Contours in Visualization

Von der Fakultät für Mathematik und Informatik
der Universität Leipzig
angenommene

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

DOCTOR RERUM NATURALIUM
(Dr. rer. nat.)

im Fachgebiet

Informatik

Vorgelegt

von Dipl.-Inf. Christian Heine
geboren am 30. August 1980 in Leipzig

Die Annahme der Dissertation wurde empfohlen von:

1. Professor Dr. Gerik Scheuermann (Leipzig)
2. Professor Dr. Heidrun Schumann (Rostock)

Die Verleihung des akademischen Grades erfolgt mit Bestehen der Verteidigung am 03.06.2013 mit dem Gesamtprädikat *magna cum laude*.

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Dissertation selbständig und ohne unzulässige fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angeführten Quellen und Hilfsmittel benutzt und sämtliche Textstellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, und alle Angaben, die auf mündlichen Auskünften beruhen, als solche kenntlich gemacht. Ebenfalls sind alle von anderen Personen bereitgestellten Materialen oder erbrachten Dienstleistungen als solche gekennzeichnet.

.....
(Ort, Datum)

.....
(Unterschrift)

Acknowledgments

First, I would like to thank my advisor Gerik Scheuermann, who always gave me the freedom to explore the field of visualization like I saw fit and lots of human resources to work with. In case of a problem he always took the time to discuss it with me.

Furthermore, I would like to thank Gerhard Heyer and Stefan Bordag from the Natural Language Processing Group at the University of Leipzig, Christoph Flamm and Ivo Hofacker from the Institute for Theoretical Chemistry at the University of Vienna, Peter F. Stadler and Marc Hellmuth from the Bioinformatics Group at the University of Leipzig, as well as Hamish Carr from the School of Computing at Leeds University, for providing me with interesting and, more importantly, real applications. Without their involvement, the methods I developed would have been less successful because of a lack of utility. I extend my thanks to Maciej Rosolowski for providing the datasets that were used in Chapter 3, Vijay Natarajan for pointing out the similarity of barrier and contour trees, and Frank van Ham for the suggestion to apply the concepts of barrier-tree-sequence drawing to dynamic clustering results.

For stimulating discussions, I would like to thank the students I have supervised or cosupervised during their bachelor or Diploma thesis or university project; sometimes the best way to understand something is to explain it to someone. I thank Patrick Oesterling and Stefan Jänicke, whose thesis resulted in scientific publications and with whom I continue to collaborate fruitfully. I thank André Reichenbach, Roxana Bujack, Nico Korn, Robert Frohl, Christian Fritzsche, and Waldemar Beser, whose work provided significant building blocks and milestones for larger research endeavors. I would like to thank Igor Strasser, Nurettin Tayfur, and Jing Wang for exploring unorthodox visualization solutions with me, as well as Axel Fischer, Clemens Fritzsch, and Sebastian Volke for much of their programming work and carving out software details.

From the members of the Image and Signal processing group a special thanks goes to Mathias Goldau who by reciting the mantra "care about your craft" sparked my mania in reading to a level he hopefully cannot fathom. I would particularly thank Dominic Schneider, Markus Rohrschneider, and Wieland Reich for collaborating on publications. Like Sherlock Holmes, my favorite vacation is studying other things for a while, and I thank them for pardoning my intrusion on their topics and valuing my opinion. In the same spirit I would like to thank Faisal Cheema, Heike Jänicke, Stefan Koch, Stefan Philips, Steven Schlegel, and Alexander Wiebel, the later also for turning me inadvertently into a walking C++ ISO standard. I thank Sebastian Eichelbaum and Mario Hlawitschka for long and in-depth discussions on technical details, and the later also especially for his moral support in the last weeks of writing this thesis. Along with the current and past members named above, I also thank Silvia Born, Julia Ebling, Albert Pritzkau, Tobias Salzbrunn, and Karin Wenzel for making the working place likable to the extend of a second home.

Finally, I would like to thank my parents Marina and Hans-Peter for providing me with the opportunity to study and my sister Melanie and her son Julian, as well as my good friends Sebastian, Kai, and Jan for numerous distractions and reminding me of the other fun things in life.

Contents

1 Motivation

To visualize means to "form a mental image" of something ([Soa03]). In English, the words "to see" and "to understand" can often be used synonymously. Although understanding does not always require one to view an image, it is often a tremendous help. Popular historical examples of images used for successful communication and visualization of data are Playfair's statistical charts of England's economy (1786), Snow's map of the London cholera outbreak in 1854, which gave a convincing plan for how to counter it, and Minard's *Carte Figurative*, that shows Napoleon's troop losses during his Russia campaign along with environmental factors over time on a map (1869). A detailed account of these examples is given by Tufte ([Tuf01, Ch. 1], [Tuf97, Ch. 2]). All three demonstrate a clear superiority of images over plain representation of numbers in tables and text. Anscombe showed a set of four very small datasets that are equivalent in different statistical measures but show substantial differences when plotted ([Ans73]), thereby demonstrating that sometimes images can be more effective than statistics.

As Donald Norman once put it, "It is things that make us smart" [Nor93], claiming that tools are not merely helpers but essential in the process of problem solving. In the context of visualization, Card *et al.* even speak of external cognition ([CMS99, Preface]). Humans have a quite limited memory for images, probably intentionally so: as the eye can sample the world very rapidly, the world is its own memory ([War08, p. 2]). Munzner says that "visualization allows people to offload cognition to the perceptual system, using carefully designed images as a form of *external memory*" ([Mun09, emphasis in original]). A suitable image can thus serve as a tool to understand large information resources or complex processes, and computers with their vast arsenal of methods for generating and interactively manipulating images have become indispensable tools in this process. According to Cart, Mackinglay, and Shneiderman,

visualization is therefore "The use of computer-supported, interactive, visual representations to amplify cognition" [CMS99, p. 6].

But not just any image is helpful. In the seminal article "Why a Diagram is (Sometimes) Worth Ten Thousand Words", Larkin and Simon list several necessary conditions for when information put in diagrammatic representation is more effective than in a sequential representation like a list or a text. But even for the same data, multiple visualizations are possible; some being more effective than others. The field of visualization thus studies the design space of visual representations and their properties.

According to Stuard Card ([War04, Preface]), the phase of exploratory design in information visualization, i.e. exploration of the design space using point designs, is over and the characterization phase is imminent, where a taxonomization of methods and a structuring of the design space takes place. However it appears that this transition is not yet finished as point designs still dominate the field and existing taxonomies do not have sufficient descriptive power to enable visualization specifications. There is an abundance of possible reasons for this: the design space for static visualizations is not yet fully explored, basic cognitive science research constantly produces new insight into how humans process visual information and enables better designs (e.g. [PW12]), the interaction of human memory with animation and interactive visualizations is not yet fully understood ([Lam08]), the value of visualizations is difficult to judge ([vW06]), it is not fully clear how to measure insight and integrate other resources, like text and observer knowledge, into the analytic process ([Nor06]), and, of course, the fact that data grows much faster than screen space.

Another reason might be that the design space extends over multiple fields with fuzzy borders: scientific, information, geographic, and software visualization, graphic design, graph drawing, diagrammatic reasoning, visual analytics, etc., each concerned with the creation or study of visual representations, but with different restrictions on dataset types, use cases, or applicable methods. An indicator for this is that e.g. the currently most popular model for the description of effective scientific and information visualizations based on visual attributes and preattentive processing is quite different than e.g. the aesthetic criteria of drawing node-link or Euler diagrams (see Chapter 2). The explosion of dataset sizes in scientific visualization can be countered by feature extraction and then a presentation of features and their relations, which cannot be done with methods from scientific visualization, but requires information visualization or graph drawing methods instead (e.g. [CS03, SWC+08,

SHCS12]).

This thesis studies some topics that lie on the boundary of information visualization, scientific visualization, graph drawing and Euler diagrams. Chapter 3 is concerned with the interactive manipulation of clusterings based on Euler diagrams. Chapter 4 then presents a GPU-based approach to render Euler diagrams to extend the first method to larger datasets. Chapter 5 presents an algorithm to draw contour trees, used in scientific visualization to find features in scalar fields and allows for the efficient selection of isosurfaces and contours. Chapter 6 presents an algorithm for a similar problem from bioinformatics: dynamic barrier trees. Barrier trees are in many ways similar to contour trees: they are defined on discrete rather than continuous observation spaces. The chapter also provides a building block for dynamic contour tree drawing.

Overview of Publications

This thesis is based on the following publications by the author:

[HSF+06] Christian Heine, Gerik Scheuermann, Christoph Flamm, Ivo L. Hofacker, and Peter F. Stadler, "Visualization of barrier tree sequences," *IEEE Transactions on Visualization and Computer Graphics*, 2006.

[HSF+07] Christian Heine, Gerik Scheuermann, Christoph Flamm, Ivo L. Hofacker, and Peter F. Stadler, "Visualization of barrier tree sequences revisited," In *Proceedings Visualization in Medicine and Life Sciences*, 2007.

[HS07] Christian Heine and Gerik Scheuermann, "Manual clustering refinement using interaction with blobs," In *Proceedings of EuroVis*, 2007.

[HBHS08] Christian Heine, Stefan Bordag, Gerhard Heyer, and Gerik Scheuermann, "Euler-diagram rendering on the GPU with applications to document analysis," Submitted to the *IEEE Conference on Information Visualization*.

[HSCS11] Christian Heine, Dominic Schneider, Hamish Carr, and Gerik Scheuermann, "Drawing contour trees in the plane," *IEEE Transactions on Visualization and Computer Graphics*, 2011.

The author furthermore collaborated on the following publications: Hofacker *et al.* [HFH+10], Rohrschneider *et al.* [RHR+09], Jänicke *et al.* [JHH+10], Jänicke *et al.* [JHS12], Oesterling *et al.* [OHJS10], Oesterling *et al.* [OHJ+11], Oesterling *et al.* [OHWS12], Reich *et al.* [RSH+12], and Schneider *et al.* [SHCS12].

2 Background

Existing visualizations can be categorized under various aspects. Each visualization can be described as a interactive mapping from data to graphical objects which are then rendered interactively. This chapter describes some of the current models for visualization as far as they affect the remaining chapters. A more extensive discussion of the foundations and principles of information visualization can be found in [Maz09, WGK10, Spe07, SM00, Tel08, CMS99, War04, War08]. It is particularly noteworthy that information visualization, graph drawing, and Euler diagrams have different notions of what constitutes a good diagram. It remains a challenge to unify the models and quality criteria for these fields.

2.1 Data

Stevens [Ste46] proposed a simple and powerful model of scales of measurements, stemming from the need in psychophysics to measure external stimulus on human senses. He classified scales as nominal, ordinal, interval, and ratio. A *nominal* data scale only allows to determine the equality of values, *ordinal* data incorporate an intrinsic ordering of values. Additionally, *interval* allows to determine the equality of intervals or differences, and *ratio* even the determination of equality of ratios. Stevens also listed permissible statistics on these scales.

A dataset is often constructed from multiple measurements both from making multiple observations under different aspects. Typically there is a measurement for each observation-aspect pair. All measurements from the same observation give a data point and the different aspects are referred to as variables. The variables are called independent if they are systematically varied by the measurement process and dependent

otherwise. E.g., a trajectory dataset has two independent variables: the tracked object's identifier and the time, and one dependent variable: its spatial position at that time. Most of information visualization is concerned with the display of data with at least one independent variable.

It is not always possible or necessary to systematically vary variables for observations. Some high-level features, e.g. cities, and their structural relations, e.g. roads, only allow a hypothetical space, e.g. each pair of cities is connected by roads, of which the observations form a subspace. Similarly, there can be inclusion relations between entities, e.g. a particular country includes a particular state and other types of associations, e.g. a particular compound partakes in a particular chemical reaction. These relations can also be expressed graphically, although the perceptual mechanics of these representations differ from systematically-varied relations, as we will see shortly.

2.2 Visual Variables

More than 40 years ago, cartographer Jacques Bertin [Ber11] laid the foundation for a theory of visualization. He proposed that all graphic representations are ultimately a set of marks on a plane which differ in their appearance based on so-called visual variables. The marks could be points, lines, and areas, and the variables planar position, size, value, texture, color, orientation, and shape. He classified the data types into the levels: qualitative (nominal), ordered, and quantitative (interval-ratio), These levels have an inclusion relationship, e.g. quantitative data is also ordered, which is also qualitative. He proposed a similar classification for visual variables, namely selective, associative, ordered, and quantitative, and postulated that a graphic design is successful if the level of the data matches the level of the visual variable. Ware [War04, p. 6] judged this proposal:

> Jacques Bertin attempted to classify all graphic marks in terms of how they could express data. For the most part, this work is based on his own judgment, although it is a highly trained and sensitive judgment. There are few, if any, references to theories of perception or scientific studies.

Cleveland and McGill [CM86] partially evaluated and ranked the visual variables with respect to which can be read by a human observer with the highest precision. For this purpose they surveyed psychophysics research and augmented it with own studies. The ranking was used

by Mackinlay [Mac86] in the automatic presentation tool (APT) to design graphics automatically according to these guidelines using methods from artificial intelligence. Mackinlay also distinguished between a graphic's expressiveness, i.e. the property that it shows all the facts and nothing but the facts that are supported by the data, and a graphics effectiveness, i.e. the amount of exploiting the abilities of the rendering device and the human visual system. The first criterion can be seen as equivalent to Bertin's suggestion.

In a similar automatic design system, Roth [RM90] refined the data classification so that established presentation conventions for specific measurements such as time (dates) are respected. Casner [Cas90] extended the general approach by recognizing that every image also has a communicative purpose and if the goal of the observer is specified, the drawing can be tuned to highlight that aspect of the data. He extended the levels of Bertin's visual variables to a large set of perceptual operators that visual variables support and matched them with the observer's logical operations required to achieve the goal. The most current general framework for graphic design is given by Wilkinson and Willis [WW05].

The search for automatic design rules is hampered by the lack of a full understanding of human perceptual processes. More recent research is finding new and refining old visual channels and their perceptional properties. Of these properties, the ability to process marks preattentively currently receives most attention. Preattentive processing allows to process marks very fast and can be used to distinguish or "pop out", sum or average, and even group items. An excellent survey over the current state of the art is given by Healey and Enns [HE12], but Ware [War04, Ch. 5] is also a good reference. These sources add mainly time-varying visual variables (e.g. frequency and phase of simple circular or elliptical motion), but also shading cues.

Visual variables have varying resolution, i.e., the number of values which can be discriminated by a human observer varies [Mil56]. But it is also known that the visual variables can interfere with each other, meaning that combinations of visual variables may not be separated easily by the mind. This complicates automatic graphic design. Ware [War04, p. 180] summarizes pair combinations of variables that were experimentally verified to be integrable or separable. Examples of integrable pairs are the red-green and yellow-blue color channels, their value combinations are perceived as one hue, but also size and shape/color, i.e. if a mark gets smaller, its shape and color becomes difficult to determine. Munzner gave examples of how integrable variables can be used to combine visual channels of low discriminability into one perceived

channel of high discriminability [Mun09]. Whether there are interactions between three variables is still an open question [War04, p. 182].

2.3 Grouping Channels

While the visual variables provide an excellent framework for designing charts and statistical graphs, relational information that describes links or grouping of data cannot be directly expressed in this framework. Instead Gestalt laws ([War04, Ch. 6]) describe the principles under which marks are perceived as a group. Of the many principles I wish to point out the following:

Proximity Elements that are closely together are perceived as a group.

Similarity Elements that look similar are perceived as a group.

Connectedness Elements that are connected are perceived as a group.

Closure Elements that are inside a common closed contour are perceived as a group.

In the Bertin model, the law of proximity and similarity can be rephrased as: marks whose visual variables are similar are perceived as a group (recall that planar position was one of the visual variables). But the two other principles are fundamentally different and usually represented differently in graphics. The principle of connectedness is used in graph drawing and the principle of closure in Euler diagrams. An early work by Marks [Mar91] tried to incorporate these Gestalt principles into the automated graphic design of diagrams.

2.4 Graph Drawing

We use the same terminology in this thesis as Diestel [Die05]. A *graph* $G = (V, E)$ is a pair of a finite *vertex* set V and an *edge* set $E \subseteq \{\{u, v\} | u, v \in V, u \neq v\}$. Each edge $e = \{u, v\}$ is *incident* on u and v; u and v are then called *adjacent*. A *digraph* $G = (V, E)$ is a graph where edges are ordered pairs rather than sets. In an edge $e = (u, v)$ the vertex v is called the head and u is called the tail of e. A *path* is a sequence of unique edges that connects vertices u and v. In a digraph the sequence of edges must satisfy that each head of an edge is the tail of the successor in the sequence. A *cycle* is a non-empty path from u to u. A graph or digraph is *acyclic* if

it contains no cycle. A graph is *connected* if for each pair of vertices u, v there exists a path from u to v. A *tree* is a connected acyclic graph, and a *directed acyclic graph*, or *DAG*, is an acyclic digraph. A directed edge (u, v) is *transitive*, if there is a directed path excluding (u, v) that connects u and v. A *transitive reduct* of a directed graph G is the largest subgraph of G that contains no transitive edges. A (di-)graph (V', E') is a *subgraph* of a (di-)graph (V, E), if $V' \subseteq V$ and $E' \subseteq E$. A (di-)graph (V', E') is a *minor* of a (di-)graph (V, E), if $V' \subseteq V$ and for each edge (u, v) in E' there is a path from (u, v) in (V, E); (V, E) is then called an *expansion* of (V', E').

Drawing a graph is the process of transforming topological properties of the graph to geometric objects in a graphical representation. This process is mostly determined by the generation of a layout for that graph, that *places* vertices in a vector space and *routes* edges to connect the vertices. Formally, a graph layout is a pair $L = (p, r)$ of a function p that maps each vertex of G onto a point in $\mathbb{R}^d$ (typically $d = 2, 3$) and a function r that maps each edge to a simple Jordan arc in the same space. The field of static graph layout creation has been intensely studied in the past decades. Introductions and surveys for general graph drawing can be found in [BETT99, KW01, JM04, BETT94, HMM00, Tam99]. Effective graph drawings are based on conventions, constraints, and aesthetic criteria, that capture the user's expectations of visual clarity ([BETT99]).

The conventions lay down the graphical elements used, e.g. whether edges are straight lines, polylines, curved, or orthogonal, i.e. segments of horizontal and vertical lines. Other conventions can be the restriction of vertices and edge bends to integer positions, forcing a planar drawing, i.e. a drawing without edge crossings, and requiring that directed edges follow only one direction, e.g. downwards. One may think of these conventions as design constraints, and are used to tailor the appearance of certain graph classes.

The other constraints are user-provided and can be used to specify positions of single or groups of vertices in a center, a common area, or at the boundary of the layout.

The layout of a graph has properties that can be measured with certain cost functions, e.g., area of the layout, number of edge crossings, distribution of vertices and edges, congruence of isomorphic structures, etc. Aesthetic criteria try to capture the notion of visually pleasing drawings. Among these criteria are minimizing the number of edge crossings, layout area, total or maximum edge length, total or maximum number of edge bends, maximizing angular resolution between edges meeting at a vertex or crossing each other, making vertices and edge bends uniformly distributed, striving for a given aspect ratio, or increasing symmetry.

For a full discussion of these conventions, constraints, and aesthetic criteria see Battista *et al.* [BETT99]. For the purposes of this thesis I want to emphasize a few points:

- It is known that not all aesthetic criteria can be fulfilled simultaneously. A layout algorithm typically has to find a trade-off between the different competing requirements.
- Graph drawing is not only concerned with the creation of images for visualization, it also has applications in circuit board design. Some conventions and aesthetic criteria are therefore less relevant for visualization.
- The aesthetic criteria were defined from the common sense of researchers in the field of graph drawing without any psychological foundation. When a study was finally carried out ([Pur97]), it was found that only minimizing edge crossings had a non-negligible impact on human performance for simple navigation tasks in small graphs.
- The conventions and aesthetic criteria are conceptually quite different from the concepts of quality in information visualizations named above.

In Chapter 3 and Chapter 4 we use and extend a force-directed layout algorithm for the distribution of elements on the screen. Generally, such a layout algorithm determines the layout of a graph G using a physical simulation. Vertices of G are treated as particles, sometimes having a mass or electrical charge, and edges of G are treated as springs or spring-like connections between the vertex particles. A simulation then computes the physical forces between the particles and moves them according to these forces towards an equilibrium, typically a configuration of minimum energy.

The force-directed method was first proposed by Eades [Ead84], who used electrically charged particles that repel each other and springs that force vertices connected through an edge to attract each other. This early approach was refined by Fruchterman and Reingold [FR91], who improved the force formulas and gave a simple optimization for the problem of computing the pairwise repelling forces. Kamada and Kawai [KK89] modeled the system using springs exclusively. Most formulations of these force-directed layout algorithms require the graph to be connected and suffer from long running times, often because of bad convergence. To avoid local minima, simulated annealing can be used. Newer

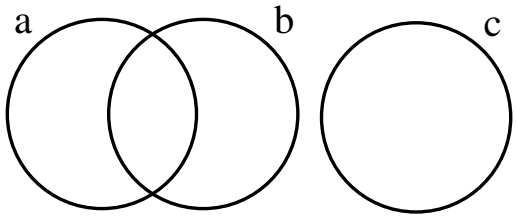


Figure 2.1: A concrete Euler diagram for the abstract Euler diagram $(\{a, b, c\}, \{\emptyset, \{a\}, \{b\}, \{a, b\}, \{c\}\})$. It expresses that there are elements shared in the categories a and b, but c does not share any elements.

formulations solve these problems by calculation of the layout on multiple levels or resolutions of the graph (e.g. [HJ04, HK00, ACL04, CCLP03]). Hachul and Jünger [HJ05] experimentally compared these methods for visual and computational performance. More recent research ports these approaches to the GPU [FT07a, GHGH08, IMO09].

2.5 Euler Diagrams

An Euler diagram is a set of contours, usually simple curves, that partition the plane into connected regions called zones or faces. In contrast to Venn diagrams, where the set of zones is equal to the power set of contours, some zones may be omitted in an Euler diagram. The following definition is adapted from Flower and Howse [FH02]. An *abstract Euler diagram* is a pair: $D = (C(D), Z(D))$ where

1. $C(D)$ is a finite set of contours. These contours are purely symbolic and not actual curves in a plane.
2. $Z(D) \subseteq 2^{C(D)}$ is the set of zones of D, i.e., for each $z \in Z(D)$ holds $z \subseteq C(D)$.
3. $\bigcup_{z \in Z(D)} z = C(D)$
4. $\emptyset \in Z(D)$

This abstract diagram can be transformed to a concrete Euler diagram on the plane in two phases: the layout of the zones and the construction of contours, i.e. the actual Euler representation. Figure 2.1 shows an example. A concrete Euler diagram is well-formed if it meets all of the following criteria.

1. Contours are simple curves, i.e. they do not cross themselves.
2. All zones are connected, they are not represented by multiple faces in the diagram.
3. A curve segment does not represent 2 or more contours.
4. Curves cannot touch without crossing.
5. Only two contours can intersect at any given point.

There are algorithms for automatic construction of Euler diagrams, e.g. work by Flower and Howse [FH02] and Chow [Cho07]. The authors admit, that these methods are only applicable for a small number of elements and contours. Any abstract Euler diagram allows for either zero or an infinite number of well-formed representations. For instance, the abstract Euler diagram $(\{a, b\}, \{\emptyset, \{a, b\}\})$ cannot be presented without breaking rule 3, or inserting a zone $\{a\}$ or $\{b\}$. Unfortunately, many real-world datasets are usually of the type that cannot be represented in a well-formed manner. As motioned by Fish and Stapleton [FS06] as well as Verroust and Viaud [VV04], the class of representable diagrams can be extended by relaxing rules. Recently, Rodgers *et al.* [RZP12] gave recommendations under which circumstances wellformedness criteria can be dropped.

Euler diagrams have seen an increased interest in the information visualization community recently, e.g. Collins *et al.* [CPC09] extracted contours to augment UML diagrams with package information. Simonetto and Auber [SA08, SA09, SAA09] proposed dedicated layout algorithms for Euler diagrams. They proposed to remove some of the wellformedness characteristics of Euler diagrams, too, in order to enable every input to be drawn. Henry-Riche and Dwyer [RD10] in particular allowed elements to be drawn more than once, contours to be represented by more than one face, and showed very aesthetic diagrams. Some research in Euler diagrams, e.g. by Wilkinson [Wil12], is geared towards generating diagrams where the faces have prescribed sizes, generalizing on treemaps. Euler diagrams regularly use graph drawing for placing zones, and the relation to treemaps indicates a deep connection with information visualization.

3 Manual Clustering Refinement using Interaction with Blobs

The huge amount of different automatic clustering methods emphasizes one thing: there is no optimal clustering method for all possible cases. In certain application domains, like genomics and natural language processing, it is not even clear if any of the already known clustering methods suffice. In such cases, an automatic clustering method is often followed by manual refinement. The refined version may then be used as either an illustration, a reference, or even an input for a rule-based or other machine learning algorithm as a new clustering method.

In this chapter we describe a novel interaction technique to manually refine cluster structures using Euler diagrams and the metaphor of soap bubbles, represented by special implicit surfaces (blobs). For instance, elements can simply be moved inside and outside of these blobs, adding or removing them, respectively, from the contours the blobs represent.

3.1 Design

There are two goals that have to be achieved for the manual improvement of a clustering. The first is to effectively communicate the current structure of the clustering to the user and the other is to provide him with means to change that structure.

One method of achieving the first goal is to lay out the elements as point marks in a scatter plot, positioning similar elements very close to each other. A cluster is indicated by a high density of points at one location. This can be used to quickly communicate attributes like size and extend of a cluster but often suffers from a poor usage of screen space. Large parts of the screen usually stay blank. As an improvement, it is

possible to use additional information like color or shape of marks to indicate which element belongs to which cluster. Using this method, the position of the point mark has less importance, or – more importantly – there is a larger degree of freedom for the placement of elements. This freedom may be used to distribute the elements more uniformly on the drawing area, but in general, only a small number of colors and shapes can be properly distinguished by a human ([War04]). So most existing systems combine scatter plots with colors and shape and neglect to use the freedom of positioning elements evenly on the screen.

There is another possibility for increasing the degree of freedom for placement. Again point marks are drawn so that similar elements are in similar places. These marks are then surrounded by a graphical shape per cluster that directly indicates that all elements inside the region of this shape belong to the cluster. As long as these graphical shapes do not overlap each other except for elements that they share, this method communicates the clustering properly. The elements may be distributed more uniformly on the screen as long as neighborhood relationships between the marks are maintained. This design replaces the Gestalt law of proximity, that is typically used for visualization clustering results, by the law of closure. We use special implicit surfaces, called blobs, to define the contours representing clusters. The strength of this diagram is that clusters may overlap arbitrarily, i.e. they do not have to be hierarchical.

Blobs appear to most lay observers like soap bubbles. In reality soap bubbles are too fragile, but we allow elements to be pushed inside or pulled outside of bubbles. For generality, we also allow elements to be part of multiple clusters, although, again in reality, it is impossible for two soap bubbles to intersect each other.

While the system is of value to any clustering problem where manual refinement of clustering or layout is necessary, we have two applications areas as major targets that led us to the development of the system. First, we are working closely with biologists looking at clusterings generated from gene expression data. The initial automatic clustering based on the data contradicts knowledge from other experiments to some extend, so they asked us to provide them with a tool. Second, we cooperate also with the natural language processing group in our department. They work intensively on topic maps and novel text clustering algorithms, where automatic clustering and manual refinement based on human understanding are an important subtopic.

3.2 Clustering Graphs as Euler Diagrams

The structure of a clustering can be described by an acyclic directed graph. Each entity and each cluster is a vertex in that graph, and there is an edge from vertex A to B, if and only if there are corresponding clusters C_A and C_B with $C_A \supset C_B$ and there is no vertex C with corresponding cluster C_C and $C_A \supset C_C \supset C_B$. The layout of a DAG can also be used to visualize the clustering (Figure 3.1), but we will use this constructed DAG only as a supporting data structure later on. The duality of DAG and Euler diagrams is illustrated in Figure 3.1.

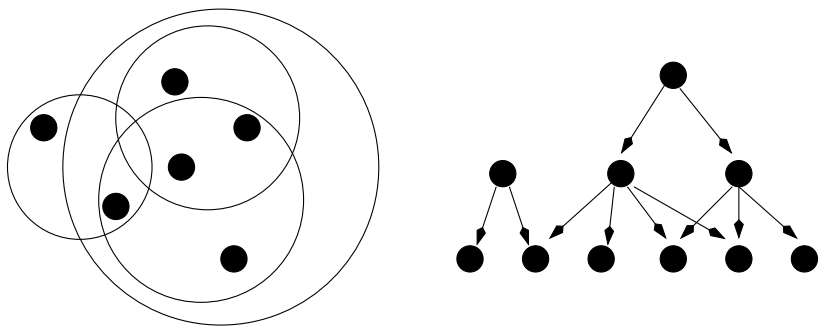


Figure 3.1: A clustering graph and its corresponding Euler representation. In this illustration, additional zones were inserted so that the shapes can be circles. This, however, is not misleading, because the elements are also drawn inside their respective zones, indicating that some faces are not zones.

As a hierarchy can be represented by a rooted tree, an arbitrary clustering may be represented by a directed acyclic graph $G = (V(G), E(G))$. The vertex set $V(G)$ is then simply one vertex for each element and cluster. The edges $E(G)$ correspond to the clustering structure, i.e. there is a directed edge from u to v, if and only if v is an element and u is a cluster that contains this element or if v is a subcluster of u. The sinks of G, i.e. the vertices having no successor, represent the elements and all other vertices represent the clusters. We implicitly assume that there are no empty clusters. Multiple clusterings may be combined into one DAG by constructing the supergraph, i.e. the smallest graph that contains all other graphs as subgraphs.

This DAG could be laid out using standard graph layout techniques, however, for the observer to determine, which element belongs to which cluster, which elements does a cluster contain, and where areas overlap,

an Euler diagram usually is the better choice as most people can comprehend these diagrams because their mathematical education included an introduction in set theory, where these diagrams are used to familiarize with set relations.

