 5

[61] Armand Joulin and Francis Bach. A convex relaxation for weakly supervised classifiers. In *International Conference on Machine Learning*, pages 1279–1286, New York, NY, USA, 2012. ACM. 56

[62] Olivier Juan and Yuri Boykov. Active graph cuts. In *Conference on Computer Vision and Pattern Recognition*, volume 1, pages 1023–1029. IEEE, 2006. 4, 13

[63] Minyoung Kim and Fernando Torre. Gaussian processes multiple instance learning. In *International Conference on Machine Learning*, pages 535–542, 2010. 57, 65, 68

[64] Sungwoong Kim, Sebastian Nowozin, Pushmeet Kohli, and Chang D Yoo. Task-specific image partitioning. *Transactions on Image Processing*, 22(2):488–500, 2013. 37

[65] Sungwoong Kim, Sebastian Nowozin, Pushmeet Kohli, and Chang Dong Yoo. Higher-order correlation clustering for image segmentation. In *Advances in Neural Information Processing Systems*, pages 1530–1538. MIT Press, 2011. 37

[66] KNIME. Konstanz Information Miner. `http://www.knime.org`. 71

[67] Graham Knott, Herschel Marchman, David Wall, and Ben Lich. Serial section scanning electron microscopy of adult brain tissue using focused ion beam milling. *The Journal of Neuroscience*, 28(12):2959–2964, 2008. 21

[68] Pushmeet Kohli and Philip HS Torr. Measuring uncertainty in graph cut solutions–efficiently computing min-marginal energies using dynamic graph cuts. In *European Conference on Computer Vision*, pages 30–43. Springer, 2006. 13

[69] Iasonas Kokkinos, Rachid Deriche, Olivier Faugeras, and Petros Maragos. Computational analysis and learning for a biologically motivated model of boundary detection. *Neurocomputing*, 71(10):1798–1812, 2008. 37

[70] Scott Konishi, Alan L. Yuille, James M. Coughlan, and Song Chun Zhu. Statistical edge detection: Learning and evaluating edge cues. *Transactions on Pattern Analysis and Machine Intelligence*, 25(1):57–74, 2003. 37

[71] Peter Kontschieder, Samuel Rota Bulo, Horst Bischof, and Marcello Pelillo. Structured class-labels in random forests for semantic image labelling. In *International Conference on Computer Vision*, pages 2190–2197. IEEE, 2011. 37

[72] Peter Kontschieder, Samuel Rota, Antonio Criminisi, Horst Bischof, Pushmeet Kohli, and Pelillo. Context-sensitive decision forests for object detection. In *Advances in Neural Information Processing Systems*. MIT Press, 2012. 37

[73] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956. 6

[74] James T Kwok and Pak-Ming Cheung. Marginalized multi-instance kernels. In *International Joint Conference on Artifical Intelligence*, volume 7, pages 901–906. ACM, 2007. 56

[75] Qiangfeng Peter Lau, Mong Li Lee, Wynne Hsu, and Tien Yin Wong. Simultaneously identifying all true vessels from segmented retinal images. *Transactions on Biomedical Engineering*, 60(7):1851–1858, 2013. 5

[76] Christian Leistner, Amir Saffari, and Horst Bischof. Miforests: Multiple-instance learning with randomized trees. In *European Conference on Computer Vision*, pages 29–42. Springer, 2010. 52, 56, 57, 68

[77] Fuxin Li and Cristian Sminchisescu. Convex multiple-instance learning by estimating likelihood ratio. In *Advances in Neural Information Processing Systems*, volume 10, pages 1360–1368. MIT Press, 2010. 50, 55

[78] Wen Li, Lixin Duan, IW Tsang, and Dong Xu. Batch mode adaptive multiple instance learning for computer vision tasks. In *Conference on Computer Vision and Pattern Recognition*, pages 2368–2375. IEEE, 2012. 56