We can generate an abstract Euler diagram from a DAG. Let $L(v)$ denote the set of all sinks that are descendants of a vertex v in G. It can be constructed recursively by

$$L(v) = \begin{cases} \{v\} & \text{if } v \text{ is a sink in } G \\ \bigcup_{(v,u)\in E(G)} L(u) & \textit{else} \end{cases}$$

For any possible zone $z \in 2^{C(D)}$ let $L(z) = \bigcap_{v\in z} L(v)$ be the elements it contains, the abstract Euler diagram is then $D = (C(D), Z(D))$ with

$$C(D) = \{v \in V(G) | v \text{ is not a sink in } G\} \subseteq V(G)$$

$$Z(D) = \{\emptyset\} \cup \left\{ z \in 2^{C(D)} \middle| L(z) \neq \emptyset \right\}.$$

The clusters are represented by contours and each sink will be contained in exactly one zone precisely describing which clusters it belongs to. Although this transformation is interesting from a theoretical point of view, as it compares the expressive power of DAGs and Euler diagrams, our implementation stores the DAG, as it is more compact.

The clustering graph is a rooted tree in the absence of overlapping clusters, i.e. clusters that share some elements but neither one is fully contained in the other.

3.3 Related Work on Cluster Visualization

Clustering deals with the identification and grouping of similar entities according to a given metric. Let S be a set of entities, then a *clustering* C is a set of clusters $C \subseteq 2^S$ (2^S denotes the *power set* of S). A *partition* or *classification* is a clustering C which satisfies

$$\bigcup_{c\in C} c = S$$

and

$$\forall a, b \in C : a \neq b \rightarrow a \cap b = \emptyset.$$

A *strictly hierarchical clustering* is a clustering C where

$$\bigcup_{c\in C} c = S$$

and

$$\forall a, b \in C : a \cap b \neq \emptyset \rightarrow (a \subseteq b \vee a \supseteq b).$$

Other types of clustering will be referred to as "general".

Cluster results are typically shown using dendrograms [Har75], which is a tree diagram using orthogonal edges. The problem with this type of representation is that it does not extend to general clusterings and that it requires some training to solve the task of determining which elements belong to a given cluster visually. Often the task is solved by the computer rather than the human observer by highlighting elements when hovering over a cluster. Dendrograms are also often only used to visually determine the maximum element distance that can be used for a partition clustering.

To solve the element-cluster task visually, many possibilities for drawing shapes around clusters have been proposed. Frishman and Tal [FT04] draw rectangles around clusters, Kumar and Garland [KG06] use shaded circles, van Ham and van Wijk [vHvW04] position elements in 3D and draw one sphere per entity. Closely-positioned elements automatically clump together. Sprenger *et al.* [SGEK97] construct ellipses around entities of the same cluster using principle component analysis to determine the orientation and length of the ellipses' axes. None of these techniques can guarantee that shapes will not overlap elements from foreign clusters.

Hierarchies may also be presented using space-filling techniques like treemaps [JS91]. These techniques have been significantly improved using squarified treemaps [BHvW00], cushion treemaps [vWvdW99], and Voronoi treemaps [BD05].

Gross *et al.* [GSF97] and Sprenger *et al.* [SBG00] use blobs in 3D, but only consider partition and hierarchical clusterings. To represent polyarchies, i.e. multiple hierarchies sharing elements, Robertson *et al.* [RCCR02] uses multiple tree views. However, the method only shows the clustering structure in the context of a very small subset of elements. De Chiara and Fish [CF07] present EulerView: a tree-view-like widget for representing and manipulating arbitrary groupings. The method actually presents an explorer-like tree view and automatically inserts nodes for group pairs with non-empty intersection.

3.4 Cluster Display

We visualize clusters by blobs. Let S be a set of elements. A (simple) blob of a cluster $C \subseteq S, C \neq \emptyset$ is an implicit surface where each $\vec{x}$ on this

surface satisfies:

$$\sum_{e \in C} \frac{\omega(e)}{\|\vec{x} - \vec{x_e}\|^k} = \gamma$$

$\vec{x_e}$ is the position of the element e, γ is an arbitrary positive threshold and k an arbitrary positive number. For our implementation we chose $k = 2$ and for simplicity $\gamma = 1$. $\omega(e)$ is the importance of an element, which can be used to give elements a larger area around them. One can safely set $\omega(e) = 1$ for all elements $e \in E$. The points that satisfy the equation not necessarily form a connected structure. So we either have to guarantee that the blob is always connected, or indicate blobs belonging together by an additional attribute, e.g., color. We chose to indicate such enclaves by color, and do not draw the boundary contour but only the interior of every blob. A point $\vec{x}$ belongs to the interior if it satisfies:

$$\sum_{e \in C} \frac{\omega(e)}{\|\vec{x} - \vec{x_e}\|^k} \geq \gamma$$

However, we modified this formula to the following one:

$$\sum_{e \in E} \frac{g_C(e)\omega(e)}{\|\vec{x} - \vec{x_e}\|^k} \geq \gamma \tag{3.1}$$

where

$$g_C(e) = \begin{cases} 1 & e \in C \\ -0.25 & e \notin C \end{cases}$$

This avoids blobs accidentally overlapping elements that are not part of its cluster. Even very closely positioned elements would rather create a hole in the blob than to be accidentally overlapped by it.

Instead of using an isosurface algorithm to extract an approximation of the blob's contour, we simply test each pixel of the screen if it is inside a certain cluster. As we allow elements to be part of any cluster, i.e., the clustering is not restricted to a partition or a hierarchy, a pixel may belong to multiple clusters. When presented on the screen the pixel will get the a mixture of all colors that have been assigned to each cluster overlapping the pixel.

A straightforward implementation can be very slow, because each pixel requires the computation of its distance to each element's position and the summation of the contributions to each cluster. This results in a complexity of $O(A \cdot N \cdot M)$, with N being the number of elements, A being the area, i.e. the number of pixels, and M being the number of clusters. A meaningful approximation of the area A in our case is $A \in O(N)$

as our layout algorithm will give each element the same amount of screen space, and typically clusterings will satisfy $M \in O(N \log N)$. So the overall complexity of this method can be approximated by $O(N^3 \log N)$. If the clustering is a strict partitioning, then $M \leq N$ and the method is bounded by $O(N^3)$.

If the clustering is strictly hierarchical, then the contribution of each element to each pixel has only to be computed for the leaves of the hierarchy tree and can be propagated in $O(M + N)$ time upwards along the tree edges. So the overall complexity can be reduced to $O(A(N + N + M)) = O(N^2 \log N)$. This idea can be extended to general clusterings. The clustering can be described by a directed acyclic graph with $M + N$ vertices, and a propagation tree that spans this DAG may easily be constructed and maintained when the clustering changes.

The rendering can be sped up using the following quad-tree-based heuristic. Most of the time, neighboring pixels belong to the same clusters. First we determine the set of clusters for each pixel of a coarse rectilinear grid. Then we test the four adjacent points of each mesh cell for the equality of the set of clusters. If they are all equal, we assume that each pixel of the cell belongs to the same clusters. If one of them differs, we split the cell in four equally-sized cells, compute the five new points, and proceed recursively, terminating when the cell's area equals exactly one pixel. This is illustrated in Figure 3.2. It makes the method much faster, while still staying in the aforementioned order of time complexity. Unfortunately, it can introduce artifacts, e.g., if the cells of the mesh are too large, and a blob is fully contained inside it. To avoid this, our implementation uses a grid size that is the biggest power of two that is still smaller than the diameter of a blob with a single element.

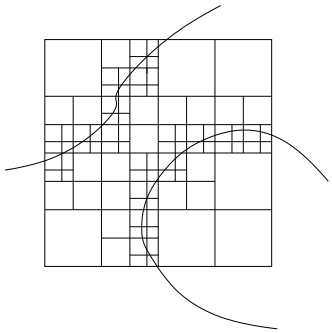


Figure 3.2: quad-tree test to approximate boundary

But we can still do better. When testing a certain pixel, we might

neglect elements that are far away from it, because their contribution to Equation 3.1 is minimal. So it suffices to just consider elements that are very near. One option is to compute the Voronoi diagram of the elements' positions as input sites. This partitioning can then be used to find the nearest site to each pixel considered in the quad-tree test and the neighbors of that site. As a site may have up to $N-1$ neighbors, this does not reduce the upper bound of the time complexity but can be expected to perform much better.

An alternative is to compute the Delaunay triangulation of the elements' positions, find for each considered pixel the surrounding triangle, and consider the contribution of the three elements incident on the triangle and optionally also the incident elements on the three adjacent triangles. Using this method a maximum of six elements contribute to Equation 3.1. The correct Voronoi cell or Delaunay triangle can be searched for in $O(\log N)$ time using trapezoidal maps ([dBvKOS08]) so the overall complexity reduces to $O(N \log N + A \cdot (\log N + h))$. The first part is given by the construction of the Voronoi diagram or Delaunay triangulation, the second part consists of determining the Voronoi cell or Delaunay triangle and the propagation of the contribution of the neighboring sites through the directed acyclic graph for each considered pixel. h is the height of the propagation tree in the DAG describing the clustering structure. This height is seldom outside $O(\log N)$ so the overall complexity can be approximated by $O((N + A) \log N)$. It has to be noted, however, that the later optimizations may result in discontinuities along cell or triangle boundaries. As long as the elements are evenly distributed, the artifacts are negligible.

3.5 Distributing Elements

Our system implements a simple spring embedder that has been modified to avoid these two problems. We chose a single-level, force-directed layout, mainly because the user of the system can watch it work and it has a high "dynamic stability", i.e., if the graph is altered after its layout has been determined the layout of the new graph will look almost the same.

While these algorithms are useful for an initial layout of the entities on the screen, during the interaction phase, we actually prefer the simpler force-directed algorithms for their higher "dynamic stability" and because it is easier to trap them in a local minimum. This leaves more freedom for the user to arrange entities after their own fashion and taste.

The layout process uses a modified force-directed layout algorithm. Most force-directed layout algorithms require the graph to be connected. However, some of the datasets we are provided with contain only elements without any edges between them. Furthermore, we want to show the clustering structure and not the full relationships between elements. We use a simplified van-der-Waals force for the general forces between particles:

$$\vec{F}_r(e) = \sum_{n \in X \setminus \{e\}} \Omega\left(\frac{\|\vec{x}_e - \vec{x}_n\|}{2 \cdot \gamma}\right) \frac{\vec{x}_e - \vec{x}_n}{2 \cdot \gamma}$$

where

$$\Omega(d) = \begin{cases} -2 & d < \frac{1}{2} \\ \frac{1}{d} - \frac{1}{d^2} & \frac{1}{2} \leq d \end{cases}$$

It repels particles if their distance is less than $2 \cdot \gamma$ and attracts them otherwise. So this force is sufficient to nicely distribute the potentially disconnected elements uniformly on the screen.

If we would implement this method, each element would create a force on another one, so our algorithm would be $O(N^2)$ per iteration. However, because the force vanishes quickly with increasing distance, it is sufficient to look only at the closest elements. Again, the set of neighbors of each element can be computed using either a Voronoi diagram or Delaunay triangulation. The formula becomes:

$$\vec{F}_r(e) = \sum_{n \in N(e)} \Omega\left(\frac{\|\vec{x}_e - \vec{x}_n\|}{2 \cdot \gamma}\right) \frac{\vec{x}_e - \vec{x}_n}{2 \cdot \gamma}.$$

Because of planarity the number of neighbor pairs to consider lies in $O(N)$. Therefore, the computation of the Voronoi diagram or Delaunay triangulation become the driving factor for the asymptotic run-time. Because of that, the general forces may be computed in $O(N \log N)$ time.

However, if we were initially provided with edges, we use them in our layout. We use simple logarithmic springs to determine the current force, setting the optimal spring length to $2 \cdot \gamma$. The complexity of one layout iteration is then $O(N \log N + E)$, E being the number of edges. Much more interesting, however, is how to enforce elements of the same cluster to be grouped together. The straightforward way would be to add edges or springs that connect elements of the same cluster. But because each element might be connected with each other element of the same cluster and each element may belong to multiple clusters, this introduces far too many edges. Instead, we implemented a probabilistic algorithm. First we choose a cluster randomly, but respect the size of the cluster, i.e.,

large clusters have a linearly greater probability for being chosen. Inside this cluster we choose two distinct elements and compute the force of a logarithmic spring between them. We repeat the process $O(N \log N)$ times. The ideal spring length is set to γ. This achieves the desired goal of elements being closer if they are in the same cluster, but leaves enough freedom for the user to tune the arrangement of elements.

3.6 Interaction

If the number of elements is large, it may be preferable to show only a subset of them. For that purpose we use the standard zoom and pan technique. We also allow the user to collapse and expand clusters. We represent collapsed clusters by a blob of one element. Subclusters are no longer visible, i.e., there will be no blob for them. There is a problem when elements are part of more than one collapsed cluster. In this case we use one representative for the intersection of both clusters and one for the rest of the cluster. Usually the elements are represented on screen by their name. We do not automatically determine the optimal name of a collapsed cluster based on its contents as this is highly application-dependent. If an application does not provide a name for a collapsed cluster, we show only the number of elements in this cluster.

We allow the user to arrange the elements on the screen in any way he sees appropriate. He can do this by moving either single elements, in which case he has to click on an element and drag it around with the mouse. To move clusters, he clicks somewhere inside a blob, and the program will determine the correct cluster or clusters just like it would determine the set of blob that contain the pixel. If the mouse is then dragged, all elements of all selected clusters follow accordingly. Because the layout process is active during this interaction for deselected elements, these will automatically make way. If the element or the cluster is released, the layout process will again arrange the elements so that they are uniformly distributed, but, in general, it will preserve their horizontal and vertical ordering.

A user can change a given clustering using one of the following methods:

- The user moves one element very close to another element that is in one or more clusters. If the distance of both elements drops below a certain threshold δ the moved element is assigned to all the clusters the other element belongs to. It appears as if the element

was pushed into the soap bubble. This process is illustrated in Figure 3.3. We chose $\delta = \frac{\gamma}{2}$.

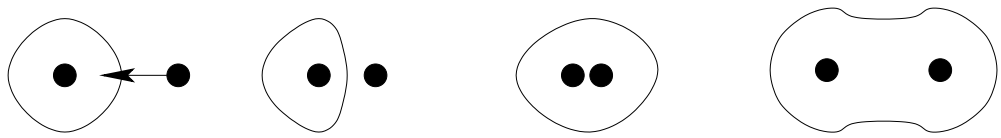


Figure 3.3: Addition of an element to a cluster.

- A single element is moved away from elements of the same cluster, thereby stretching the cluster. Once the distance to all the other elements grows beyond a certain threshold Δ, it is removed from the cluster. It appears as if the element was pulled out of the soap bubble. This process is illustrated in Figure 3.4. In our implementation we use $\Delta = 2 \cdot \gamma$.

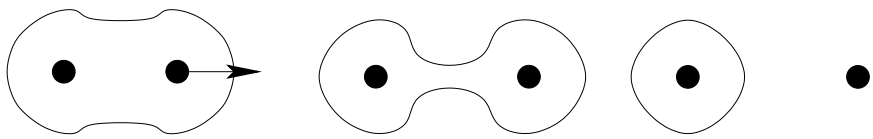


Figure 3.4: Removal of an element from a cluster.

- The user draws a freehand line around some elements. These elements will be assigned to a new cluster. This appears as if the user has made a new soap bubble around some elements (Figure 3.5).
- The user double-clicks on a cluster, thereby deleting it. The bubble appears to be pierced and bursts.

Because the layout process is active during this interaction, the user has to move the elements quickly enough, so that the other elements do not flee to fast (in case of pushing an element into a cluster) or follow to fast (in case of pulling an element from a cluster). In our implementation we update the layout exactly 50 times a second and enforce an upper bound on the velocity of an element empirically. An alternative would be to use a modifier key (e.g. "Shift") to slow down the layout update. We also use the "CTRL" modifier key, if we drag elements around and do not want them to leave their current clusters regardless of how far

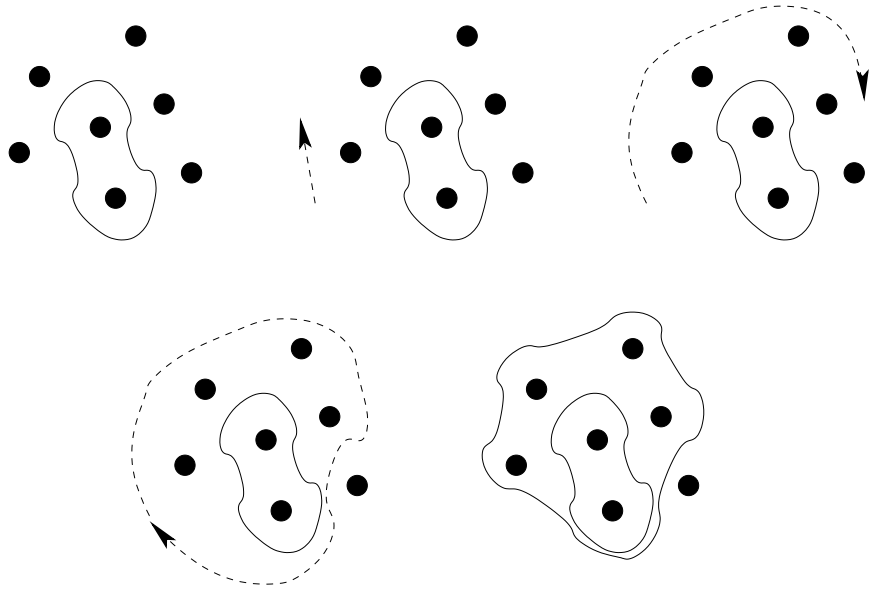


Figure 3.5: Lasso selection of multiple elements into a new cluster.

they are dragged from their original position. This is really helpful in situations where we want one element of an cluster be part of another one as well, but leave the others as they are.

3.7 Results

We implemented our method in a system called "BubbleClusters" in Java. The optimization techniques used make layout recomputations and redisplay of blobs fast enough to be interactive on a Pentium IV 2.54GHz. The first users of our system found it very intuitive and "fun to work with". Figure 3.6 shows the system working on an example that was inspired by a common task of the domain experts that we developed the application for. The application depicted is from the field of genomics, where a given set of genes have been clustered for same functionality, but genes often are part of multiple clusters, as they behave differently in the presence of other genes. Biologists would like to create a partition of the genes to gene groups using the knowledge they already have from specific experiments. Figure 3.7 shows a real-world dataset to illustrate how complex the clusterings can become that a domain expert will refine.

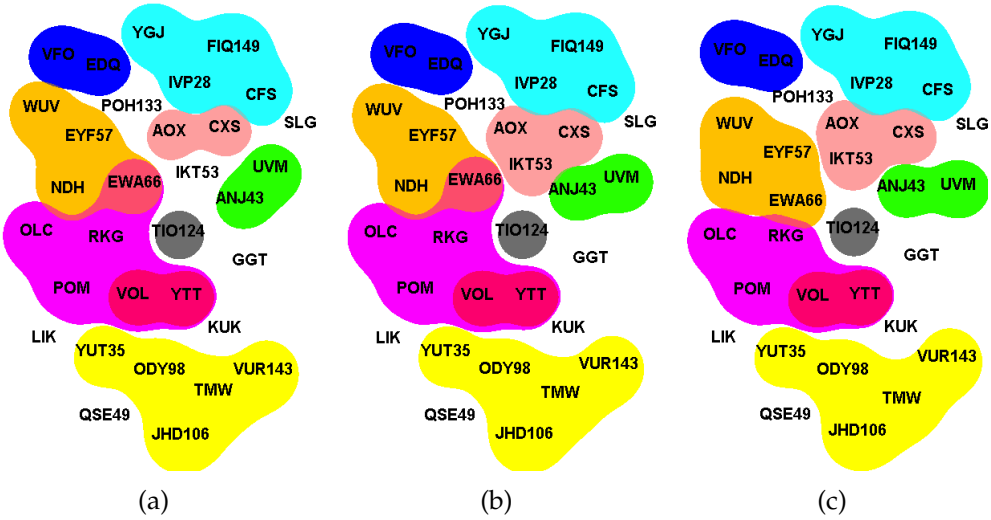


Figure 3.6: (a) Our method's grouping technique was used to create this initial clustering to some randomly created gene names. (b) IKT53 has been pushed inside a cluster. (c) EWA66 has been pulled outside a cluster.

3.8 Summary

We presented a pixel-based algorithm to show Euler diagrams using blobs. The blobs are reminiscent of soap bubbles. The method is most effective when elements are distributed uniformly on the screen, therefore we gave a force-directed layout algorithm that uses van-der-Waals forces, which converges quickly to uniformly-distributed configurations. We presented and implemented multiple ideas on how to improve the layout and rendering speed to enable interactive manipulation of clusterings, using well-known data structures from computational geometry and exploiting the DAG structure of the clustering to reuse partial results. First user tests demonstrated a "fun factor" that gives some evidence for the intuitive understanding of the system.

The presented solution has some drawbacks. First, where many clusters overlap, colors mix to various gray tones. We can counter this problem by using and mixing textures instead. Also we do not explicitly show contours, but these could help in resolving zone borders where mixing colors produces indistinguishable colors. Also the speed is not sufficient for datasets larger than approximately 100 elements. These problems will be addressed in the next chapter.

Figure 3.7: Euler diagram of a complex dataset. The dataset shown is a randomly selected part of a real-world dataset of a correlation analysis of genes. It consists of 78 elements and 24 clusters. On average each element is part of 4 clusters.

4 Euler-Diagram Rendering on the GPU with Applications to Document Analysis

Efficient human exploration of document collections is still a major challenge. Common approaches combine automatic natural language processing techniques and interactive visualization approaches. Typically, automatic text analysis creates an information space and projection methods allow interactive visual exploration of the space. Since the information space creation is usually not perfect and projection increases the problem, users often wish to change the projection and sometimes the information space during the exploration phase to obtain an improved model of the document collection. We present a fast GPU-based rendering of the clustering structure of the document collection using an Euler-diagram representation. As the clusters may overlap arbitrarily we use colors and patterns to distinguish different sets including the various intersections. We apply this approach to a combination of document and paragraph clustering based on novel methods from computational linguistics. Users can interactively change cluster, document, and paragraph layout. Furthermore, the user can, in principle, also change the clustering which may lead to effective steering of the information space creation in the future. Finally, the visualization allows the selection of particularly coherent or incoherent parts of texts with the low effort.

4.1 Problem Statement

Efficient human exploration of document collections is a major challenge of the information age. Common approaches combine natural language

processing with visualization to reach this goal. Typically, automatic text analysis regards documents as elements in a high-dimensional space. In this space, a metric [AKS+02], self-organizing map [LSM91, RI96], or clustering [AKS+02, JTP+95, Ren94] defines a neighborhood relation creating an information space. This space is projected to two or three dimensions using principle component analysis [FMG05], multidimensional scaling [AKS+02, CC92, FMG05], or the low-dimensional structure of the Kohonen maps [LSM91, RI96]. The results are visualized as annotated point clouds [AKS+02, Ren94], or landscapes after applying a low-pass filter [FMG05, JTP+95].

Classical graphical interaction metaphors like panning and zooming allow exploration of the collection. Additional information like keywords associated with the mouse position is provided [FMG05]. Some systems support active querying of documents based on interaction with the landscape [JTP+95]. While all this research is promising, there are potentially shortcomings. The basic process of creating the information space is usually not perfect – typical recall in controlled experiments reaches around 90%.

Usually the information space is a partition or a hierarchical structuring of the document collection. But many natural phenomena such as natural language contain ambiguities and a higher recall may be obtained by allowing an arbitrary rather than a strict hierarchical clustering of the document collection. As another example, documents may be categorized with respect to different aspects and the union of these categorizations usually does not fit naturally in a hierarchy.

Furthermore, a user may want to personalize the projection, e.g. changing the cluster layout, based on his growing understanding of the information space during exploration. This is especially typical if the information space is created based on more than one similarity measure, e.g. document and paragraph similarity. Finally, the user may want to adjust the results of the information creation process in certain areas. Ideally, this information should be used to adapt the parameters of the process generating the neighborhood relation. The exploration and adjustment, however, require a system that effectively communicates the clustering structure before and after changes, preserving the mental map and in an adequate response time.

Recently, the field of computational linguistics, or more specifically text mining [HQW05], has seen the appearance of a variety of algorithms that allow extracting information from raw texts, such as morphology [KCT08], word similarity [Bor07] or terminology [Wit05]. These algorithms can be used in common clustering or classification algorithms to

reveal the relations between texts or parts of texts, such as paragraphs or sentences. Such paragraph and text similarity enables the computation of an accurate clustering of documents.

In this chapter, we concentrate on information spaces based on clusterings that may be non-hierarchical. We combine novel natural language processing algorithms with a new GPU-based rendering of the resulting information space. Given a set of input documents, we construct a grouping of documents and an independent grouping of paragraphs using a specialized clustering algorithm. Chinese whispers [Bie06a] and an optional module taking similarity between words into account [Bor07] allows to group documents that do not necessarily share a single word with each other, as long as the content is related. This framework can be easily extended to take morphological segmentation, part-of-speech tags, syntactic patterns or other linguistic information into account. A terminology extraction algorithm is then used to find cluster labels consisting of the three most significant terms.

The visualization process consists of an initial layout of the elements representing documents and paragraphs using a force-directed layout algorithm. After this step, the clustering structure is shown in an Euler diagram with contours and overlapping patterns. Our method does not impose a restriction on the amount of overlap but is limited by the ability of the human observer to distinguish multiple overlapping patterns and colors. We render the Euler diagram using a local approximation. It ensures that no element is drawn inside a contour that it does not belong to and results in contours that are relatively slim. We use the GPU extensively for this local approximation to achieve interactive frame rates for up to multiple hundred simultaneously visible elements and clusters even on dated graphics hardware. We repeatedly show the clustering, change the currently visible part of the hierarchy, move elements, or even change the clustering structure. After such operations, the elements are distributed again uniformly on the screen, while preserving the mental map.

4.2 Linguistic Processing

Given a set of documents, the main goal of this work is to uncover similarities between them. Assuming that the texts in question are machine readable, i.e. not stored as images, a simulation of semantic similarity between two texts, or parts of it, e.g. paragraphs, is usually computed by measuring the angle between two vectors in a high-dimensional vector

space [Seb02]. The vector space is defined such that each word has an own dimension, entirely orthogonal to all other dimensions (i.e. words). A document is then represented by a point in this vector space that corresponds to the words occurring in it (and their frequency). However, for example, highly frequent words tend to have less differentiation power than less frequent words. Therefore a measure such as *tf.idf* [Seb02] is used to transform pure frequencies of words into weights that are highest for words that occur frequently, but in very few documents, or put differently, for the most document-specific words.

Once a basic vector space model is defined, any clustering algorithm can be used to find groups of texts belonging together. The undirected weighted graph that contains one vertex for each document or, depending on the case, parts of the document, like paragraphs, and the similarities between them as edges will be referred to as the *document* or *paragraph similarity graph*. A clustering algorithm augments this graph by adding vertices representing the new clusters as well as new edges for the similarities that can be computed among the new clusters and the elements already contained in the graph. We will call this graph the *similarity graph*. The main output of the clustering describes relationships between the clusters, i.e. which clusters are subclusters of each other and which elements each cluster contains. Because this information defines an order on the elements and clusters, the result of the clustering can be described by a directed acyclic graph we call the *clustering graph*. This graph usually omits transitive information.

Chinese Whispers (CW) [Bie06a], a probabilistic graph clustering algorithm, is particularly well suited for clustering natural language material (such as texts, words, etc.). It is both very scalable (time-linear complexity) and it produces excellent results, as evidenced in several applications such as unsupervised language separation [BT05], part of speech clustering of words [Bie06b] or even detecting spam [BW07]. CW produces flat clusterings by default (or rather it partitions the graph).

The last step of the natural language processing part is finding short, but descriptive labels for each cluster. To this end, we decided to use the terminology extraction algorithm developed by Witschel [Wit05] which combines cues from syntax and morphology with a differential analysis to produce a ranking of most significant words for a given document. To find a label for a cluster, all texts of that cluster are combined into one long text and then the terminology extraction is run on that text. Since the terminology extraction also includes a tagger, all words extracted were filtered such that no words tagged as stop words are shown.

4.3 Layout

We use the force-directed layout algorithm by Fruchterman and Reingold [FR91] for the initial layout of the similarity graph, independent of the clustering graph. In case the similarity graph is not available, we construct it from the clustering graph simply by estimating the similarity of elements based on the number of clusters they both reside in. The Fruchterman-Reingold algorithm does not scale well, but we do not consider this to be a disadvantage of our method, as it can be replaced by faster layout algorithms that also have a better convergence behavior (e.g. see the comparison by Hachul and Jünger [HJ05]).

The set of visible elements and clusters will be represented by *nodes* on the screen. This set may change through the expansion or collapse of nodes. If this happens, the user shifts nodes, or changes the clustering the layout of the visible part of the graph is adapted by further iterations of the Fruchterman-Reingold algorithm, but instead of computing all the $\frac{n(n-1)}{2}$ repelling forces, we calculate the Delaunay triangulation on the position of the nodes and exert repelling forces only between neighbors in the triangulation. We show all iterations of this layout adaption to help the user understand the transition in order to preserve the mental map.

The layout of the nodes can result in Euler representations with disconnected zones. We use colors and patterns to indicate disconnected zones of the same cluster. We use this effect as a visual cue for inspection as it highlights regions where the clustering failed for ambiguities or systematic reasons.

4.4 Euler-Diagram Rendering

Concerning the wellformedness criteria of Euler diagrams, our rendering method guarantees that contours are simple curves, but zones may be disconnected. This relaxation is required to allow any abstract Euler diagram to be rendered. We minimize the effect of this disadvantage by using colors and patterns for each region. Unfortunately, also contours can touch each other without crossing and there are curve segments that can represent more than one contour because of the local approximation we use when rendering the Euler diagram for speed reasons. We compensate the effect of breaking these rules by varying the thickness of lines to indicate the number of contours they represent.

We present an overview over the algorithm first and then discuss the single steps.

1. Compute the Delaunay triangulation of the locations of the nodes, i.e. the currently visible elements and clusters.

2. Add a border to the triangulation by replicating vertices of the convex hull.

3. Replace long lines on the new convex hull of the triangulation by further insertion of points.

4. For each triangle, precalculate and mix the patterns that occur in it. The number of mixtures per triangle is limited by a constant.

5. Draw each triangle using a certain interpolator and the premixed patterns.

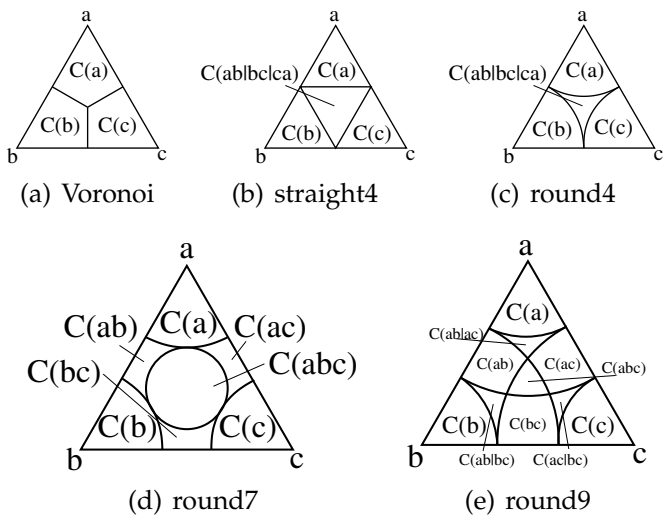
4.4.1 Zone Mapping

A concrete Euler diagram can be defined as a function mapping each contour $c \in C(D)$ to a set of points $P_c \subseteq R^2$ in the plane interior to its curve. This can also be described by a relation $T \subseteq C(D) \times R^2$ where it is clear that a dual function can be created mapping each point $p \in R^2$ to a zone and therefore to a (possibly empty) set of clusters $C_p = \{c|(c,p) \in T\}$.

The input to this stage of the algorithm is a finite set of points P resulting from the node layout and a set of contours $C(D)$. Furthermore, there is a function that maps each point $p \in P$ to its zone $z(p) \in Z(D)$. To compute $z(p)$, we use a simple traversal upwards starting from p in the clustering graph. This operation can be costly but is computed beforehand and only needs to be updated when the user changes the clustering graph. If P were the entire plane, or at least the part that is shown on screen, the mapping $z(p)$ could be inverted to get an Euler representation. Unfortunately, P is only a finite subset of the plane and an interpolation has to determine the set of clusters for at least the points presented on the screen. To interpolate efficiently, we interpolate on triangles of a triangulation.

Because we do not interpolate scalar values but rather finite sets of clusters, we partition each triangle of the triangulation into up to nine connected regions as illustrated in Figure 4.1. It shows which set of clusters is assigned to each point of the interior of the triangle.

Using the triangulation, it is clear that only points inside the convex hull of the original point set can be interpolated. When the outside is neglected, some contours are not closed but rather end at the convex hull.



(a) Voronoi (b) straight4 (c) round4

(d) round7 (e) round9

Figure 4.1: Five different interpolators. The interpolators partition the triangle into connected regions being equivalent in the set of clusters and contour pieces between pairs of neighboring regions. $C(a)$ means the set of all clusters point a belongs to. $C(ab)$ means $C(a) \cap C(b)$ and $C(ab|bc)$ means $C(a) \cap C(b) \cup C(b) \cap C(c)$. (a) is the simplest interpolator and gives a Voronoi-like representation, (e) is the most complex but gives the best results in practice. (d) and (e) also have the advantage of being configurable with respect to the radius of the smaller circles around the end vertices. The radii of the center or the larger circles is then determined so that the circles touch the smaller circle. Note that for all interpolators contours are C_0 continuous across triangle boundaries. With the exception of (b), the region borders are perpendicular to the triangle borders.

To extrapolate outside the original point set we add further points that lie outside. These points belong to the zone $\emptyset$. We simply duplicate each of the original points that lie on the convex hull and position the duplicate at unit distance from the original in the direction opposite the barycenter of all points. Inserting these points in the Delaunay triangulation can create sharp border triangles and the contours can get very distorted. This happens especially where the convex hull has a long edge. We counter this effect by adding more virtual points on these long edges. The points are positioned at least one and at most two unit distances away from their next neighbor. The number of additional points added in this step can only be bounded trivially by of the number of nodes as it depends on the circumference of the drawing. Fortunately, the layout keeps the

drawing compact so the number of additional points required is usually low compared to the number of nodes.

4.4.2 Colors and Pattern Mixing

In cases where the zones become disconnected, a different type of visual information is needed to communicate the connectivity. This can be done by using colors and textures and also has the benefit of making legal Euler diagrams visually more appealing as demonstrated by Ware [War04, p. 198]. Therefore, we apply this coloring to all clusters and defer the problem of finding disconnected zones.

Where point sets overlap, these textures have to be mixed. The textures can be mixed at each point but it is more efficient to mix them for each triangle and even for the whole triangulation. For m textures there are 2^m possible mixtures, but only a very small subset are actually used. A triangulation of n points cannot contain more than $2n - 5$ triangles. As we have seen in the previous section, each triangle is partitioned into a maximum of nine regions, many of them shared across triangle boundaries, so we never need more than a constant amount of mixtures per node.

In our implementation, the user has the ability to freely assign colors and patterns to each cluster. Clusters may not need to have a color or pattern, in this case they are simply distinguished by their contour line. When the ROUND7 and ROUND9 interpolators are used, there is always a border region between clusters that do not share any elements, i.e. contours cannot touch from the outside. In this case, it is impossible to mistake a partition for an inclusion, as Figure 4.2 shows. We therefore do not have to mix the patterns of all clusters that contain the point, but rather can use the most specific clusters, i.e. all the clusters in the set that have a pattern assigned and do not have any descendant in the set.

4.4.3 Implementation

Before actually drawing the triangulation, we iterate through it to determine all the mixtures we actually need. These mixtures are then generated and stored in one large texture that is used when the triangulation is drawn. Because mixtures are shared across triangles we use a cache to avoid multiple mixing and keep the size of the texture small. The images in the result section have been drawn with just one 512x512 texture containing 8x8 pixel sizes tiles. Our implementation does not limit the number of base textures, and any texture may be used as base texture.

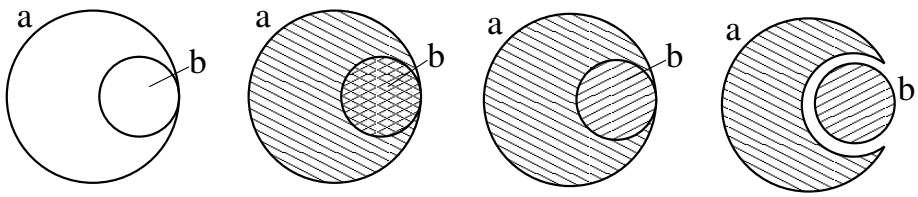


Figure 4.2: How to avoid mixing textures unnecessarily. The left Euler diagram can mean either that b is a subset of a (contour b touches a from the inside) or that they do not share any elements (b touches a from the outside). This ambiguity can be removed by mixing of textures. The second image therefore represents b is a subset of a and the third that they don't intersect just like the fourth diagram. When contours cannot touch from the outside, image 3 can always be interpreted as b is a subset of a without the need to mix the textures.

Currently our implementation uses for the base textures twelve patternless colors and eight simple line patterns in six different colors each.

For brevity, we describe only the ROUND7 interpolator. The other interpolators can be implemented similarly. Given a point $\vec{p}$ inside the triangle, as well as the three end points $\vec{a}, \vec{b}, \vec{c}$ of the triangle, we compute the barycentric coordinates $\beta_A, \beta_B, \beta_C$ of $\vec{p}$. These coordinates are used to map to a point in an equilateral triangle. In this equilateral triangle, there is a circle around each end point of the triangle with radius δ for some $0 < \delta \leq \frac{1}{2}$. There is also a circle at the barycenter of the triangle with radius $\frac{1}{\sqrt{3}} - \delta$. Figure 4.3 illustrates these zones. The circles do not overlap but rather touch each other. The parameter δ is used to define the relative size of the circle inside and the circles on the end points of the triangle. We determined $\delta = 0.4$ experimentally to produce balanced results.

Using barycentric coordinates, we can treat every triangle as if it were equilateral and effectively map the regions of the equilateral to the original triangle. The region borders in the interpolators are perpendicular to the triangle border. When the regions are mapped to the original triangle, these circles become ellipses and do not necessarily cross perpendicularly. As a result, the combination of contour pieces from neighboring triangles may not be C_1 continuous. To minimize this effect we chose the Delaunay triangulation as the triangulation of the original point set because it maximizes the minimum angle in all triangles. It allows any layout of nodes to be drawn with the method. An alternative would be

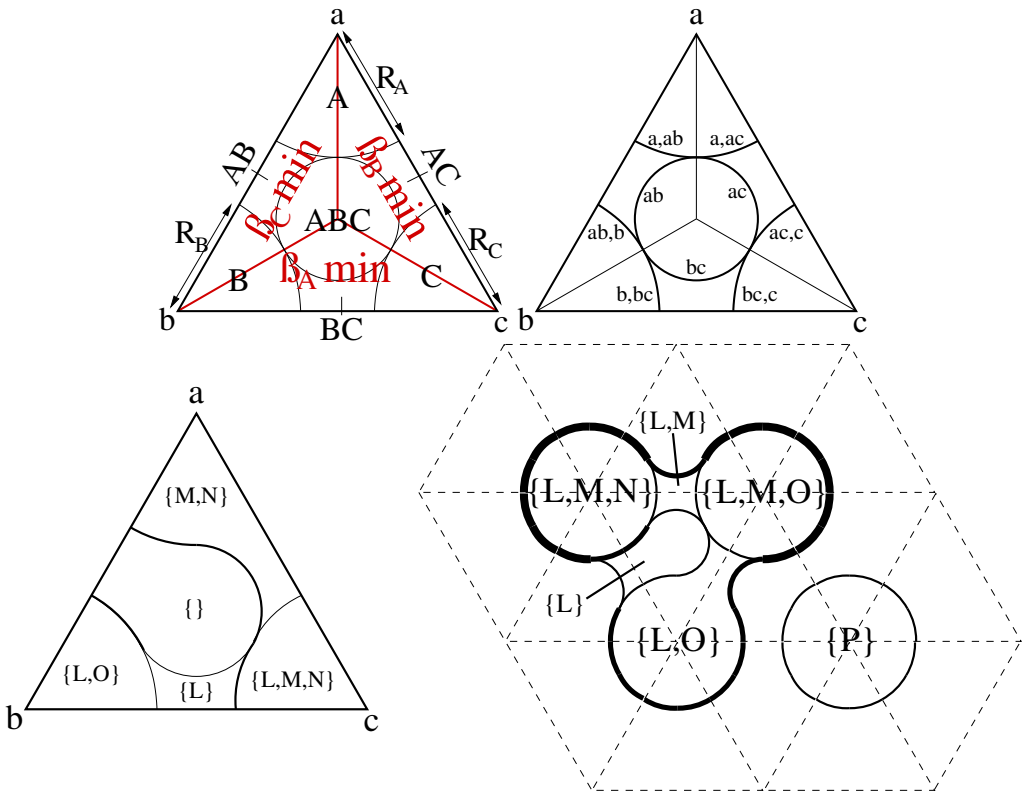


Figure 4.3: Determining the regions and contour pieces. The point-region test is performed using a simple point-inside-circle test on the four circles and a minimum barycentric coordinate test for the remaining three regions. This test also splits the contours in 9 contour pieces. In the lower left image, an example is shown where each contour piece is assigned a thickness based on the number of clusters in the symmetric difference of the adjacent region's cluster sets. The elements a, b, and c belong to the clusters $\{M, N\}$, $\{L, O\}$, and $\{L, M, N\}$ respectively. In the lower right image, an example is shown that combines all contour pieces to contours. Note that virtual points have to be added outside the convex hull of the point set so that contours are closed.

to use a Kohonen-map-based node layout on a mesh with equilateral triangles.

An OpenGL vertex and fragment shader compute the barycentric coordinates, perform the region test, classify border pixels based on the given contour piece thicknesses and even anti-alias the borders. When classifying border pixels, the distance to a contour is not calculated for the equilateral triangle but for the original triangle in order to guarantee constant line width, independent of the actual size and shape of the triangle. The fragment shader calculates the two-dimensional vectors $\vec{\Delta}_{p,X}$ as the difference of the point p as projected into the equilateral triangle and the center of each circle $X \in A, B, C, ABC$. This difference vector is then projected to a vector that gives the difference vector in the original triangle. Its length is determined and shortened by the projected circle radius in that direction. Because the projection is linear, the final tests can be expressed simply as:

$$k_{p,X} = \left|\left| J \cdot \vec{\Delta}_{p,X} \right|\right| \cdot \left(1 - \frac{R_X}{\left|\left| \vec{\Delta}_{p,X} \right|\right|} \right)$$

$k_{p,X} < 0$ means p is inside the circle X. $\frac{1}{2}|k_{p,X}| < d_K$ means p is a border pixel on one of the four circles $X \in A, B, C, ABC$, and a contour piece K with thickness d_K that lies on the circle X. J is a linear transform from the equilateral triangle with coordinates $\left(0, \frac{\sqrt{3}}{2}\right), \left(-\frac{1}{2}, -\frac{1}{\sqrt{3}}\right), \left(\frac{1}{2}, -\frac{1}{\sqrt{3}}\right)$ to the actual triangle with coordinates $(a_x, a_y), (b_x, b_y), (c_x, c_y)$.

$$J = \begin{pmatrix} b_x - c_x & \frac{1}{\sqrt{3}}\left(2a_x - b_x - c_x\right) \\ b_y - c_y & \frac{1}{\sqrt{3}}\left(2a_y - b_y - c_y\right) \end{pmatrix}$$

4.4.4 Analysis

In the following, let n denote the number of nodes. When the layout is updated, an iteration first computes the Delaunay triangulation in $O(n \log n)$ time. The number of edges in the triangulation is $O(n)$ and repulsive forces are only computed for neighbors in the triangulation. Attracting forces are calculated for all edges of the visible part of the similarity graph. It takes time in the order of $O(n \log n + E)$ to make one iteration of the layout.

We then compute the convex hull of the nodes' locations. This can be done in $O(n)$ time using $O(n)$ space when it is extracted from the

Delaunay triangulation. The vertices of the convex hull are then duplicated and additional points are inserted on long edges. Usually, their number is much smaller than n. The new Delaunay triangulation can be constructed in $O(n \log n)$ time. We use CGAL [CGA] for both the convex hull as well as the Delaunay triangulation. Colors are mixed in $O(n)$ time and $O(n)$ space. The actual rendering draws each triangle once and the triangulation does not contain more that $O(n)$ points, so the communication with the GPU takes $O(n)$ time. For each pixel the number of operations to be executed can be bounded by a constant and we render each pixel at most once. Let A denote the number of pixel of the drawing area then the overall complexity is $O(A + n \log n)$ to generate one image.

4.5 Interaction

The user may pick one ore more nodes using a mouse click or by drawing rectangles. The modifier keys known from file management systems are available and behave similarly. The selected nodes may be moved around the screen by dragging them with the mouse. After release the layout is updated towards a uniform redistribution of nodes. This allows the user to change the layout to his desire.

Text documents collections often contain a huge number of documents. Usually, only a small fraction of them can be presented on the screen because the representation is limited both by the size and the resolution of the screen. Also the perceptional abilities of the observer is limited to only a few nodes. We provide simple zoom and pan interactions as well as the expansion and the collapse of nodes. These techniques are well known from other applications for exploring hierarchies but the operations are more complex for arbitrary clusterings. A simple implementation can use an aggressive collapse strategy, where all nodes are hidden when their parent is collapsed, regardless of whether they belong to some clusters that are expanded. On the other hand, there is a safe collapse strategy as well, where only the nodes are hidden that are not inside other expanded clusters.

The user is also able to change the clustering by a gesture similar to file explorers. The current selection is dragged (picked up) and released over a point specified by the mouse pointer. As this point belongs to exactly one zone of the Euler diagram, we can add all selected nodes to all the clusters the zone represents, and remove them from all others. If the CTRL key is pressed while releasing the nodes, they are not removed from their old clusters imitating a "copy" behavior. The "move" opera-

tion can be used for removal from clusters as well, e.g., the removal from all clusters can be accomplished by dragging the nodes to a point on the zone that belongs to no cluster. For convenience, we also provide the user with a pop-up menu showing all visible clusters and allowing him to directly select and deselect clusters.

These techniques can be used with any clustering, not necessarily based on documents but we provide further means to inspect the data. When hovering over a node, its text is shown in a large tool tip window. Selected nodes have small transparent tool tips shown near them. In these tool tips either the text or some significant words of it are shown. Because the tool tip windows may not be large enough to show the whole content, it is scrolled automatically. On the selected nodes, further linguistic processing can occur. Generally, the flexibility of this visualization framework with respect to interactivity allows nearly every conceivable operation to be performed on masses of text, such as refining a clustering, ordering a number of documents into different folders or finding natural groupings for an enhanced exploration of unknown texts.

4.6 Results

To illustrate the method, a small collection of 35 newspaper texts was created. Documents concerning three topical issues were taken from six main German online newspapers (such as www.spiegel.de), namely about the Russian presidential elections, the riots in Burma, and a political issue in Northern Korea. After removing paragraphs with less then 10 words, the total number of paragraphs is 288. On the described experimental data, the clustering should theoretically produce three clusters, each containing all texts from one topic. However, the typical result is that there are up to 8 clusters with three large clusters correctly representing the three topics and several small clusters of texts that did not fit either of the big clusters. This means that the precision is usually nearly perfect, whereas recall is somewhere between 70% and 90%.

Using a standard force-directed layout, Figure 4.4 shows the structure of the text similarity graph along with a typical outcome of the CW color based clustering. As can be seen, the 'Putin' cluster is very dense with many high similarities in between its texts, whereas the other two clusters have several inter-cluster similarities, which is because both topics discuss military and unrest in a country. The three most significant terms for each cluster of one randomly chosen clustering outcome of the experimental setup are shown below (along with the number of texts contained

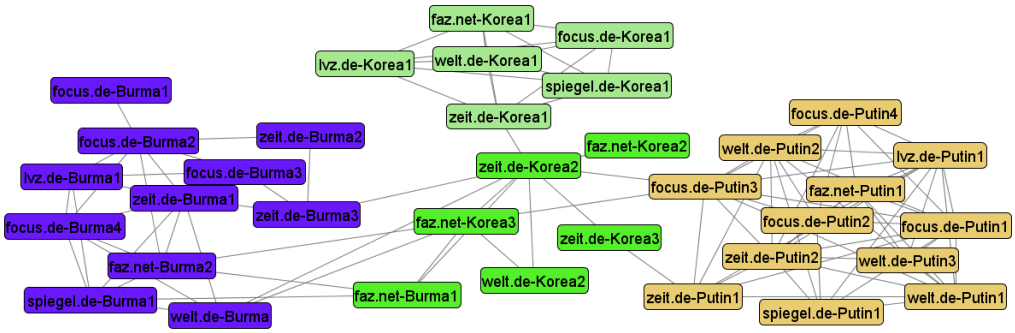


Figure 4.4: Typical result of Chinese Whispers clustering of the 35 experimental documents.

in that cluster):

1. (10) Putin, Subkow, Russland
2. (6) Roh, Pjöngjang, Grenze
3. (6) Burma, Burmesen, Junta
4. (4) Nordkorea, Kim, Pjöngjang
5. (4) Mönche, Birma, Nagai
6. (3) Gambari, Rangun, Birma
7. (2) Iwanow, Putin, Subkow

After locating the overlaps in the Euler diagram we then used the tool tips to investigate the data further. A screen shot from this session is presented in Figure 4.5. This session revealed that paragraph clustering can help improving the clustering as one can find useful interrelations between the text clusters. The observer can immediately recognize that the red cluster is separated. Investigation into the data reveals that it could be merged with the white, magenta and blue cluster as they deal with the same topic. For the same reason, the green and the yellow cluster might be merged as well. There are only two overlaps: one between the green and the cyan text cluster and also one overlap between the red, white, and magenta text cluster. This indicates that the clustering on the paragraphs mostly concur with the clustering of the text. After merging all clusters to their correct topic, only the overlap between the green and the

cyan persist. Investigating the texts reveals that the one paragraph of the green cluster, which deals with a political issue in North Korea, mentions economic relationship with Russia. A political issue in Russia is the topic of the cyan cluster.

Our program SIMDOCBROWSE shows two clusterings: one for the texts and one for the paragraphs. Both were generated with the Chinese whispers algorithm. We were mainly interested, where the clustering of the paragraphs disagrees with clustering of the texts and whether this information can be used to correctly identify the topics. We used this dataset to first determine the best interpolator of the five we have presented. The interpolators can be switched at the run-time of our program to examine their differences in detail. Figure 4.6 shows the visualization on the same dataset and the same layout of the elements using the different interpolators. At first, we were impressed that although the layout of the elements is determined independently of the clustering graph, only few clusters are disconnected. This indicates the high quality of the CW clustering algorithm. The parts where clusters are disconnected seem to be most visible when using the interpolators of the ROUND class. The ROUND7 and ROUND9 clusters furthermore show overlaps of clusters better than the others. We prefer the ROUND9 over the ROUND7 interpolator because contours are less jaggy and therefore cause less stress to the observer.

In Figure 4.7 the image is separated into patterns and contours allowing to study the contribution to their combination. The contours mask the patterns, which is why in the combination the paragraph clustering is not as salient as the text clustering using colors. Figure 4.8 shows an example where too many contours produce simply too much clutter and hide the essentially simple clustering just using grayscale colors, indicating that colors and patterns may be more efficient in communicating set relations than contours.

4.7 Summary

We implemented a general framework for representing non-hierarchical clustering structures as Euler diagrams using different interpolators. It makes the implementation of new interpolators easy, in fact, the ROUND9 interpolator was designed last and was implemented in less than half an hour. The presented visualization algorithm has proved to be a very intuitive and sufficiently fast way of representing the relations between texts and parts of texts. The computational linguistics algorithms, responsi-

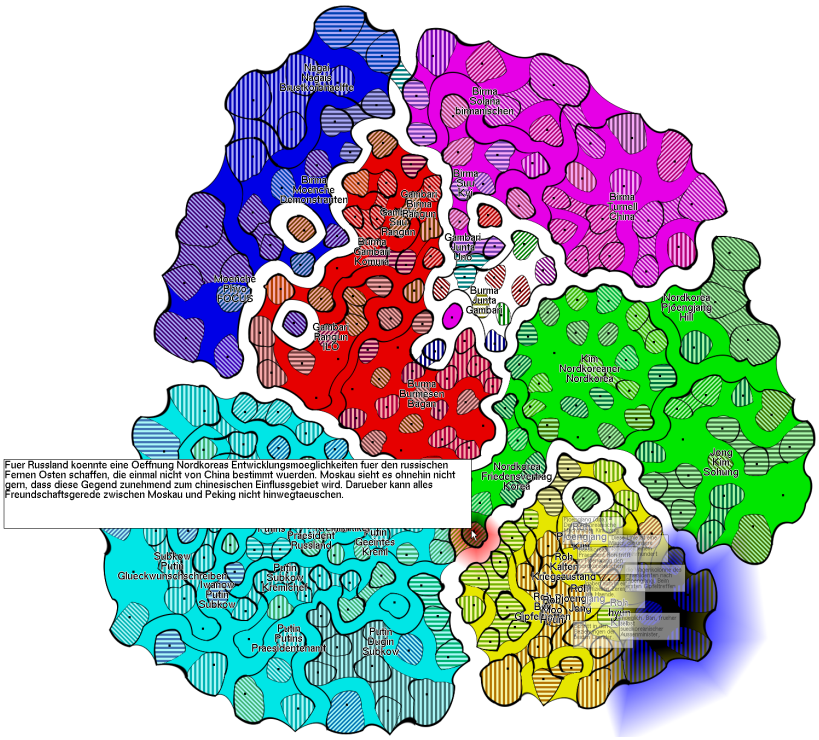
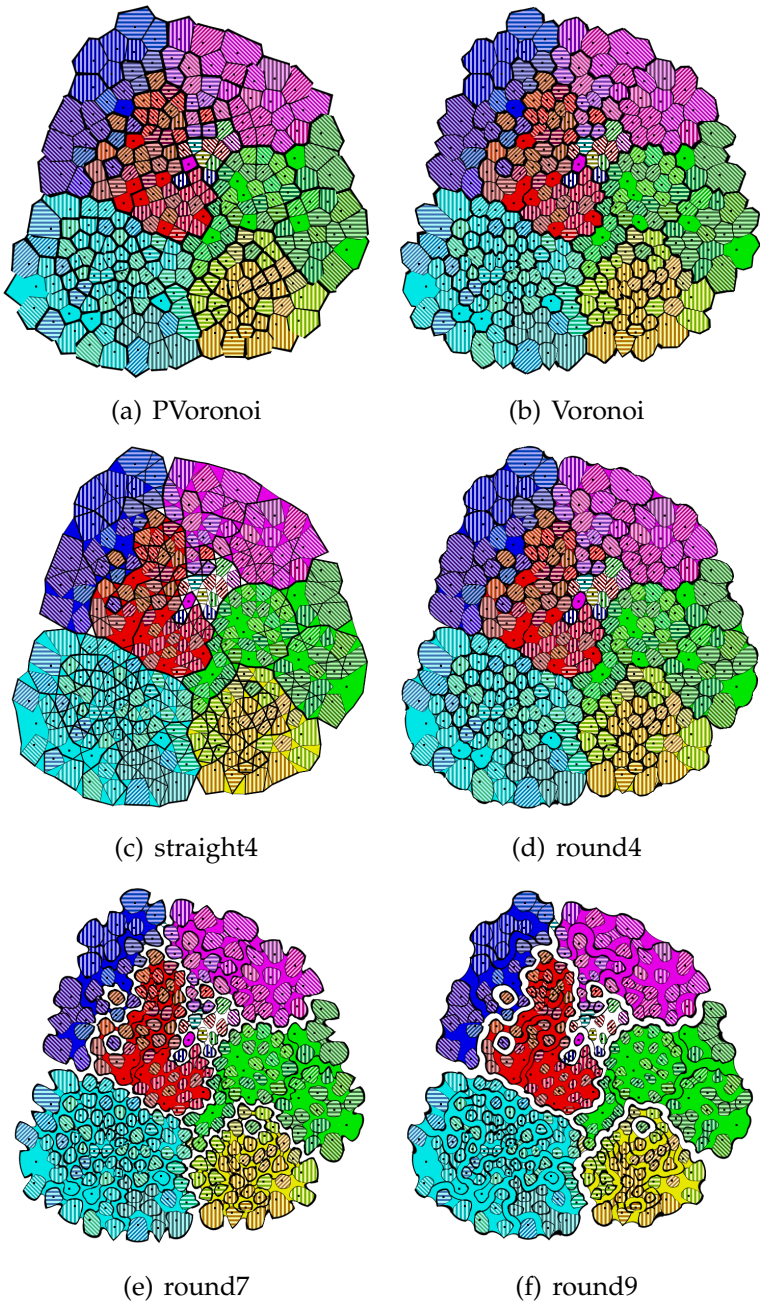


Figure 4.5: An Euler diagram of a clustered collection of text documents. In the example, there are 35 texts that have been partitioned into 7 clusters and 288 paragraphs partitioned into 66 clusters. The texts themselves are also clusters for the paragraphs but have no texture, only a border. Clusters that have only one element (text or paragraph) have no texture attached, e.g., there is a "white" cluster of just one text that only has a border. In total, there are 108 clusters and 6 of them have a color, 60 of them have a pattern attached. Because we only have 48 patterns available, some of them were assigned multiple times. In the yellow cluster, some paragraphs have been selected and the mouse currently hovers over a text and the content of it is shown.

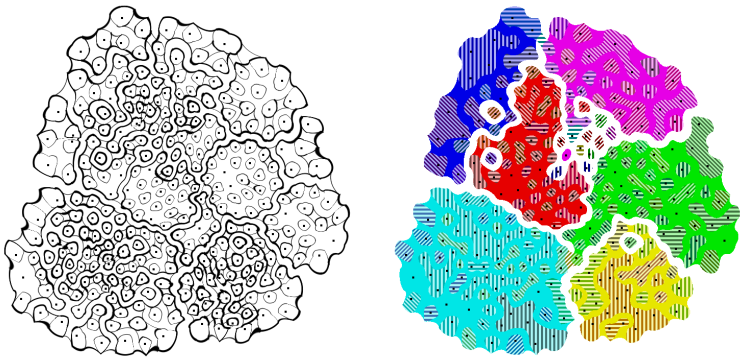


(a) PVoronoi (b) Voronoi

(c) straight4 (d) round4

(e) round7 (f) round9

Figure 4.6: The same clustering structure using different interpolators. Labels have been omitted. The PVORONOI interpolator emulates a Voronoi treemaps to give an impression what the data would look like if that method were used instead, although the emulator lacks the shading of cells and their thicker borders.



(a) without patterns (b) without borders

Figure 4.7: Demonstration how the combination of contours and textures support each other towards an improved perception of the clustering. The same dataset as in Figure 4.5 is used, using the ROUND9 interpolator.

ble for the clusterings and cluster labels, proved to be very powerful and also produce very fitting relations. Therefore, it is not surprising that even though Figure 4.5 contains a total of 323 elements in 73 clusters, it still is easy to grasp and renders sufficiently fast to allow interactive exploration (at a resolution of 1280x800 with 17-21 FPS on a GeForce Go 6100). The actual rendering is very fast, we determined the performance bottlenecks of our application to be the rendering of labels and, at large numbers of visible elements, the Delaunay triangulation and the layout refinement that is based on it.

Further research will concentrate on improving even further the quality of the clusterings by adding more linguistic knowledge, such as automatic morpheme segmentation or automatically acquired word associations. Another huge improvement regarding the rendering speed can be obtained if the Delaunay triangulation were not to be computed in each step of the layout rearrangement, but replaced by a dynamic Delaunay triangulation with lower average run-time complexity. Because the elements are moved only slightly during layout rearrangement the topology of the Delaunay triangulation mostly stays the same. Because of that, mixtures only have to be determined when the topology changes. In our current implementation, it is cumbersome to detect these events. This could be improved if a dynamic Delaunay triangulation including event handlers is used. Finally, many ideas regarding the possible interactions with the underlying data will be explored in more detail.

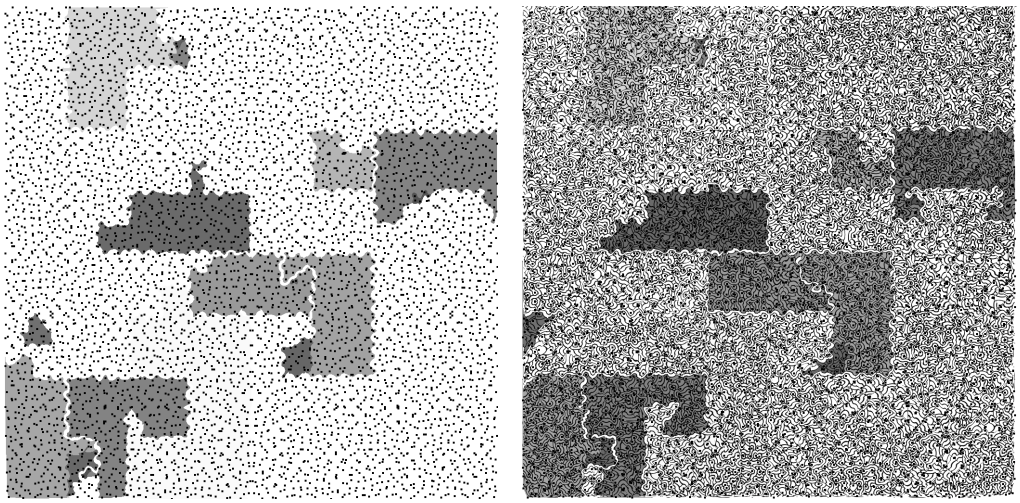


Figure 4.8: A clustering of the isolet dataset. The isolet dataset consisting of 6238 data points in 25 dimensions has been clustered using a divisive hierarchical clustering. The points have been laid out by depth-first traversal of the hierarchy and positioning them on a Hilbert curve ([MM08]). The region of each point is colored using grayscale according to the height of the point's parent, i.e. the distance at which he was merged into a cluster. Using this simple scheme already shows the clusters, while showing all contours leads to clutter. The uniform region colors indicate that the intracluster distances are homogeneous but vary by cluster and therefore a single maximal distance for partitioning the points could not be found.