[79] Roberto Lotufo and W Silva. Minimal set of markers for the watershed transform. In *International Symposium on Mathematical Morphology*, pages 359–368. CSIRO publications, 2002. 13, 14

[80] Aurélien Lucchi, Kevin Smith, Radhakrishna Achanta, Graham Knott, and Pascal Fua. Supervoxel-based segmentation of mitochondria in em image stacks with learned shape features. *Transactions on Medical Imaging*, 31(2):474–486, 2012. 12

[81] Filip Malmberg, Robin Strand, and Ingela Nyström. Generalized hard constraints for graph segmentation. In *Image Analysis*, pages 36–47. Springer, 2011. 37

[82] Olvi L Mangasarian and Edward W Wild. Multiple instance classification via successive linear programming. *Journal of Optimization Theory and Applications*, 137(3):555–568, 2008. 56

[83] Laszlo Marak, Jean Cousty, Laurent Najman, Hugues Talbot, et al. 4d morphological segmentation and the miccai lv-segmentation grand challenge. In *MICCAI Workshop on Cardiac MR Left Ventricle Segmentation Challenge*, number 1, pages 1–8. MIDAS, 2009. 5

[84] Oded Maron and Tomás Lozano-Pérez. A framework for multiple-instance learning. *Advances in Neural Information Processing Systems*, pages 570–576, 1998. 56, 60

[85] 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 *International Conference on Computer Vision*, volume 2, pages 416–423. Springer, July 2001. 4

[86] David R Martin, Charless C Fowlkes, and Jitendra 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. 37

[87] Brian W Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta - Protein Structure*, 405(2):442–451, 1975. 38

[88] Nicolai Meinshausen. Forest garrote. *Electronic Journal of Statistics*, 3:1288–1304, 2009. 58, 62

[89] Bjoern H Menze, B Michael Kelm, Daniel N Splitthoff, Ullrich Koethe, and Fred A Hamprecht. On oblique random forests. In *Machine Learning and Knowledge Discovery in Databases*, pages 453–469. Springer, 2011. 60

[90] Fernand Meyer. Minimum spanning forests for morphological segmentation. In *Mathematical Morphology and its Applications to Image Processing*, pages 77–84. Springer, 1994. 25, 26

[91] Fernand Meyer and Serge Beucher. Morphological segmentation. *Journal of Visual Communication and Image Representation*, 1(1):21–46, 1990. 25, 26

[92] Fernand Meyer and Jean Stawiaski. A stochastic evaluation of the contour strength. In *Pattern Recognition*, pages 513–522. Springer, 2010. 13, 16

[93] NI. LabVIEW. `http://www.ni.com/labview/`. 71

[94] Hannes Nickisch, Carsten Rother, Pushmeet Kohli, and Christoph Rhemann. Learning an interactive segmentation system. In *Indian Conference on Computer Vision, Graphics and Image Processing*, pages 274–281. ACM, 2010. 20

[95] Dennis Nilsson. An efficient algorithm for finding the M most probable configurations in probabilistic expert systems. *Statistics and Computing*, 8(2):159–173, 1998. 25, 26

[96] Sebastian Nowozin, Carsten Rother, Shai Bagon, Toby Sharp, Bangpeng Yao, and Pushmeet Kohli. Decision tree fields. In *International Conference on Computer Vision*, pages 1668–1675. IEEE, 2011. 37, 39

[97] OpenAlea. . `http://openalea.gforge.inria.fr`. 72

[98] George Papandreou and Alan L Yuille. Perturb-and-map random fields: Using discrete optimization to learn and sample from energy models. In *International Conference on Computer Vision*, pages 193–200. IEEE, 2011. 13

[99] Nadia Payet and Sinisa Todorovic. Sledge: Sequential labeling of image edges for boundary detection. *International Journal of Computer Vision*, 104(1):15–37, 2013. 37