5 Drawing Contour Trees in the Plane

Scientific and medical visualization often involve scalar fields. A common visualization technique is the isosurface: a geometric surface defined by a single function value. Connected components of an isosurface are called *contours*, analogous to contour lines on a map.

Large datasets can be studied with topological analysis of the contours. For an arbitrary manifold, the Reeb graph [Ree46] describes contour evolution as a topological skeleton in graph form. For the special case of functions on simply-connected domains, the Reeb graph is also connected and acyclic: the *contour tree*.

Since the contour tree summarizes contours of a scalar field, it can be used as a visual abstraction of the field in user interfaces [BPS97]. This requires layouts that emphasize topological and geometric properties. In particular, it is desirable to fix the y-coordinates of the tree nodes using the isovalues of the corresponding contours in the tree.

Existing graph-drawing algorithms work poorly for contour trees, as they assume that vertices can be given arbitrary (x, y) positions, or that constraints in y are based on graph properties, not vertex or edge attributes.

We therefore formalize aesthetic constraints for drawing contour trees, adapt and evaluate some existing graph drawing approaches to contour trees, and propose a novel algorithm for automated 2-D layout of large contour trees subject to these constraints. We choose 2-D layout rather than 3-D, because abstract information such as graphs is more easily understood in 2-D [War04] and because existing 3-D layouts struggle to maintain a user's sense of orientation to the isovalue dimension.

5.1 The Contour Tree

A scalar field is a continuous function $f : \mathbb{R}^n \to \mathbb{R}$, whose range is a single (scalar) value. We assume the domain to be simply-connected. Level sets, or isosurfaces, are sets $S(w) = f^{-1}(w)$ for some *isovalue* w. Figure 5.1 shows an example of isosurfaces in a small dataset.

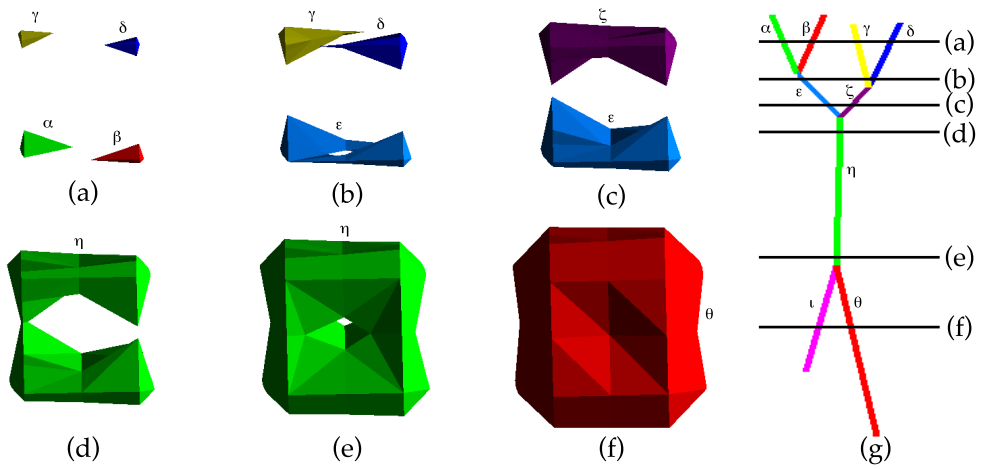


Figure 5.1: A set of isosurfaces with corresponding contour tree. By setting the y-coordinate of the vertices based on the isovalue, we preserve the property that a horizontal line cuts exactly one edge of the tree for every contour at the corresponding isovalue. In this example, it is possible to choose x-coordinates so that no edge crossings occur. Moreover, secondary attributes such as color can be used to emphasize the relationship between individual contours and the contour tree. Edge ι represents a set of contours not visible in the images because they live inside the contours of edge θ.

A *contour* is a connected component of $S(w)$. As w increases, contours appear at local minima, join or split at saddles, and disappear at local maxima. Contracting each contour to a point gives the *contour tree*, which tracks this evolution (Figure 5.1). The contour tree is then the nesting diagram of all possible contours [BR63] and has a 1–1 mapping from points in the tree to contours of f.

More formally, Morse theory [Mil63] shows that topological changes in f occur at *critical points* where the differential of f vanishes. Leaf nodes represent local extrema, while interior nodes represent saddles at which global connectivity changes, but not saddles at which surface

genus changes. All other contours at w occur at the intersection of edges with a horizontal line $y = w$.

This property is fundamental to the utility of the contour tree: it implies that each contour can be represented uniquely by a single point in the contour tree. Since the isovalue is important both to the definition of the contours and to human perception of the abstraction, this leads to an unusual constraint for 2-D graph layout: the y-coordinate is fixed, but the x-coordinate is free.

The contour tree has been used to index contours [BR63, CS03], to simplify data topologically [CL03, CSvdP04, PCMS04, TNTF04], and to compare fields directly by graph matching [ZBB04, SWC+08]. In user interfaces, the contour tree is displayed to help the user explore and analyze the dataset [BPS97] and as an abstraction for indirect manipulation of individual contours [CS03, CSvdP04, SWC+08] or regions of the data [TTFN05, WDC+07].

Simplifying the contour tree and the underlying topology is performed by pruning (removing) branches of the tree in order of importance [CSvdP04, PCMS04, TNTF04], as shown in Figure 5.2. Longer and longer sets of edges are joined by sequences of collapses – a process known as *branch decomposition*. Each *child branch* c connects at a saddle s to a more stable *parent branch* p. This parent-child relation defines the *branch hierarchy*, with the most stable branch being the *root branch*. The desire to perform pruning dynamically leads to an additional criterion when drawing the contour tree: vertices should be inserted onto their parent edge, as shown in Figure 5.2, fixing both x and y coordinates. Moreover, simplification can be driven by geometric properties such as persistence (range of isovalues), volume (of the branch), and hypervolume (integral of the function inside the branch) [CSvdP04]: the desire to represent these properties also leads to a criterion.

5.2 Related Work on Contour Tree Drawing

While substantial literature exists on the computation and uses of the contour tree, relatively little work has been reported on its visual layout and presentation.

Shinagawa and Kunii [SKK91] gave an algorithm for 2-D layout of the Reeb graph, using iconic descriptions of the nesting of contours. The images are hard to interpret because the viewer has to know the meaning of the icons.

Bajaj, Pascucci, and Schikore [BPS97] showed contour trees and geo-

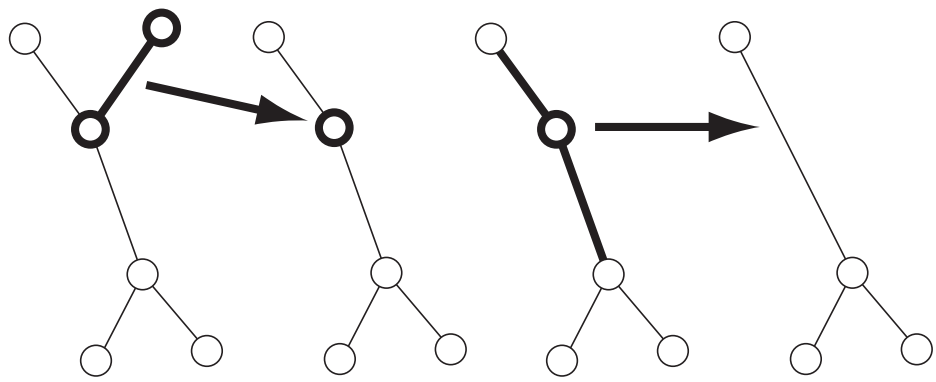


Figure 5.2: Simplifying a contour tree by pruning an edge (left) and collapsing a redundant node (right).

metric properties to guide isovalue selection, but did not discuss algorithms for contour tree layout. Carr and Snoeyink [CS03] extended this work to use contour trees to manipulate contours indirectly by dragging tags through a tree drawing, representing the evolution of a particular contour as the isovalue varied. This implies that smooth contour evolution should be represented by smooth tag movement, penalizing angular drawings of edges. They also noted that some contour trees cannot be laid out without crossings (Figure 5.3a), and instead provided manual layout tools and the ability to use *dot* [GKNV93] to lay out trees of 256 vertices or less, but at the expense of run-time on the order of several minutes.

Pascucci, Cole-McLaughlin, and Scorzelli [PCMS04] introduced the *toporrery*, which laid the contour tree out radially in x and y, then set the z-coordinate to the isovalue. L-shaped edges were used to connect the vertices. While these L-shaped edges help the user understand 3-D orientation, the interface requires non-intuitive 3-D interaction with an abstraction of the data, and does not avoid occlusion due to edge crossings in projection). Moreover, rotation and perspective projection hamper the user's ability to compare isovalues visually.

Weber *et al.* [WBP07] generated landscapes with specific contour trees in 3-D and were able to indicate both the branch hierarchy and an additional geometric property (they chose volume), but again this method has the problem of choosing the right projection because mountains often occlude smaller mountains or valleys.

The automatic visualization of the more general Reeb graphs seems

even more problematic. Doraiswamy and Natarajan [DN08] used the toporrery method [PCMS04] on a contour tree that spans the Reeb graph. A more recent work by Doraiswamy and Natarajan [DN08] used the Sugiyama-style layout of Tulip [Aub04].

Recently, Takahashi *et al.* [TFO09] extended the isomap [TSL00] framework to automated computation and presentation of approximate contour trees. We discuss the part of this work that deals with contour tree drawing and propose modifications in Section 5.3.3.

5.3 Graph Drawing

As previous work shows, visual representations of contour trees using graph drawing techniques usually use a Sugiyama-style layout (e.g. *dot* [GKNV93], Tulip [Aub04]). However, a formal discussion of the application of standard graph drawing techniques to contour trees has been lacking. In this section, we remedy this with an account of how graph drawing can be applied to contour trees, and identify the shortcomings of existing methods.

Existing basic algorithms for tree drawing, e.g. rooted tree drawing [RT81] and radial layout [Ead92], are not well-suited to the contour tree, as they do not indicate the isovalues of the critical points. Tree drawing algorithms that support drawing vertices with one coordinate fixed can draw dendrograms [Har75], which show cluster relations, and phylogenetic trees [PLCB04], which show the evolution of species: species being leafs and inner nodes being hypothetical predecessors where evolution diverged. The drawings of our algorithms resemble their style, but combinatorially, these trees are a subclass of contour trees, because they fan out in only one direction.

These shortcomings lead us to consider more general techniques for drawing contour trees and Reeb graphs.

5.3.1 Layered Graph Layout with *dot*

Reeb graphs and contour trees can be described as DAGs, with edges oriented from lower to higher isovalue. It is therefore possible to apply DAG drawing, such as the Sugiyama approach [STT81], which splits the problem in three phases: Rank, Order, and Position. *Rank* assigns vertices to ordered *layers* with edges directed from lower to higher layers. *Order* sweeps layers and swaps vertices and edges in each layer to minimize edge crossings. Finally, *Position* adjusts vertices to minimize edge lengths

but preserves their intralayer order. Most DAG drawing algorithms fit this framework: our proposed algorithm may also be thought of in these terms.

A popular algorithm for drawing DAGs is *dot* [GKNV93]. It extends the *Rank* phase in the Sugiyama framework by adding *augmented* vertices to each layer that an edge spans. Crossing minimization in the *Order* phase can then be done by swapping vertices. *Position* ranks vertices horizontally to determine final x-positions. *dot* also allows minimum vertex separation and edge importance to be specified. These are used to keep important edges mostly free of crossings and bends. *dot* ends with a post-processing phase that translates augmented vertices to control points of splines so that edges are drawn as smooth curves. However *dot* may need to add as many as $O(|V| \cdot |E|)$ augmented nodes [Fri96]. Thus, *dot* works best when the number of augmented nodes is small.

dot defines the *ranking* problem as: Given a DAG (V, E), let $\delta : E \to \mathbb{R}^+$ be the minimum edge length, $\omega : E \to \mathbb{R}^+$ be the importance of an edge, then a *ranking* $\lambda : V \to \mathbb{R}$ can be found by the linear program:

$$min \sum_{(u,v)\in E} \omega(u,v)(\lambda(v) - \lambda(u))$$

$$\text{subject to: } \delta(u,v) \leq \lambda(v) - \lambda(u) \quad \forall (u,v) \in E.$$

The importance of an edge indicates the desire to keep that edge short. Gansner *et al.* [GKNV93] give an algorithm to solve instances of this problem. We also use it in the last phase of our algorithm (Section 5.5).

dot is very popular but also dated. More recently, Auber introduced *Tulip* [Aub04] based on the Sugiyama framework. Tulip reduces memory requirements and run-time by inserting fewer augmented nodes: no more than two augmented nodes per edge, reducing the requirements to $O(|V| + |E|)$ space and $O(|V| \cdot |E|)$ run-time. Eiglsperger *et al.* [ESK04] then further improved the run-time to $O(|V| + |E| \log |E|)$ under the same constraints. Both publications provide results only for small graphs with few or no crossings. A loss in quality compared with *dot* was noted [ESK04] but not shown.

Applying *dot* to Contour Trees

Trivially, *dot* applies to contour trees by ignoring the actual isovalues and preserving only edge orientation. But this ignores the principal visual constraint on contour trees: the y coordinate must reflect the isovalue. While the y-coordinate can be reset to the isovalue, this leads to occlusions and additional edge crossings.

This can be avoided by exploiting *dot*'s layers. One layer is defined for each isovalue containing all vertices at that value. Thus, resetting the y-coordinate introduces no new edge crossings or overlaps. This layering is achieved by subdividing every edge at every isovalue and forcing vertices to their correct layer by issuing *samerank* statements in the *dot* input file. In practice, this requires *dot* to augment many vertices and causes a prohibitive run-time. For example, the gas furnace chamber dataset (see Section 5.6) augmented from $1,556$ to $99,137$ vertices (unpruned) or from 214 to $4,750$ vertices (pruned).

A partial solution quantizes the isovalue of each vertex to a layer $l(v) = k \cdot f(v) \cdot (w_{min} - w_{max})^{-1}$, where w_{min} and w_{max} denote the minimum and maximum isovalue, respectively. The y-position assigned to each vertex is then close to the desired value, so additional crossings or overlaps are unlikely when correcting the y-value. However, the user must choose the number of levels k, and the run-time of the algorithm increases significantly with the number of levels due to the increased number of augmented nodes (Table 5.1).

A layout generated by *dot* using 50 layers is shown in Figure 5.7c. As we can see, contour tree drawings using layer-constrained *dot* contained very few edge crossings (see Section 5.6), but at an unacceptable cost in time.

An example drawing using [ESK04] is shown in Figure 5.7d and Figure 5.8b. While this has successfully reduced the number of crossings, the branch decomposition structure of the contour tree is not apparent. However, the loss of quality predicted by Eiglsperger [ESK04] was not observed, even for very large contour trees.

In summary, then, *dot* is unsuitable for large contour trees. Either the y-constraint is not respected, or extra crossings are generated when resetting y-coordinates, or layering with augmenting vertices induces a prohibitive run-time, even with recent optimizations of *dot*. And in all Sugiyama-style layouts, the branch decomposition structure of the contour tree is not apparent.

5.3.2 Stress-Based Graph Layout

A popular general graph drawing algorithm by Kamada and Kawai [KK89] embeds the data points in a low-dimensional space where the distances used for drawing resemble the original distances d_{ij} between points i, j in an high-dimensional space, or the graph-theoretic distances between vertices i, j. The quality of such a mapping can be quantified using the

stress:

$$stress = \frac{1}{2}\sum_{i<j} w_{ij}(d_{ij} - ||p_i - p_j||)^2$$

where p_i is the position in the low-dimensional space, and $w_{ij} = d_{ij}^{-2}$ gives the importance of fitting the distance d_{ij}. This cost function can be interpreted as the energy of a system of springs connecting points that have an ideal rest length of d_{ij} and stiffness w_{ij}.

There are many known techniques for finding embeddings optimizing the stress, such as stress majorization [GKN04]. Koren and Harel [KH03] described a method which minimizes stress for each coordinate in turn. Since we wish to fix the y-coordinates of the vertices, this method is easily modified, simple to implement, and can be extended to more than one dimension if needed, although Gansner *et al.* [GKN04] claim that stress majorization and axis stress minimization are only equivalent in the one-dimensional case.

The axis stress algorithm computes optimal x-positions given fixed y-positions using simple linear algebra techniques. When the y-positions are fixed, the stress is defined as

$$stress = \sum_{i<j} w_{ij}(\sqrt{d_{ij}^2 - (y_i - y_j)^2} - |x_i - x_j|)^2.$$

The algorithm approximates this functions by a slightly simpler function that bounds *stress* from above and for which one set of x_i can be computed exactly by solving a system of linear equations. This solution is used to construct a better upper bound on *stress* and new x_i are computed. The process is iterated until no better bound can be found and the last set of x_i is then returned.

Applying Stress Based Layout to Contour Trees

Unfortunately, this approach performs very badly for contour trees, as can be seen in Section 5.6 and Figure 5.7b). Starting from an initial random configuration, the algorithm is very easily trapped in a local minimum. Even when another approach is used to find a reasonable starting configuration, the axis stress method often fails after only a few iterations, or generates a configuration that is actually worse than the initial configuration. We believe that this is because the strong constraints on the contour tree and the inability to move nodes vertically generate many local minima and many impassable barriers to the assumptions behind stress-based optimization.

5.3.3 HDE-Based Graph Layout

Harel and Koren also gave a fast algorithm [HK04] that draws general undirected graphs on n vertices by embedding them in a k-dimensional space ($k \ll n$), then using a maximum-variance projection onto the plane. They called it *high-dimensional embedding* (HDE).

The initial embedding is chosen to be the shortest distance from each vertex to k landmark vertices. These landmarks are equally distributed in the graph and are determined by an approximate k-centers problem: the first landmark vertex is chosen at random, and each additional landmark is chosen to be the vertex with maximum shortest distance to all previous landmarks. The shortest distances of each vertex i to each of the k landmarks are stored in k-dimensional vectors X_i which are then centered $\hat{X}_i = X_i - \frac{1}{n}\sum X_i$. Then a covariance matrix is computed $S_{ij} = \frac{1}{n}\hat{X}_i^T \cdot \hat{X}_j$. The two eigenvectors e_1, e_2 for the two biggest eigenvalues of S are computed and used as projection directions to give each vertex i a position $(e_1, e_2) \cdot \hat{X}_i$ in the plane. This approach is not particularly well-suited to drawing trees, as they usually span large high-dimensional subspaces ([HK04]). Selecting only one or two eigenvectors for projection then normally results in visual overlaps.

Applying HDE Layout to Contour Trees

Close observation of Takahashi *et al.* [TFO09] reveals that their method of drawing contour trees is essentially the HDE approach on the x and y coordinates of the reconstructed domain mesh. Thus, only x and y are computed using HDE, while z is set to the isovalue of each vertex. Moreover, during computation of shortest distances the difference in isovalue of their incident vertices is used for each edge's weight (i.e. length). What the method fails to address is why this approach works, and we will investigate this in the remainder of this section.

Takahashi *et al.* [TFO09] introduced the geodesic distance between vertices, which is the sum of the absolute isovalue differences along the shortest path sp in the given mesh graph $M = (V, E)$

$$d_{ij} = \sum_{\{u,v\} \in sp_M(i,j)} |f(u) - f(v)| \,.$$

Although not stated explicitly, geodesic distance d in a mesh M is related to geodesic distance d' in the Reeb graph R computed from M. In fact, d bounds d' from above and the difference, if any, is reduced if M's resolution is increased. Moreover, where the shortest path sp uses a small

number of mesh edges, d tends to be a better approximation of d'. Thus, positioning vertices using d and M produces results similar to using d' and R, without needing to compute either R or d' from the data. However, this method produces only a visual arrangement of points, from which the Reeb graph must be inferred. This works well with simple Reeb graphs or contour trees, but tends to break down due to visual occlusion in large graphs, or where many of the branches are short (i.e. have few vertices).

Because geodesic distances in the mesh and Reeb graph are related, the method can also be applied directly by using the contour tree or Reeb graph as a substitute for the mesh (in which case $d = d'$).

We then apply HDE using the geodesic distances and compute one eigenvector-eigenvalue pair for the x coordinate and set y of each vertex i to the isovalue $f(i)$. To make the geometric distances in the drawing a better match for the geodesic distances, we follow Koren and Harel [KH03] and subtract $y = f$ from the distance computation, as this already ensures separation in the drawing. To that end we replace d_{ij} by

$$d'_{ij} = \sqrt{d_{ij}^2 - (f(i) - f(j))^2}.$$

In both Takahashi *et al.* [TFO09] and HDE, the landmarks are determined heuristically using the k-centers problem. We instead use the contour tree itself. As we have already noted, there is no significant visual difference between using all the vertices of the mesh and using only critical vertices as landmarks. Essentially, any good drawing of the contour tree shows the relationship of the critical vertices. It therefore follows that using the critical vertices as landmarks is likely to lead to a suitable result. We further optimize by considering only the local extrema, as in most cases they have the greatest influence on the shape of the contour tree.

Even if these modifications are applied to Takahashi *et al.* [TFO09] their approach is no longer successful for larger contour trees, because subtrees tend to overlap at the borders of the drawing, as can be seen in Figure 5.7a and Figure 5.8a. This is consistent with the observation that HDE performs poorly for trees in general.

5.3.4 Summary of Existing Methods

We have seen that while various existing methods can be modified or coerced to draw contour trees, these drawings are rarely successful in accentuating the fundamental structure of the trees. This is primarily

because the layout of contour trees is strongly constrained in ways that are inconsistent with the assumptions of these methods. In short, these methods have been applied on an *ad hoc* basis, rather than in a considered fashion.

5.4 Contour Tree Aesthetics

In general terms, graph drawing algorithms operate best when there is a clear statement of what constitutes a good drawing, expressed as a set of *drawing aesthetics*. We therefore start our analysis by considering the aesthetics relevant to contour tree drawing. We begin by observing that the contour tree can be viewed as a layered free (i.e. rootless) tree with additional semantic, topological, and geometric information associated with it. This is best displayed when the following aesthetics are observed:

- *crossing line:* cutting the tree at height $y = w$ results in the same number of crossings as there are contours at the isovalue w,
- *branch stability:* parent branches in the branch hierarchy (more important) should be more stable in the layout than child branches,
- *smooth movement:* moving a tag on the contour tree by a small amount in y direction should not result in a large movement in x direction,
- *crossing minimization:* edge crossings convey no information about the contour tree, increase visual clutter, and should therefore be minimized. Prefer crossings between child branches over crossings between parent branches, and
- *geometric properties:* geometric properties such as volume, etc. should be presented as ancillary (attributed) information.

Usually, the isovalue w of a critical point is closely associated with its y-value in the drawing. Conceptually, the isovalue derives from some physical property, e.g. pressure, and is therefore of a different domain than a screen coordinate. So the actual mapping may be described by a function $y = c \cdot w$ for some constant c. It is also possible to generalize the *crossing line* property by replacing y by any strict monotonic function that preserves the relative vertical ordering of critical and regular points. We present an example for such a function in Section 5.5.7 that improves the resolution in regions with very similar critical values. Note that to achieve the *crossing line* property, the edges' Jordan arcs also have to be

monotonic in the y direction and do not exceed the y-values of their incident critical points.

The *branch stability* property must be strengthened when the drawing may be simplified after pruning. When a saddle is removed as a result of a child branch being pruned, the parent branch will deform unless the saddle lies exactly on the parent branch as drawn in the pruned contour tree. We therefore require that the saddle between a parent and a child branch should be positioned on the direct line between the highest and the lowest point of the parent branch.

For the *smooth movement* property, there is no direct algorithmic implication. One may model it as: edges' curves have to be *strictly* monotonic in y direction, or: never exceed a specified angle with the y-axis. Note that these two models are similar in nature: if all edges are strict monotonic there is always a non-singular scaling in x-direction that results in all angles being lower than the threshold. Small angles should also be avoided to allow the visual discrimination of edges.

Geometric properties can be shown by varying an edge's thickness or drawing a partially transparent shape of varying thickness around it. Multiple thicknesses and colors can then be used to show more than one geometric property. Note that using the mapping $y = c \cdot w$, the *persistence* of edges and branches is directly visible.

As usual in graph drawing, not all criteria may be met simultaneously. Also, individual applications will rank their importance differently. Figure 5.3a shows that it is impossible to avoid edge crossings fully if the y-values are fixed. But even if tree edge crossings can be avoided preserving the isovalues for some contour tree, the smooth evolution is sometimes impossible (Figure 5.3b).

As Sugiyama-style layouts are driven by minimizing edge crossings, they naturally respect the *crossing minimization* criterion. Ensuring the *crossing line* criterion requires either a post-processing by shifting the vertices to their correct y-value or a proper layering different from the original implementation. Details were given in Section 5.3.1. The post-processing phase of *dot* makes edges smooth, thereby partially respecting *smooth movement*. Both the stress-based and HDE-based approach conform to the *crossing line* and *smooth movement* aesthetic criteria. Note that we can not attest the *crossing minimization* criterion to either method, because Kamada and Kawai [KK89] already observed that distance-based drawings – in general – favor symmetry over minimization of edge crossings. All methods fail to show *branch stability* and *geometric properties* (other than persistence).

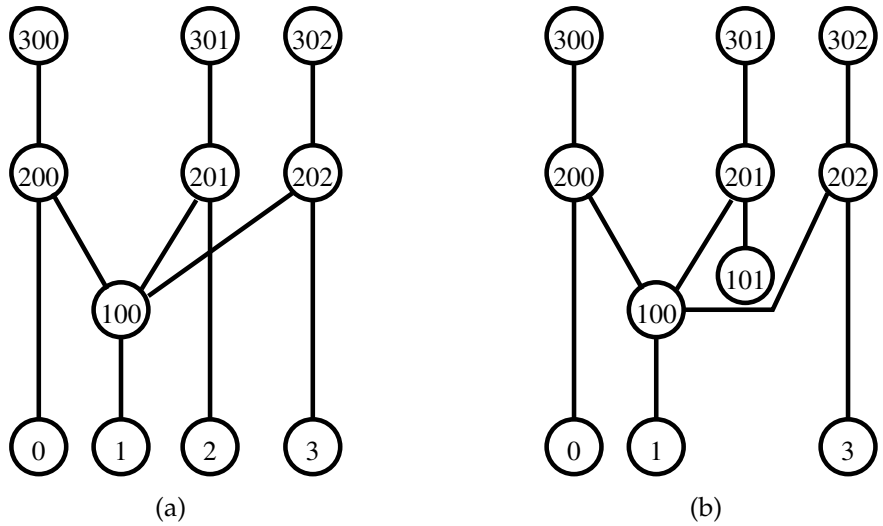


Figure 5.3: Unavoidable edge crossings. (a) a contour tree with an unavoidable edge crossing if y-values are constrained by isovalues. (b) a contour tree with an unavoidable near-horizontal edge (either $(100, 200)$) or $(100, 202)$, without violating y-value fixation or allowing an edge crossing. Note that allowing 101 to come arbitrarily close to 100 is also not desirable as then two edges would nearly lie on each other, i.e. their angle becomes very small.

The *yFiles* layout of Eiglsperger [ESK04] can be modified to place all vertices of each branch to lie on a line. The crossing minimization then sweeps through all layers and places each vertex close to the vertices in the adjacent layers. Unfortunately, the *branch stability* constraint then fixes this so rigidly that the crossing minimization gets stuck directly in a local minimum after one sweep.

In this chapter, we present two novel drawing methods for contour trees. The *diagonal* algorithm (see Section 5.5.8) respects the *crossing line*, *branch stability*, and *smooth movement* properties. It ensures the last property by drawing all edges diagonal, so the movement of a tag in x-direction always equals the movement in y-direction. The *orthogonal* algorithm (see Section 5.5) respects all but the *smooth movement* criterion.

5.5 *Orthogonal* Contour Tree Drawing

The *orthogonal* method is centered around *crossing minimization*. Branches are drawn as vertical lines and are connected by saddles which are drawn as horizontal lines rather than points. This restriction obviously violates the *smooth movement*, because tags can "jump" when they pass a saddle, but it allows us to divide the layout problem into four phases that transform the pure combinatorial part of the layout (i.e. the branch hierarchy plus its attributes) step by step into the final layout. The phases of the algorithm, as illustrated in Figure 5.4 are: s

- SIMPLIFY: Partition the hierarchy in *branch groups* that can always be drawn without edge crossing.
- PERMUTE: Order the branch groups to minimize edge crossings.
- ORDER: Partially order all branch groups and their contained branches.
- POSITION: Position vertices horizontally, preserving order and minimizing total horizontal edge length.

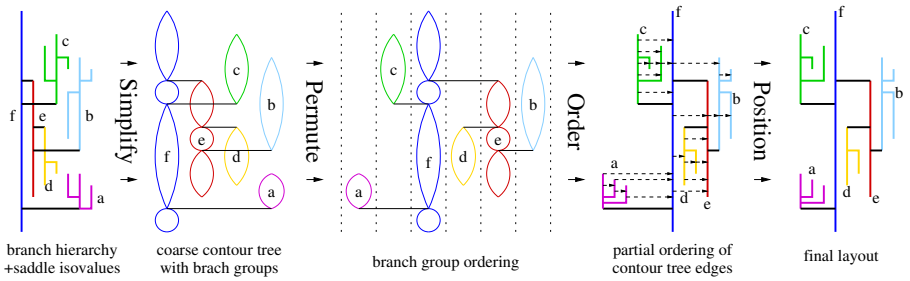


Figure 5.4: The phases of the *orthogonal* algorithm. The contour tree is simplified to a coarse tree of branch groups. The branch groups are then ordered and minimum separation constraints are computed from the ordering from which finally x-positions are computed. See main text for details.

Note that the last two phases of the algorithm are named after phases in the Sugiyama framework because they are similar in spirit. Our algorithm omits the *Rank* phase and treats all vertices as belonging to the same layer. One may also think of the SIMPLIFY phase as a *Rank*ing into vertical layers and then *Order*ing them rather than individual vertices.

The method shows *geometric properties* by using branch silhouettes representing the varying contour size along each branch. Integrating the size over the isovalue would give the volume the branch contains, therefore drawing the contour sizes also shows this volume via the area of the branch's silhouette. We assume that the varying contour size has been computed beforehand, usually during the contour tree computation. In our implementation, we use the approach presented in the contour spectrum [BPS97] to that aim. For faster rendering, we resample the actual varying contour size along each branch using a fixed step size and use piece-wise linear interpolation.

5.5.1 The SIMPLIFY Phase

The SIMPLIFY phase finds substructures of the contour tree $T = (V, E)$ that can be laid out trivially. These substructures are simplified to obtain a *coarse contour tree* T', for which the run-time is much reduced and the convergence of the PERMUTE phase is improved.

A *branch group* is a set of connected branches that satisfy a *crossing criterion*: for any saddle s of the branch group connecting it to a different branch group, there is only one branch in the group that has an isovalue range that includes s. Because branches will be drawn as vertical lines and saddles as horizontal lines connecting them, this can also be described as: each saddle s crosses only one branch of its group. This criterion ensures that the layout inside the branch group is independent of the global ordering of branch groups.

The algorithm for this phase starts with each branch of the branch hierarchy being its own branch group and then successively joins branch groups c that are leaves in the hierarchy with their parent unless there is a sibling s of c where the isovalue of the saddle connecting p and s lies within the isovalue range of c, thus ensuring the crossing criterion. A child that joined its parent is removed from the hierarchy and if all children of a branch have been joined, that branch too becomes a leaf. The algorithm terminates when no branches can be joined without violating the crossing criterion.

Figure 5.4 shows a sub-optimal partition of a contour tree into branch groups. The shape of branch groups was chosen to indicate that branches may have varying "thickness" (number of edges at a given isovalue), although only one edge at the saddles connecting to other branch groups. In the illustration, branch group a could join with the branch group f, however, it is not done for illustrative purpose. The group b cannot join e because of the saddle between d and e and similarly d cannot join e. e

may not join f because it is no leaf.

We observed that this phase joins branches that "grow" mostly in one direction, i.e. have only short "back branches". Furthermore, the result of the algorithm is not dependent on the actual order in which the joining takes places. The number of branches in the hierarchy is one less than the number of leaves in the contour tree. For contour trees without multisaddles, this automatically means that the coarse contour tree is at most half the size of the original tree. For real-world datasets, we even observed a typical reduction by 75% and a maximum reduction by 97%.

5.5.2 The PERMUTE Phase

The PERMUTE phase finds an ordering of branch groups that minimizes the weighted number of edge crossings.

Let $T' = (G, E')$ be the coarse contour tree with G being the set of branch groups and E' the set of edges between groups. Let $c : G \times E' \to \mathbb{R}$ be a function that gives a *weighted* crossing number $c(g, e)$ when the branch group g is crossed at the saddle's isovalue e represents. This weighted crossing number is the sum of all crossed branches' weights. If all weights are 1, this number equals the number of edge crossings. For the weight, we simply use the persistence of a branch, although this can be substituted by other stability measures, e.g. the inverse of the hierarchy depth, the volume, hypervolume, etc. Finally let $\sigma : G \to \mathbb{N}$ denote the (unique) position of a branch group in a permutation and $l(u, v) = \min(\sigma(u), \sigma(v))$ and $h(u, v) = \max(\sigma(u), \sigma(v))$. The PERMUTE phase finds the σ that minimizes:

$$\sum_{\substack{g \in G, e \in E' \\ \sigma(g) \in \\ (l(e), h(e))}} c(g, e).$$

This cost function effectively sums over all crossings that actually take place given a fixed permutation σ. The weights are chosen so that important branches are kept crossing free.

It is not known, whether there is a polynomial time algorithm that solves this minimization problem. We use a random walk combined with simulated annealing to solve it, making this the only non-deterministic phase of our algorithm. We start with a random permutation of the branch groups and refine it iteratively. For each iteration i, our algorithm takes one branch group, reinserts it elsewhere in the ordering, and reevaluates the cost function. Improvements are always kept, and if the

cost deteriorates by Δ, the new configuration is kept with probability $\exp(-\frac{i\Delta}{\tau})$. We experimentally found $\tau = |G|$ to give good convergence. We remember the best configuration found yet and terminate when no improvements over this configuration's cost were observed within the last $|G|$ iterations.

5.5.3 The ORDER Phase

In the ORDER Phase, we extend the ordering of branch groups to an ordering of branches before assigning final horizontal positions.

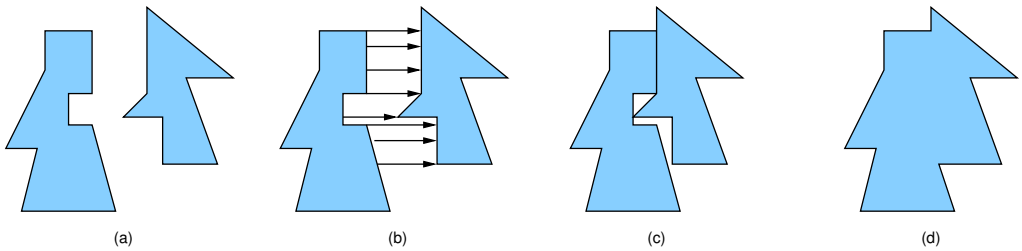


Figure 5.5: Merging two silhouettes into a new one. To avoid overlaps, a partial ordering of points on the silhouette has to be preserved (b). The silhouettes then are fitted together (c) and treated as one (d).

We describe the partial ordering of branches by a DAG. Each branch of the original contour tree is represented as a vertex in the DAG and there is an edge from branch b_1 and b_2 if they have a non-empty overlap in isovalues and

1. they belong to the same branch group and b_2 is a descendant of b_1 in the branch hierarchy, or

2. they belong to different branch groups g_1 and g_2 and $\sigma(g_1) < \sigma(g_2)$, with σ being the result of the PERMUTE phase.

Furthermore, each edge of the DAG can be annotated with a minimum length, which ensures a minimum distance between branches after the POSITION phase. This minimum separation is chosen to avoid overlaps of branches' shapes which show geometric properties, e.g. volume distribution. Let $t_i : \mathbb{R} \to \mathbb{R}^+$ denote the "thickness" of the branch b_i at a given value inside its isovalue range $[w_{min}(b_i), w_{max}(b_i)]$; it is assumed to

be zero outside this range. The minimum separation between branches b_i and b_j is then given by

$$\delta(b_i, b_j) = c + \max_{w \in O} t_i(w) + t_j(w),$$

if b_i and b_j have a non empty overlap O in isovalue and zero otherwise. $c > 0$ is chosen to ensure that all branches are separated even if their thickness is zero.

The DAG can be huge because it contains transitive information. To compute the transitive reduction, we use the silhouette idea of the Reingold-Tilford layout [RT81]. In contrast to this method, our silhouettes do not have a fixed shape. We only track edges that are currently on the silhouette border and successively join silhouettes while updating both the DAG and δ. Initially, each branch has a silhouette with all its edges on both its left and right border and the DAG contains only one vertex for each branch. For all silhouettes S, let LB_S denote the set of edges on the left border of S and RB_S the edges of the right border of S. When two silhouettes u and v are joined to a silhouette w and u should be left of v, then LB_w is LB_u plus all edges of LB_v that are not spanned by the isovalues of the edges of LB_u. We proceed similarly to generate RB_w by adding edges of RB_u to RB_v. Edges are inserted into the transitive-reduct DAG under the same conditions as in the transitive DAG, but only the edges of RB_u and LB_v are considered.

5.5.4 The POSITION Phase

In the POSITION phase, we augment some edges to the partial order DAG and compute a *ranking* to define the x-position of each branch. Note that this phase does the exact same thing as the *Position* phase of *dot* [GKNV93], only on a different graph.

Because of the orthogonal drawing convention, any weighted topological numbering[1] on the DAG and δ computed in the ORDER phase gives a horizontal positioning of elements that avoids overlaps and preserves the number of edge crossing as computed by the PERMUTE phase. However, we observed that child branches are often not close to their parent and horizontal edge segments can be very long. We solve this problem by using the *ranking* and use the same trick as *dot*. Recall that in *dot*'s *ranking* (Section 5.3.1) δ denoted the minimum separation between elements and ω denoted the importance of an edge to be short. First we

[1]See [BETT99] for definition and an algorithm to compute it in linear time. Suffice it so say that it is a simpler problem than *ranking*.

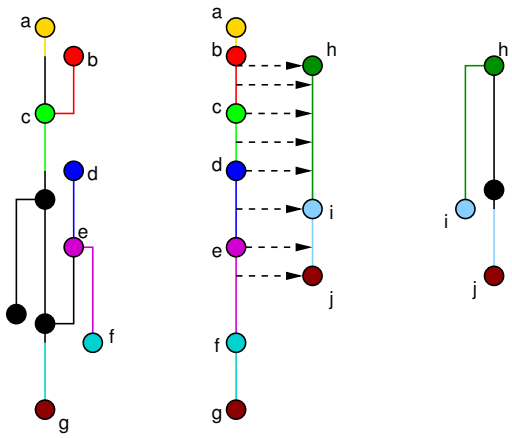


Figure 5.6: Merging in close detail. Shown on the outer are the two trees to be merged and their inner silhouette compressed to a line in the middle. The partial order of silhouette points can be described by a partial order of the graph elements they originate from. Colors and letters indicate that correspondence.

set ω of the original DAG edges (they only represent ordering relations) to 0 so that they have no effect on the length of horizontal edges. Their δ value is as was computed in the ORDER phase. For each horizontal segment, i.e. saddle between branches u, v, we insert a new vertex $e_{u,v}$ into the DAG and edges $(e_{u,v}, u)$ and $(e_{u,v}, v)$. We set δ for these augmented edges to 0 and their weight ω to 1. We then solve the linear program to finally get the x-position of each branch.

5.5.5 Implementation

A straightforward implementation of the algorithm can be very slow. In this section, we present three data structures that help making the elementary operations needed in the three phases of our algorithm faster.

To quickly determine the weighted number of crossings of an isovalue with a branch group, we store for each branch group a sequence of pairs $(isovalue, nCross)$, sorted by isovalue. To find the crossing weight for a given isovalue w, we search the sequence for the largest isovalue lower than w by binary search, and retrieve $nCross$ from the corresponding pair. When branch groups are joined, we apply a kind of merge sort on their pair sequence, at each step adding $nCross$ of both sides.

Since the PERMUTE phase takes most of the run-time, we have opti-

mized it, in particular the evaluation of crossing costs. We observed that many pairs of horizontal lines (saddles) and branch groups can never cross, either because their isovalue ranges do not intersect, or because they are incident. In fact, as the tree becomes larger, each edge tends to have a smaller range of isovalues, and thus the proportion of other edges that it can intersect decreases. We represent all potential intersections in a *conflict graph*. In this bipartite graph, each branch group g_i and edge $e_{j,k} = (g_j, g_k)$ of the coarse contour tree T' is a vertex and $(g_i, e_{j,k})$ is an edge if $i \neq j$ and $i \neq k$ and the isovalue of the saddle between g_j and g_k lies between the minimum and maximum isovalue of the branch group g_i.

Another improvement is based on the observation that many iterations do not change the crossing cost, because the branch group was not moved sufficiently far in the ordering. Here, for any branch group g_i, if the movement changes the number of crossings because of some edge $e_{j,k} = (g_j, g_k)$, then $(g_i, e_{j,k})$ must be an edge in the conflict graph. More precisely, if the order of g_i and all the g_j that have a graph-theoretic distance of 2 in the conflict graph is changed, the global cost changes too. Thus, instead of picking an entirely random new position for g_i, we pick a random new position between the g_j other than the interval that g_i currently resides in. Although this computation takes significantly more time than just randomly picking a new position, it greatly reduces the number of iterations needed.

In the ORDER phase, we describe the silhouettes of branches and branch groups by an alternating list of edges and switch isovalues, ordered by switch isovalues for both the left and right border. The switch isovalues indicate at which isovalue another of an edge (part) appears on the silhouette. Note that because of varying edge lengths, edges may occur multiple times and/or only in part on the border. Joining is done by simple merge-sort-like traversing of list pairs.

The presence of multisaddles and regular vertices in the contour tree do not pose a problem to the *diagonal* and the *orthogonal* algorithm, the layout will even be the same, with the regular vertices sitting peacefully on the lines of the branches they reside on. Regular vertices will increase the run-time in an uncritical fashion, i.e. having them while running the algorithm will be as fast as running the algorithm after removing them.

5.5.6 Extension to Reeb Graphs

Our algorithm can be extended to the more general Reeb graphs. The biggest challenge is that for Reeb graphs there is no proper definition of

branch hierarchy so for the time being one has to compute the branch hierarchy of a "spanning" contour tree. This is similar to the idea of "loop surgery" [TGSP09]. The only changes to our algorithm are trivial. The crossing criterion in the SIMPLIFY phase has to be checked against all Reeb graph saddles. The PERMUTE phase has to deal with more saddles and therefore more horizontal lines and potential crossings. The ORDER phase does not take any saddles into account and therefore stays unchanged. The POSITION phase is essentially the one from *dot* which is already applicable to general graphs and therefore extends naturally. However, our algorithm is tuned for contour trees and there may be more or other desired aesthetic criteria for Reeb graphs.

5.5.7 Resolution Improvement

This phase is an optional preprocessing step that maps isovalues to y-values distorting edge lengths (and persistence). It preserves the relative ordering of vertices and distributes them more evenly in the y-direction.

We can approximate the density of isovalues by letting each value be the source of a small Gaussian kernel with fixed standard-deviation σ. Let $d(x)$ be the density function:

$$d(x) = \frac{1}{|V|} \sum_{v \in V} \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-f(v))^2}{2\sigma^2}}$$

Clearly $d(x)$ is high in populated regions and low elsewhere. The same is true for the steepness of the integral:

$$D(X) = \int_{-\infty}^{X} d(x)dx$$

We observed that using $y = D(f(v))$ rather than $y = c \cdot f(v)$ distributes the vertices more evenly, improving on the resolution in dense areas and attracting outliers. Furthermore, we noticed, when all values are pairwise different and σ approaches ∞, $D(f(v))$ becomes the sorting number of $f(v)$.

Unfortunately, integrating Gaussians cannot be done analytically, so we use the logistic function to approximate:

$$D(x) = \frac{1}{|V|} \sum_{v \in V} \frac{1}{1 + e^{(f(v)-x)\sigma^{-1}}}$$

We determined $\sigma = (max_{v \in V} f(v) - min_{v \in V} f(v)) / \sqrt{|V|}$ empirically to give good results.

5.5.8 The *Smooth Movement* property

As the *orthogonal* algorithm only lacks the *smooth movement* property, we investigated how an algorithm would look like that achieves it. In fact, it is quite simple to construct an algorithm that respects the *smooth movement*, *crossing line*, and *branch stability*: the *diagonal* algorithm.

Given the branch hierarchy of a contour tree, the *diagonal* algorithm simply draws the root branch as a diagonal line from (y_{min}, y_{min}) to (y_{max}, y_{max}) and proceeds recursively on its child branches, each one drawing themselves orthogonally and incident to their parent branch at the saddles y-value that connects them. Figure 5.7f shows an example of such a drawing. This method is deterministic and can be implemented to run in linear time. Furthermore, it has the interesting property of *dynamic stability*: adding a new child branch does not change the layout of the other ones, thereby ensuring the *branch stability* property without effort. The method, however, suffers from many edge crossings, often making the drawing unreadable.

It is very difficult to combine *smooth movement* and *crossing minimization*. Furthermore, the *smooth movement* at one point requires that branches are not drawn as vertical lines. In this case, the silhouette showing varying contour size should be skewed rather than rotated in order not to distort information. But as this skewing gets stronger, the viewer gets a more distorted impression of the actual contour sizes, i.e. the branch appears thinner. Although branch attributes may be represented in another fashion, it indicates that *smooth movement* may not be combined with the *geometric property* aesthetic criterion.

5.6 Results

We implemented the *diagonal* and *orthogonal* algorithms in C++ and tested them on several simulated flow fields. These datasets were the pressure field of the flow inside a gas furnace chamber, and the pressure and velocity field of the unsteady flow around a cuboid. The sizes of the datasets are 32440 and 100^3 points, respectively. Computing the contour tree took less than 10 seconds in each case, and the contour trees have 788, 6665 and 1493 branches, respectively. The branch hierarchy was created and topologically simplified using persistence as measure of stability. The Sugiyama layouts were generated using *yFiles* [WEK01] which is a Java program. The *dot* layouts were generated with *graphviz* [GN00]. We use an own implementation of the *HDE-based* and *axis stress*

methods in C++ that build on the GNU Scientific Library [Gal].

Figure 5.7 and Figure 5.8 give example drawings of the gas furnace chamber's contour trees with the different methods presented. Figure 5.8 shows layouts of rather large contour trees and aims at qualitative evaluation of the algorithms. The irregular and overlapping subtrees of the *HDE-based* layout in Figure 5.8a renders this method unsuitable even in a system providing zooming capabilities. Although the regular look of the Sugiyama layout in Figure 5.8b is rather pleasing, it lacks to show which branches are important and in fact which edges constitute branches. The *orthogonal* layout of Figure 5.8c precisely shows the branching, and the varying contour sizes indicate important branches and areas of exploration. It suffers a bit from long horizontal edges, but generally unimportant structures are drawn very small.

Figure 5.9 shows the influence of showing geometric properties and using resolution improvement with the simplified contour tree of the gas furnace chamber data set. Notice how contour size rapidly increases in the vicinity of saddles and how this attracts attention towards further exploration. Figure 5.10 shows contour trees for the pressure and velocity of the cuboid dataset. Note the zig-zag of branches in Figure 5.10a and the long horizontal edges of Figure 5.10b, that show where the algorithm could be improved further.

Table 5.1 gives run-times and the number of edge crossings in the final drawing as a simple measure of quality. We chose edge crossings as it has been verified to be the measure of most importance on human understanding [Pur97]. All measurements took place on one core of a 1.8GHz dual-core Opteron. Run-times are an average of 10 runs with the exception of the *dot* layout for the unsimplified gas furnace chamber contour tree where the first run required more than 7 days and was aborted. The 214 and 1556 layer *dot* layouts refer to the exact layering, i.e. one isovalue per layer, whereas the other *dot* layouts were generated using fixed-step isovalue quantization (Section 5.3.1). Because the orthogonal algorithm is non-deterministic, we present the minimum, median, and maximum number of crossings in the resulting images.

The *orthogonal* algorithm is able to find a crossing free layout for the contours trees of a couple of hundreds of vertices and is generally faster than *dot* unless *dot* uses very few layers. The run-time of the *orthogonal* algorithm for the large contour trees is dominated by the PERMUTE phase. It does not scale properly with input size because of the varying success of the SIMPLIFY phase.

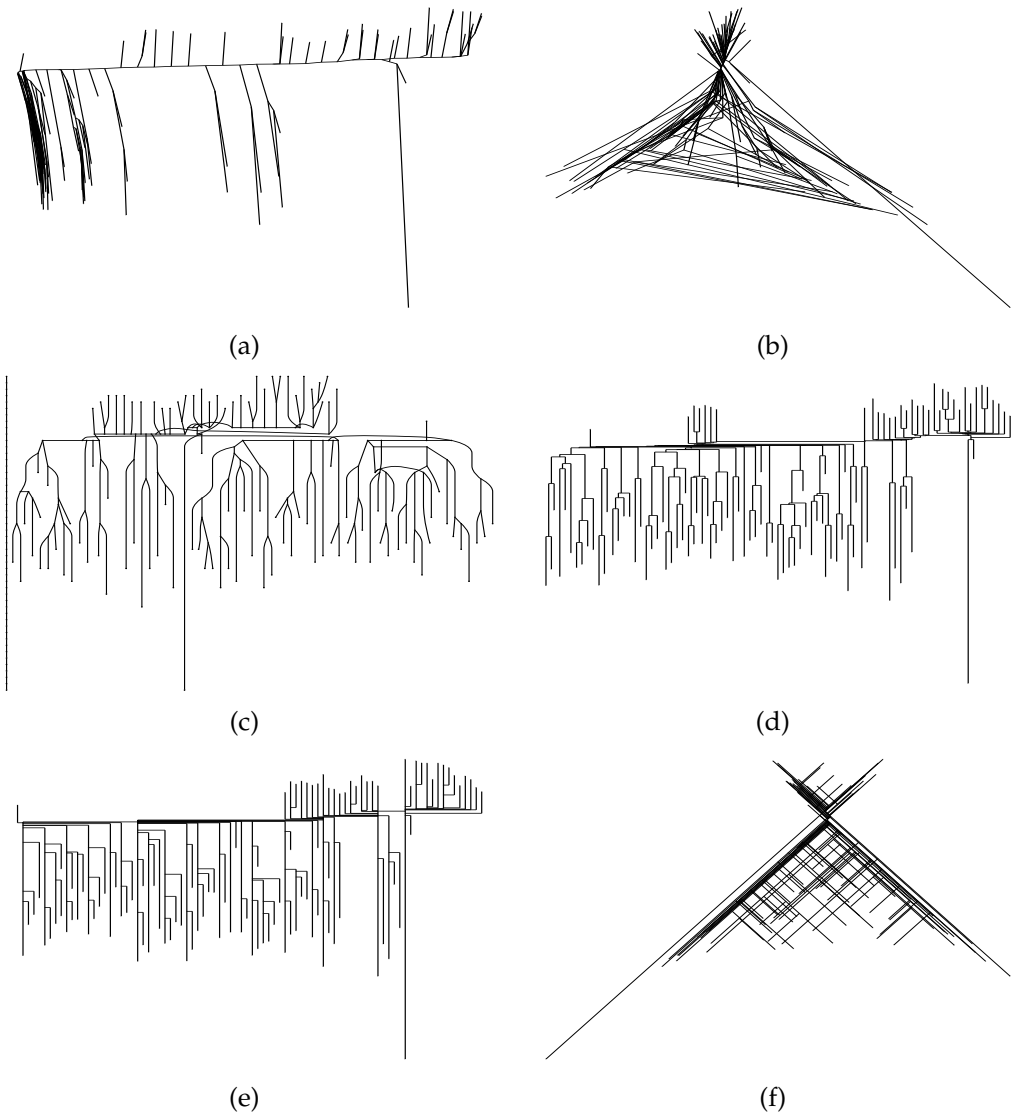
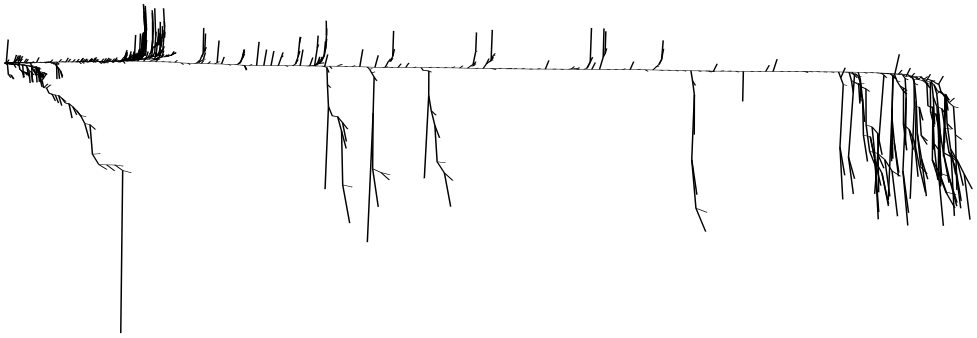
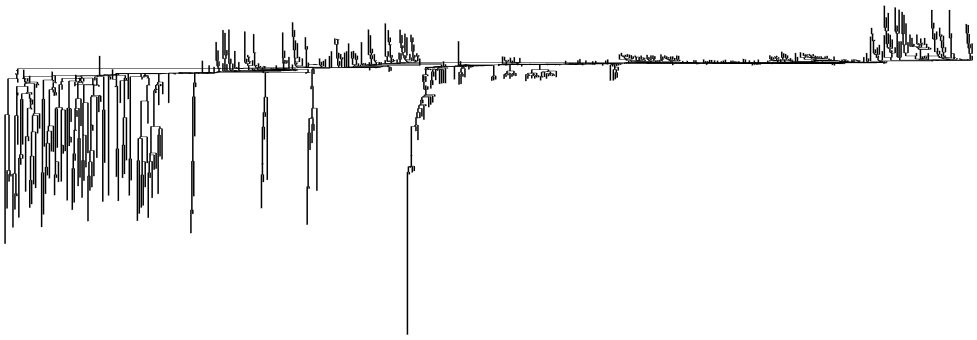


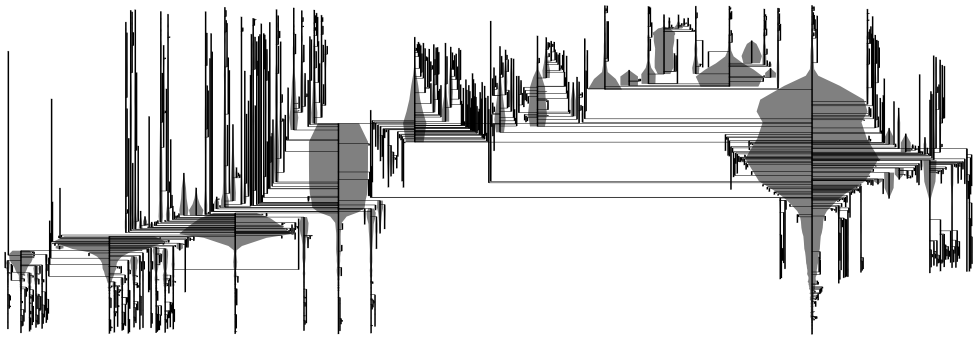
Figure 5.7: Drawings for the contour tree of the flow inside a gas furnace chamber topologically simplified to 107 branches. (a) *HDE-based* layout, note the "wrapping" at the border resulting in overlaps, (b) *axis-stress* layout, stuck in a local minimum, (c) *dot* layout using 50 layers suffering from some edge crossings, isovalues are not correct, (d) *Sugiyama* layout, 2 edge crossings, branch hierarchy is hard to discern, (e) *orthogonal* layout showing branch hierarchy without any edge crossings, contour size not shown and no resolution improvement occurred, (f) *diagonal* layout showing branch hierarchy and allowing *smooth movement* but suffering from lots of edge crossings.



(a)



(b)



(c)

Figure 5.8: Drawings for unsimplified contour tree of the flow inside a gas furnace chamber (788 branches). (a) *HDE-based* layout suffering from many overlaps. (b) *Sugiyama* layout, 375 edge crossings, branch hierarchy is hard to discern. (c) *orthogonal* layout, 58 edge crossings, using resolution improvement, showing varying contour size, and branch hierarchy. Note how branches of high volume attract attention and topological noise is drawn very small.

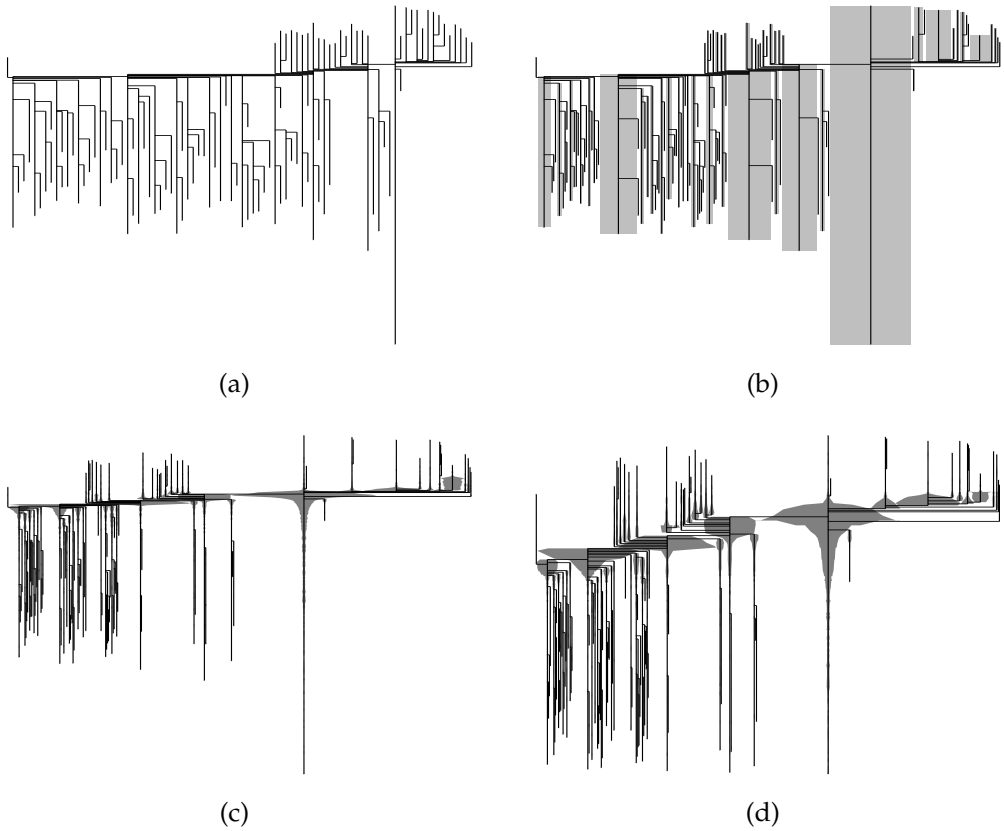
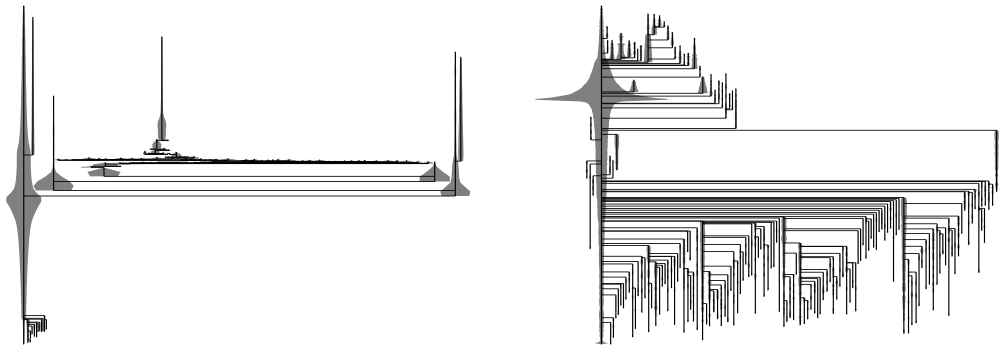


Figure 5.9: *orthogonal* drawings for the contour tree of a gas furnace chamber, 107 branches. (a) No volume shown. (b) average volume is drawn around each branch. (c) varying contour size is drawn around each branch. (d) same as (c) but with resolution improvement.