[100] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957. 6

[101] Gwénolé Quellec, Mathieu Lamard, Michael D Abràmoff, Etienne Decencière, Bruno Lay, Ali Erginay, Béatrice Cochener, and Guy Cazuguel. A multiple-instance learning framework for diabetic retinopathy screening. *Medical Image Analysis*, 16(6):1228–1240, 2012. 57

[102] John Ross Quinlan. *C4.5: programs for machine learning*, volume 1. Morgan kaufmann, 1993. 58

[103] Vikas C Raykar, Balaji Krishnapuram, Jinbo Bi, Murat Dundar, and R Bharat Rao. Bayesian multiple instance learning: automatic feature selection and inductive transfer. In *International Conference on Machine learning*, pages 808–815. ACM, 2008. 57

[104] Jos BTM Roerdink and Arnold Meijster. The watershed transform: Definitions, algorithms and parallelization strategies. *Fundamenta Informaticae*, 41(1):187–228, 2000. 5

[105] Emma Rollon, Natalia Flerova, and Rina Dechter. Inference schemes for M-best solutions for soft CSPs. In *Workshop on Preferences and Soft Constraints*, volume 2, 2011. 26

[106] B Seroussi and JL Golmard. An algorithm directly finding the K most probable configurations in bayesian networks. *International Journal of Approximate Reasoning*, 11(3):205–233, 1994. 26

[107] Burr Settles. Active learning literature survey. *University of Wisconsin, Madison*, 52:55–66, 2010. 11

[108] Burr Settles, Mark Craven, and Soumya Ray. Multiple-instance active learning. In *Advances in Neural Information Processing Systems*, pages 1289–1296. MIT Press, 2008. 56

[109] Pramod Sharma, Chang Huang, and Ram Nevatia. Unsupervised incremental learning for improved object detection in a video. In *Conference on Computer Vision and Pattern Recognition*, pages 3298–3305. IEEE, 2012. 57

[110] Christoph Sommer, Christoph Straehle, Ullrich Köthe, and Fred A Hamprecht. ilastik: Interactive learning and segmentation toolkit. In *International Symposium on Biomedical Imaging: From Nano to Macro*, pages 230–233. IEEE, 2011. 10, 69

[111] Christoph Straehle, Melih Kandemir, Ullrich Köthe, and Fred A Hamprecht. Multiple instance learning with response-optimized random forests. In *International Conference on Pattern Recognition*. IEEE, 2014. 55

[112] Christoph Straehle, Ullrich Koethe, and Fred A. Hamprecht. Weakly supervised learning of image partitioning using decision trees with structured split criteria. In *International Conference on Computer Vision*. IEEE, 2013. 35

[113] Christoph Straehle, Ullrich Koethe, Graham Knott, Kevin Briggman, Winfried Denk, and Fred A Hamprecht. Seeded watershed cut uncertainty estimators for guided interactive segmentation. In *Conference on Computer Vision and Pattern Recognition*, pages 765–772. IEEE, 2012. 12

[114] Christoph Straehle, Ullrich Köthe, Graham Knott, and Fred A Hamprecht. Carving: scalable interactive segmentation of neural volume electron microscopy images. In *Medical Image Computing and Computer-Assisted Intervention*, pages 653–660. Springer, 2011. 5, 12, 21, 25, 26, 32

[115] Christoph Straehle, Sven Peter, Ullrich Köthe, and Fred A Hamprecht. K-smallest spanning tree segmentations. In *Pattern Recognition*, pages 375–384. Springer, 2013. 25

[116] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Symposium on Theory of Computing*, pages 343–350. ACM, 2000. 40

[117] Andrew Top, Ghassan Hamarneh, and Rafeef Abugharbieh. Active learning for interactive 3d image segmentation. In *Medical Image Computing and Computer-Assisted Intervention*, pages 603–610. Springer, 2011. 11, 13

[118] Srinivas. Turaga, Kevin L Briggman, Moritz Helmstaedter, Winfried Denk, and H Sebastian Seung. Maximin affinity learning of image segmentation. *arXiv preprint arXiv:0911.5372*, 2009. 37