5.7 Summary

We identified five aesthetic criteria for drawing contour trees in the plane. These are: fixing isovalue, emphasizing branch hierarchy, minimizing edge crossings, avoiding sharp bending edges, and displaying geometric properties. We also presented two algorithms conforming to three and four of these aesthetic criteria.

The *orthogonal* algorithm is very efficient for drawing contour trees of up to 200 vertices and computes readable layouts on sizes up to 2000 vertices in a reasonable time (up to 10 minutes). For even larger trees it is still possible to perform the topological simplification first, before



(a) (b)

Figure 5.10: *orthogonal* drawings for contour trees of a flow around a cuboid. Both trees have been topologically simplified and the varying contour size is shown. The drawings also use resolution improvement. (a) pressure, 153 branches, 1.74s, 7 crossings. (b) velocity, 163 branches, 0.32s, 0 crossings.

running the algorithm. The images of the large contour trees are still hard to read as a whole, but we note that this algorithm preserves the branch hierarchy, and therefore it is now possible to provide a layout of the more important branches as an overview and show the fine structures on demand (e.g. when zooming in) without having to recompute the layout and therefore preserving the mental map of the image.

In the future, we intend to investigate the challenges of showing multiple additional branch properties using the *orthogonal* algorithm. We also intend to investigate methods to detect subsets of orthogonal edges that can be replaced by smooth edges.

method name (Fig.)	**layout time**	**number of crossings**
gas furnace chamber, 107 *branches*		
HDE (5.7a)	0.136s	143
axis stress (5.7b)	1.05s	625
dot, 50 layers (5.7c)	0.464s	34*
dot, 100 layers	2.16s	12*
dot, 214 layers	22.7s	0*
Sugiyama (5.7d)	5.88s	2
orthogonal (5.7e)	0.948s	$0, 0, 0$
diagonal (5.7f)	0.108s	438
gas furnace chamber, 788 *branches*		
HDE (5.8a)	3.48s	1025
axis stress	57.4s	5071
dot, 256 layers	150s	831*
dot, 512 layers	4573s	401*
dot, 1556 layers	$> 7days$	
Sugiyama (5.8b)	158s	375
orthogonal (5.8c)	322s	$58, 66, 116$
diagonal	0.288s	4566

Table 5.1: Results for run-time and number of edge crossing for various algorithms on the gas furnace chamber dataset. (*) Because spline-spline crossings are hard to compute, we approximated their number by polyline crossings to evaluate *dot*.

6 Visualization of Barrier-Tree Sequences

Dynamical models that explain the formation of spatial structures of RNA molecules have reached a complexity that requires novel visualization methods that help to analyze the validity of these models. Here, we focus on the visualization of so-called folding landscapes of a growing RNA molecule. Folding landscapes describe the energy of a molecule as a function of its spatial configuration; thus they are huge and high-dimensional. Their most salient features, however, are encapsulated by their so-called barrier tree that reflects the local minima and their connecting saddle points. For each length of the growing RNA chain there exists a folding landscape. We visualize the sequence of folding landscapes by an animation of the corresponding barrier trees.

The animation is created by an adaption of the foresight-layout-with-tolerance algorithm for dynamic graph layout problems. Since it is very general, the main ideas for the adaption are presented: construction and layout of a supergraph, and how to build the final animation from its layout. We present some new heuristics that improve the readability of the final animation.

6.1 Biological Background

Ribonucleic acid (RNA) is a linear biopolymer, i.e. a chain of covalently connected units (nucleotides) of which there are four types: adenine (A), guanine (G), cytosine (C), and uracil (U). RNA molecules play an important role in many biological contexts, e.g. protein synthesis. The biological function of an RNA molecule is determined predominantly by its spatial structure which in turn is determined by the sequence of nu-

cleotides. When an RNA molecule is produced in the cell, it folds back to form double helical regions consisting of paired nucleotides. The list of helices or (equivalently) of base pairs is known as the *secondary structure* of the RNA molecule. Since helices stabilize the structure while the intervening unpaired loops are destabilizing, each secondary structure can be assigned a free energy equivalent to the energy released when the molecule folds. To a large extent, the secondary structure already determines the function of RNA.

Various methods have been proposed to explain and predict the structures of RNA molecules. Typically, one considers the structure with the lowest free energy, i.e. the one for which the folding process that starts from the completely unfolded state releases the maximum amount of energy. This structure is the most stable one, and according to the laws of statistical mechanics, the one that is most frequently attained in thermodynamic equilibrium. The folding process itself can, however, take a long time so that the equilibrium state that will be reached after an infinite waiting time may not be biologically relevant. Instead, the folding process may pause in metastable structures from which it is hard to escape due to high energy barriers. The folding process of an RNA molecule can be modeled as a Markov process whose states are the individual secondary structures [CHS96]. Transitions are allowed only between "neighboring configurations", i.e. those that differ by only one base pair [FFHS00], and transition rates are proportional to $\exp(\Delta E/RT)$, where ΔE is the difference in energy, T is the ambient temperature, and R is a constant. In practice, however, the transition matrix is much too large to solve the resulting master equation directly.

A refined model transforms the configuration space into a large graph, whose vertices are secondary structures and whose edges connect neighboring structures. The neighbor graph along with the energy specific to each configuration can be imagined as a discrete energy landscape. A folding or refolding process can then be described by a path in the graph or a walk in the energy landscape. For each such path there exists one structure of maximal free energy, the *maximum* of the path. The *barrier* between two configurations is the smallest maximum of all paths between the two configurations. If a structure refolds, it has to overcome at least this energy barrier. These barriers partition the graph into "basins" that are centered around local energy minima (secondary structures of which all neighbors are less stable). An approximate model is now obtained by considering the basins as effective states of the RNA molecules. Transition rates between basins can be derived from the more detailed model under the assumption that the folding process is nearly equilibrated lo-

cally within each basin [WSSF+04].

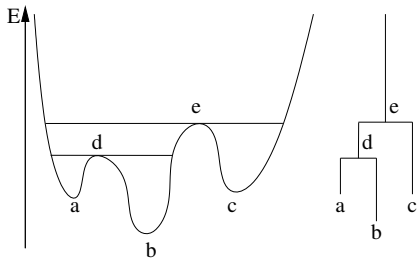


Figure 6.1: A very simple landscape and barrier tree. In contrast to normal trees, each vertex of a barrier is drawn at a height that reflects the free energy of the folding configuration it represents. To determine the energy barrier between two local minima, one has to find the barrier-tree vertex that has both leaves representing the local minima as descendants and the greatest topological distance to the root of the tree.

The relevant information can now be stored in the so-called *barrier tree* T of the landscape. The leaves of T correspond to the local minima of the energy landscape together with their basins of attraction, while inner vertices represent the barriers (also called saddle-points) between the basins. Figure 6.1 shows an example of a barrier tree for a very simple landscape. This example is just for illustrative purposes; we consider mainly landscapes where individual points do have a high and varying number of neighbors, making the landscape a high-dimensional object. Barrier trees are constructed by successively "flooding" the basins of the landscape. A barrier is found at the point where the lakes of two basins would join. These two joined basins are considered to be one when the "flooding" is continued. See Flamm *et al.* [FHSW02] for a detailed description.

In reality, however, RNA molecules are not "born" as a whole. Rather, they are "transcribed" nucleotide by nucleotide from their DNA template, so that the molecule is still growing while it already starts to fold [MM04]. The structures that are formed are thus dependent upon the relative rates of folding and transcription. Similar effects are observed when an RNA molecule travels through a narrow pore, where it must unfold on one side and refold on the other [GBH04]. Again the kinetics of folding is coupled to the speed with which the molecule is pulled through the pore. Instead of single static energy landscape, we thus have to deal with a situation where the energy landscape, and hence the rules

of folding, changes with each step of the second dynamical process. Since the latter proceeds in small steps, it only causes moderate changes in the energy landscape. Thus, there is a natural correspondence between a local energy minimum x before and a (unique) local minimum x' after a step of the second dynamics: Structure x is modified to some structure x^* i.e. by appending a single unpaired nucleotide. Then x^* relaxes to the local minimum x' to whose basin it belongs to. Note that multiple local minima can map to the same local minimum in the next step, and that local minima might arise that are not mapped from any local minimum of the previous step.

From the biophysical point of view, the problem is thus to understand the dynamics of folding combined with another process such as transcription or pore traversal. As in the static case, this can be done by approximating the folding energy landscape at each step by its barrier tree. The second dynamics is then represented by transitions between corresponding local minima. While the folding process in the static case is relatively easy to interpreted as a movement on the barrier tree, we now have to consider a movement on a series of barrier trees whose vertices are connected in a specific way.

In numerical simulations, one observes, that for some RNAs the fraction of folding trajectories that reach the ground state of a certain fully grown chain depends in a non-trivial way on the relative speed of transcription. Both for very slow and very fast transcription the molecule reaches the ground state quickly, while in an intermediate regime most of the trajectories become trapped in a metastable, very different, secondary structure. In order to understand this phenomenon it is necessary to compare the trajectories in the barrier-tree sequence and to pinpoint the step(s) in which escape from local minima occurs at the same time scales as chain elongation. The same type of questions naturally arise in other settings where the folding energy changes, i.e., whenever the temperature or salt concentration changes.

6.2 Visualization Problem

Without an appropriate visualization tool it is virtually impossible to find the time steps and transition at which timescale differences have a drastic effect, as there is little or no *a priori* coherence between the layouts of the individual barrier trees in a series. It is thus very tedious to actually follow a trajectory through a series and to determine the likely transitions. The mapping of local minima, however provides information that, as we

shall see, can be utilized to enhance the coherence of adjacent trees in a series.

The barrier trees thus share common information that should be presented accordingly, i.e. it should not attract more attention than the parts that differ. Instead of visualizing a sequence of barrier trees that have some redundancy, one can also say that there is just one barrier tree that changes with time in a way that the barrier trees of the sequence are snapshots of the dynamic tree at certain points of time. In this work, we will thus view this problem as a dynamic graph drawing problem. As an abstraction, we define the problem as follows: *Given a sequence of barrier trees and leaf mappings, where leaves of one tree are mapped on leaves of the following tree, determine the layout of all trees such that in a presentation the mental map is retained.*

To solve the visualization problem, our algorithm is split in several parts, which we will describe in sections Section 6.4 to Section 6.6. Given the barrier trees $T_i = (V_i, E_i, e_i)$, ($e_i : V_i \to R$ is a function that gives the energy of each vertex) and the leaf mappings $f_i : V_{i-1} \to V_i$ between them, we first find equivalent vertices. These vertices are then arranged in an order that minimizes an objective function which is mainly determined by the number of visible edge crossings for the whole sequence at presentation time. We use simulated annealing to determine this order. Given this order, we directly derive the layout of the single trees that make up the barrier-tree sequence and present them in an animation with transitions that help to communicate the changes.

6.3 Related Work on Dynamic Graph Drawing

The first attempts toward dynamic graph drawing were very specific. Moen [Moe90] presents an algorithm that shows a part of an ordered tree. Although the tree itself stays the same, the selected subset may change through replacement of subtrees by leaves and vice versa. Cohen *et al.* [CBT+92] gives detailed algorithms and data structures for a number of dynamic graph classes. These allow visualizing popular data structures, e.g. AVL trees, and adjusting the layout of a graph, if it is being edited or browsed. Both approaches share a common motivation: they reduce the computation time of the layout by reusing information about the previous layout. This has the side effect of making the layout of the changed graph similar to the unchanged, but accumulation of many elementary changes can result in an aesthetically unpleasing drawing.

North [Nor95] measures the quality of an algorithm to make good

dynamic drawings based on *incremental* or *dynamic stability*, i.e. the property of an algorithm to produce very similar layouts for graphs that differ only slightly. He applies his concepts to the drawing of dynamic directed acyclic graphs. Misue *et al.* [MELS95] introduce the concept of *mental distance*. It formally describes the difference of two layouts and can be used to measure the perceived stability of a dynamic graph layout. They define the aesthetic criterion "preserving the mental map" for any dynamic graph drawing problem, and refine it to three models. In the *orthogonal ordering* the left-to-right, and up-down order of vertices stays the same. *Proximity relations* are preserved, if the relative distances of vertices and edges do not change. The *topology* is preserved, if vertices and groups of vertices of one region stay in that region. The *mental distance* of two layouts is the number of times or the amount by which a rule is broken. Frishman and Tal [FT04] present an algorithm that draws dynamic clustered general graphs using an incremental force-directed method. Their algorithm generally preserves the mental map by reusing the earlier layout, but improves the layout slightly, if a static graph drawing aesthetic criteria is not met any more. They recently generalized their method to unclustered graphs ([FT07b]).

If the layout process cannot be formulated to minimize the mental distance between successive layouts, a local transition or morphing of the layouts has to take place. Friedrich and Eades [FE02] describe a method to make sure that the transition preserves the mental map. To do that, an affine transformation that registers both layouts is determined and performed. Using a force-directed approach, vertices are moved to their final positions while avoiding occlusions and other visual artifacts linear interpolation would bring forth. Fortunately, our algorithm produces layouts that are stable enough not to require these forms of transition.

Erten *et al.* [EHK$^+$03] describe a method to layout general dynamic graphs using a force-directed method. Vertices of the evolving graph that are equivalent are connected by virtual springs that contract in the force-directed method. As a result, vertices referring to the same instance at different times are positioned closely together. This ensures a good stability of the dynamic layout. We do not use this general approach, because we feel that the final animation should at least resemble the look and feel of barrier trees.

Diehl and Görg [DG02] propose a general scheme to layout dynamic graphs when all graphs of the sequence are known prior to layout creation. This scheme is independent of the class of the graphs and the layout algorithm used. Their *foresight-layout-with-tolerance* algorithm makes a trade-off between static and dynamic graph drawing aesthetic crite-

ria based on a tolerance parameter. In a first phase a *supergraph* is constructed that contains all graphs of the sequence as subgraphs. Then the layout of this (static) supergraph is determined and used as a blueprint for the layout of the subgraphs. The layout of the subgraphs can be further improved with respect to static graph drawing aesthetic criteria, but its mental distance may not differ by more than the tolerance parameter from the blueprint layout. Presentation of the sequence can be done using morphing geometry information between the single subgraphs. Görg *et al.* [GBPD04] further improve the scheme with the notion of the *importance* of a vertex or edge. This importance is a measure for the number of times a vertex or edge is present in the graph sequence and is used to improve the visual quality of the layouts.

A similar idea is presented by Gaertler and Wagner [GW05]. Instead of an animation, a $2\frac{1}{2}$D visualization, i.e. a 3D view of a stack of static 2D layouts – each showing the graph at a certain point of time – is generated. Brandes *et al.* [BDS03] also use $2\frac{1}{2}$D visualization to show a set of similar metabolic pathways. They create the layouts of the acyclic directed graphs representing the pathways using a layout of an union of all graphs, and also determine the optimal ordering of layouts. Both approaches share the notion of the supergraph, local adjustments like in the foresight-layout-with-tolerance algorithm are not performed. Dwyer and Schreiber [DS04] also use $2\frac{1}{2}$D to visualize a set of similar phylogenetic trees. Phylogenetic trees are very similar in structure to barrier trees. In contrast to the other two approaches instead of a supergraph only a *minimal leaf ordering* is determined. This neglects the identification of equivalent inner vertices, which becomes necessary, if transitions are to be shown between key frames. It also requires each inner vertex to have exactly two children, a property which barrier trees do not have in general.

In this chapter we adapted the foresight-layout-with-tolerance algorithm. Since it is very general, we optimized each of the phases to fit our dynamic barrier-tree application.

The layouts of the subgraphs that is generated from the supergraph layout can also be used in a $2\frac{1}{2}$D visualization. However, we found this to be inappropriate, because the barrier-tree sequences under consideration were highly dynamic. In our datasets we observed that almost any tree at time t has nearly nothing in common with the tree at time $t + 5$. A $2\frac{1}{2}$D visualization would therefore exhibit much visual clutter. Also, the energy of a vertex, and thus its vertical position, can change between subgraphs. In a $2\frac{1}{2}$D visualization one would have to indicate such events with visual links between slices; we found it more natural to indicate that

in an animation with a movement of the vertex. In general, we think that the animation of transitions between subgraph layouts can be efficiently used to communicate the changes the barrier-tree topology to the user.

6.4 Supergraph Construction

In the following, $path_G(u, v)$ shall be true, if and only if there exists a directed path in G starting at u and ending at v. $odeg_G(v)$ denotes the number of edges of G, whose tail is v. $T_i = (V_i, E_i)$ is a rooted tree and also a directed acyclic graph, where all edges are oriented to point away from the root toward the leaves. Note that each leaf v satisfies $odeg_{T_i}(v) = 0$. L_i denotes the set of leaves of the tree T_i and F_i an arbitrary subset of L_i. $L_G(v)$ is the set of all vertices w that satisfy both $path_G(v, w)$ and $odeg_G(w) = 0$. In a tree, these vertices are leaves, in a directed acyclic graph, they are sinks. Thus $L_G(v)$ assigns the set of leaves/ sinks that can be reached from v to each vertex v. 2^M denotes the set of all subsets of M.

6.4.1 Problem Definition

The problem of the supergraph of a sequence of trees with leaf mappings is: given a sequence of rooted trees $T_0, \ldots, T_n$ with

$$\forall 0 \leq i, j \leq n : \big(i \neq j \rightarrow V_i \cap V_j = \emptyset\big)$$

and

$$\forall 0 \leq i \leq n : \forall v \in V_i : odeg_{T_i}(v) \neq 1$$

and a sequence of leaf mappings $f_1, \ldots, f_n$ with $f_i : F_{i-1} \rightarrow L_i$, find the smallest graph $G = (V, E)$ and a global mapping of tree vertices on supergraph vertices $k = \bigcup_{i=0}^{n} k_i$, $k_i : V_i \rightarrow V$, k_i injective, such that

1. G contains all trees:

   $$\forall 0 \leq i \leq n : (k_i(V_i) \subseteq G \wedge \forall (u, v) \in E_i : path_G(k_i(u), k_i(v)))$$

   and each path from u to v does not visit vertices from $k_i(V_i)$ except u and v.

2. G conforms to the leaf mapping:

   $$\forall 1 \leq i \leq n : \forall u, v \in V_{i-1} : (f_i(u) \neq f_i(v) \rightarrow k_i(f_i(u)) \neq k_i(f_i(v)))$$

3. G conforms to the topological properties of all trees:

   $$\forall 0 \leq i \leq n : \forall u, v \in V_i : \neg\, path_{T_i}(u, v) \rightarrow \neg\, path_G(k_i(u), k_i(v))$$

6.4.2 Motivation

The first step of the foresight-layout-with-tolerance algorithm [DG02] is to construct a supergraph of all the graphs in a sequence. The supergraph is the smallest graph that contains all graphs of the sequence as subgraphs. To accomplish this, it is necessary to know which vertices of the graphs should be considered equivalent. Leaf mappings between successive trees are used as a base for this process, however, this can only be applied directly to some of the leaves of the trees. The identification of equivalent inner vertices and leaves that result from merging leaves in the previous tree is non-trivial. We did not motivate this identification by graph-theoretic minimization, but decided that the supergraph should reflect properties of the corresponding landscapes. This has the advantage, that the supergraph may be used as an alternative and static representation of the barrier-tree sequence.

A barrier tree not only stores energy barriers between local minima, it also gives a rough and abstract view on the topology of a landscape. The shape of the barrier tree illustrates the order of the unification of basins. This unification order will be used to identify equivalent inner vertices. If, for instance, an inner vertex u has two leaves as its children that are mapped to two different leaves of the following tree having the same parent v, the inner vertex u and the parent v can be seen as topologically equivalent. If the leaf mapping is extended by this new information, further parts of the trees can be processed to further identify inner vertices as equivalent, and to quickly identify isomorphic structures between the barrier trees that conform to the leaf mapping. This takes only the topology of the barrier tree into account. The energy information about each vertex is neglected.

This procedure ends abruptly, as soon as there is the slightest topological difference in a barrier tree. In practice, this strict behavior results in a large number of vertices that are not considered to be equivalent. This can be avoided by identifying equivalent inner vertices based on the set of local minima that can be reached from the corresponding barrier by descending in the landscape. In Figure 6.2a, vertex e and j are considered to be equivalent, because the sets of leaves that can be reached from them are equal considering the leaf mapping. Vertices d and i are not considered to be equivalent because the set of leaves that can be reached from them, $\{a, b\}$ and $\{g, h\}$ respectively, are not equal considering the leaf mapping. Such cases are very common and are generated mostly, when the height of barriers between successive trees change. The supergraph is in that case no longer a tree, but a directed acyclic graph (DAG). This

is unavoidable, but the supergraph will always be at most a DAG.

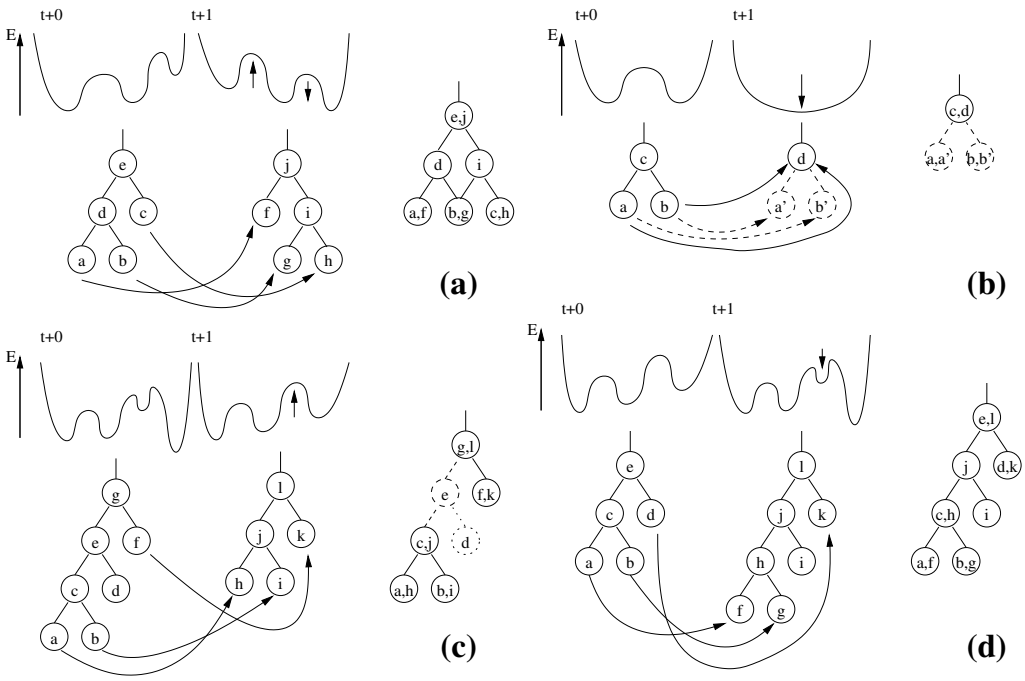


Figure 6.2: Examples of elementary landscape and barrier-tree changes. Each figure shows, how the energy landscape changes, illustrates the barrier trees (only the topology is shown) and the leaf mappings and shows, how the supergraph should look like in the cases: barrier swap (a), leaf merging (b), leaf vanishing (c), leaf creation (d).

Imagine, that the barrier swap from Figure 6.2a is reverted at time $t + 2$. The tree at time $t + 2$ conveys exactly the same information as the tree at time $t + 0$. It contains an inner vertex that is not equivalent to any vertex of tree $t + 1$, but equivalent to vertex d. This vertex should not be inserted in the supergraph, as it does not represent "new" information. But this fact cannot be concluded by looking at tree $t + 1$ alone. Considering all past trees can get quite complicated, it is much easier to just look into the supergraph for the past trees. The supergraph can and will be used as a data structure to quickly identify equivalent inner vertices of the barrier trees. It is efficient to construct the supergraph iteratively. To determine the supergraph for the trees T_0 to T_{n+1}, we use the supergraph of the trees T_0 to T_n for identification of equal vertices and add any new information we gain from tree T_{n+1}.

Figure 6.2b shows another common case of change in the energy landscape. Often barriers disappear, and local minima get merged. Obviously our "set of leaves" approach fails in this case, the vertices c and d would not be considered equivalent ($\{a, b\}$ vs. $\{d\}$). The solution is to temporarily add the mirror vertices a' and b' as children to d and modify the leaf mapping. This methodology is a must, if more than two leaves merge or the merging leaves do not share the same parent. Merged leaves must be marked as inactive in the supergraph, so they will not be considered for the "set of leaves" of other inner vertices.

In Figure 6.2c a leaf vanishes, i.e. it is not part of the leaf mapping. This may happen, because the number of leaves is usually reduced to the most relevant ones, and a relevant leaf may have a non-relevant successor. In such a case the leaf (d) is marked as inactive and is not considered for the set of leaves. This leaves us with the problem, that the vertices c and e of tree $t + 0$ have the same set of leaves ($\{a, b\}$), and thus vertex j is considered equivalent to both vertices. In that case, the vertex farthest from the root (c) is selected. What becomes apparent now is, that the tree $t + 1$ is not really a subgraph of the supergraph, because it lacks an edge from g, i to c, j. The supergraph is still an expansion of tree $t + 1$.

In Figure 6.2d a leaf is added to the tree. This is the inverse of the previous case. The edge from e to c is replaced by a path (i, j, h) and the new leaf is added at the appropriate location. Again the supergraph is an expansion of tree $t + 0$. The removal of transitive edges has little to no effect on the quality of the final presentation, but reduces the size of the supergraph and greatly improves the performance, when the layout of the supergraph is determined.

These four operations are considered elementary and are the only operations we observed in our datasets. However, it is expected that multiple elementary operations take place between successive trees of the sequence. Because creation, deletion, and merging cannot happen to the same leaf of a tree simultaneously, these operations and the supergraph modifications they imply do not affect each other. Also creation, deletion, and merging happen at or near the leaves, while *barrier swaps* only add inner vertices. So these operations also do not affect each other and can be done separately.

6.4.3 Construction

For each directed graph $G = (V, E)$ define the function $mark_G$ as:

$$mark_G : 2^V \rightarrow 2^V$$

$$M \mapsto \left\{ v \mid L_G(v) \subseteq \bigcup_{u \in M} L_G(u) \right\}$$

The operation of this function may be described as this: Starting from the vertices of M, all incoming edges are marked. If all outgoing edges of a vertex get marked in that process, that vertex is added to M and the process continues. The process ends, if no more vertices can be added to M. Figure 6.3 illustrates this. Obviously $M \subseteq mark_G(M)$ and $M = \emptyset$, if and only if $mark_G(M) = \emptyset$. Unlike the example, M does not have to contain leaves/ sinks only.

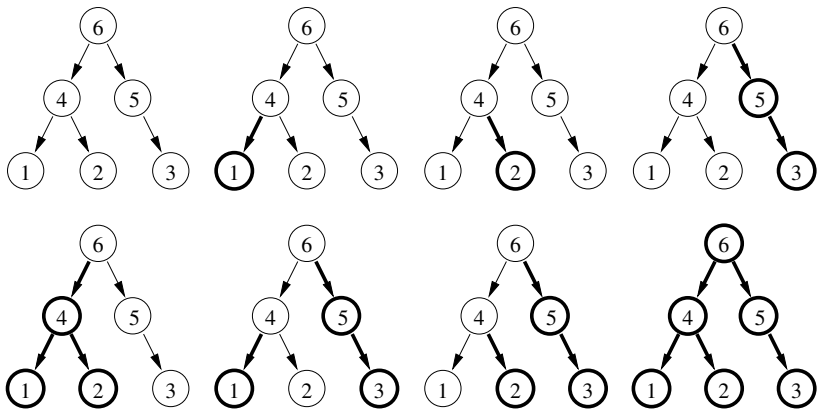


Figure 6.3: Example for the $mark_G$ function The result is illustrated by vertices with thick circles. top left to right: $mark_G(\emptyset)$, $mark_G(\{1\})$, $mark_G(\{2\})$, $mark_G(\{3\})$. bottom left to right: $mark_G(\{1,2\})$, $mark_G(\{1,3\})$, $mark_G(\{2,3\})$, $mark_G(\{1,2,3\})$.

The function $match_G$ reduces a mark to the topmost layer:

$$\begin{aligned} match_G : 2^V &\rightarrow 2^V \\ M &\mapsto mark_G(M) \cap \{v \mid \forall (u,v) \in E : u \notin mark_G(M)\} \end{aligned}$$

For the example in Figure 6.3: $match_G(\emptyset) = \emptyset$, $match_G(\{1\}) = \{1\}$, $match_G(\{3\}) = \{5\}$, $match_G(\{1,2\}) = \{4\}$, $match_G(\{2,3\}) = \{2,5\}$, $match_G(\{1,2,3\}) = \{6\}$.

Construct G iteratively: $G_0 = T_0$, $\forall v \in V_0 : k_0(v) = v$. Construct $G_i = (V'_i, E'_i)$ and $k_i : V_i \rightarrow V'_i$ from $G_{i-1} = (V'_{i-1}, E'_{i-1})$, $k_{i-1} : V_{i-1} \rightarrow V'_{i-1}$, $T_i = (V_i, E_i)$ and f_i as follows:

Determine the active part of the supergraph G_{i-1}, this is much easier than tracking inactive (deleted or merged) parts of the supergraph:

$$G'_i = (A_i, K_i)$$

$$A_i = \left\{ v |\, v \in V'_{i-1} \wedge \exists l \in L_i : path_{G_{i-1}}(v, k_{i-1}(l)) \right\}$$

$$K_i = E'_{i-1} \cap A_i \times A_i$$

For each vertex of the tree T_i determine the set of leaves of T_i that can be reached from that vertex:

$$\begin{aligned} M_i : V_i &\rightarrow 2^{L_i} \\ u &\mapsto \left\{ v |\, v \in L_i \wedge path_{T_i}(u, v) \right\} \end{aligned}$$

For each vertex of the tree determine its *leaf set*, i.e. the set of vertices of the active part of the supergraph, that map on a leaf in M_i because of the leaf mapping:

$$\begin{aligned} B_i : V_i &\rightarrow 2^{V_i} \\ v &\mapsto \{ k_{i-1}(w) |\, f_i(w) \in M_i(v) \} \end{aligned}$$

Using the $match_G$ function find vertices of the active part of the supergraph with the most similar set of leaves:

$$l_i(v) = match_{G'_i}(B_i(v))$$

Determine all children of a tree vertex that have an empty *leaf set*. These children are vertices that are created in the current barrier tree. Note that, if all children of a tree vertex have an empty *leaf set*, that vertex will also have an empty *leaf set* also and is thus a newly created inner vertex of the barrier tree.

$$n_i(v) = \{ w |\, (v, w) \in E_i \wedge B_i(w) = \emptyset \}$$

Barrier-tree vertices can now be categorized:

- $fresh(v)$, iff $l_i(v) = \emptyset$. v is a new vertex in the current barrier tree.
- $matching(v)$, iff $|l_i(v)| = 1 \wedge n_i(v) = \emptyset$. In that case an equivalent vertex has been found in the supergraph. This vertex is the one element of $l_i(v)$ and no child of v is *fresh*.
- $matchfresh(v)$, iff $|l_i(v)| = 1 \wedge n_i(v) \neq \emptyset$. An equivalent vertex has been found in the supergraph. At least one child of v is *fresh*.
- $recomb(v)$, iff $|l_i(v)| > 1$. An equivalent vertex could not be found. $l_i(v)$ contains the most similar vertices.

Each vertex of the tree must be inserted in the supergraph, unless an equivalent vertex had been found.

$$V_i' = V_{i-1}' \cup \{\, v \mid v \in V_i \wedge \neg\text{matching}(v)\}$$

$$k_i(v) = \begin{cases} u & l_i(v) = \{u\} \wedge n_i(v) = \emptyset \\ v & l_i(v) = \{u\} \wedge n_i(v) \neq \emptyset \end{cases}$$

The inserted edges are:

$$\begin{aligned} E_i'' = E_{i-1}' \quad & \cup \quad \{\, (u,v) \mid v \in V_i \wedge (u,w) \in E_{i-1}' \wedge \text{matchfresh}(v)\} \\ & \cup \quad \{\, (v,w) \mid v \in V_i \wedge l_i(v) = \{w\} \wedge n_i(v) \neq \emptyset\} \\ & \cup \quad \{\, (k_i(v), w) \mid v \in V_i \wedge w \in l(v) \wedge \neg\text{matching}(v)\} \\ & \cup \quad \{\, (k_i(u), k_i(v)) \mid (u,v) \in E_i\} \end{aligned}$$

Transitive edges may be removed:

$$E_i' = \left\{ (u,v) \mid (u,v) \in E_i'' \wedge \neg\exists\, path_{(V_i', E_i'')}(u,w) \neq (u,w) \right\}$$

The final supergraph G is equal to the supergraph G_n, i.e. the supergraph after inserting each tree of the sequence.

6.4.4 Example

Figure 6.4 shows a nontrivial example for one iteration of the supergraph construction process. It has been chosen to show all four elementary operations that can modify barrier trees. k_i, mapping the vertices of T_i to vertices of G_i is:

$$k_i = \{(a,1), (b,2), (c,3), (d,4), (e,5), (f,6), (g,8), (h,7), (i,10)\}$$

T_i is thus very similar to G_i, only the edge (i,h) of the tree is represented by the path $(10,9,7)$ in G_i. The vertex f does not occur in the leaf mapping, i.e. it is deleted. The active part of G_i is thus: $A_{i+1} = \{1,2,3,4,5,7,8,9,10\}$. Because of the leaf mapping the *leaf sets* of the vertices of T_{i+1} are:

$$\begin{aligned} B_{i+1} \quad = \quad & \{(j,\emptyset), (k,\emptyset), (l,\{1\}), (m,\{2,4\}), (n,\{7\})\} \\ \cup \quad & \{(o,\emptyset), (p,\{1,2,4\}), (q,\{7\}), (r,\{1,2,4,7\})\} \end{aligned}$$

After $mark_{(A_{i+1},K_{i+1})}$ and $match_{(A_{i+1},K_{i+1})}$ have been determined, l_{i+1} and n_{i+1} result to:

$$\begin{aligned} l_{i+1} \quad = \quad & \{(j,\emptyset), (k,\emptyset), (l,\{1\}), (m,\{2,4\}), (n,\{9\})\} \\ \cup \quad & \{(o,\emptyset), (p,\{5\}), (q,\{9\}), (r,\{10\})\} \\ n_{i+1} \quad = \quad & \{(j,\emptyset), (k,\emptyset), (l,\emptyset), (m,\emptyset)\} \\ \cup \quad & \{(n,\emptyset), (o,\{j,k\}), (p,\emptyset), (q,\{o\}), (r,\emptyset)\} \end{aligned}$$

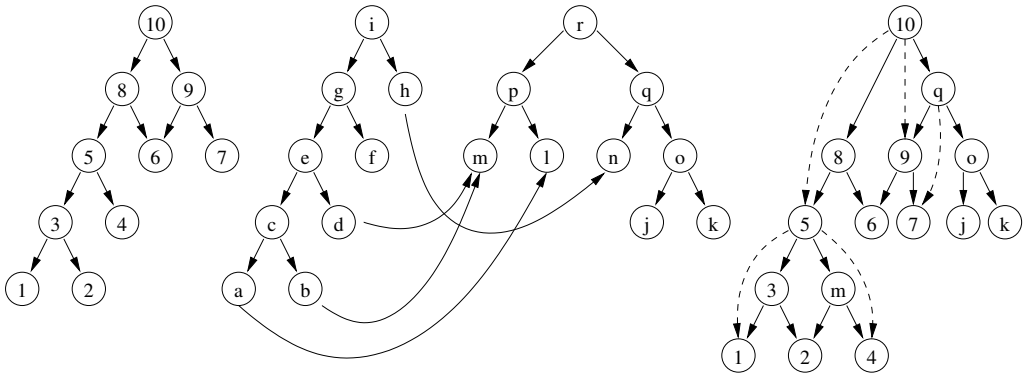


Figure 6.4: Example construction of the supergraph of two trees. Left to right: the supergraph G_i, the tree T_i, the tree T_{i+1}, and the supergraph G_{i+1}. Arrows between T_i and T_{i+1} indicate the leaf mapping. The dashed lines in G_{i+1} indicate edges that can be replaced by a path. The exact description, how the trees are embedded in the supergraph and how the supergraph is modified in this iteration, are found in the main text.

The vertices of T_{i+1} are categorized as follows: $fresh(j), fresh(k), match(l), recomb(m), match(n), fresh(o), matching(p), matchfresh(q), matching(r)$. Therefore the following vertices have to be added to the supergraph, and k_{i+1} results to:

$$V_{i+1} = V_i \cup \{j, k, m, o, q\}$$

$$k_{i+1} = \{(j,j), (k,k), (l,1), (m,m), (n,9), (o,o), (p,5), (q,q), (r,10)\}$$

Insertion of the edges is left as an exercise to the reader. Some transitive edges may be removed.

6.4.5 Postprocessing

Unfortunately, the use of the supergraph as a data structure to find similar leaf sets often requires the insertion of edges that are not needed for the final solution. Some edges are inserted to ensure correct results for the *match* and *mark* functions, but are not required for the supergraph to be an expansion of all trees. Removal of these edges decreases both the possibility of edge crossings in and the running time of the layout process.

These edges are identified as a side-product in a postprocessing phase. In this phase each edge of the supergraph is annotated with the set of all trees it occurs in. The motivation for this will be explained in the next

section. Usually a tree edge corresponds to a path in the supergraph. Therefore, each edge of the path is annotated with the tree. Quite frequently, there are multiple possible paths for one tree edge. In such cases only the edges of the longest path are annotated. After annotation, there will be many edges which do not belong to any tree. These can be removed safely. Choosing the longest path is a simple and quick heuristic that favors edges with a high probability of reuse. In practice this removes 5–20 percent of all edges of the supergraph plus any transitive edges. However, a proper problem definition for this phase would be: find the largest set of edges that can be removed without violating the constraint, that the supergraph is an expansion of each tree.

6.5 Supergraph and Subgraph Layout

The second step of the foresight-layout-with-tolerance algorithm creates the layout of the supergraph. In general, the supergraph will be a DAG. Because of the special nature of the supergraph, we benefit from using a special supergraph layout, that reflects properties of the resulting tree layouts, rather than employing known layout algorithms suitable for DAGs. The routing of edges is not relevant for the layout of the supergraph. The edges will be routed only in the subgraphs.

6.5.1 Supergraph Layout

We use the barrier trees directly as an input for the supergraph layout. We try to find an order σ of the equivalence classes such that the sum of all edge crossings in all trees is minimized, if the barrier-tree vertices were drawn using this order as the horizontal order.

$$\sigma = \underset{O}{argmin} \sum_{i=1}^{N} \left(\alpha \cdot crossings(T_i, O_{V_i}) + \beta \cdot localorder(T_i, O_{V_i}) \right)$$

where

- $T_i = (V_i, E_i, e_i)$ is the i-th tree in the sequence,
- $G = (V, E)$ is the supergraph of the tree sequence,
- V is the set of equivalence classes,
- $k : \bigcup_{i=1}^{N} V_i \rightarrow V$ maps each tree vertex to its equivalence class,

- $O \subset V \times V$ is an ordering relation,
- O_{V_i} is that ordering relation restricted to V_i and satisfies $(u, v) \in O_{V_i}$, if and only if $(k(u), k(v)) \in O$ for all $u, v \in V_i$,
- $crossings(T_i, O_{V_i})$ denotes the number of edge crossings if the tree T_i was drawn with the horizontal order of the vertices given by O_{V_i},
- $localorder(T_i, O_{V_i})$ names approximately the number of times a parent vertex is not drawn between its children, and
- α, β constants, which we set to 1 and 5, respectively.

At first we minimized the above function only considering minimizing the number of edge crossings and used simulated annealing [KGV83] to that end. We were surprised that it is possible to draw the simplest sequence (ATT) with a total of 27 edge crossings for the whole sequence. We were quickly disappointed by the images themselves, as it was apparent that we neglected to encourage the father of two vertices to be drawn between them. Because of that, the use of our orthogonal drawing style resulted in hardly readable images. So we added the second term to our objective function to avoid this particular effect. It accumulates the difference of the number of vertices that are drawn to the left of their parent and the number of vertices that are drawn to the right of their parent for each vertex. If each vertex is always between its two successors, the contribution of this term to the objective function is always zero. We experimentally determined $\alpha = 1$ and $\beta = 5$ to give good final layouts. It roughly means that we rather allow 5 edge crossings than one parent that is not between its children.

There are multiple possibilities to implement the simulated annealing strategy for this particular objective function. We tested several of them, and found the following to behave the best. We start with a random order of equivalence classes and iteratively improve this order. At each iteration, we pick a random equivalence class and insert it at a random position between two other equivalence classes. Then we reevaluate the objective function for all trees and compare it to the old value. If we improved, we keep the new order, otherwise we only keep it with the probability

$$p = \frac{1}{1 + exp(\Delta C \, T_t^{-1})} \qquad\qquad T_t = \frac{n_t - t}{t}$$

with ΔC being the cost increase and T_t being a temperature which decreases linearly with each iteration t. We stop the process after a fixed number of iterations n_t.

Instead of recalculating the total number of edge crossings, we just calculate ΔC by considering only adjacent and incident edges on all vertices v with $h(v)$ being the equivalence class currently moved. We can do this similarly for the *localorder* term of the objective function. This greatly decreases the time per iteration and makes the process very fast.

In an earlier implementation we computed the layout of the supergraph using the *dot* algorithm by Gansner *et al.* [GKNV93]. In this algorithm most of the time is spend minimizing edge crossings in a repeated heuristic two-layer edge crossing minimization which had a time complexity of $O(N^4)$, where N is the maximum number of vertices on one layer. Although the algorithm seldom runs in that order for real world examples, it takes a very long time to find the minimal number of edge crossings for our barrier-tree sequences. Not only because we observed that there was at least one layer where one eighth of all supergraph vertices resided in, but also because the swapping of vertices often did not change the number of edge crossings directly, but a few iterations later might have allowed improvements.

Our current method has much faster iterations because the number of operations per iteration is in the order of

$$O\left(\sum_{i\in\{1,\ldots,N\}} deg(T_i)|E_i|\right) = O\left(\sum_{i\in\{1,\ldots,N\}} deg(T_i)|V_i|\right)$$

where $deg(T)$ is the degree of T, i.e. the maximum number of incident and adjacent edges on any vertex v of T. So one iteration roughly scales linearly with the total number of vertices of the whole sequence, as the degree of our trees is 2 or 3 in almost all cases, i.e. a very small constant. So one iteration lies in $O(N)$, but we require many more iterations to achieve the same quality improvement of one iteration of the *dot* algorithm.

6.5.2 Tree Layout

Until now, the energy of a vertex has been ignored. Since a vertex of the supergraph may represent multiple vertices of the tree sequence and each of these vertices may have a different energy, a supergraph vertex may not have a single energy value. Because we want one of the coordinates to indicate the energy, it is impossible to perform coordinate

assignment for the supergraph vertices. Instead it is performed for each tree separately, respecting the order generated in the *ordering* phase. This constraint preserves the *mental map*, specifically the *orthogonal ordering*. Positioning each tree separately allows us to locally improve the layout of the subgraphs. This corresponds to the third phase of the foresight-layout-with-tolerance algorithm.

Initially the horizontal position of a tree vertex v is directly gained from the number of equivalence classes smaller than $h(v)$ with respect to the global ordering relation O. The vertical position of v directly reflects its energy.

After the vertices have been positioned, edges must be routed. For simplicity, each tree edges consist of just one horizontal and one vertical line segment that directly connect the two adjacent vertices. In general, it is not always possible to draw the trees without edge crossings. We sacrificed this property for the preservation of the mental map. Drawing the edges as orthogonal line segments conforms to the style barrier trees are usually drawn. We also found that a straight-line drawing does not necessarily reduce the number of edge crossings and additionally makes tracing the edges harder than an orthogonal drawing.

Positioning each tree separately allows us to locally improve the layout of the subgraphs. This corresponds to the third phase of the foresight-layout-with-tolerance algorithm. It is trivially possible to generate the horizontal position of a tree vertex v from the number of vertices of the same tree that are smaller than v with respect to O_{V_i}. It would make a better visual impression if the key frames were studied by themselves, but it destroys a lot of the mental map; so we decided not use this option for our animations.

6.6 Animation

After the layout for each tree has been generated, the single trees could be presented using the generated layout. In practice, there can be quite a number of changes between consecutive trees. Vertices and edges may appear or disappear, and whole subtrees can change the energy of their vertices. We created methods to make the transition smooth and to indicate the type of change. Vertices that experience a change of energy are moved accordingly in the drawing area using linear interpolation of the coordinates. Barriers that appear or disappear are presented using *blending*. Edges are modified based on the changes of their adjacent vertices. Subtrees that are created or merged "grow" out of or into the vertices,

where they are created or merged into, again using linear interpolation of their coordinates.

Usually the huge number of changes would require each change to be visualized separately. In our proof-of-concept implementation, all changes are shown simultaneously using the following scheme: Each transition is given a time interval $[t_i, t_i + \Delta t)$. Vertices that change their energy are moved during $[t_i + \frac{3}{8}\Delta t, t_i + \frac{7}{8}\Delta t)$. Subtrees that grow into a vertex because of merging are scaled during $[t_i + \frac{2}{8}\Delta t, t_i + \frac{5}{8}\Delta t)$, subtrees that grow out of a vertex, do so during $[t_i + \frac{5}{8}\Delta t, t_i + \frac{8}{8}\Delta t)$. Fading out of barriers takes place during $[t_i + \frac{2}{8}\Delta t, t_i + \frac{6}{8}\Delta t)$ and fading in during $[t_i + \frac{4}{8}\Delta t, t_i + \frac{8}{8}\Delta t)$. The remaining interval $[t_i, t_i + \frac{2}{8}\Delta t)$ is used for a static presentation of tree T_i. The segments overlap intentionally. In the datasets we observed we found that using non-overlapping sections resulted in large parts of the tree simply disappear and appear and destroy the mental map of the user.

6.7 Highlighting

One common question for a domain expert that analyzes the barrier-tree sequence is "which of two given structures is the winner", i.e. which one is more probable to be found in nature. It is typically found when the folding process starts in one part of the energy landscape and, later on, a new part of the energy landscape is created which is separated by a very high barrier from the rest. Regardless of how optimal the local minima of the new part of the landscape are, it is unlikely that the molecule will fold into one of them, because the barrier is too high and the probability that it will be overcome is very low on the timescale for folding reactions.

Using a simple technique, some elements of the animation can be highlighted to emphasize such observations. We split the last tree at the root and look for the leaf of lowest energy in the left subtree and the leaf of lowest energy in the right subtree. The first one is marked blue and the later one red. When the animation is shown, predecessors of these two leaves will be drawn with the appropriate color. The predecessors are given by the leaf mapping, i.e. they are local minima that will refold into the two final configurations during the process. We also found that drawing the path from the root to the actual leaf with the highlight color is visually more attracting that just coloring the leaf and its one adjacent edge. This is the default color scheme, but the user can also select other leaves, not necessarily in the last tree, to be highlighted using colors.

6.8 Results

We evaluated our algorithm on three datasets. The ATT dataset consists of 20 barrier trees, with at most 25 leaves per tree and a total of 894 vertices in all trees. It represents a small RNA molecule, with sequence length growing from 40 to 74 nucleotides with varying step size. The LEPTO dataset consists of 47 barrier trees, with a maximum of 50 leaves per tree and a total of 3727 vertices in all trees. The sequence length of the molecule increases from 10 to 56 nucleotides. The largest example, the HOK dataset, consists of 65 trees with a maximum of 100 leaves and a total of 8635 vertices. The sequence length grows from 10 to 74 nucleotides. The inner vertices of all trees of these datasets satisfy $odeg(v) = 2$, i.e., all inner vertices have exactly two children. All datasets represent rather short RNA molecules.

One way to determine the quality of the algorithm is to look at properties of the supergraph. The number of vertices in the supergraph of the ATT, LEPTO, and HOK datasets are 392, 1874 and 4594 respectively. This means that only about half of the vertices of the trees were identified as redundant. This results from a property of the sequences that we have not yet mentioned. In each new tree of a sequence, leaves get deleted, merged, and added. The average number of leaves that are added is 5.00, 7.20, and 16.16 respectively. That means that up to 20 percent of each tree changes on average. It can be shown, however, that a graph-theoretic minimum supergraph would not be smaller than approximately half to one third of the size of our supergraphs.

More critical to the perceived quality of the layout is the number of edges. If this number is near the number of vertices, the supergraph is very similar to a tree and can thus be drawn with few edge crossings and (horizontally) short edges. Horizontally long edges in the supergraph layout are undesirable, because each edge is shown at least once. The amount of edges divided by the amount of vertices for the three datasets are 1.52, 1.69, and 1.61, respectively. Although these numbers seem to be close, the LEPTO and HOK datasets have a significantly larger number of edge crossings and long edges than the ATT dataset. This is because the edges are unevenly distributed among the layers of the supergraph layout. The animation suffers from long edges that are close together and are notoriously difficult to track.

Two preprocessing methods have been tested to determine, if a subset of the data still results in bad layouts. Surely, we do not want to reduce the number of trees, since we want to visualize the whole process.There are a number of barriers that are connected by an edge in the barrier

tree, and whose energy differs only slightly. Such barriers are merged in a preprocessing step. This process reduces the probability of *barrier swaps* and the supergraph will have less vertices. In the LEPTO and HOK datasets, the merging of barriers that differ by 0.5 or less (which is approximately two percent of the overall energy range) reduced the total number of tree vertices to 2419 and 5863, respectively, and the number of supergraph vertices to 853 and 2493, respectively. This means, that now nearly two third of the vertices were identified as redundant. Unfortunately, this method does not reduce the number of edges as much as the number of vertices, thus the supergraph suffers from a huge number of edge crossings and long horizontal edges. After applying this method, the supergraph span less layers and the edges got distributed more equal over the layers. In the final animation long edges are still visible, but they are no longer close together, so it is easier to track them.

The second method is the reduction of leaves in the barrier trees. Local minima with a low energy are generally more stable and have a high probability of being present in the next barrier tree. They are also more interesting than local minima of higher energy. For each leaf that is removed, the one barrier connecting it to the rest of the tree is removed as well. By reducing the number of leaves in the LEPTO and HOK datasets to a maximum of 31 and 66 leaves per tree, respectively, the total number of tree vertices was reduced to 2409 and 5875, respectively. The number of supergraph vertices was reduced to 732 and 2528, so again almost two third of the tree vertices have been identified as redundant. This preprocessing method removed substantially more edges than vertices, and in the LEPTO and HOK datasets the number of edges divided by the number of vertices decreased to 1.50 and 1.44, respectively, which greatly improved the supergraph layout. There were a lot less edge crossings and only a few long edges. This directly resulted in a better layout of the barrier trees.

We found our simulated-annealing-based supergraph layout, which effectively only determines an optimal vertex order for the supergraph vertices, to perform very well. We feared that most of the apparently random permutations do not improve the tree layout at all or only slowly. Indeed, we require a huge number of iterations until the images produced were useful. Whereas the dot-based algorithm used 1000 layer sweep iterations the simulated-annealing-based algorithm uses at least 100.000 iterations to generate readable animations. But because each iteration, being of a lower order of complexity, requires much less time, the new implementation is still faster. On an AMD Opteron with 2.0GHz, the dot-based algorithm with 1000 iterations required approximately 37,

3462, and 16790 seconds for the ATT, LEPTO, and HOK dataset with 381, 1531, and 3793 equivalence classes respectively. The simulated-annealing-based method using 1.000.000 iterations required approximately 27, 79, and 182 seconds, respectively.

The visual output of the two radically different methods is not directly comparable. There is one thing directly observable for all datasets. The simulated-annealing-based algorithm distributes vertices more evenly across the drawing area. But this is rather a nice side effect of the method and was not originally intended. While the ATT sequence does not improve much visually if compared to the dot-based method, it benefits from the reduced computation time. The visual quality of the LEPTO and HOK sequences improves greatly with the simulated-annealing-method, because it considers only relevant edge crossings, i.e. those that will show up in the barrier-tree layouts.

Figure 6.5 shows the key frames for the ATT data set. Terminator/anti-terminators are small RNA structure elements occurring in bacterial messenger RNAs (mRNAs) that modulate the translation of mRNA into a protein. They are switch-like elements that can form alternative structures with drastically different physiological function. In one state, the mRNA is translated and protein is produced, while the alternative RNA conformation suppresses this process. The speed of transcription determined, which of these two states is reached. The barrier-tree sequence of the growing RNA element can be used to understand the molecular mechanism for this behavior.

The panels in Figure 6.5 show the leader sequence of the pheS-pheP operon of *E. coli* [FMS+83]. In the first stages, i.e., when only the 5′ (left) part of the molecule has been transcribed, the folding landscape is dominated by a single conformation (visible as the lowest-energy subtree in panels 1 through 5. As the molecule grows, an alternative basin of attraction (left subtree in rows 4 and 5) appears. In the first stages, this class of conformation is less stable than the l.h.s. subtree, which is initially populated as it corresponds to the stable conformations in the first folding stages. In the full-size element, however, a different conformation class is thermodynamically favored, which appears as the r.h.s. subtree in the last 3 panels. The important observation is that this class of conformations is reachable only by transversing a sizable energy barrier; hence the transition into this conformation is slow. The speed of translation thus determines, whether the growing chain has sufficient time to switch into the optimal subtree (approximately in panels 13-15), or whether it remains trapped in the l.h.s. subtree, as the barrier height (and hence the necessary transition time) increases with the chain length (panels 16-18).

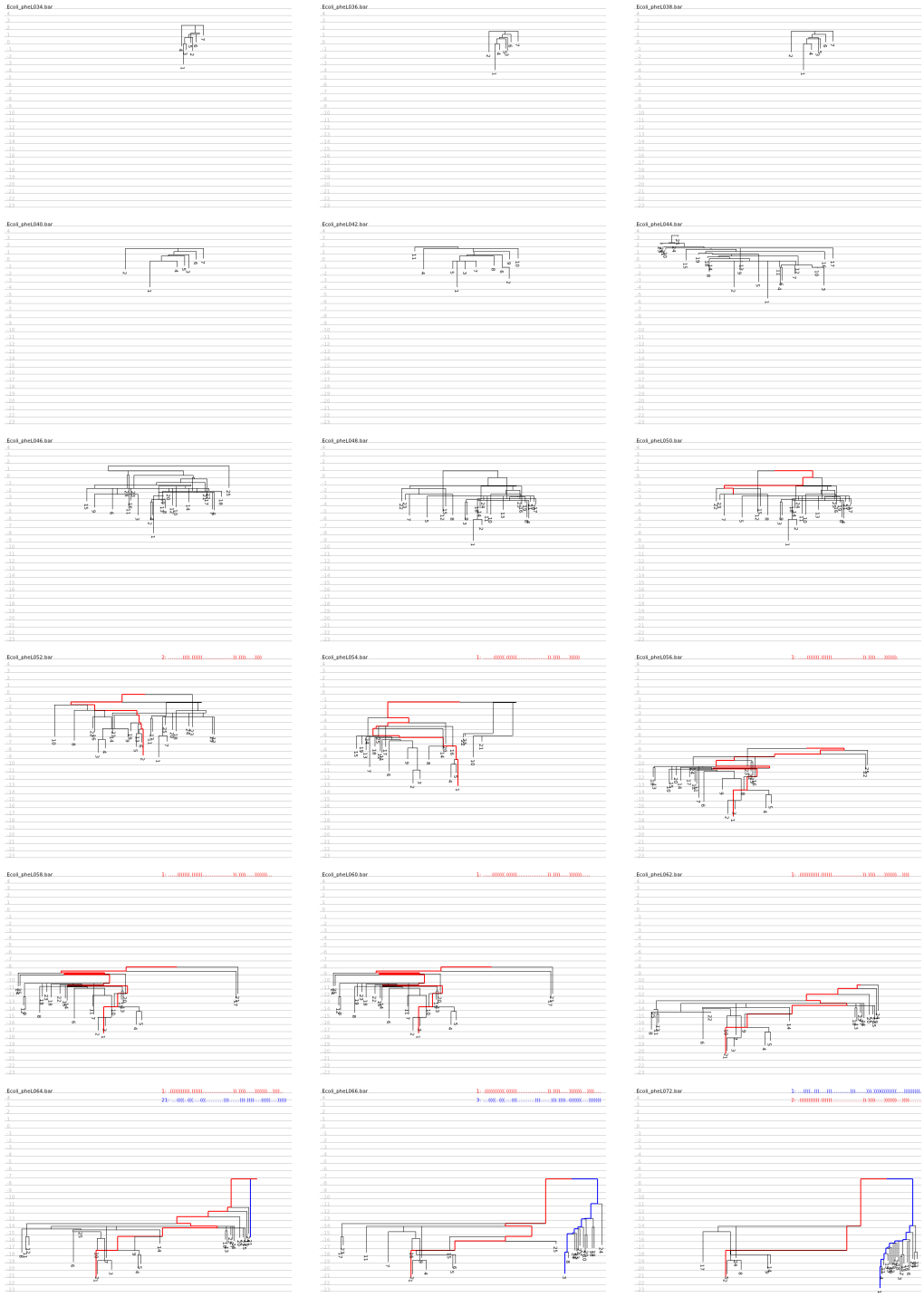


Figure 6.5: 18 subgraph layouts of the ATT sequence.

6.9 Summary

We have shown that it is possible to generate readable layouts for sequences of barrier trees using the foresight-layout-with-tolerance algorithm. For larger datasets, preprocessing may need to be applied to the sequence. While reducing barriers decreases the height of the supergraph layout, a reduction of leaves decreases the width and greatly improves the perceived quality of the layout.

We also showed, that construction of a supergraph may sometimes lead to suboptimal results if the supergraph does not use all information from the graphs it is constructed for. Our naive implementation of a combination of supergraph construction and layout clearly outperformed the version where these two were separate, both quality and run-time-complexity wise. The number of iterations needed for a given dataset seems to scale with its size, but we are yet uncertain exactly how. We are looking for methods that automatically determine the optimal number of iterations.

From the viewpoint of folding landscapes, often only a small number of leaves are of interest. These leaves and their history can be highlighted using colors. The layout of the single trees may be combined with additional information. The simulation of the folding process during the growing of the molecule under various temperatures and growing rates results in distribution functions for local minima. Because the animation of the barrier trees preserves the orthogonal ordering, annotating the barrier-tree leaves with the density of the corresponding structure configurations preserves the mental map for the annotations. The change in the densities could additionally be indicated by a flow of liquid along the tree edges. Methods that combine tree layout and additional information are also investigated.

The current methods to generate the animation leave room for further improvements. Different strategies for edge removal during the postprocessing of the supergraph construction can result in an improved layout, because fewer edges generally result in fewer edge crossings. Rather than overproducing the edges of the supergraph and reducing them afterward, a more constructive method could be proposed. In this chapter we did not pay much attention to local improvement of the subgraph layout. Especially in larger datasets this would be beneficial, because each subgraph uses only a small part of the drawing area and requires high resolution. A local improvement based on a force-directed strategy is could achieve the desired distribution.

As a transition from one tree to the next consists of many elementary

operations, instead of showing them simultaneously, it might be better to break the leaf mappings in elementary operations and show them in sequence.

The constructed supergraph is a static visualization of the whole sequence, and presentation forms other than an animation, may be investigated. An alternative would be to synthesize a 2D landscape from all barrier trees, where the folding process is visualized as a walk.

7 Conclusions and Future Work

In this thesis we examined some point designs on the boundary of information visualization, graph drawing, and Euler diagrams. It is clear that there is a large body of knowledge in these fields that can and should be transferred between the fields and at some point combined into larger theory of visualization.

Euler diagrams can benefit from the model of retinal variables, i.e. it can use colors and textures to improve perception of contours especially in the regions of overlap ([War04]). Chapter 3 discussed the drawing of Euler diagrams using blobs – a technique well known from computer graphics – which deliver smooth contours and are visually stable when elements' positions change. The chapter focused on an interactive visualization process which enabled the user to move items and, while doing so, changing their grouping. To achieve interactive frame rates for small datasets, a method to exploit the duality of DAGs and abstract Euler diagrams was presented that computes in quad-tree-fashion for each pixel which clusters it belongs to and was applied to manual refinement of clustering results. Although there is much research in the area of Euler diagrams, most of it is geared towards static set relations and static display. The work in this chapter remains to-date the only approach for dynamic set relations and enabling the user to arbitrarily position the elements.