[119] W-J Tzeng and FY Wu. Spanning trees on hypercubic lattices and nonorientable surfaces. *Applied Mathematics Letters*, 13(7):19–25, 2000. 27

[120] Markus Unger, Thomas Pock, Werner Trobin, Daniel Cremers, and Horst Bischof. TVSeg - interactive total variation based image segmentation. In *British Machine Vision Conference*, pages 40.1–40.10. BMVA Press, 2008. 3, 44

[121] Ranjith Unnikrishnan, Caroline Pantofaru, and Martial Hebert. A measure for objective evaluation of image segmentation algorithms. In *Conference on Computer Vision and Pattern Recognition - Workshops*, pages 34–34. IEEE, 2005. 36

[122] Leslie G Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979. 17

[123] Gael Varoquaux. joblib. `https://github.com/joblib/joblib`. 72

[124] Luc Vincent and Pierre Soille. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583–598, 1991. 5, 14, 26

[125] Paul Viola, John Platt, Cha Zhang, et al. Multiple instance boosting for object detection. In *Advances in Neural Information Processing Systems*, volume 2, page 5. MIT Press, 2005. 56, 68

[126] Hua-Yan Wang, Qiang Yang, and Hongbin Zha. Adaptive p-posterior mixture-model kernels for multiple instance learning. In *International Conference on Machine learning*, pages 1136–1143. ACM, 2008. 66

[127] Jun Wang and Jean Daniel Zucker. Solving multiple-instance problem: A lazy learning approach, 2000. 56

[128] Qi Wang, Yuan Yuan, Pingkun Yan, and Xuelong Li. Saliency detection by multiple-instance learning. *Transactions on Cybernetics*, 43(2):660–672, 2013. 57

[129] Yan Xu, Jun-Yan Zhu, Eric Chang, and Zhuowen Tu. Multiple clustered instance learning for histopathology cancer image classification, segmentation and clustering. In *Conference on Computer Vision and Pattern Recognition*, pages 964–971. IEEE, 2012. 57

[130] Chen Yanover and Yair Weiss. Finding the m most probable configurations using loopy belief propagation. *Advances in neural information processing systems*, 16:289–295, 2003. 25, 26

[131] Benjamin Yao, Xiong Yang, and Song-Chun Zhu. Introduction to a large-scale general purpose ground truth database: methodology, annotation tool and benchmarks. In *Energy Minimization Methods in Computer Vision and Pattern Recognition*, pages 169–183. Springer, 2007. 43, 44

[132] Stella X Yu and Jianbo Shi. Segmentation given partial grouping constraints. *Transactions on Pattern Analysis and Machine Intelligence*, 26(2):173–183, 2004. 37

[133] Qi Zhang and Sally A Goldman. EM-DD: An improved multiple-instance learning technique. *Advances in Neural Information Processing Systems*, 2001. 56, 60, 65, 68

[134] Yibiao Zhao, Song-Chun Zhu, and Siwei Luo. Co3 for ultra-fast and accurate interactive segmentation. In *International Conference on Multimedia*, pages 93–102. ACM, 2010. 44

[135] Songfeng Zheng, Alan Yuille, and Zhuowen Tu. Detecting object boundaries using low-, mid-, and high-level information. *Computer Vision and Image Understanding*, 114(10):1055–1067, 2010. 37

[136] Zhi-Hua Zhou, Yu-Yin Sun, and Yu-Feng Li. Multi-instance learning by treating instances as non-i.i.d. samples. In *International Conference on Machine Learning*, pages 1249–1256, New York, NY, USA, 2009. ACM. 50, 55, 66, 68

[137] Song Chun Zhu and Alan Yuille. Region competition: Unifying snakes, region growing, and bayes/mdl for multiband image segmentation. *Transactions on Pattern Analysis and Machine Intelligence,*, 18(9):884–900, 1996. 3