For larger set relations, the method was improved in Chapter 4 in terms of visual discriminability of sets when they overlap and rendering speed. The use of textures as a visual channel that supports discrimination in addition to colors was employed successfully to compare clustering results of small document collections with different clustering granularity. Here the duality of DAGs and Euler diagrams was exploited to generate a representation of the cluster hierarchies' union. We studied the appearance of multiple GPU-accelerated contour interpolators

and gave a recommendation for the optimum. However, for the sake of fast rendering some wellformedness criteria of Euler diagrams had to be sacrificed. New computational architectures providing faster execution times allow for new and more complex interpolators in our framework. The study of base textures that allow a large number of overlapping sets remains a challenge and open question.

In Chapter 5 we outlined the design space of contour tree representations in the form of five drawing criteria and gave two example point designs: one academic and one for real-world use. The later respects four of the criteria, and is a mix of information visualization and graph drawing design. While single branches are reminiscent of traditional line graphs, drawn vertically and without axes, the branches were arranged horizontally according to minimizing edge crossings – a graph drawing aesthetic. The method extends trivially to the more general class of Reeb graphs, although for this class of graphs, additional drawing criteria may be applicable. Future work would focus on removing the impact of long horizontal edges that dominate the drawings for very large contour trees.

In Chapter 6, we presented an algorithm that constructs an animation for a sequence of barrier trees that are compaction of so-called folding landscapes, i.e. fitness landscapes in which the structure of RNA molecules is predicted. A major part of the work was constructing the dynamic tree from tree snapshots at given points in time. The animations help to explain why certain spatial configurations that are not energy-minimal are found in nature: because of the distinct reshaping of the landscape during the RNA transcription process. The method was applied successfully for more use cases by Hofacker *et al.* [HFH+10]: the traveling of an RNA molecule through a pore, unfolding at one side and refolding at the other, and a shift in environment temperature.

Lessons learned from these four designs is that when the perceptual operation of judging nesting and inclusion relations between items should be supported by the diagram, an Euler representation is preferred over a node-link diagram. The later are more useful if graphical node properties are expressed with information visualization techniques that highlight data associated with the feature a node represents, and relations between feature are secondary. Applying the Gestalt principle of closure is more effective in conveying grouping relations than the principle of proximity as the items can be distributed more uniformly on the screen and thereby allow a higher data density as we have seen in Chapter 4.

The work on barrier trees and the work on contour trees can be combined. For time-varying scalar fields the contour tree becomes dynamic

and principles of the dynamic barrier trees carry over. Although it is possible to reconstruct a dynamic contour tree from time snapshots using the barrier method, a direct computation considering the correct interpolation will increase the resolution with respect to time. A dedicated time-varying contour tree algorithm does not yet exist, but an algorithm for the more general case of Reeb graphs does ([EHM+08]), providing an alternative vantage point.

Another upcoming topic is the analysis of the topology of multiple scalar fields [SHCS12]. The relationship of contour tree drawings and tree map algorithms was established by [HW10]. As Euler diagrams can be seen as a generalization of tree maps, they could be used to show the overlapping regions of representative contours as indicated by the contour trees of the single fields. Area-proportional Euler diagrams ([Wil12]) could highlight interesting topological overlaps from the mass of overlaps that are likely to result due to noise in the dataset. A very long-term goal could be to combine concepts of all three methods to show time-varying multifield topology using animations of dynamic Euler diagrams.

Bibliography

[ACL04] Reid Andersen, Fan R. K. Chung, and Lincoln Lu. Drawing power law graphs. In Pach [Pac04], pages 12–17.

[AKS+02] K. Andrews, W. Kienreich, V. Sabol, J. Becker, G. Droschl, F. Kappe, M. Granitzer, P. Auer, and K. Tochtermann. The infosky visual explorer: Exploiting hierarchical structure and document similarities. *Information Visualisation*, 1(3):166–181, 2002.

[Ans73] F. J. Anscombe. Graphs in statistical analysis. *American Statistician*, 27:17–21, 1973.

[Aub04] David Auber. Tulip - a huge graph visualization framework. In Jünger and Mutzel [JM04], pages 105–126.

[BD05] Michael Balzer and Oliver Deussen. Voronoi treemaps. In *IEEE Symposium on Information Visualization*, page 7. IEEE Computer Society, 2005.

[BDS03] Ulrik Brandes, Tim Dwyer, and Falk Schreiber. Visualizing related metabolic pathways in two and a half dimensions. In Liotta [Lio04], pages 111–122.

[Ber11] Jacques Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. Esri Press, 2011.

[BETT94] Guiseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, 1994.

[BETT99] Guiseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[BHvW00] Mark Bruhls, Kees Huizing, and Jarke J. van Wijk. Squarified treemaps. In *Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42. IEEE Computer Society, 2000.

[Bie06a] Chris Biemann. Chinese whispers - an efficient graph clustering algorithm and its application to natural language processing problems. In *Proceedings of the Workshop on Textgraphs at the HLT/NAACL*, New York City, NY, USA, June 2006.

[Bie06b] Chris Biemann. Unsupervised part-of-speech tagging employing efficient graph clustering. In *Proceedings of the Student Research Workshop at the COLING/ACL*, Sydney, Australia, July 2006.

[Bor07] Stefan Bordag. *Elements of Knowledge-free and Unsupervised lexical acquisition*. PhD thesis, Department of Natural Language Processing, University of Leipzig, Leipzig, Germany, 2007.

[BPS97] Chandrajit L. Bajaj, Valerio Pascucci, and Daniel R. Schikore. The contour spectrum. In *IEEE Visualization 1997*, pages 167–173. IEEE Computer Society, 1997.

[BR63] Roger L. Boyell and Henry Ruston. Hybrid techniques for real-time radar simulation. In *AFIPS '63 (Fall): Proceedings of the 1963 Fall Joint Computer Conference*, pages 445–458. ACM, 1963.

[BT05] Chris Biemann and Sven Teresniak. Disentangling from babylonian confusion - unsupervized language identification. In *Proceedings of the Sixth International Conference on Intelligent Text Processing and Computational Linguistics (CICLing), LNCS 3406*, Mexico City, Mexico, February 2005. Springer.

[BW07] Chris Biemann and Hans Friedrich Witschel. Webspam detection via semi-supervised graph partitioning. In *Pro-*

ceedings of GraphLab 2007, Workshop on Graph Labelling at ECML/PKDD, Warsaw, Poland, 2007.

[Cas90] Stephen Michael Casner. *Task-Analytic Design of Graphic Presentations*. PhD thesis, University of Pittsburgh, 1990.

[CBT+92] Robert F. Cohen, Giuseppe Di Battista, Roberto Tamassia, Ioannis G. Tollis, and Paola Bertolazzi. A framework for dynamic graph drawing. In *Symposium on Computational Geometry*, pages 261–270, 1992.

[CC92] M. Chalmers and P. Chitson. Bead: Explorations in information visualization. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 330–337. ACM, 1992.

[CCLP03] D. Chan, K. Chua, C. Leckie, and A. Parhar. Visualisation of power-law network topologies. In *Proceedings of the Eleventh IEEE International Conference on Networks*, pages 69–74, 2003.

[CF07] Rosario De Chiara and Andrew Fish. EulerView: a non-hierarchical visualization component. In *VL/HCC*, pages 145–152. IEEE Computer Society, 2007.

[CGA] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[Cho07] Stirling Chow. *Generating and Drawing Area-Proportional Euler and Venn Diagrams*. PhD thesis, University of Victoria, 2007.

[CHS96] Jan Cupal, Ivo L. Hofacker, and Peter F. Stadler. Dynamic programming algorithm for the density of states of RNA secondary structures. In R. Hofstädt, T. Lengauer, M. Löffler, and D. Schomburg, editors, *Computer Science and Biology 96 (Proceedings of the German Conference on Bioinformatics)*, pages 184–186. Universität Leipzig, 1996.

[CL03] Yi-Jen Chiang and Xiang Lu. Progressive simplification of tetrahedral meshes preserving all isosurface topologies. *Computer Graphics Forum*, 22(3):493–504, 2003.

[CM86] William S. Cleveland and Robert McGill. An experiment in graphical perception. *International Journal of Man-Machine Studies*, 25(5):491–501, 1986.

[CMS99] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.

[CPC09] Christopher Collins, Gerald Penn, and M. Sheelagh T. Carpendale. Bubble sets: Revealing set relations with isocontours over existing visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1009–1016, 2009.

[CS03] Hamish Carr and Jack Snoeyink. Path seeds and flexible isosurfaces using topology for exploratory visualization. In *VISSYM '03: Proceedings of the symposium on data visualisation 2003*, pages 49–58. Eurographics Association, 2003.

[CSvdP04] Hamish Carr, Jack Snoeyink, and Michiel van de Panne. Simplifying flexible isosurfaces using local geometric measures. In *IEEE Visualization 2004*, pages 497–504. IEEE Computer Society, 2004.

[dBvKOS08] Mark de Berg, Marc van Kreveld, Mark Overmars, and Ottfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[DG02] Stephan Diehl and Carsten Görg. Graphs, they are changing. In Stephen G. Kobourov and Michael T. Goodrich, editors, *Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 23–30. Springer, 2002.

[Die05] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, 3rd. edition, 2005.

[DN08] Harish Doraiswamy and Vijay Natarajan. Efficient Output-Sensitive Construction of Reeb Graphs. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *ISAAC*, volume 5369 of *Lecture Notes in Computer Science*, pages 556–567. Springer, 2008.

[DS04] Tim Dwyer and Falk Schreiber. Optimal leaf ordering for two and a half dimensional phylogenetic tree visualisation. In Neville Churcher and Clare Churcher, editors, *InVis.au*,

volume 35 of *CRPIT*, pages 109–115. Australian Computer Society, 2004.

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[Ead92] Peter Eades. Drawing free trees. *Bulletin of the Institute of Combinatorics and its Applications*, 5:10–36, 1992.

[EHK+03] Cesim Erten, Philip J. Harding, Stephen G. Kobourov, Kevin Wampler, and Gary V. Yee. Graphael: Graph animations with evolving layouts. In Liotta [Lio04], pages 98–110.

[EHM+08] Herbert Edelsbrunner, John Harer, Ajith Mascarenhas, Valerio Pascucci, and Jack Snoeyink. Time-varying Reeb graphs for continuous space-time data. *Computational Geometry*, 41(3):149–166, 2008.

[ESK04] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing. In Pach [Pac04], pages 155–166.

[FE02] Carsten Friedrich and Peter Eades. Graph drawing in motion. *Journal of Graph Algorithms and Applications*, 6(3):353–370, 2002.

[FFHS00] Christoph Flamm, Walter Fontana, Ivo L. Hofacker, and Peter Schuster. RNA folding at elementary step resolution. *RNA*, 6:325–338, 2000.

[FH02] Jean Flower and John Howse. Generating Euler Diagrams. In Mary Hegarty, Bernd Meyer, and N. Hari Narayanan, editors, *Diagrams*, volume 2317 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2002.

[FHSW02] Christoph Flamm, Ivo L. Hofacker, Peter F. Stadler, and Michael T. Wolfinger. Barrier trees of degenerate landscapes. *Zeitschrift für Physikalische Chemie*, 216(2):155–173, 2002.

[FMG05] B. Fortuna, D. Mladenic, and Marko Grobelnik. Visualization of text document corpus. *Informatica*, 29:497–502, 2005.

[FMS+83] Guy Fayat, Jean-François Mayaux, Christine Sacerdot, Michel Fromant, Mathias Springer, Marianne Grunberg-Manago, and Sylvain Blanquet. *Escherichia coli* phenylalanyl-tRNA synthetase operon region : Evidence for an attenuation mechanism. Identification of the gene for the ribosomal protein L20. *Journal of Molecular Biology*, 171(3):239–352, 1983.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[Fri96] Arne Frick. Upper Bounds on the Number of Hidden Nodes in Sugiyama's Algorithm. In Stephen C. North, editor, *Graph Drawing*, volume 1190 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 1996.

[FS06] Andrew Fish and Gem Stapleton. Defining Euler Diagrams: Simple or What? In Dave Barker-Plummer, Richard Cox, and Nik Swoboda, editors, *Diagrams*, volume 4045 of *Lecture Notes in Computer Science*, pages 109–111. Springer, 2006.

[FT04] Yaniv Frishman and Ayellet Tal. Dynamic drawing of clustered graphs. In Matthew O. Ward and Tamara Munzner, editors, *10th IEEE Symposium on Information Visualization (InfoVis 2004)*, pages 191–198. IEEE Computer Society, 2004.

[FT07a] Y. Frishman and A. Tal. Multi-Level Graph Layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1319, 2007.

[FT07b] Yaniv Frishman and Ayellet Tal. Online dynamic graph drawing. In Museth et al. [MMY07], pages 75–82.

[Gal] M. Galassi. *GNU Scientific Library Reference Manual*, 3rd edition.

[GBH04] Ulrich Gerland, Ralf Bundschuh, and Terence Hwa. Translocation of structured polynucleotides through nanopores. *Physical Biology*, 1(1):19–26, 2004.

[GBPD04] Carsten Görg, Peter Birke, Mathias Pohl, and Stephan Diehl. Dynamic graph drawing of sequences of orthogonal and hierarchical graphs. In Pach [Pac04], pages 228–238.

[GHGH08] Apeksha Godiyal, Jared Hoberock, Michael Garland, and John C. Hart. Rapid Multipole Graph Drawing on the GPU. In Ioannis G. Tollis and Maurizio Patrignani, editors, *Graph Drawing*, volume 5417 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2008.

[GKN04] Emden R. Gansner, Yehuda Koren, and Stephen C. North. Graph drawing by stress majorization. In Pach [Pac04], pages 239–250.

[GKNV93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, 2000.

[GSF97] Markus H. Gross, T. C. Sprenger, and J. Finger. Visualizing informationon a sphere. In *IEEE Symposium on Information Visualization 1997*, pages 11–16. IEEE Computer Society, 1997.

[GW05] Marco Gaertler and Dorothea Wagner. A hybrid model for drawing dynamic and evolving graphs. In Healy and Nikolov [HN06], pages 189–200.

[Har75] John A. Hartigan. *Clustering Algorithms*. Wiley, 1975.

[HBHS08] Christian Heine, Stefan Bordag, Gerhard Heyer, and Gerik Scheuermann. Euler diagramm rendering on the GPU with applications to document analysis, 2008. Submitted to IEEE International Conference on Information Visualization.

[HE12] Christopher Healey and James Enns. Attention and visual memory in visualization and computer graphics. *IEEE Transactions on Visualization and Computer Graphics*, 18(7):1170–1188, 2012.

[HFH+10] Ivo L. Hofacker, Christoph Flamm, Christian Heine, Michael T. Wolfinger, Gerik Scheuermann, and Peter F. Stadler. Barmap: RNA folding on dynamic energy landscapes. *RNA*, 16:1308–1316, 2010.

[HJ04] Stefan Hachul and Michael Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In Pach [Pac04], pages 285–295.

[HJ05] Stefan Hachul and Michael Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In Healy and Nikolov [HN06], pages 235–250.

[HK00] David Harel and Yehuda Koren. A fast multi-scale method for drawing large graphs. In Joe Marks, editor, *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 183–196. Springer, 2000.

[HK04] David Harel and Yehuda Koren. Graph drawing by high-dimensional embedding. *Journal Graph Algorithms and Applications*, 8(2):195–214, 2004.

[HMM00] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 06(1):24–43, 2000.

[HN06] Patrick Healy and Nikola S. Nikolov, editors. *Graph Drawing, 13th International Symposium, GD 2005, Limerick, Ireland, September 12-14, 2005, Revised Papers*, volume 3843 of *Lecture Notes in Computer Science*. Springer, 2006.

[HQW05] Gerhard Heyer, Uwe Quasthoff, and Thomas Wittig. *Wissensrohstoff Text. Text Mining: Konzepte, Algorithmen, Ergebnisse.* W3L-Verlag, Bochum, Germany, 2005.

[HS07] Christian Heine and Gerik Scheuermann. Manual clustering refinement using interaction with blobs. In Museth et al. [MMY07], pages 59–66.

[HSCS11] Christian Heine, Dominic Schneider, Hamish Carr, and Gerik Scheuermann. Drawing contour trees in the plane. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1599–1611, 2011.

[HSF+06] Christian Heine, Gerik Scheuermann, Christoph Flamm, Ivo L. Hofacker, and Peter F. Stadler. Visualization of barrier tree sequences. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):781–788, 2006.

[HSF+07] Christian Heine, Gerik Scheuermann, Christoph Flamm, Ivo L. Hofacker, and Peter F. Stadler. Visualization of barrier tree sequences revisited. In Lars Linsen, Hans Hagen, and Bernd Hamann, editors, *Proceedings Visualization in Medicine and Life Sciences*, Mathematics and Visualisation, pages 275–292. Springer, 2007.

[HW10] William Harvey and Yusu Wang. Topological landscape ensembles for visualization of scalar-valued functions. *Computer Graphics Forum*, 29(3):993–1002, 2010.

[IMO09] S. Ingram, T. Munzner, and M. Olano. Glimmer: Multilevel MDS on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 15(2):249–261, 2009.

[JHH+10] Stefan Jänicke, Christian Heine, Marc Hellmuth, Peter F. Stadler, and Gerik Scheuermann. Visualization of graph products. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1082–1089, 2010.

[JHS12] Stefan Jänicke, Christian Heine, and Gerik Scheuermann. Comparative visualization of geospatial-temporal data. In *IVAPP - International Conference on Information Visualization Theory and Applications*, 2012. Best Student Paper Award.

[JM04] Micheal Jünger and Petra Mutzel, editors. *Graph Drawing Software*. Mathematics and Visualization. Springer, 2004.

[JS91] B. Johnson and Ben Shneiderman. Tree maps: A space-filling approach to the visualization of hierarchical information structures. In *IEEE Visualization*, pages 284–291, 1991.

[JTP+95] J.A.Wise, J.J. Thomas, K. Pennock, D. Lantrip, M.Pottier, A. Schur, and V. Crow. Visualizing the non-visual: Spatial analysis and interaction from text documents. In *IEEE Information Visualization 2005 Proceedings*, pages 51–58. IEEE CS, 1995.

[KCT08] Mikko Kurimo, Mathias Creutz, and Ville Turunen. Morpho Challenge evaluation by IR experiments. In *Proceedings of the CLEF 2007 Workshop*, Lecture Notes in Computer Science. Springer, 2008.

[KG06] Gautam Kumar and Michael Garland. Visual exploration of complex time-varying graphs. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):805–812, 2006.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KH03] Yehuda Koren and David Harel. Axis-by-axis stress minimization. In Liotta [Lio04], pages 450–459.

[KK89] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[KW01] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer, 2001.

[Lam08] Heidi Lam. A framework of interaction costs in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1149–1156, 2008.

[Lio04] Giuseppe Liotta, editor. *Graph Drawing, 11th International Symposium, GD 2003, Perugia, Italy, September 21-24, 2003, Revised Papers*, volume 2912 of *Lecture Notes in Computer Science*. Springer, 2004.

[LS87] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, 1987.

[LSM91] X. Lin, D. Soergel, and G. Marchionini. A self-organizing semantic map for information retrieval. In *Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 262–269. ACM, 1991.

[Mac86] Jock D. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986.

[Mar91] Joseph William Marks. *Automating the Design of Network Diagrams*. PhD thesis, Harvard University, 1991.

[Maz09] Riccardo Mazza. *Introduction to Information Visualization*. Springer, 2009.

[MELS95] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[Mil56] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, 1956.

[Mil63] John Willard Milnor. *Morse Theory*. Princeton University Press, 1963.

[MM04] Irmtraud M. Meyer and Istvan Miklos. Co-transcriptional folding is encoded within RNA genes. *BMC Molecular Biology*, 5(10), 2004.

[MM08] Chris Muelder and Kwan-Liu Ma. Rapid graph layout using space filling curves. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1301–1308, 2008.

[MMY07] Ken Museth, Torsten Möller, and Anders Ynnerman, editors. *EuroVis07: Joint Eurographics - IEEE VGTC Symposium on Visualization, Norrköping, Sweden, 23-25 May 2007*. Eurographics Association, 2007.

[Moe90] Sven Moen. Drawing dynamic trees. *IEEE Software*, 7(4):21–28, 1990.

[Mun09] Tamara Munzner. Visualization. In *Fundamentals of Computer Graphics*, pages 675–707. AK Peters, 2009.

[Nor93] Donald A. Norman. *Things That Make Us Smart: Defending Human Attributes in the Age of the Machine*. Perseus Books, 1993.

[Nor95] Stephen C. North. Incremental layout in DynaDAG. In *Graph Drawing*, pages 409–418, 1995.

[Nor06] Chris North. Toward measuring visualization insight. *IEEE Computer Graphics and Applications*, 26(3):6–9, 2006.

[OHJ+11] Patrick Oesterling, Christian Heine, Heike Jänicke, Gerik Scheuermann, and Gerhard Heyer. Visualization of high-dimensional point clouds using their density distribution's topology. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1547–1559, 2011.

[OHJS10] Patrick Oesterling, Christian Heine, Heike Jänicke, and Gerik Scheuermann. Visual analysis of high-dimensional point clouds using topological landscapes. In *PacificVis*, pages 113–120. IEEE, 2010.

[OHWS12] Patrick Oesterling, Christian Heine, Gunter H. Weber, and Gerik Scheuermann. Representing a point cloud's topology by a 1D height field. *IEEE Transactions in Computer Graphics and Visualization*, 2012. In press.

[Pac04] János Pach, editor. *Graph Drawing, 12th International Symposium, GD 2004, New York, NY, USA, September 29 - October 2, 2004, Revised Selected Papers*, volume 3383 of *Lecture Notes in Computer Science*. Springer, 2004.

[PCMS04] Valerio Pascucci, M. Cole-McLaughlin, and G. Scorzelli. Multi-resolution representation of topology. In *IASTED Visualization, Imaging, and Image Processing 2004*, pages 452–290, 2004.

[PLCB04] Cynthia Sims Parr, Bongshin Lee, Dana Campbell, and Benjamin B. Bederson. Visualizations for taxonomic and phylogenetic trees. *Bioinformatics*, 20(17):2997–3004, 2004.

[Pur97] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In Giuseppe Di Battista, editor, *Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 1997.

[PW12] Daniel Pineo and Colin Ware. Data visualization optimization via computational modeling of perception. *IEEE Transactions on Visualization and Computer Graphics*, 18(2):309–320, 2012.

[RCCR02] George Robertson, Kim Cameron, Mary Czerwinski, and Daniel Robbins. Polyarchy visualization: visualizing multiple intersecting hierarchies. In *CHI '02: Proceedings of the*

SIGCHI conference on Human factors in computing systems, pages 423–430, New York, NY, USA, 2002. ACM.

[RD10] Nathalie Henry Riche and Tim Dwyer. Untangling Euler Diagrams. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1090–1099, 2010.

[Ree46] G. Reeb. Sur le points singuliers d'une forme de pfaff complètement intégrable ou d'une fonction numérique. *Comptes Rendus de l'Acadèmie des Sciences de Paris*, 222:847–849, 1946.

[Ren94] E. Rennison. Galaxy of news: An approach to visualizing and understanding expansive news landscapes. In *ACM Symposium on User Interface Software and Technology*, pages 3–12. ACM, 1994.

[RHR+09] Markus Rohrschneider, Christian Heine, André Reichenbach, Andreas Kerren, and Gerik Scheuermann. A novel grid-based visualization approach for metabolic networks with advanced focus&context view. In David Eppstein and Emden R. Gansner, editors, *Graph Drawing*, volume 5849 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 2009.

[RI96] D.A. Rushall and M.R. Ilgen. Depict: Documents evaluated as pictures. In *IEEE Information Visualization 1996 Proceedings*, pages 100–107. IEEE CS, 1996.

[RM90] Steven F. Roth and Joe Mattis. Data characterization for intelligent graphics presentation. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, pages 193–200. ACM, 1990.

[RSH+12] Wieland Reich, Dominic Schneider, Christian Heine, Alexander Wiebel, Guoning Chen, and Gerik Scheuermann. Combinatorial vector field topology in 3 dimensions. In Ronald Peikert, Helwig Hauser, Hamish Carr, and Raphael Fuchs, editors, *Topological Methods in Data Analysis and Visualization II*, Mathematics and Visualization, pages 47–59. Springer, 2012.

[RT81] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.

[RZP12] Peter Rodgers, Leishi Zhang, and Helen C. Purchase. Well-formedness Properties in Euler Diagrams: Which Should Be Used? *IEEE Transactions on Visualization and Computer Graphics*, 18(7):1089–1100, 2012.

[SA08] Paolo Simonetto and David Auber. Visualise Undrawable Euler Diagrams. In *IV*, pages 594–599. IEEE Computer Society, 2008.

[SA09] Paolo Simonetto and David Auber. An heuristic for the construction of intersection graphs. In *IV*, pages 673–678. IEEE Computer Society, 2009.

[SAA09] Paolo Simonetto, David Auber, and Daniel Archambault. Fully automatic visualisation of overlapping sets. *Computer Graphics Forum*, 28(3):967–974, 2009.

[SBG00] T. C. Sprenger, R. Brunella, and Markus H. Gross. H-blob: A hierarchical visual clustering method using implicit surfaces. In *IEEE Visualization 2000*, pages 61–68. IEEE Computer Society, 2000.

[Seb02] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.

[SGEK97] T. C. Sprenger, Markus H. Gross, A. Eggenberger, and M. Kaufmann. A framework for physically-based information visualization. In *Proceedings of Eurographics Workshop on Visualization '97 (Boulogne sur Mer, France, April 28-30, 1997)*, pages 77–86, 1997.

[SHCS12] Dominic Schneider, Christian Heine, Hamish Carr, and Gerik Scheuermann. Interactive comparison of multifield scalar data based on largest contours. *Computer Aided Geometric Design*, 2012. In press.

[SKK91] Yoshihisa Shinagawa, Tosiyasu L. Kunii, and Yannick L. Kergosien. Surface coding based on morse theory. *IEEE Computer Graphics and Applications*, 11(5):66–78, 1991.

[SM00] Heidrun Schumann and Wolfgang Müller. *Visualisierung: Grundlagen und allgemeine Methoden*. Springer, 2000.

[Soa03] Catherine Soanes, editor. *The Oxford Dictionary of English*. Oxford University Press, 2nd edition, 2003.

[Spe07] Robert Spence. *Information Visualization: Design for Interaction*. Pearson Education, 2nd edition, 2007.

[Ste46] S. S. Stevens. On the theory of scales of measurement. *Science*, 103(2684):677–680, 1946.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[SWC+08] Dominic Schneider, Alexander Wiebel, Hamish Carr, Mario Hlawitschka, and Gerik Scheuermann. Interactive comparison of scalar fields based on largest contours with applications to flow visualization. In *IEEE Visualization 2008*, pages 1475–1482. IEEE Computer Society, 2008.

[Tam99] Roberto Tamassia. Advances in the theory and practice of graph drawing. *Theoretical Computer Science*, 217(2):235–254, 1999.

[Tel08] Alexandru C. Telea. *Data Visualization: Principles and Practice*. A.K. Peters, 2008.

[TFO09] Shigeo Takahashi, Issei Fujishiro, and Masato Okada. Applying manifold learning to plotting approximate contour trees. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1185–1192, 2009.

[TGSP09] Julien Tierny, Attila Gyulassy, Eddie Simon, and Valerio Pascucci. Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1177–1184, 2009.

[TNTF04] Shigeo Takahashi, Gregory M. Nielson, Yuriko Takeshima, and Issei Fujishiro. Topological volume skeletonization using adaptive tetrahedralization. In *Geometric Modelling and Processing 2004*, pages 227–236, 2004.

[TSL00] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[TTFN05] Yuriko Takeshima, Shigeo Takahashi, Issei Fujishiro, and Gregory M. Nielson. Introducing topological attributes for objective-based visualization of simulated datasets. In *Volume Graphics 2005*, pages 137–146, 236, 2005.

[Tuf97] Edward Rolf Tufte. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, 1997.

[Tuf01] Edward Rolf Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

[vHvW04] Frank van Ham and Jarke J. van Wijk. Interactive visualization of small world graphs. In *IEEE Symposium on Information Visusalization 2004*, pages 199–206. IEEE Computer Society, 2004.

[VV04] Anne Verroust and Marie-Luce Viaud. Ensuring the Drawability of Extended Euler Diagrams for up to 8 Sets. In Alan F. Blackwell, Kim Marriott, and Atsushi Shimojima, editors, *Diagrams*, volume 2980 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 2004.

[vW06] Jarke J. van Wijk. Views on visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):421–433, 2006.

[vWvdW99] Jarke J. van Wijk and Huub van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *IEEE Symposium on Information Visualization*, pages 73–78, 1999.

[War04] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufman, 2nd edition, 2004.

[War08] Colin Ware. *Visual Thinking for Design*. Morgan Kaufmann, 2008.

[WBP07] Gunther Weber, Peer-Timo Bremer, and Valerio Pascucci. Topological landscapes: A terrain metaphor for scientific data. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1416–1423, 2007.

[WDC+07] Gunther Weber, Scott Dillard, Hamish Carr, Valerio Pascucci, and Bernd Hamann. Topology-controlled volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):330–341, 2007.

[WEK01] Roland Wiese, Markus Eiglsperger, and Michael Kaufmann. yFiles: Visualization and Automatic Layout of Graphs. In *Graph Drawing*, pages 453–454, 2001.

[WGK10] Matthew Ward, Georges C. Grinstein, and Daniel Keim. *Interactive Data Visualization: Foundations, Techniques, and Applications*. A.K. Peters, 2010.

[Wil12] Leland Wilkinson. Exact and Approximate Area-Proportional Circular Venn and Euler Diagrams. *IEEE Transactions on Visualization and Computer Graphics*, 18(2):321–331, 2012.

[Wit05] Hans Friedrich Witschel. Terminology extraction and automatic indexing – comparison and qualitative evaluation of methods. In *Proceedings of Terminology and Knowledge Engineering (TKE)*, Copenhagen, Denmark, 2005.

[WSSF+04] Michael T. Wolfinger, W. Andreas Svrcek-Seiler, Christoph Flamm, Ivo L. Hofacker, and Peter F. Stadler. Exact folding dynamics of RNA secondary structures. *Journal Phys. A: Math. Gen.*, 37:4731–4741, 2004.

[WW05] Leland Wilkinson and Graham Willis. *The Grammar of Graphics*. Springer, 2nd edition, 2005.

[ZBB04] Xiaoyu Zhang, Chandrajit L. Bajaj, and Nathan Baker. Fast matching of volumetric functions using multi-resolution dual contour trees. Technical report, Texas Institute for Computational and Applied Mathematics, Austin, Texas, 2004.